# UML Class Diagrams

**Presentation by:**
**Ms. J.K. Josephine Julina**
**Assistant Professor**
**Department of Information Technology**
**SSN College of Engineering**

# Introduction

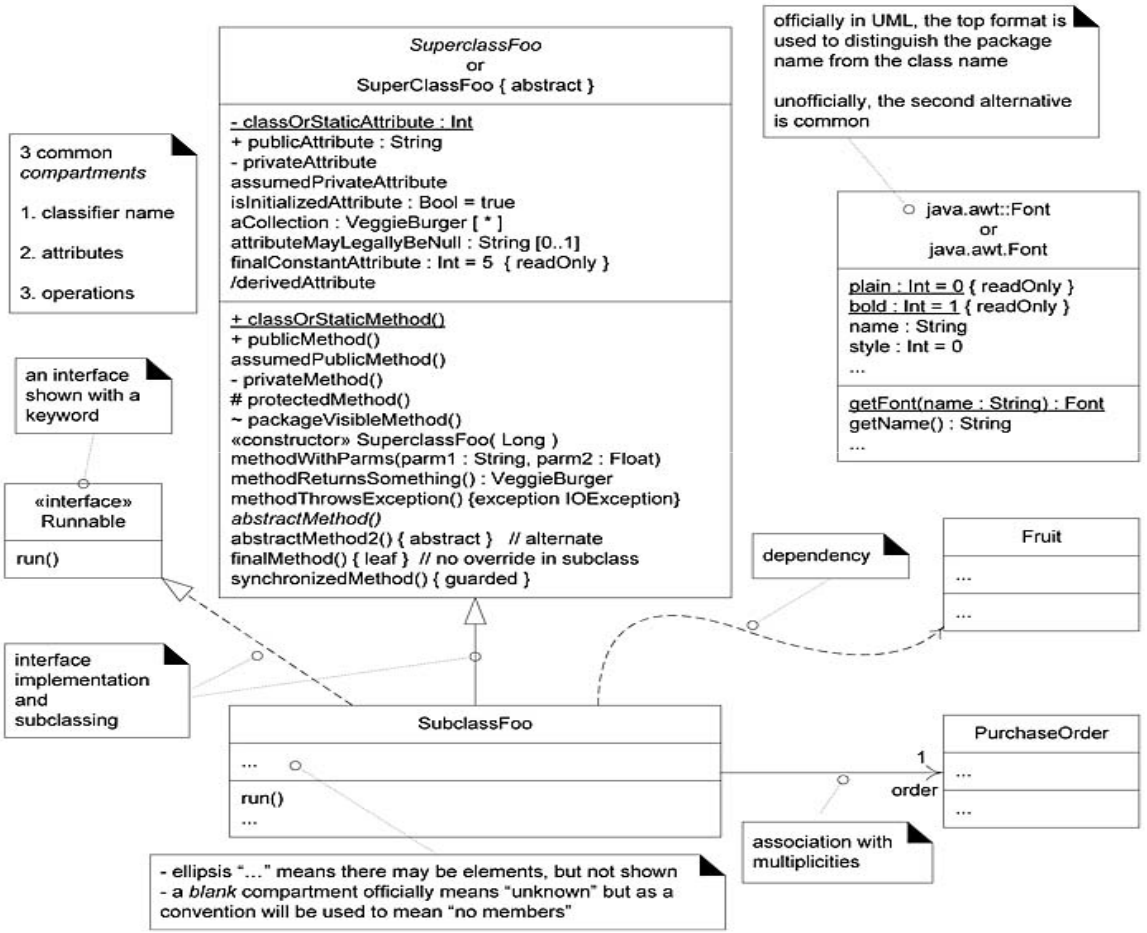
Objective

Provide a reference for frequently used UML class diagram notation

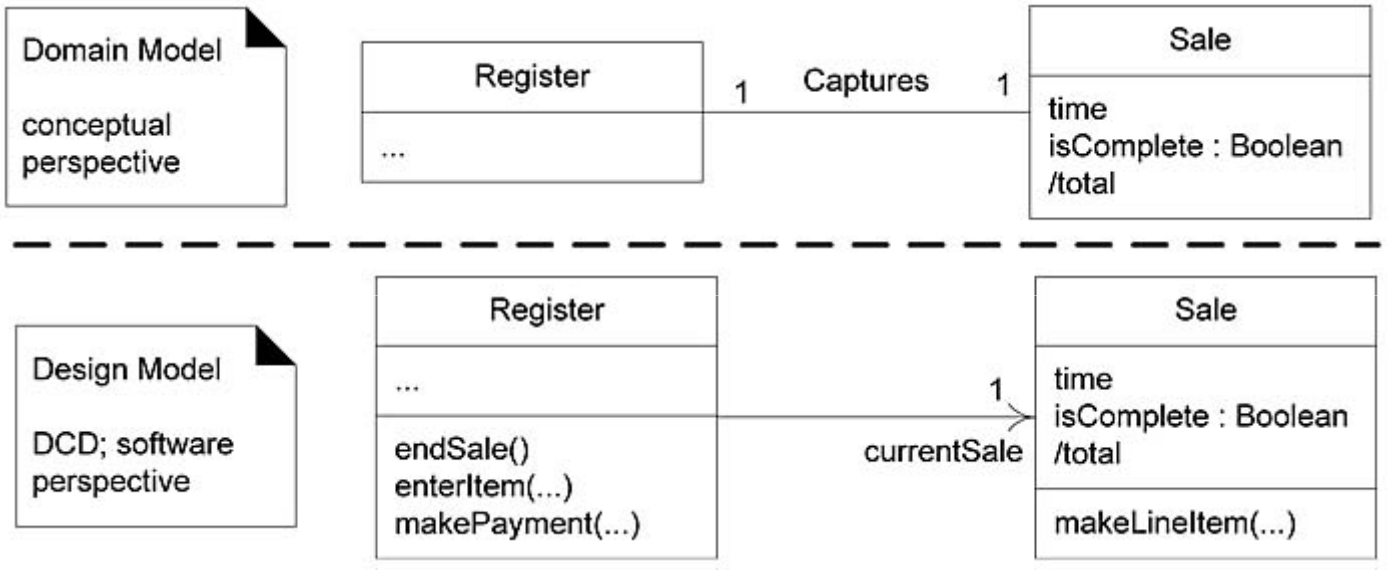
Illustrates classes, interfaces and their associations.
They are used for static object modeling.

Conceptual class in domain model.
Class in software or design perspective.
Design model – Set of all Design Class  Diagram (DCD) and other parts include UML interaction and package diagram.

# Introduction



# Two perspectives



# Classifier

A UML classifier is a “model element that describes behavioral and structure features”.

Classifiers can also be specialized.

Most common classifiers are regular classes and interfaces.

# Attribute Text and Association line

A UML classifier is a “model element that describes behavioral and structure features”.

Attributes of a classifier (structural properties)

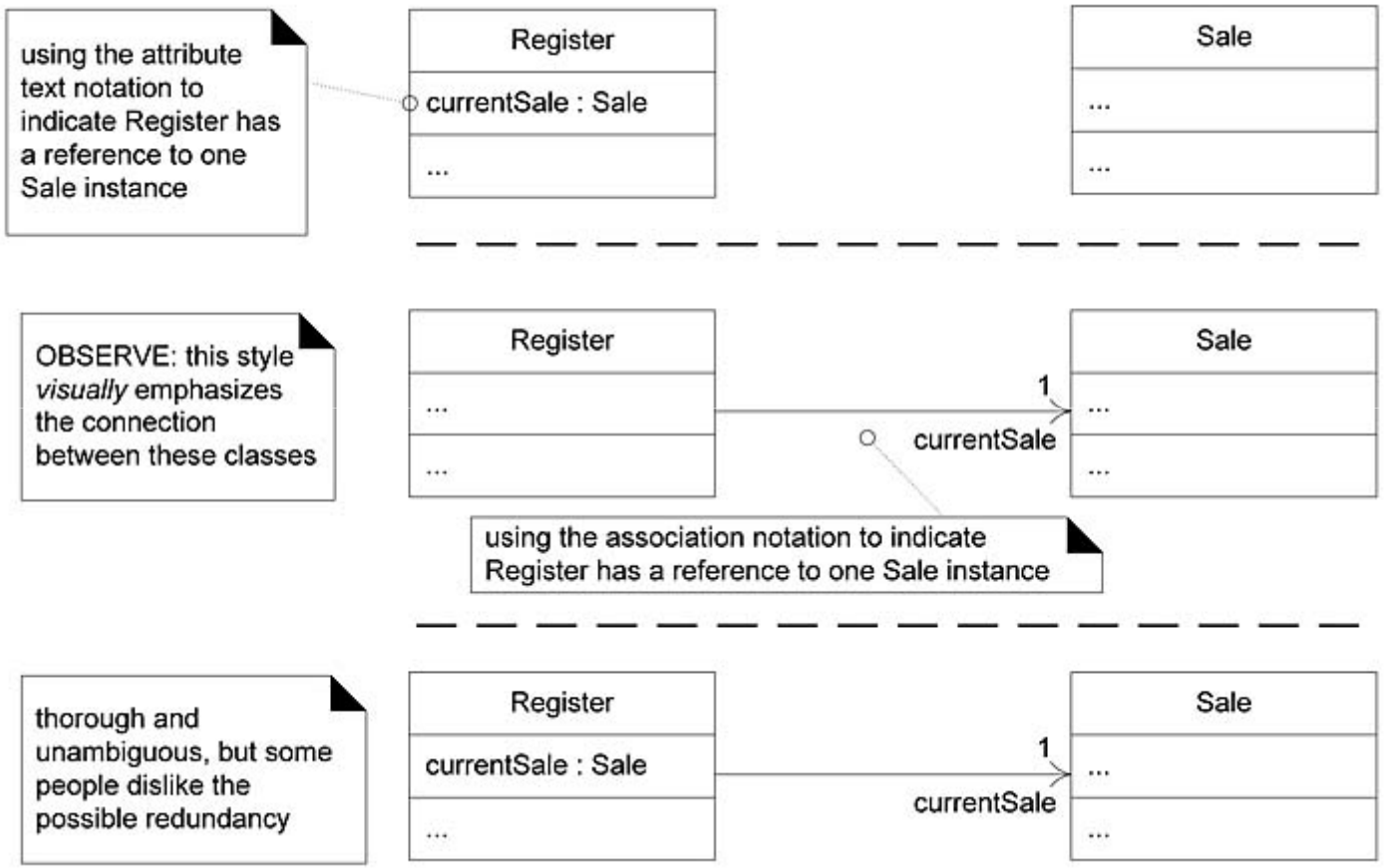
- attribute text notation

visibility name: type multiplicity = default [property-string]

- association line notation
- both together

attribute-as-association line has the following style:
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
use the multiplicity notation
a **rolename** (*currentSale*) only at the target end to show the attribute name
no association name

# Attribute text versus association line notation



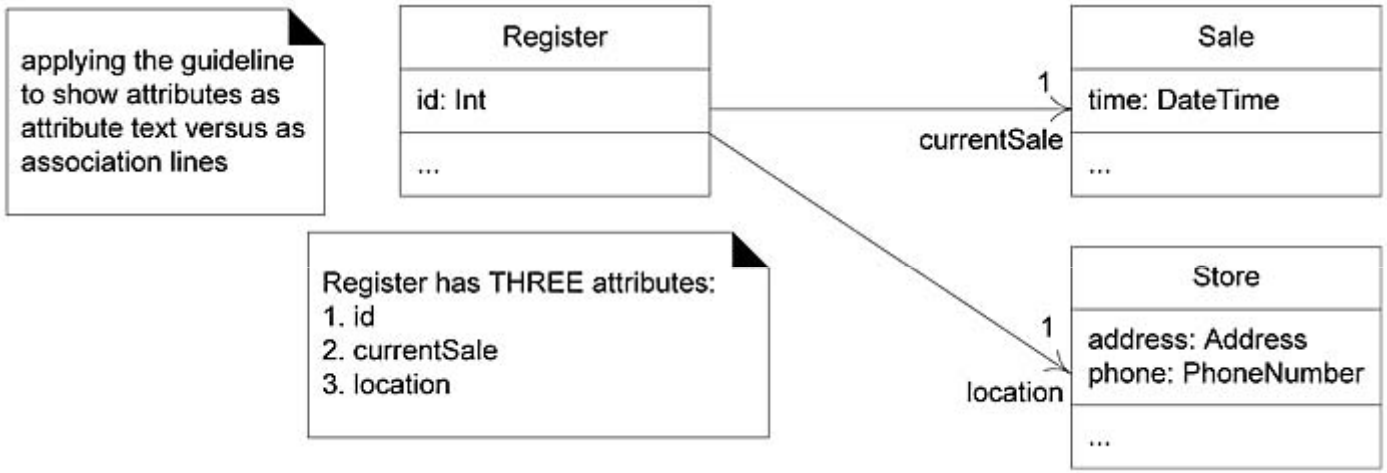
# When to Use Attribute Text versus Association Lines for Attributes?

This question was first explored in the context of domain modeling. To review, a **data type** refers to objects for which unique identity is not important. Common data types are primitive-oriented types such as:
Boolean, Date (or DateTime), Number, Character, String (Text), Time, Address, Color, Geometrics (Point, Rectangle), Phone Number, Social Security Number, Universal Product Code (UPC), SKU, ZIP or postal codes, enumerated types
***Guideline***: Use the attribute text notation for data type objects and the association line notation for others.

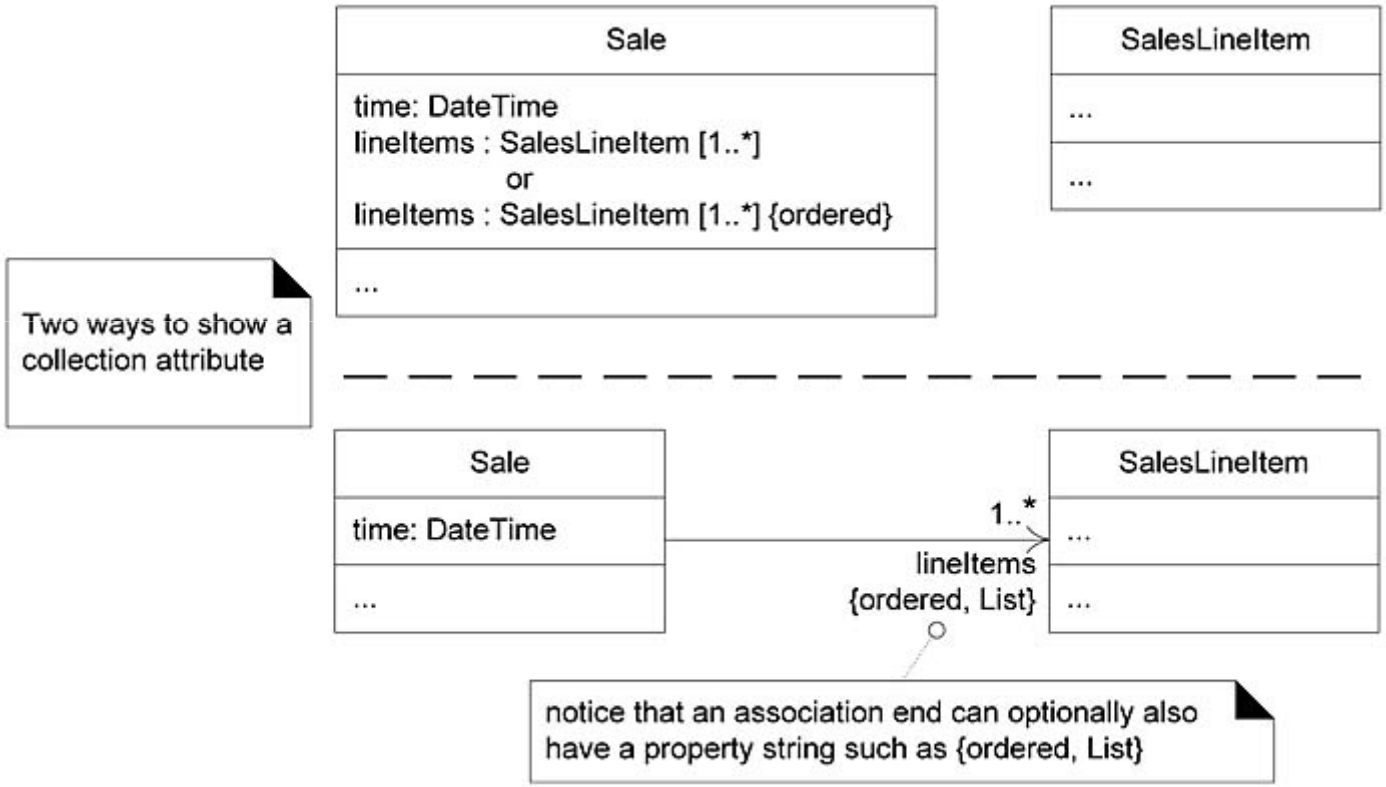
```
public class Register
{ private int id;
private Sale currentSale;
private Store location; // … }
```

The end of an association can have a navigability arrow. It can also include an *optional***rolename** (officially, an **association end name**) to indicate the attribute name. And of course, the association end may also show a **multiplicity** value such as '*' or '0..1'.

# Applying guidelines to show attributes in two notations



# Two ways to show a collection attribute in uml



# Note symbols

Note symbols can be used on any UML diagram, but are especially common on class diagrams. 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
a UML **constraint**, in which case it must be encased in braces '{…}'
a **method** body—the implementation of a UML operation

# Operations and Methods

One of the compartments of the UML class box shows the signatures of operations. official format of the operation syntax is:
**visibility name (parameter-list) {property-string}**
**visibility name (parameter-list) : return-type {property-string}**
The property string contains arbitrary additional information, such as exceptions that may be raised, if the operation is abstract, and so forth.
An operation is *not* a method. A UML **operation** is a *declaration*, with a name, parameters, return type, exceptions list, and possibly a set of *constraints* of pre- and post-conditions.
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»

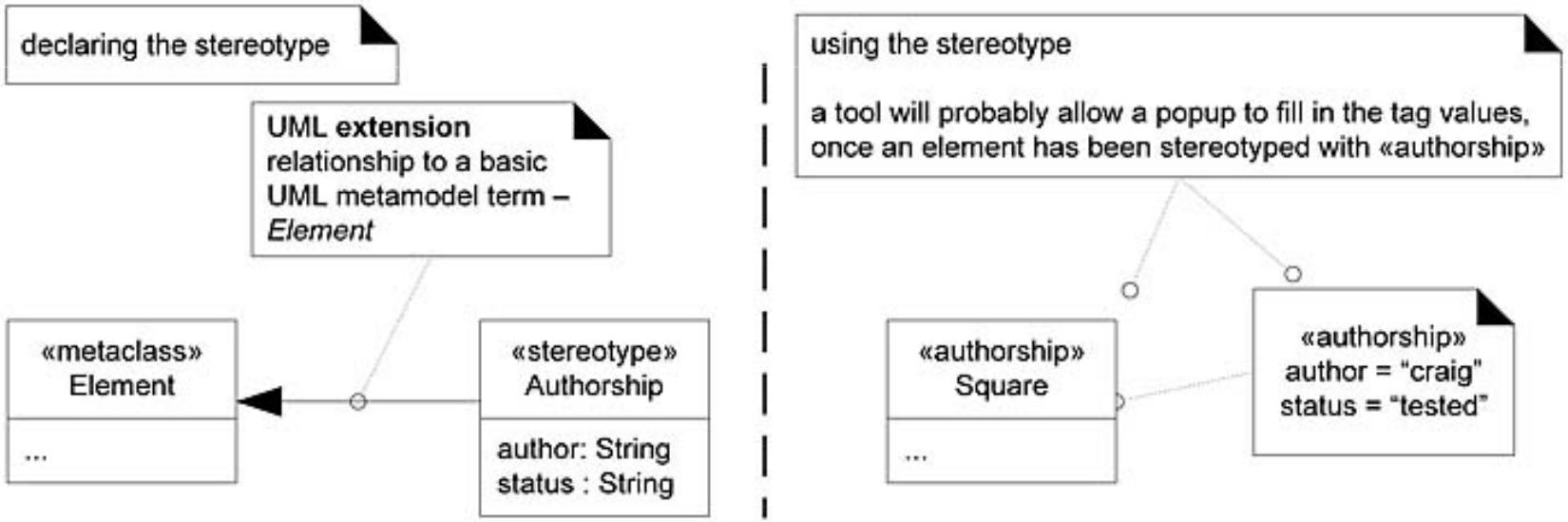
# Keywords

| 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 and Tags

A **stereotype** represents a refinement of an existing modeling concept and is defined within 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.



# UML Properties and Property Strings

In the UML, a **property** is “a named value denoting a characteristic of an element. A property has semantic impact.”

Properties of elements may be presented in many ways, but a textual approach is to use the UML **property string** *{name1=value1, name2=value2}* format, such as *{abstract, visibility=public}*.

Some properties are shown without a value, such as *{abstract}*; this usually implies a boolean property, shorthand for *{abstract=true}*. Note that *{abstract}* is both an example of a *constraint* and a property string.

# Generalization, Abstract Classes and Abstract Operations

**Generalization** in the UML is shown with a solid line and fat triangular arrow from the subclass to superclass.

*Generalization—A taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an indirect instance of the general classifier. Thus, the specific classifier indirectly has features of the more general classifier.*

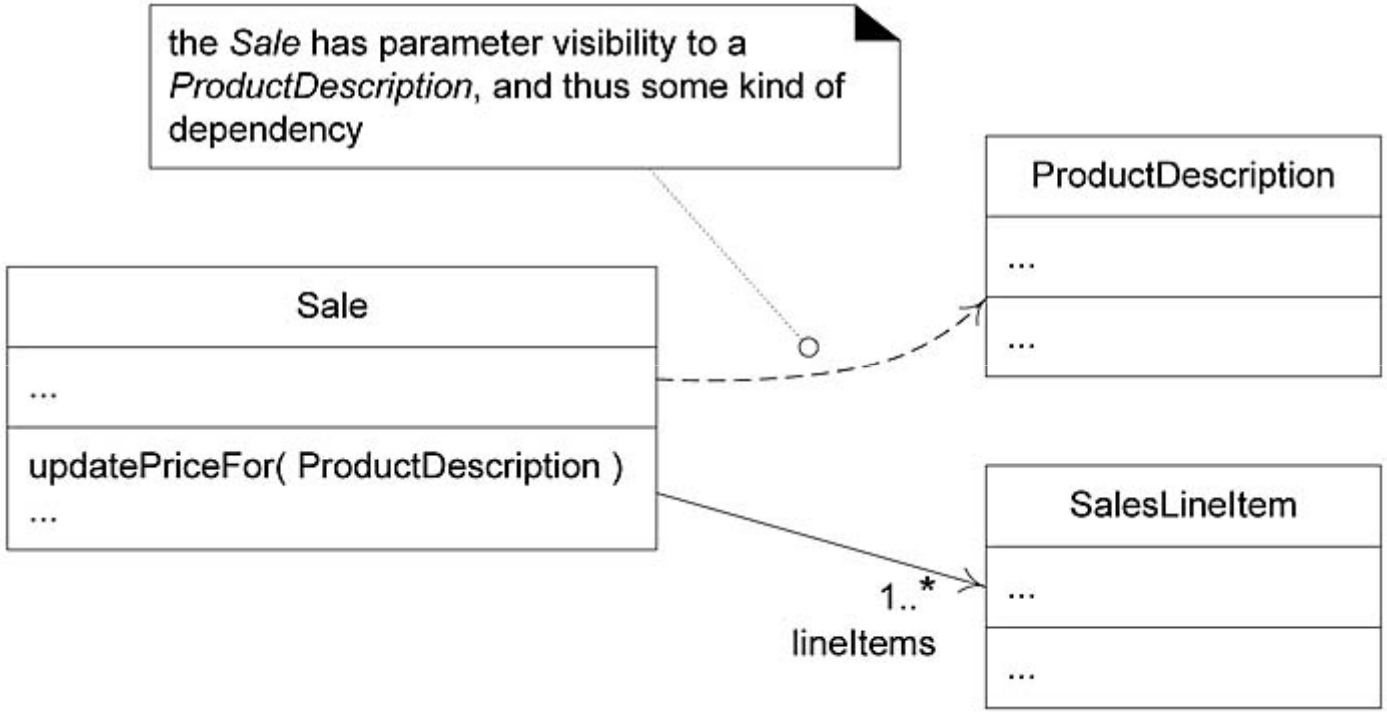
**abstract classes** and operations can be shown either with an *{abstract}* tag (useful when sketching UML) or by italicizing the name (easy to support in a UML tool).
The opposite case, **final classes** and operations that can't be overridden in subclasses, are shown with the *{leaf}* tag.

# Dependency

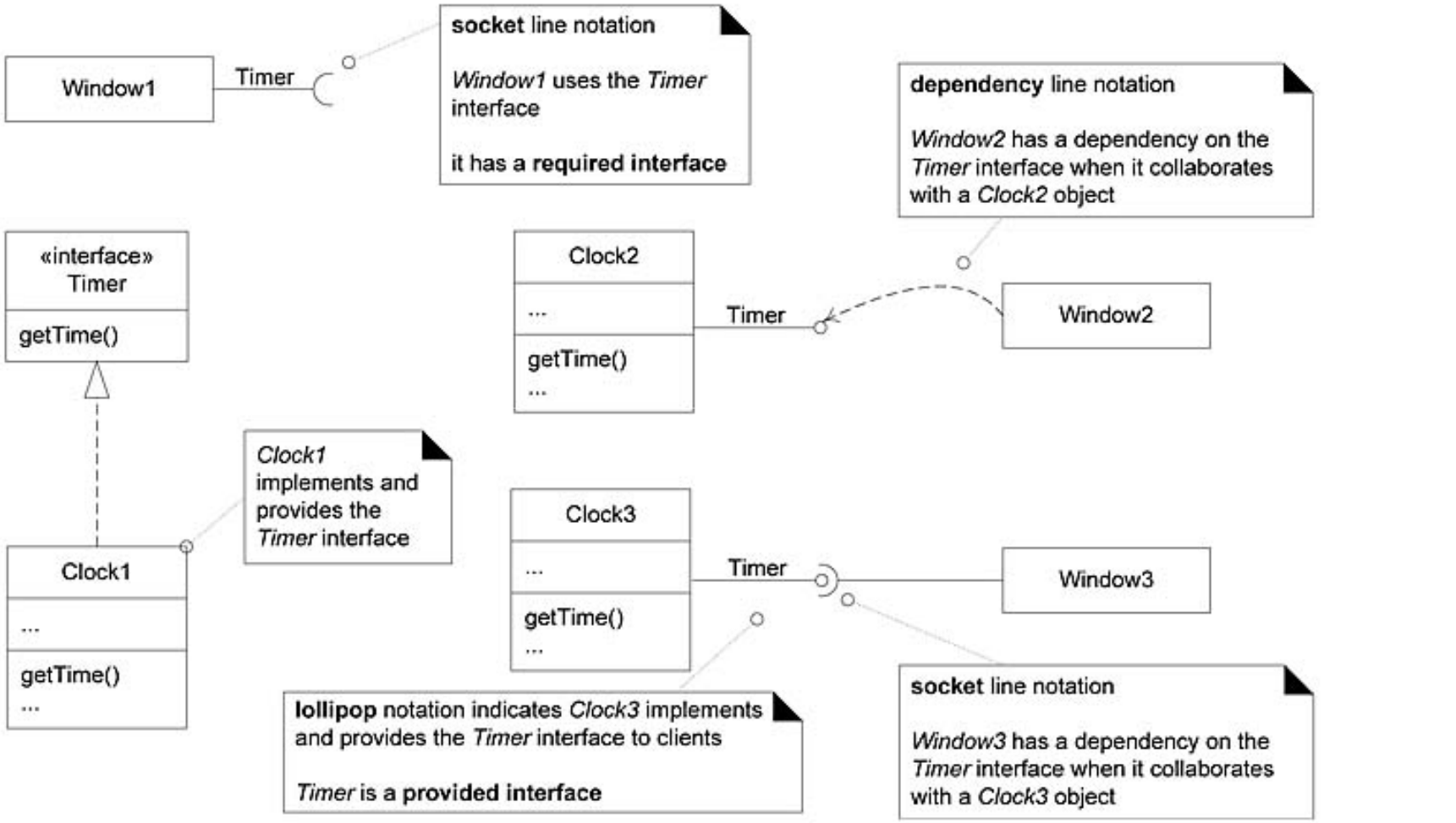
Dependency lines may be used on any diagram, but are especially common on class and package diagrams.
Dependency can be viewed as another version of **coupling**, a traditional term in software development when an element is coupled to or depends on another.

There are many kinds of dependency; here are some common types in terms of objects and class diagrams:
having an attribute of the supplier type
sending a message to a supplier; the visibility to the supplier could be:
    an attribute, a parameter variable, a local variable, a global variable, or class visibility (invoking static or class methods)
receiving a parameter of the supplier type
the supplier is a superclass or interface

# Dependency


the *Sale* has parameter visibility to a *ProductDescription*, and thus some kind of dependency
ProductDescription
...
...
Sale
...
updatePriceFor( ProductDescription )
...
SalesLineItem
...
...
1..*
lineItems

# Interfaces


Window1
Timer
socket line notation
Window1 uses the Timer interface
it has a required interface
dependency line notation
Window2 has a dependency on the Timer interface when it collaborates with a Clock2 object
«interface»
Timer
getTime()
Clock2
...
getTime()
...
Timer
Window2
Clock1
implements and provides the Timer interface
Clock1
...
getTime()
...
Clock3
...
getTime()
...
Timer
Window3
lollipop notation indicates Clock3 implements and provides the Timer interface to clients
Timer is a provided interface
socket line notation
Window3 has a dependency on the Timer interface when it collaborates with a Clock3 object

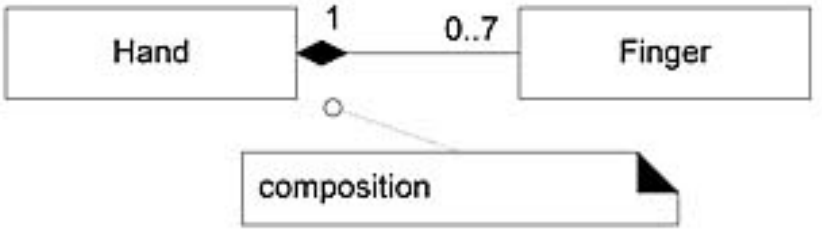
# Composition over Aggregation

**Aggregation** is a vague kind of association in the UML that loosely suggests whole-part relationships (as do many ordinary associations).

It has no meaningful distinct semantics in the UML versus a plain association, but the term is defined in the UML.

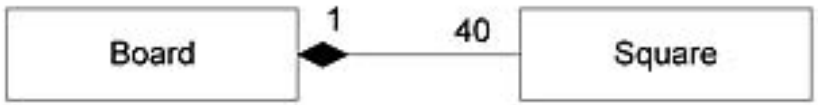
**Composition**, also known as **composite aggregation**, is a strong kind of whole-part aggregation and *is* useful to show in some models.
- an instance of the part (such as a *Square*) belongs to only *one* composite instance (such as one *Board*) at a time
- the part must *always belong* to a composite (no free-floating *Fingers),*
- the composite is responsible for the creation and deletion of its parts—either by itself creating/deleting the parts, or by collaborating with other objects.

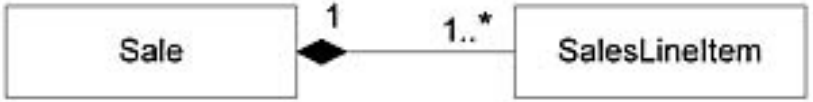
# Composition over Aggregation




composition means
-a part instance (*Square*) can only be part of one composite (*Board*) at a time

-the composite has sole responsibility for management of its parts, especially creation and deletion

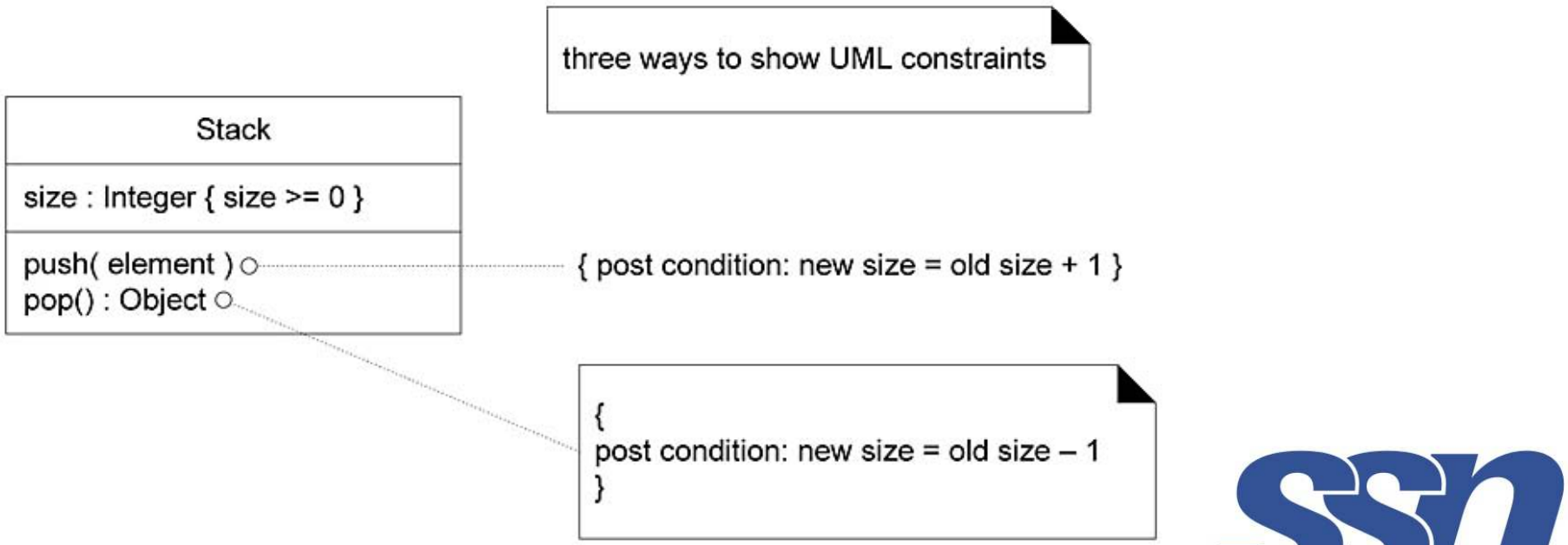


# Constraints

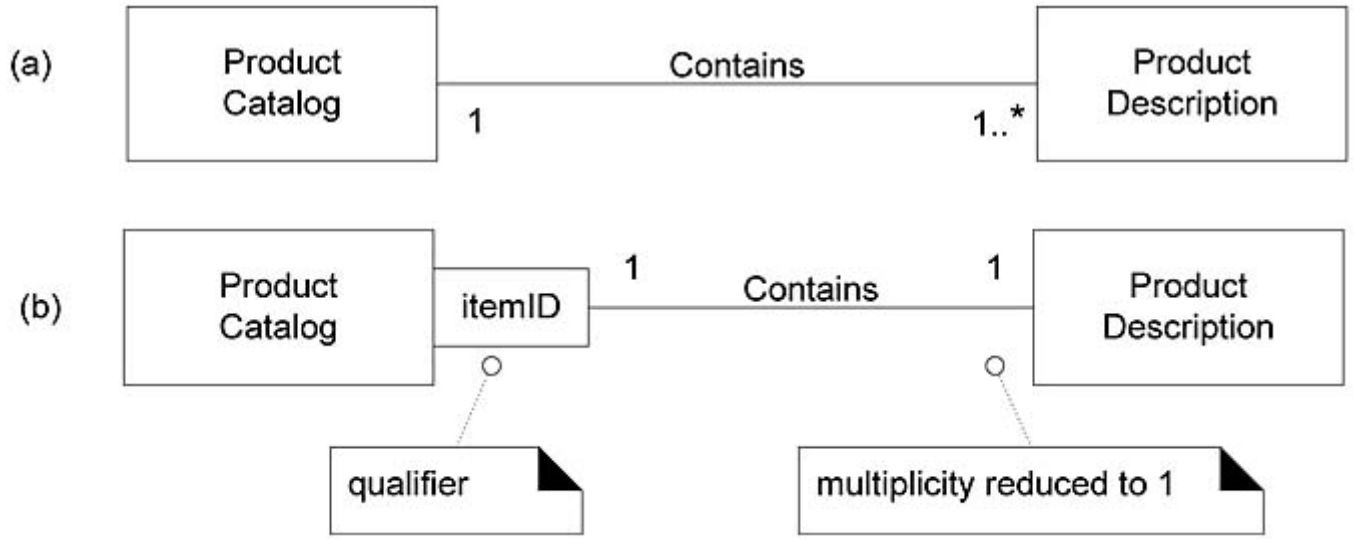
Constraints may be used on most UML diagrams, but are especially common on class diagrams.
A UML **constraint** is a restriction or condition on a UML element. It is visualized in text between braces; for example: *{ size >= 0 }*.

The text may be natural language or anything else, such as UML's formal specification language, the **Object Constraint Language** (OCL)



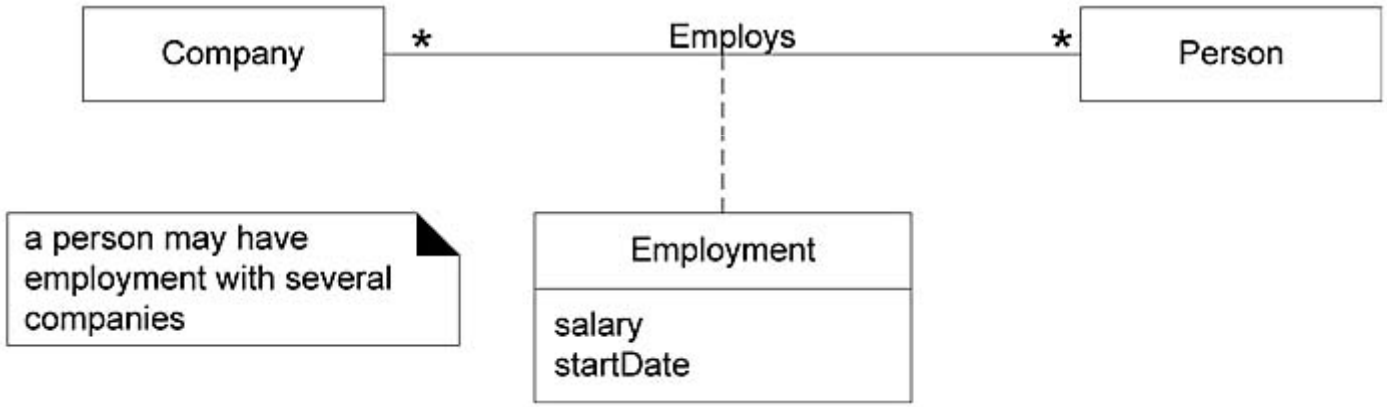
# Qualified Association

A **qualified association** has a **qualifier** that is used to select an object (or objects) from a larger set of related objects, based upon the qualifier key.



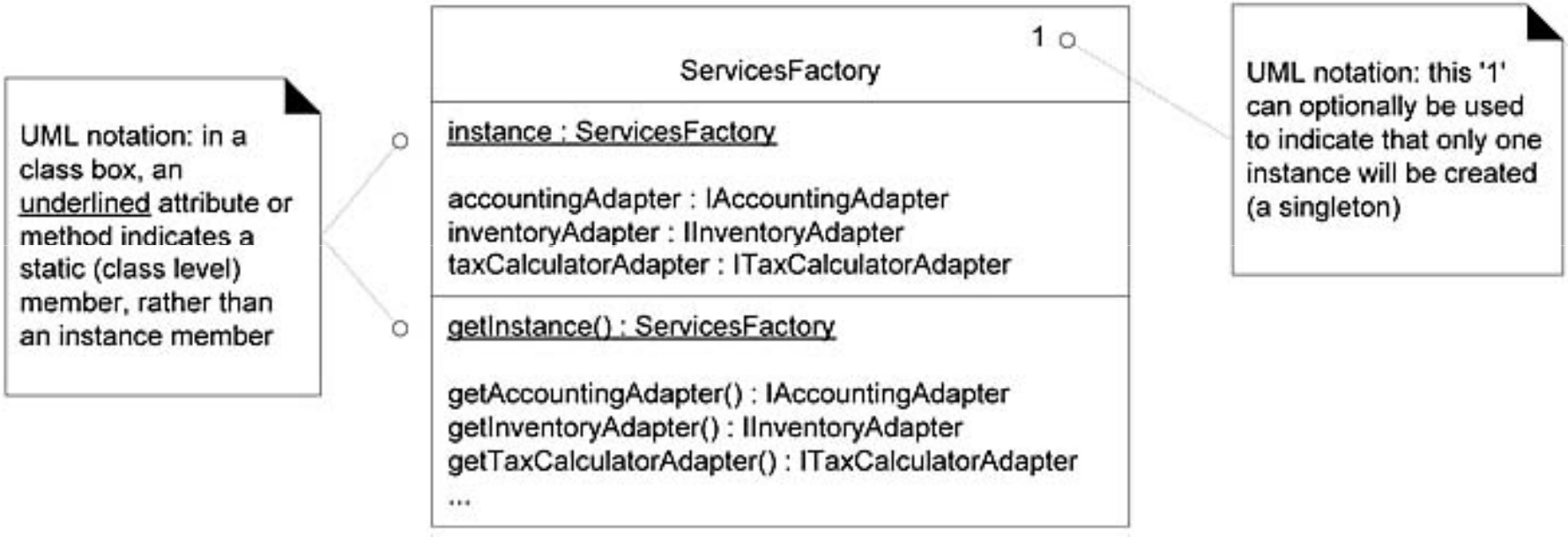
# Association Class

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



# Singleton Classes

*Singleton instance* - *one* instance of a class instantiated — never two.



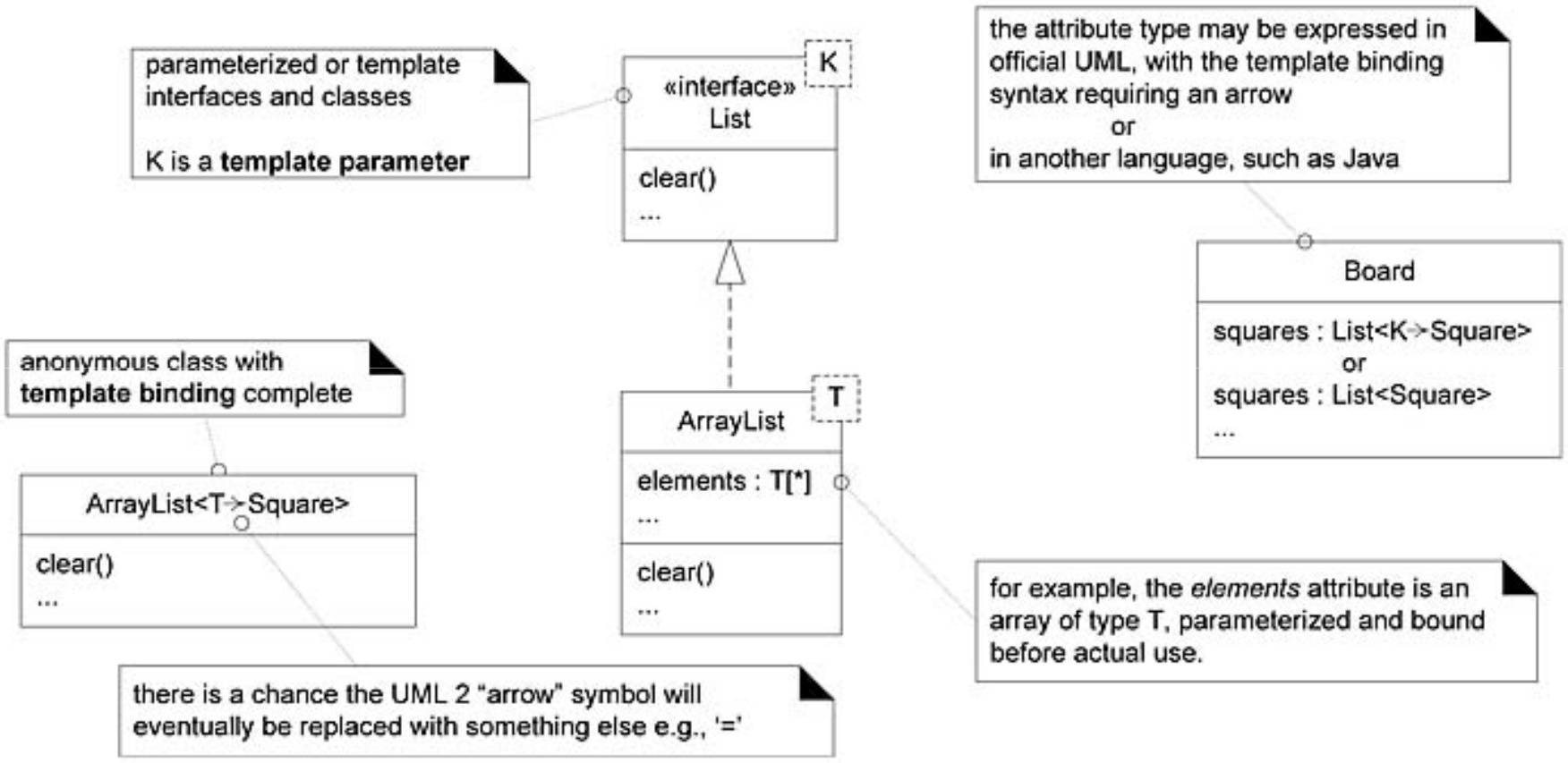
# Template Classes and Interfaces

Many languages (Java, C++, …) support **templatized types**, also known (with shades of variant meanings) as **templates**, **parameterized types**, and **generics**.

They are most commonly used for the element type of collection classes, such as the elements of lists and maps.

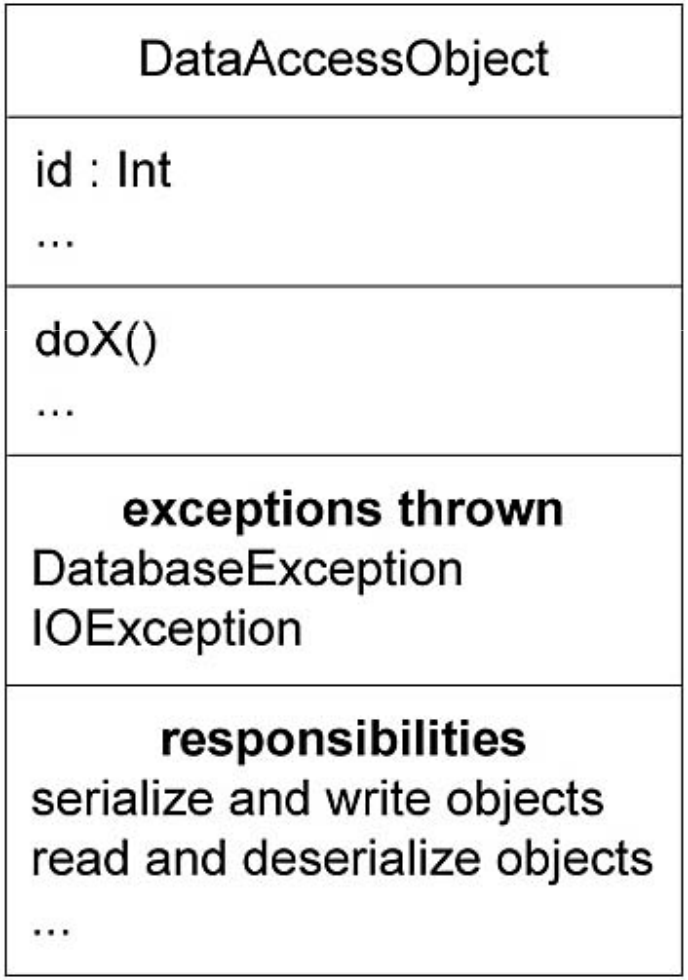
```
public class Board
{ private List<Square> squares = new ArrayList<Square>(); // … }
```

# Template Classes and Interfaces



# User-Defined Compartments

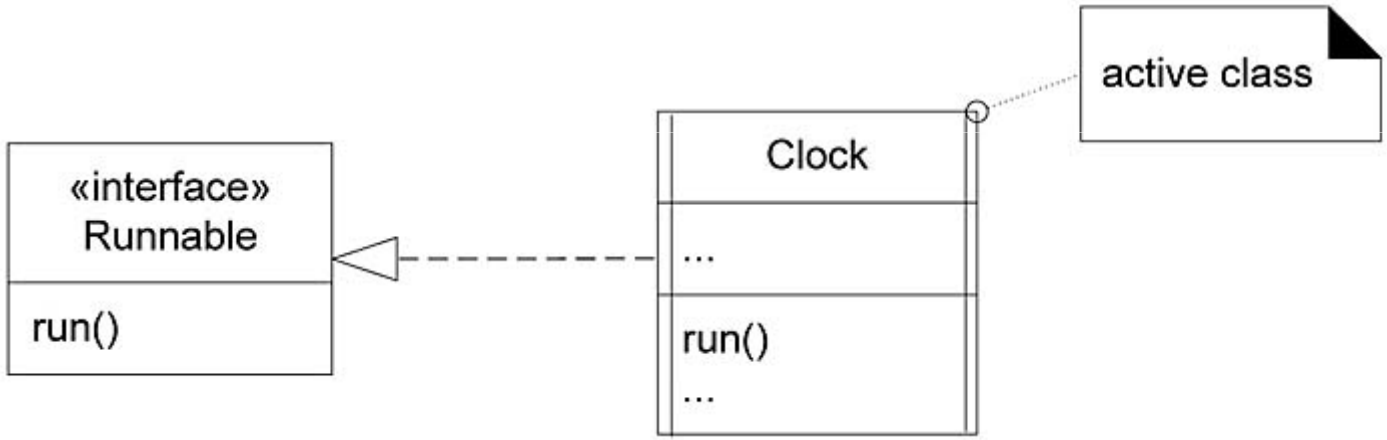
In addition to common predefined **compartments** class compartments such as name, attributes, and operations, user-defined compartments can be added to a class box.




# Active Class

An **active object** runs on and controls its own thread of execution. Not surprisingly, the class of an active object is an **active class**.

In the UML, it may be shown with double vertical lines on the left and right sides of the class box.



# The influence of interaction diagrams on class diagrams