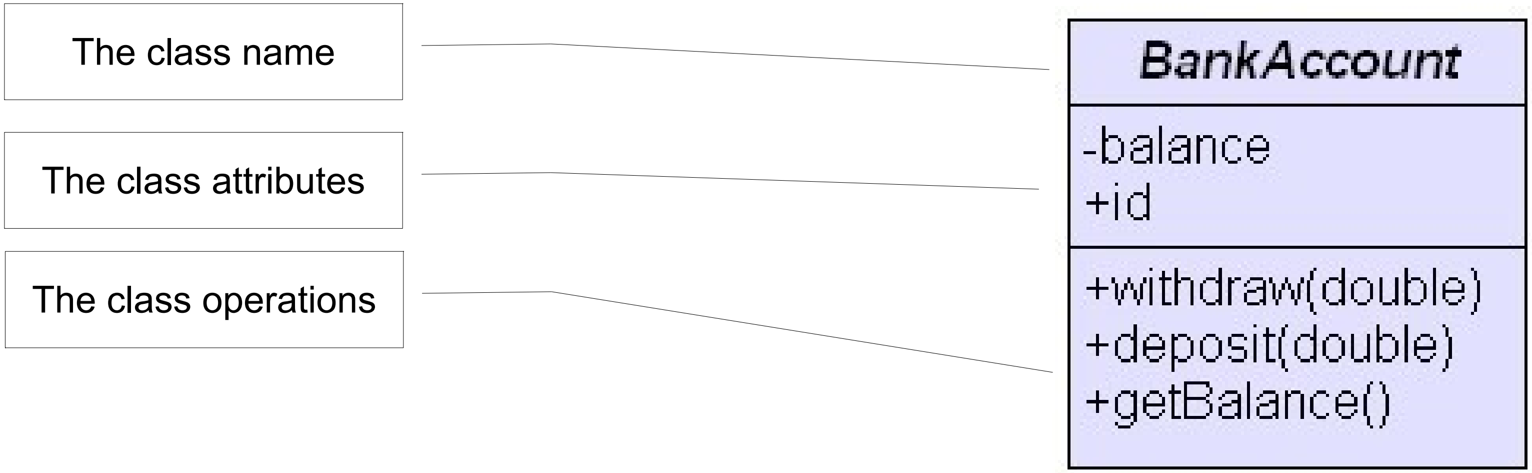
UML Class Diagrams

Introduction

- The UML Class diagram provides information about the classes we are going to declare, their relationships with each other, their attributes and their operations.
- The UML Class diagram depicts the detailed static design of our object oriented planned software.
- A Class is represented with a rectangular box divided into compartments used for holding its name, its attributes and its operations.

Introduction



The class name

The class attributes

The class operations

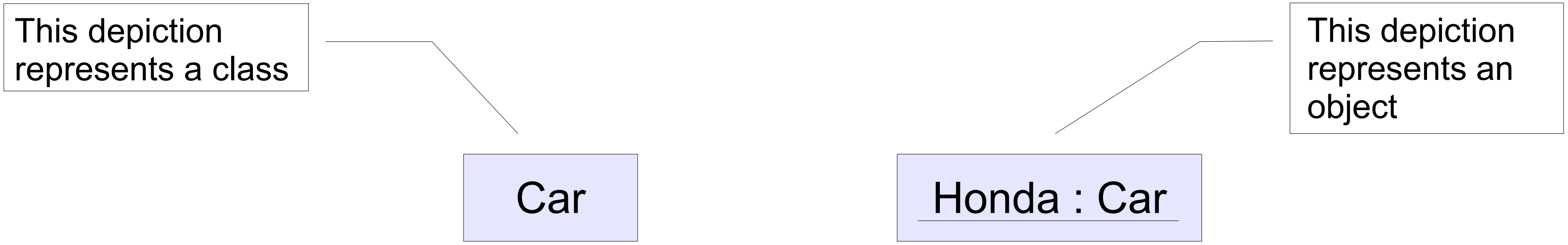
BankAccount

-balance
+id

+withdraw(double)
+deposit(double)
+getBalance()

Introduction

- An object is an instance of a class. Depicting an object in our diagram is done by drawing an empty rectangle and writing the object name + ':' + its type and an underline. The underline will differentiate this depiction from a class description. A Class depiction doesn't include the underline.

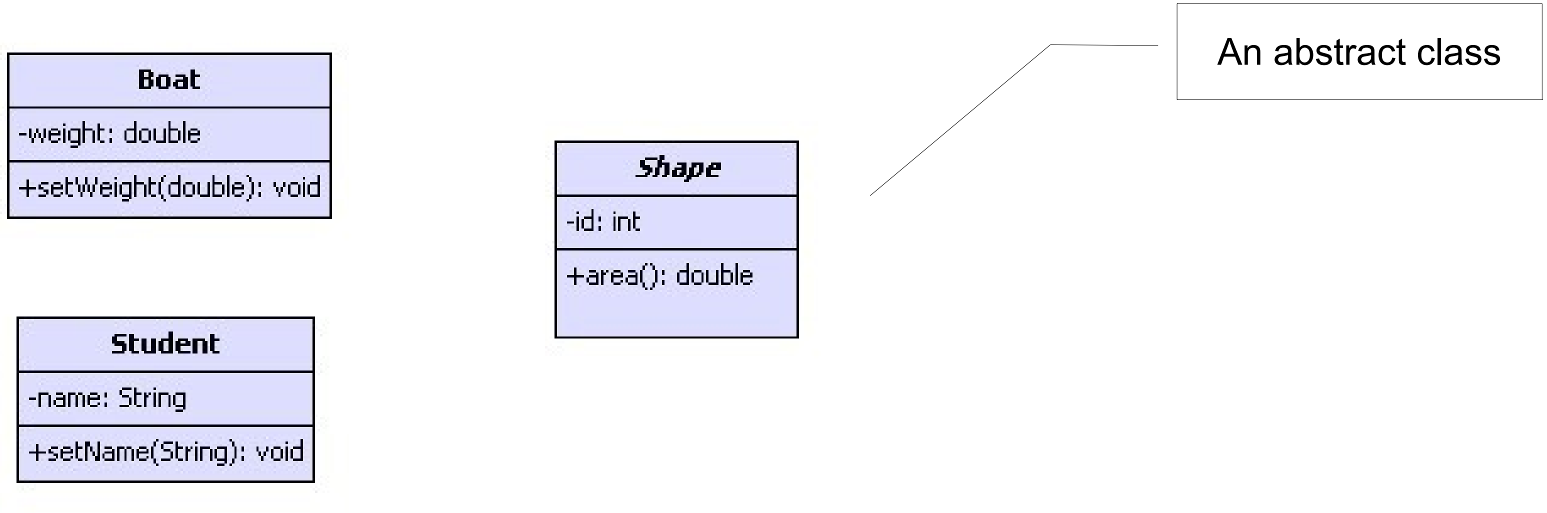


This depiction represents a class

This depiction represents an object

The Class Name

- UML suggests that a class name should start with a capital letter, be centered in the top compartment, be written in a boldface font and be written in italics if the class is abstract.



Boat
-weight: double
+setWeight(double): void

Student
-name: String
+setName(String): void

Shape
-id: int
+area(): double

An abstract class

Visibility Possibilities

- UML Class diagram allows using four different visibility levels:

- Private

+ Public

# Protected

~ Package

Representing Class Attributes

- Representing attributes should use the following notation:

  visibility / name : type multiplicity = default

  {property strings and constraints}

  visibility - should be one of these symbols: +, -, # or ~

  / - indicates whether the attribute is a derived one.

  name - is a noun or a short phrase naming the attribute.

  type - this is the type of the attribute... can be either a class type or a primitive one.

  multiplicity - specifies the number of instances of the attribute type.

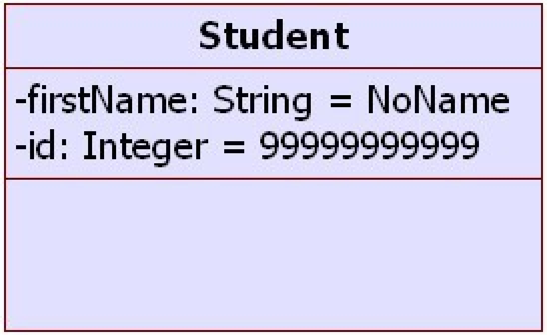
  default - this is the default value of the attribute.

  property strings - is a collection of properties (or tags) that can be attached.

  constraints - one or more restrictions placed on an attribute.

Representing Class Attributes

- The attribute should be placed within the second compartment of the class (below the top compartment where we write the class title).

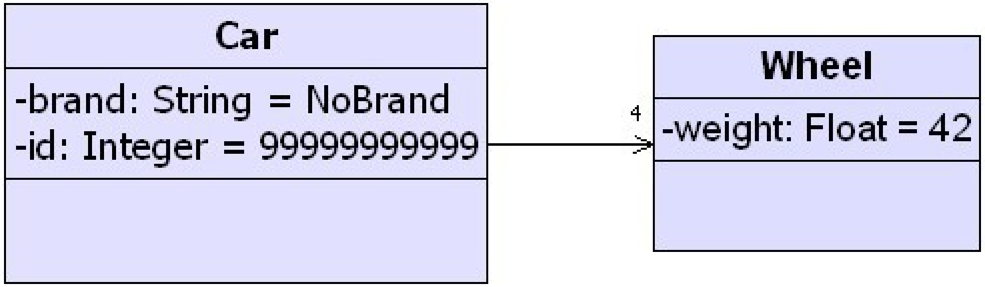


Student

-firstName: String = NoName
-id: Integer = 99999999999

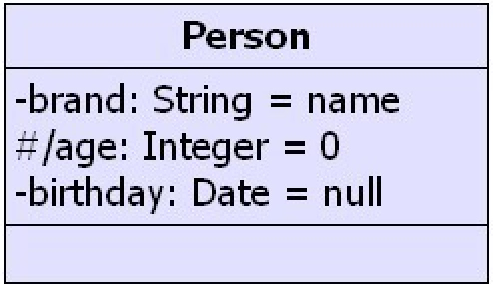
Attributes via Class Relationships

- You can represent attributes using the relationship notation connecting between the class you now describe and another class that will be instantiated in order to get an object that its reference will be saved within the class attribute we now describe.



Derived Attributes

- The derived notation (/) indicates a redundant attribute, that its value depends on the value of other attribute/s in the same object.



Person

-brand: String = name
#/age: Integer = 0
-birthday: Date = null

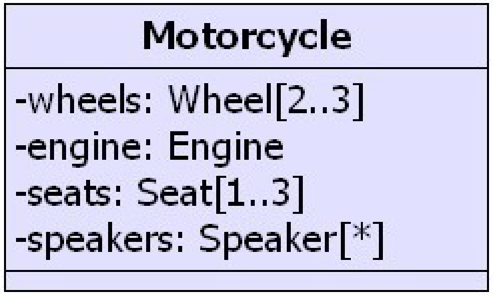
Attributes Multiplicity

- The multiplicity characteristic of attribute denotes how many instances of the attribute type are created.

  When omitting the multiplicity default is 1.

  When specifying a range of possible values the * represents infinity.

  When specifying * only, it means zero or more (infinity).



Motorcycle

-wheels: Wheel[2..3]
-engine: Engine
-seats: Seat[1..3]
-speakers: Speaker[*]

Attribute Properties

- An attribute might have a number of properties that convey additional information:

readOnly specifies the attribute can't be modified once its initial value is set.

union specifies that the attribute type is a union of possible values.

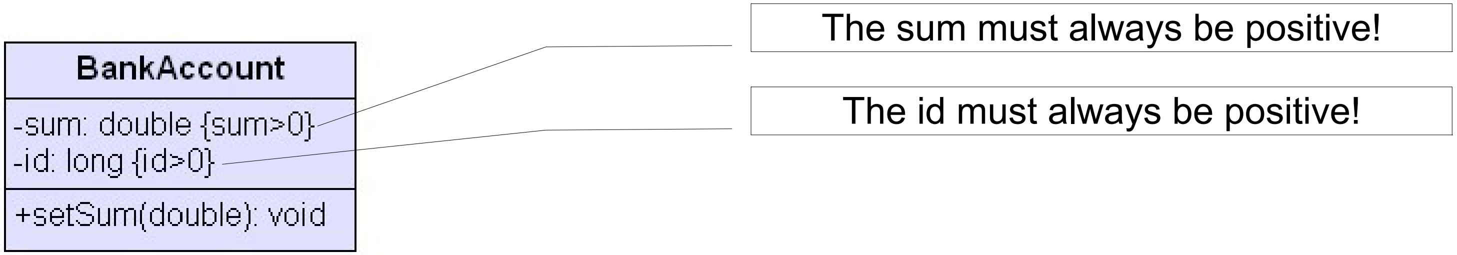
subsets <attribute name> specifies that the possible values for this attribute are a subset of all valid values for the other attribute.

redefines <attribute name> specifies that this attribute is kind of an alias for the other attribute.

composite specifies that this attribute takes part of a relationship between this class and other\s.

Attribute Constraints

- A constraint represents a restriction we set for a specific attribute.
- We can depict a constraint writing it inside curly braces { } as a boolean expression that should be true.



BankAccount

-sum: double {sum>0}
-id: long {id>0}

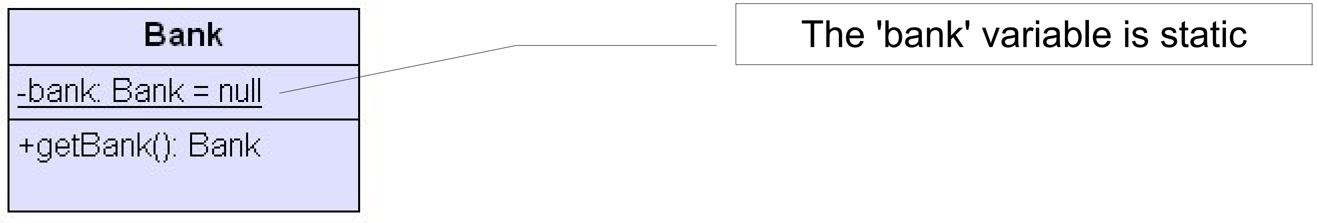
+setSum(double): void

The sum must always be positive!

The id must always be positive!

Static Attributes

- Static attributes are represented by placing an underline beneath the attribute specification.



The 'bank' variable is static

Representing Class Operations

- Representing operations should use the following notation:

  visibility name (parameters) : return type {properties}

  Each parameter should use the following notation:

  direction parameter_name : type [multiplicity] = default_value {properties}

visibility	indicates the visibility of the operation (+,-,# or ~)
name	this is the operation name
return type	this is the type of information that will be returned
properties	specifies constraints and properties associated with the operation
direction	can be one of the following: in, inout, out or return.
parameter_name	starts with a lowercase subsequent with a capital one.
type	class type or a primitive type.

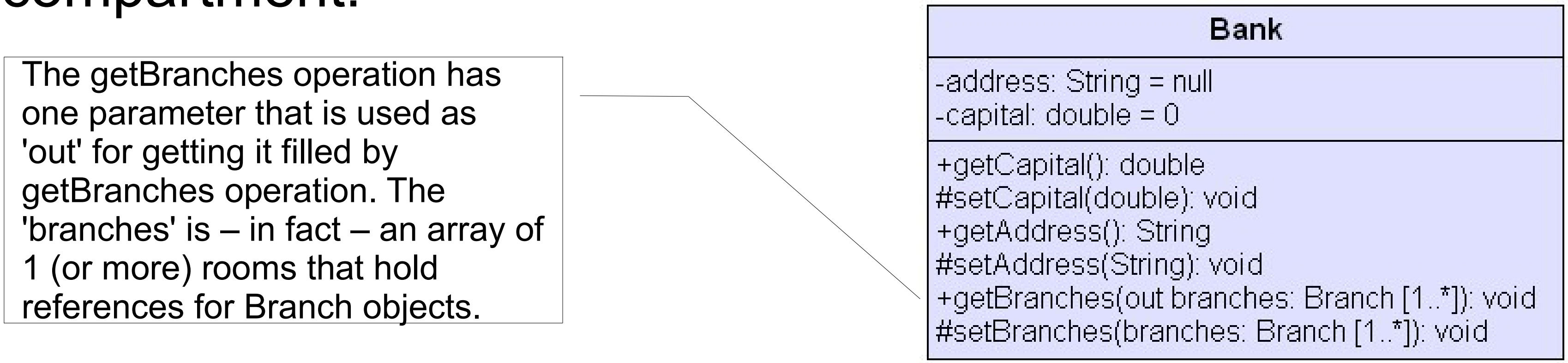
Representing Class Operations

multiplicity specifies the number of instances of the parameter type

default_value default value that will be set if no argument is sent

properties specifies within curly brackets parameter properties such as read only and unique.

- The operations are specified within the separated bottom compartment.



The getBranches operation has one parameter that is used as 'out' for getting it filled by getBranches operation. The 'branches' is – in fact – an array of 1 (or more) rooms that hold references for Branch objects.

Bank

-address: String = null
-capital: double = 0

+getCapital(): double
#setCapital(double): void
+getAddress(): String
#setAddress(String): void
+getBranches(out branches: Branch [1..*]): void
#setBranches(branches: Branch [1..*]): void

Operation Constraints

- Constraints (Preconditions & Postconditions) can be placed within curly brackets immediately after the operation signature.

- Preconditions

  Condition the system needs to meet before the operation call.

  Will be denoted using the following notation:

  {precondition: boolean_expression}

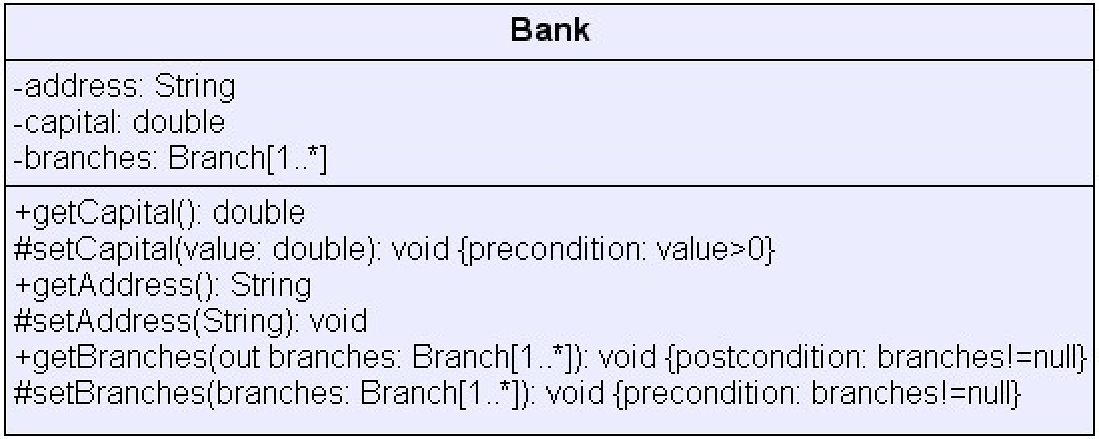
- Postcondition

  Condition the system needs to meet after the operation call.

  Will be denoted using the following notation:

  {postcondition: boolean_expression}

Operation Constraints



Bank

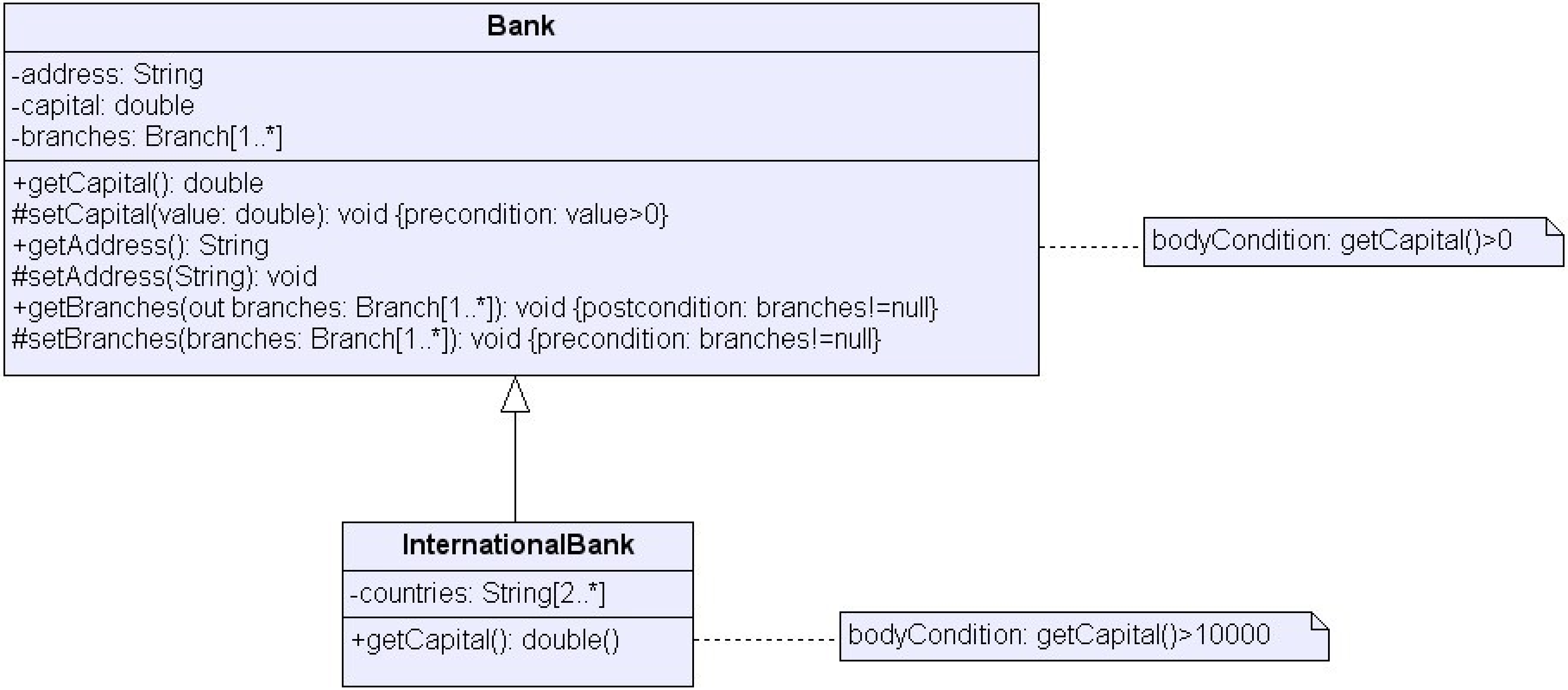
-address: String
-capital: double
-branches: Branch[1..*]

+getCapital(): double
#setCapital(value: double): void {precondition: value>0}
+getAddress(): String
#setAddress(String): void
+getBranches(out branches: Branch[1..*]): void {postcondition: branches!=null}
#setBranches(branches: Branch[1..*]): void {precondition: branches!=null}

Body Conditions

- The body condition describes a condition the returned value needs to meet.
- Unlike Postcondition, the body condition can be replaced with another condition by extending classes.
- The body condition is presented within a note connected with a dashed line to the operation specification it refers.

Body Conditions



Bank

-address: String
-capital: double
-branches: Branch[1..*]

+getCapital(): double
#setCapital(value: double): void {precondition: value>0}
+getAddress(): String
#setAddress(String): void
+getBranches(out branches: Branch[1..*]): void {postcondition: branches!=null}
#setBranches(branches: Branch[1..*]): void {precondition: branches!=null}

bodyCondition: getCapital()>0

InternationalBank

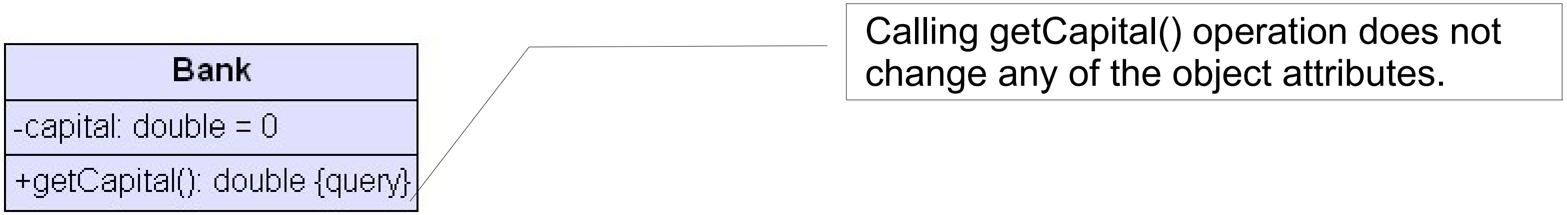
-countries: String[2..*]

+getCapital(): double()

bodyCondition: getCapital()>10000

Query Operations

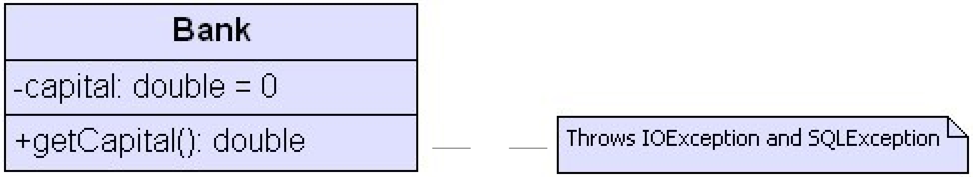
- A query operation is an operation that does not modify any of the object's attributes.
- Marking an operation as a query one is done by appending the {query} constraint to the operation signature.



Calling getCapital() operation does not change any of the object attributes.

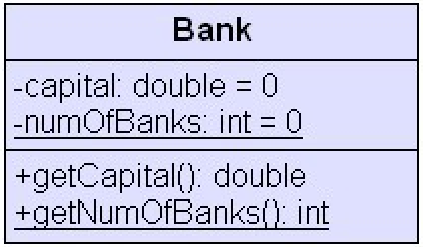
Operations Exceptions

- It is possible to inform about exceptions that might be thrown using a small note connected with a dashed line to the operation it refers.



Static Operations

- A static operation is marked with an underline.



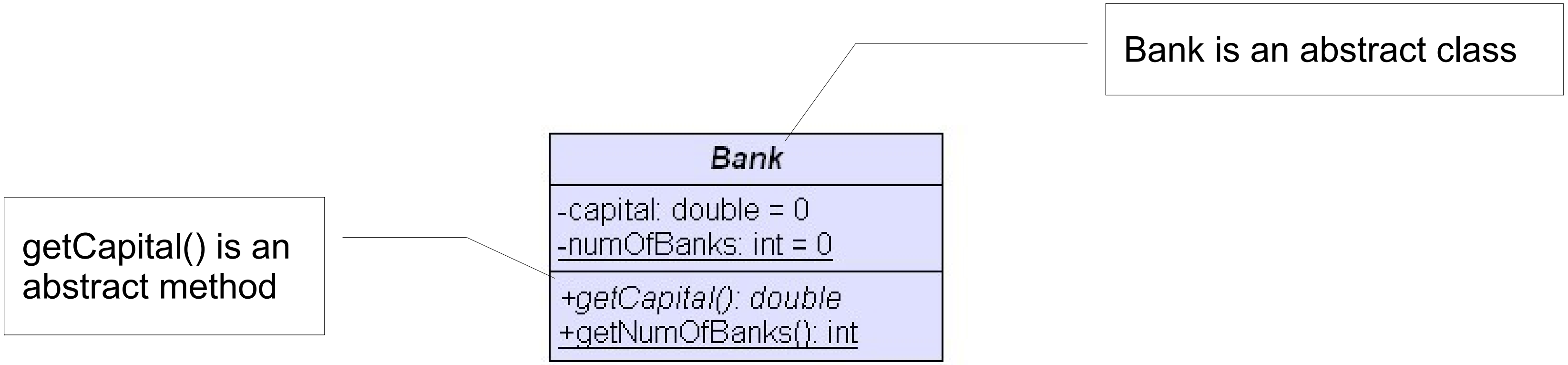
Bank

-capital: double = 0
-numOfBanks: int = 0

+getCapital(): double
+getNumOfBanks(): int

Abstract Class

- An abstract class is denoted by writing the class name in italics.
- The abstract operation is denoted by writing its name in italics as well.



Bank is an abstract class

getCapital() is an abstract method

Bank

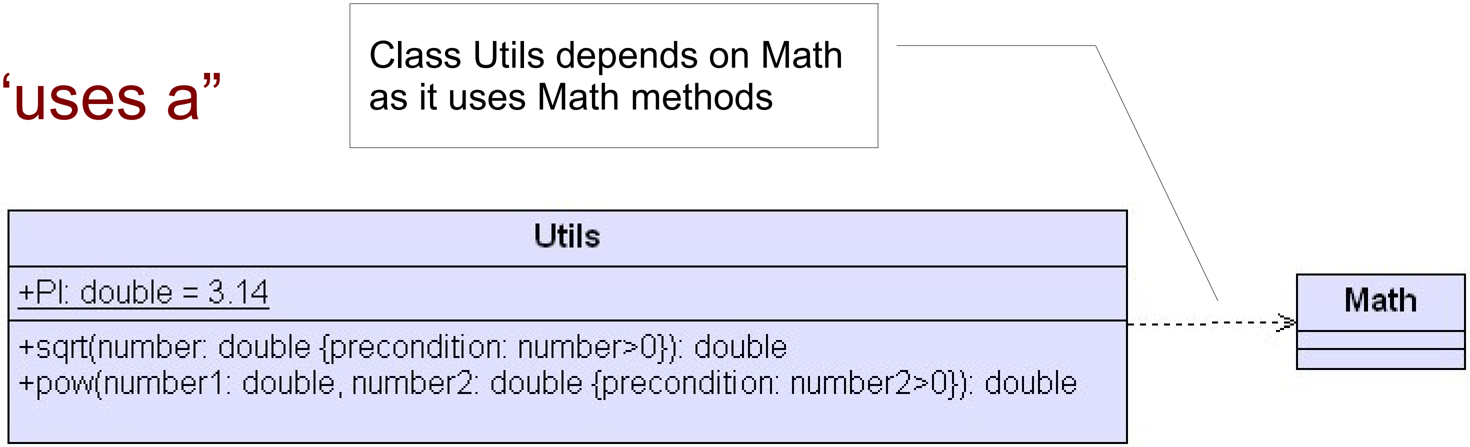
-capital: double = 0
-numOfBanks: int = 0

+getCapital(): double
+getNumOfBanks(): int

The Dependency Relationship

- One class has a dependency relationship with another class when it uses or had knowledge of it.
- The dependency relationship is usually known as “uses a”.
- Dependency relationship is marked using a dashed arrow.

“uses a”

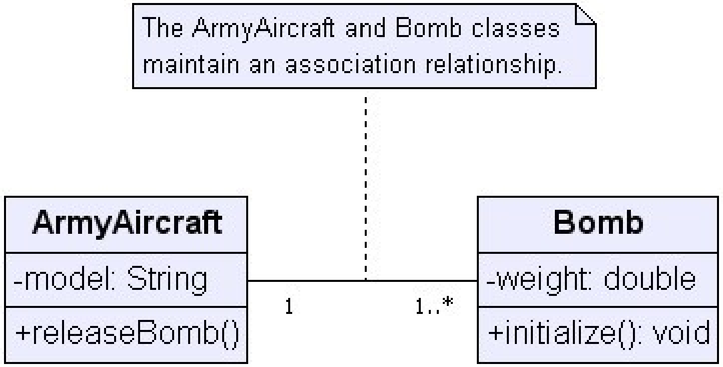

Class Utils depends on Math as it uses Math methods
Utils
+PI: double = 3.14
+sqrt(number: double {precondition: number>0}): double
+pow(number1: double, number2: double {precondition: number2>0}): double
Math

The Association Relationship

- The association relationship is a bit stronger than the dependency one.
- One class has an association relationship with another class when it retains a relationship to that class over an extended period of time and the life lines of the two objects that were instantiated from the two classes is not tied together.
- The association relationship is usually known as “has a”.
- The association relationship is denoted using a simple line.

The Association Relationship

“has a”

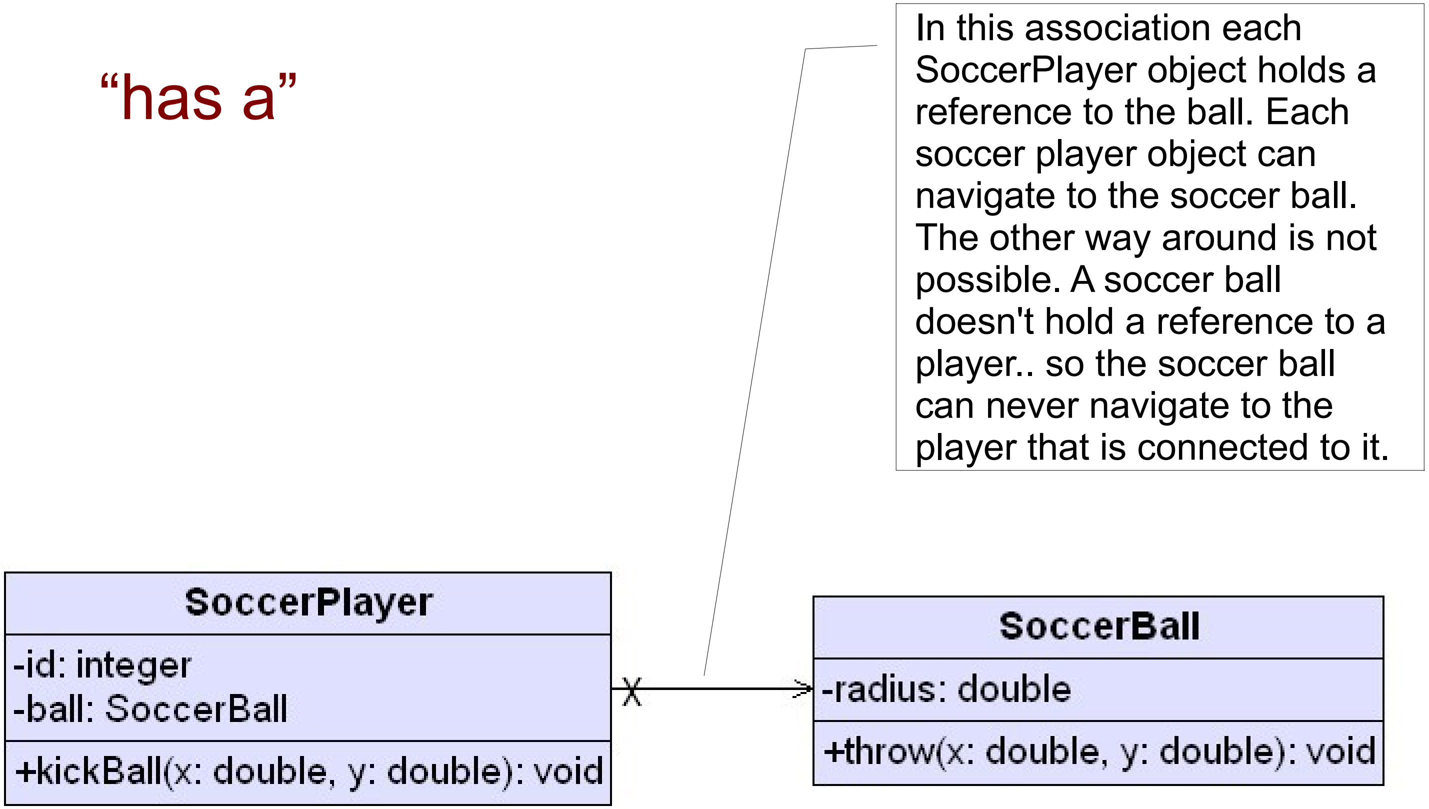


The Association Relationship

- When drawing an association relationship between two classes it is possible to indicate the possibility to navigate in a specific direction from one class to another, by adding a simple arrow to the association line.
- Placing 'X' on the association line near one of the classes will indicate that it is forbidden to navigate in its direction.

The Association Relationship

“has a”



In this association each SoccerPlayer object holds a reference to the ball. Each soccer player object can navigate to the soccer ball. The other way around is not possible. A soccer ball doesn't hold a reference to a player.. so the soccer ball can never navigate to the player that is connected to it.

SoccerPlayer

-id: integer
-ball: SoccerBall

+kickBall(x: double, y: double): void

SoccerBall

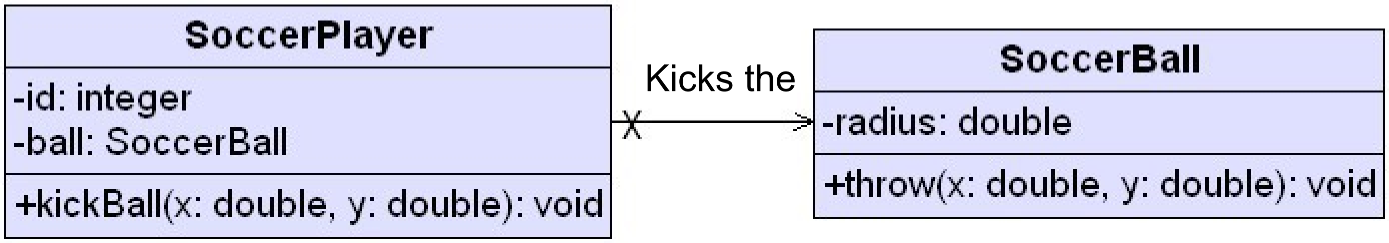
-radius: double

+throw(x: double, y: double): void

The Association Relationship

- When drawing the association line it is possible to add a small textual phrase above the line. By doing so we can provide some context.

“has a”



SoccerPlayer

-id: integer
-ball: SoccerBall

+kickBall(x: double, y: double): void

Kicks the

SoccerBall

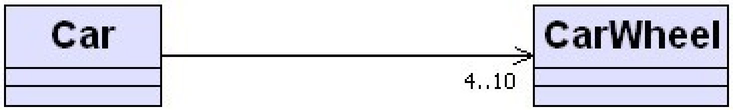
-radius: double

+throw(x: double, y: double): void

The Association Relationship

- Using the association relationship in order to indicate about a specific attribute of one of the classes allows us specifying a multiplicity value in order to indicate how many instances of a particular class are involved.

"has a"

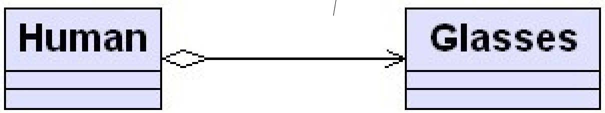


The Aggregation Relationship

- The aggregation relationship is a bit stronger than association.
- Unlike association, an aggregation relationship between two classes implies about ownership and might imply about some kind of relationship between the life lines.
- The aggregation relationship is usually known as “owns a”.
- The aggregation relationship is depicted using a line with an arrow on one end and an empty diamond on the other.

The Aggregation Relationship

“owns a”



Each human that use glasses owns a pair of glasses. Unlike the composition relationship (explained in the next slides) the glasses can be used by other humans as well. The glasses are not part of human.

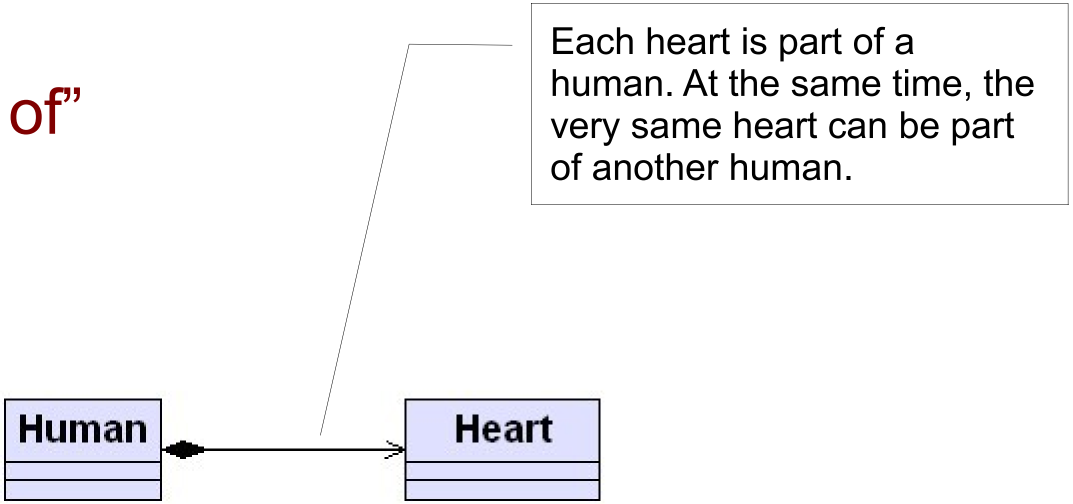
The Composition Relationship

- The composition relationship is a very strong relationship between classes. Stronger than aggregation.
- The composition relationship indicates a whole-part relationship. Unlike the other relationships, the “part” piece can be involved in one composition relationship at any given time.
- We depicted the composition relationship the same way we depict the aggregation one, just that instead of empty diamond we use a black one.

The Composition Relationship

- The life lines of any two instances involved in the composition relationships is (nearly) always linked.
- The composition relationship is usually known as “is part of”.

“is part of”



Each heart is part of a human. At the same time, the very same heart can be part of another human.

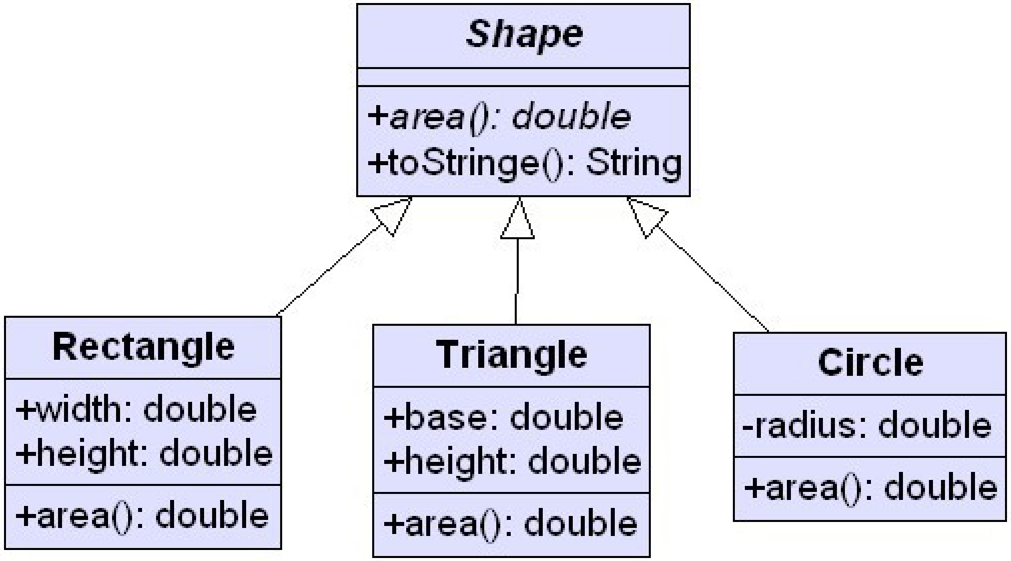
The Generalization Relationship

- The generalization relationship between two classes exists when one class is less specific (kind of “more general”) version of the other class.
- The generalization relationship is usually read as “is a”.
- The generalization relationship is shown with a solid line with a closed arrow pointing from the specific class to the general one.

Classes Relationships Summary

Relationship	Description	Arrow	Description
Dependency	“uses a”	– – >	One class uses another
Association	“has a”	———	One class retains relationship to the other over an extended period of time. The lifeline of the two is not tied.
Aggregation	“owns a”	◇——>	Ownership & some sort of relationship between the two life lines.
Composition	“is part of”	◆——>	The two life lines is (nearly) always linked.

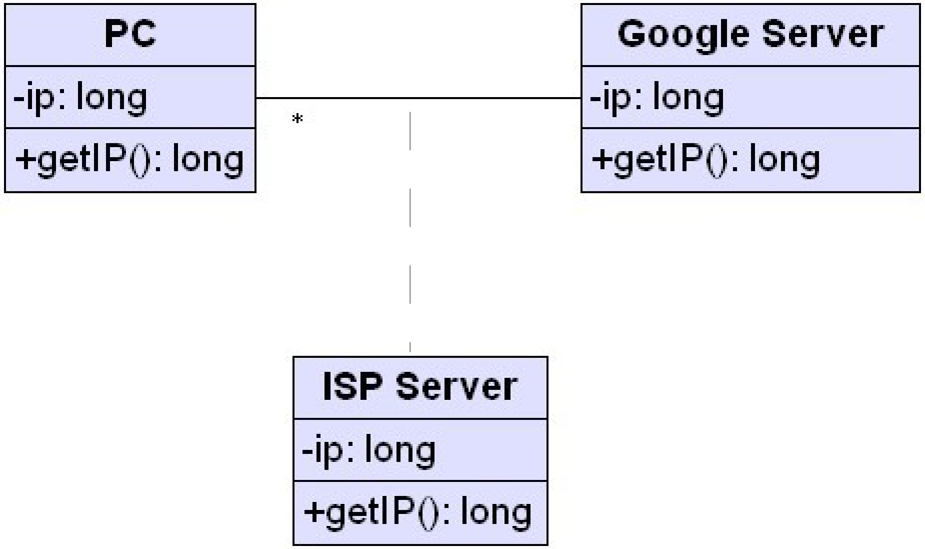
The Generalization Relationship



Association Classes

- An association class is a class that assists describing a relationship between two elements.
- Association class is depicted like any other class, connected with a dashed line to the association it represents.

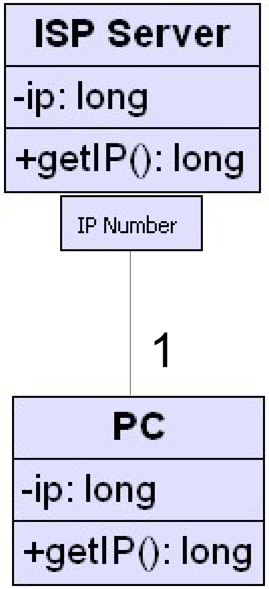
Association Classes



Association Qualifiers

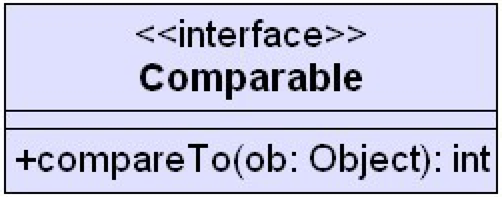
- An association qualifier is an additional information that describes an association connection between two classes.
- The association qualifier is usually an attribute of the target element.
- The association qualifier is depicted by placing a small rectangle between the association line and the source element and drawing the qualifier name inside.

Association Qualifiers



Interfaces

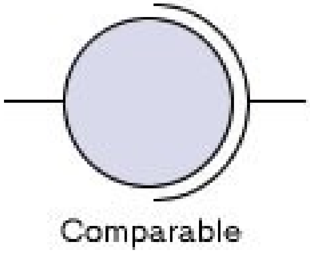
- In order to depict an interface we can use the class standard notation together with the “<<interface>>” stereotype.



<<interface>>
Comparable

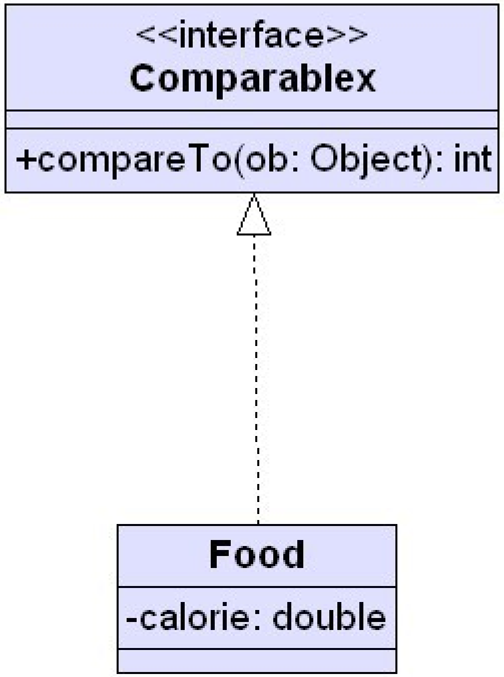
+compareTo(ob: Object): int

- Alternatively, we can use the ball-and-socket notation.

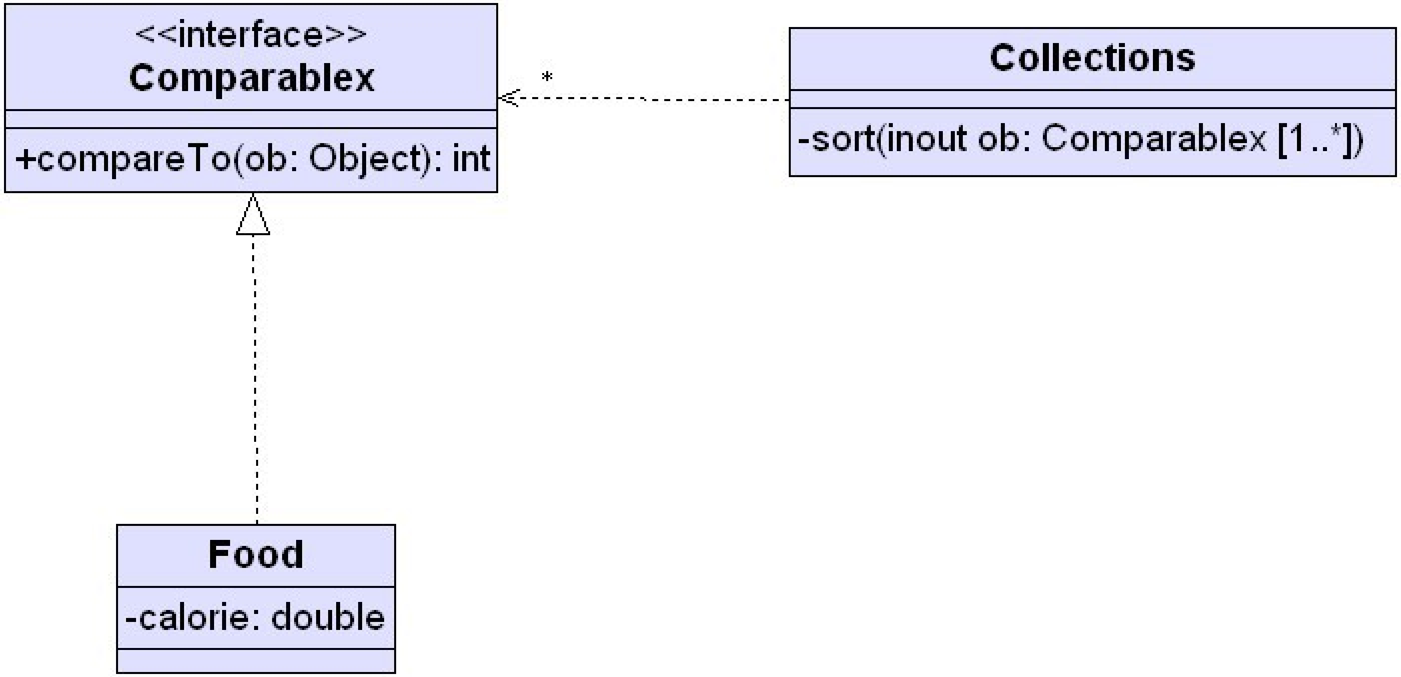


Interfaces

- Interfaces can't be instantiated. There is a need to declare a class that realize (implements) the interface.
- Depicting a class that realize (implements) an interface will be done using a dashed line starting at the class and leading to the interface. A closed arrow will be depicted at the end.

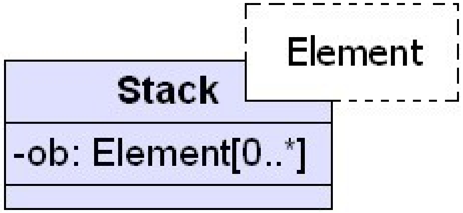


Interfaces


<<interface>>
Comparablex
+compareTo(ob: Object): int
*
Collections
-sort(inout ob: Comparablex [1..*])
Food
-calorie: double

Templates

- Indicating that a class is a template class can be done by drawing a dashed rectangle in the upper right corner of the class, and specify inside that dashed rectangle a name to act as a place holder for the actual type.



Database Schema

- Mapping DB tables to classes and table columns to attributes enables presenting a DB Schema using the UML Class Diagram.
- Primary keys, foreign keys and constraints can be displayed using UML constraints or stereotypes.
- Relationships between tables can be depicted using associations between classes.