Special Topic 12.1

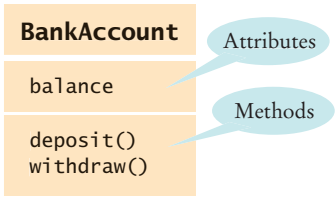
Attributes and Methods in UML Diagrams

Sometimes it is useful to indicate class *attributes* and *methods* in a class diagram. An **attribute** is an externally observable property that objects of a class have. For example, `name` and `price` would be attributes of the `Product` class. Usually, attributes correspond to instance variables. But they don't have to—a class may have a different way of organizing its data. For example, a `GregorianCalendar` object from the Java library has attributes `day`, `month`, and `year`, and it would be appropriate to draw a UML diagram that shows these attributes. However, the class doesn't actually have instance variables that store these quantities. Instead, it

internally represents all dates by counting the milliseconds from January 1, 1970—an implementation detail that a class user certainly doesn't need to know about.

You can indicate attributes and methods in a class diagram by dividing a class rectangle into three compartments, with the class name in the top, attributes in the middle, and methods in the bottom (see the figure below). You need not list *all* attributes and methods in a particular diagram. Just list the ones that are helpful to understand whatever point you are making with a particular diagram.

Also, don't list as an attribute what you also draw as an aggregation. If you denote by aggregation the fact that a `Car` has `Tire` objects, don't add an attribute `tires`.



Attributes and Methods in a Class Diagram

Special Topic 12.2

Multiplicities

Some designers like to write *multiplicities* at the end(s) of an aggregation relationship to denote how many objects are aggregated. The notations for the most common multiplicities are:

- any number (zero or more): `*`
- one or more: `1..*`
- zero or one: `0..1`
- exactly one: `1`

The figure below shows that a customer has one or more bank accounts.

An Aggregation Relationship with Multiplicities

Special Topic 12.3

Aggregation and Association

Some designers find the aggregation or *has-a* terminology unsatisfactory. For example, consider customers of a bank. Does the bank "have" customers? Do the customers "have" bank accounts, or does the bank "have" them? Which of these "has" relationships should be modeled by aggregation? This line of thinking can lead us to premature implementation decisions.

Early in the design phase, it makes sense to use a more general relationship between classes called **association**. A class is associated with another if you can *navigate* from objects of one class to objects of the other class. For example, given a `Bank` object, you can navigate to `Customer` objects, perhaps by accessing an instance variable, or by making a database lookup.

The UML notation for an association relationship is a solid line, with optional arrows that show in which directions you can navigate the relationship. You can also add words to the line ends to further explain the nature of the relationship. The figure below shows that you can navigate from `Bank` objects to `Customer` objects, but you cannot navigate the other way around. That is, in this particular design, the `Customer` class has no mechanism to determine in which banks it keeps its money.

Frankly, the differences between aggregation and association are confusing, even to experienced designers. If you find the distinction helpful, by all means use the relationship that you find most appropriate. But don't spend time pondering subtle differences between these concepts. From the practical point of view of a Java programmer, it is useful to know when objects of one class manage objects of another class. The aggregation or *has-a* relationship accurately describes this phenomenon.

An Association Relationship