Chapter 2 – Relationships between Classes

Contents

[1 Introduction 2](#_Toc156978116)

[2 One-to-One Relationships 2](#_Toc156978117)

[2.1 One Way Navigability 2](#_Toc156978118)

[2.2 Exercises 5](#_Toc156978119)

[2.3 Multiple 1-1 Relationships 6](#_Toc156978120)

[2.4 Convenience Methods & Delegation 8](#_Toc156978121)

[2.5 Exercises 10](#_Toc156978122)

[2.6 Two-way Navigability 11](#_Toc156978123)

[2.7 Exercises 12](#_Toc156978124)

[3 One-to-Many Relationships 12](#_Toc156978125)

[3.1 UML for One-to-Many 12](#_Toc156978126)

[3.2 Example: Implementing One-to-Many 13](#_Toc156978127)

[3.3 Exercises 18](#_Toc156978128)

[4 Implementing a Stack 19](#_Toc156978129)

[4.1 Exercises 21](#_Toc156978130)

[5 Object Oriented Programming 21](#_Toc156978131)

To make this document easier to read, it is recommended that you turn off spell checking and grammar checking in Word:

1. Choose: File, Option, Proofing
2. At the very bottom, check: “Hide spelling errors…” and “Hide grammar errors…”

# Introduction

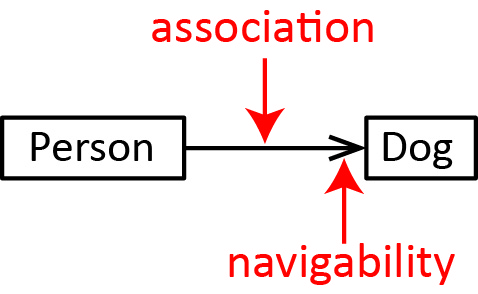
In the previous chapter, we discussed how a class can be used to model a real-world object, and how to write and test the class. In this chapter we learn how to model the *relationship* between two classes. Examples of three types of relationships is shown in the table below. In this chapter, we consider only one-to-one, and one-to-many.

|  |  |  |
| --- | --- | --- |
| **Relationship** | **Example** | **Class Diagram** |
| *One-to-one* | “A person has a dog” |  |
| *One-to-many* | “A bank has many customers” |  |
| *Many-to-many* | “A student has many courses each course has many students” |  |

# One-to-One Relationships

To build real systems, we almost always have more than one class, and they generally have relationships between them. We begin by considering the simplest type of relationship, one-to-one.

## One Way Navigability

The example in this section is found in the *example\_person\_dog\_1\_way\_navigability* package.

As shown on the right, in a class diagram, we model the one-to-one relationship between two classes with an *association* which is a solid line drawn between two classes. The arrow indicates *navigability*: “A person *has-a* dog.” An *association* is also called the *has-a* relationship.

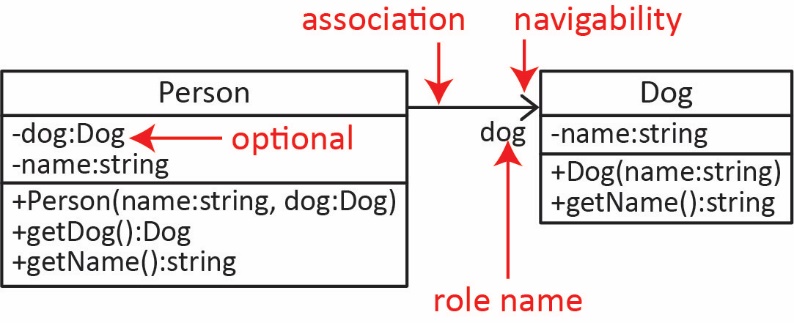
When we say a “person has-a dog”, in terms of implementation, we mean that the *Person* classhas an instance variable of type *Dog*. For example, in the example below, we see that this variable is named *dog.* This is also indicated by navigability arrow pointing from the *Person* to the *Dog.* As shown in the example below, we choose initialize the *dog* in the constructor, and a getter is provided.

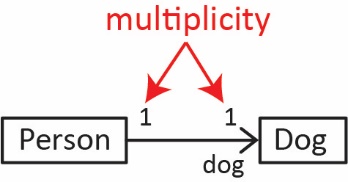
|  |  |
| --- | --- |
| ***Person* Class** | ***Dog* Class** |
| **public** **class** Person {  **private** String name;  **private** Dog dog;  **public** Person(String name, Dog dog) {  **this**.name = name;  **this**.dog = dog;  }    **public** Dog getDog() { **return** dog; }  **public** String getName() { **return** name; }  @Override  **public** String toString() {  **return** "name=" + name + ", dog=" + dog;  }  } | **public** **class** Dog {  **private** String name;  **public** Dog(String name) {  **this**.name = name;  }  **public** String getName() {  **return** name;  }  @Override  **public** String toString() {  **return** "name=" + name;  }  } |

Next, let’s see how to use this class and what is occurring in memory.

|  |  |
| --- | --- |
| **Code** | **Memory** |
| Dog d = **new** Dog("Spot"); | Object diagram showig a reference variable, d pointing to a dog object in memory. |
| Person p = **new** Person("Leah", d);  System.***out***.println(p); // name=Leah, dog:name=Spot | Object diagram showig a reference variable, p pointing to a person object in memory. The person object has a referece to a dog object. Thus, the person and the dog are linked. |
| Dog d2 = p.getDog();  System.***out***.println(d2); // name=Spot | The dog object how has a new reference to it, d2. |

Class diagram showing that an arrow on the end of an association is called navigability.
*One-way navigability* refers to a situation where two classes, *A* and *B* have an association and class *A* has a reference to class *B,* but *B* does not have a reference to *A*. We can indicate this in a class diagram by putting an arrow on the end of the association that points to the dependency as shown on the right. What this means is that, given an *A* object, you can always access the associated *B* object (as long as there is a getter for *B*). However, given a *B* object, you can’t access the associated *A* object.

In the example above, a person knows who her dog is (because of the *dog* instance variable in *Person*), but the dog does not know who its owner is (because there is no instance variable of type *Person* in the *Dog class*)*.* This illustrates *one-way navigability.* The class diagram representing these two classes is shown on the right. Note the following:

* This relationship is *one-to-one*. This means that each *Person* has exactly one *Dog* and each *Dog* can be associated with exactly one *Person*.
* The arrowhead drawn pointing to the *Dog* provides more information. It indicates that the *Person* class has an instance variable of type *Dog.* The arrowhead is referred to as *navigability*.
* The *role name* (optional) provides even more information. It indicates that the instance variable of type *Dog* in the *Person* class has the name *dog.* The role name should be at the end of the line where the navigability is shown (on the right in the diagram above).
* Typically, we do not show the *dog* instance variable in the *Person* class when we show the role name, as it is redundant.
* We can explicitly specify *multiplicity* to show that the relationship is 1-1 as shown in the class diagram on the right. However, if the multiplicity is left off, then it is understood to be 0 or 1 (a person has one dog, or possibly none). Typically, we leave a multiplicity of 1 off of the diagram. We talk more about multiplicity in a later section.

Notice in the *Person* class above, we provide a *getDog* method, which is of course a getter for the *dog* instance variable.

**public** Dog getDog() {

**return** dog;

}

If we want to allow a *Person* to change their *Dog*, we could supply a setter:

**public** **void** setDog(Dog dog) {

**this**.dog = dog;

}

We can use this method with code like this:

|  |  |
| --- | --- |
| **Code** | **Memory** |
| Dog d = **new** Dog("Spot"); | Object diagram showing a reference, d, to a dog object in memory. |
| Person p = **new** Person("Leah", d);  System.***out***.println(p); // name=Leah, dog:name=Spot | Object diagram showing a reference, p, to a person object in memory, that itself has a reference to a dog object. |
| Dog d2 = **new** Dog("Chaps"); | Object diagram showing the preceding's figure of a person that has a dog, but now also shows a new dog that has been created, referenced to be d2. |
| p.setDog(d2);  System.***out***.println(p); // name=Leah, dog:name=Chaps | Object diagram showing the reference, p, to the person object from the preceding figure now linked to the dog referenced to by d2. Thus, the original dog, d, is now not referenced to by p. |

Currently, we require that a person have a dog because the only constructor requires a dog as an arguemnt.

**public** Person(String name, Dog dog) {

**this**.name = name;

**this**.dog = dog;

}

Suppose we do not want to require that a person have a dog. Then we can add another constructor:

**public** Person(String name) {

**this**(name,**null**);

}

Of course, we would need to be careful, because the *getDog* method could return *null.* For example, this code would generate a runtime error (null pointer exception):

Person p = **new** Person("Leah");

Dog d = p.getDog();

System.***out***.println(d.toString());

This is a very common type of error. Note that, after the 2nd line executes, *d* is *null*, which means *d* does not refer to an object in memory; there is no *Dog* in memory. This does not cause the error. It is the next line, where we attempt: d.toString(). It is attempting to call *toString* on something that is not an object. As we stated earlier, this is called a *null pointer exception*. When it occurs, your program stops and displays a message like this:

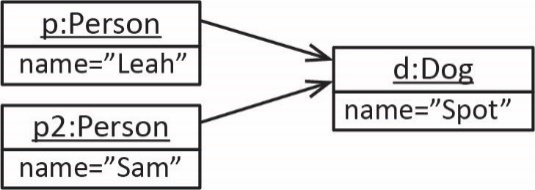
Exception in thread "main" java.lang.NullPointerException

at person\_dog\_1\_way\_navigability.PersonTest.testShowsAnError(PersonTest.java:42)

at person\_dog\_1\_way\_navigability.PersonTest.main(PersonTest.java:10)

So, what does a null pointer exception mean? It means you tried to call a method on an object that doesn’t exist (technically, an object that is *null*).

Finally, more than one *Person* could have the same *Dog* as illustrated by this sample code:

Dog d = **new** Dog("Spot");

Person p = **new** Person("Leah", d);

System.***out***.println(p); // name=Leah, dog:name=Spot

Person p2 = **new** Person("Sam", d);

System.***out***.println(p2); // name=Sam, dog:name=Spot

This may or may not be what we want to allow. If you had a strong reason to not allow this, then, most likely, you would need higher level classes to *manage* the creation of a person and dog associations. We discuss this in detail later in this chapter.

## Exercises

1. Consider the *Car* class below. Write a *Person* class with the characteristics shown below. Write some sample code to use this class. The solution is in the *exercise\_person\_car1 package*.

* A *Person has-a Car* and a name.
* A constructor that accepts the name and a car.
* A constructor that accepts just the name.
* A way to access the name (*i.e.* a getter).
* A way to access the car (*i.e.* a getter).
* A way to assign a car to the person (*i.e.* a setter).

**public** **class** Car {

**private** **double** totalDistance;

**public** Car() {

**this**.totalDistance = 0.0;

}

**public** **double** getTotalDistance() {

**return** totalDistance;

}

**public** **void** drive(**double** time, **double** rate) {

**double** distance = time\*rate;

totalDistance += distance;

}

@Override

**public** String toString() {

String msg = String.*format*("totalDistance=%.1f" + totalDistance);

**return** msg;

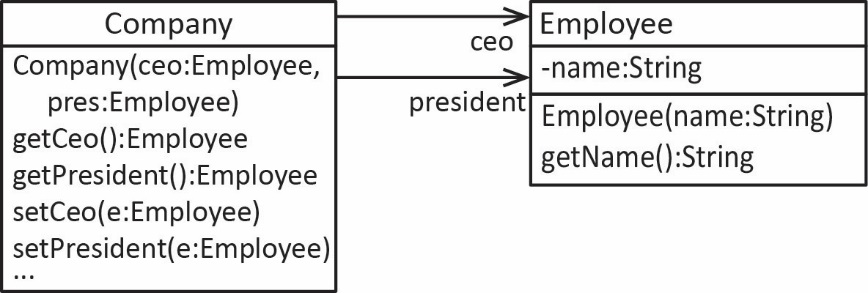
}

}

## Multiple 1-1 Relationships

The example in this section is found in the *example\_company\_employee* package.

A class can have multiple 1-1 relationships with the same class. For example, a *Company* has a *CEO* and a *President*. We would probably generalize *CEO* and *President* as employees (as we do here) or manager, *etc.* In a class diagram, we would show this is:



The code for *Company* class is shown below. Notice that we have two *Employee* instance variables: *ceo* and *president*

**public** **class** Company {

**private** Employee ceo;

**private** Employee president;

**public** Company(Employee ceo, Employee president) {

**this**.ceo = ceo;

**this**.president = president;

}

**public** Employee getCeo() {

**return** ceo;

}

**public** **void** setCeo(Employee ceo) {

**this**.ceo = ceo;

}

**public** Employee getPresident() {

**return** president;

}

**public** **void** setPresident(Employee president) {

**this**.president = president;

}

@Override

**public** String toString() {

**return** "CEO=" + ceo + ", President:" + president;

}

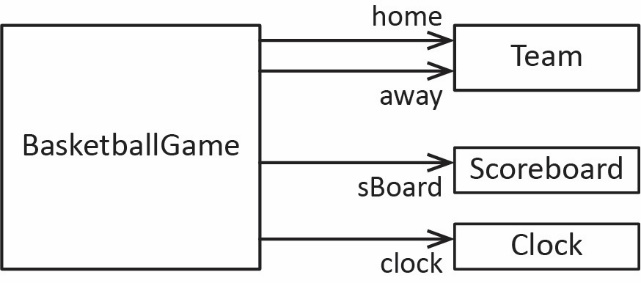
}

Sample code:

| **Code** | **Memory** |
| --- | --- |
| Employee e1 = **new** Employee("Xavier");  Employee e2 = **new** Employee("Sadie"); | Object diagram showing to employee objects in memory. |
| Company c = **new** Company(e1,e2);  System.***out***.println(c);  // CEO=name=Xavier, President:name=Sadie | Object diagram showing a company object in memory linking to two employee objects. |

Example: Write a *snippet* of code to create a company and then use the *Company* methods to swap the CEO and president.

| **Code** | **Memory** |
| --- | --- |
| // Create the company  Employee e1 = **new** Employee("Xavier");  Employee e2 = **new** Employee("Sadie");  Company c = **new** Company(e1,e2);  System.***out***.println(c);  // CEO=name=Xavier, President:name=Sadie | Object diagram showing a company object in memory linking to two employee objects. |
| // Remember the current ceo and pres  Employee oldCeo = c.getCeo();  Employee oldPres = c.getPresident(); | Object diagram showing explicity references, oldCeo and oldPres to the two employee objects from the preceding figure. |
| // Swap ceo and pres  c.setCeo(oldPres);  System.***out***.println(c);  // CEO=name=Sadie, President:name=Sadie | Object diagram showing a company object linking both ceo and president to the same employee. |
| c.setPresident(oldCeo);  System.***out***.println(c);  // CEO=name=Sadie, President:name=Xavier | Object diagram showing a company object in memory linking to two employee objects, where the two employee object have been swapped. |

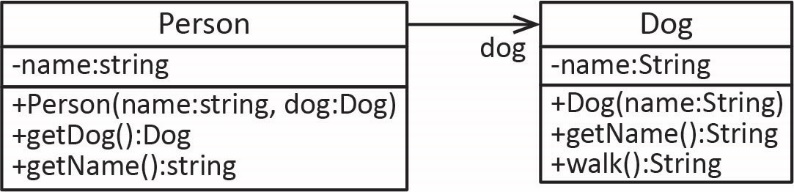
Example – As stated above, a class can also have as many associations as it needs to model a particular situation. For example, a *BasketballGame* might need associations with *Team, Scoreboard,* and *Clock* as shown on the right.

Finally, a class can have a *reflexive association*, where a class has an association with itself. In the class diagram below, a *Course* has another *Course* which is the *prerequisite.* In the object diagram below, the *Course, cs1302* has a prerequisite that is also a *Course, cs1301*. A reflexive association is the basis for linked data structures (*e.g.* Linked List, Trees, *etc.*) and will be studied in another course.

|  |  |
| --- | --- |
| **Class Diagram** | **Object Diagram** |
| Class diagram showig a course class with an association with itself, whose role name is prerequisite. | Object diagram showing a course object (cs 1302) with a link to another course object, its prerequisite (cs 1301) |

## Convenience Methods & Delegation

The example below is found in the *example\_person\_dog\_1\_way\_navigability* package.

Continuing with the the *Person, Dog* example, suppose the *Dog* has a *walk* method:

**public** String walk() {

**return** name + " is walking";

}

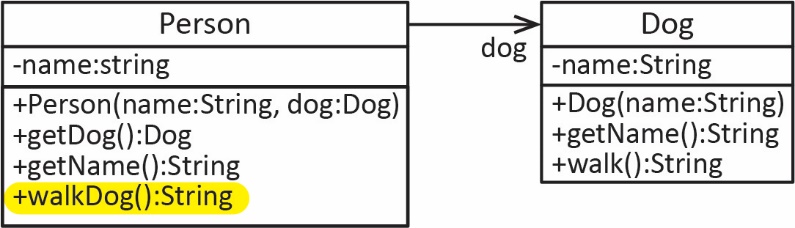
If we want a person to walk their dog, we could ask the person for their dog, and then ask the dog to walk:

Person p = **new** Person("Sandra", **new** Dog("Chaps"));

Dog d = p.getDog();

String msg = d.walk();

System.***out***.println(msg); // Chaps is walking

Or, we could add a *convenience method* to the *Person* class, *walkDog* that *delegates* to the *Dog* to accomplish the walk:

**public** String walkDog() {

**return** dog.walk();

}

Then, to walk the dog:

Person p = **new** Person("Sandra", **new** Dog("Chaps"));

String msg = p.walkDog();

System.***out***.println(msg); // Chaps is walking

Another convenience method we might add to the *Person* class:

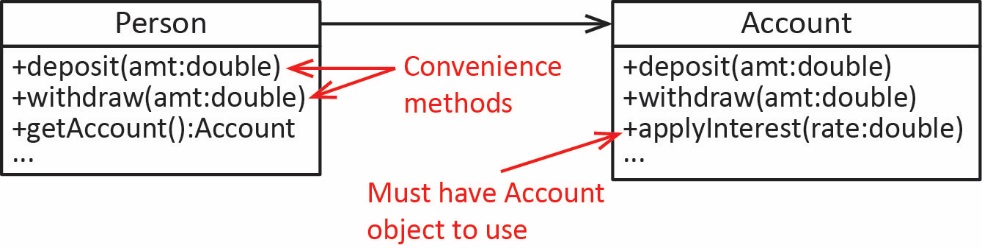
**public** String getDogName( ) {

**return** dog.getName();

}

When designing classes, we want to make them as easy to use as possible. Frequently, that involves making a *convenience* method that makes accomplishing a *use case[[1]](#footnote-1)* simpler for a programmer building a system. In general, if class *A has-a B*, we should give thought to what convenience methods we should have in *A* that delegate to *B.* Clearly, we could delegate to all methods in *B*; however, without strong reason to, this might make *A* to have too many responsibilities.

For example, suppose that a *Person* has an *Account* as shown below (code is found in the *example\_person\_account\_1\_way* package). The *Account* class has *deposit* and *withdraw* methods. However, we might add For example, perhaps we have *deposit* and *withdraw* convenience methods in the *Person* class, but require an instance of an *Account* object to *applyInterest*:



For example:

|  |  |
| --- | --- |
| ***Person* Class** | **Test Code** |
| **public** **class** Person {  **private** String name;  **private** Account account;  **public** Person(String name, Account account) {  **this**.name = name;  **this**.account = account;  }    **public** Account getAccount() {  **return** account;  }  **public** **void** deposit(**double** amount) {  **if**(amount>0) {  account.deposit(amount);  }  }  **public** **void** withdraw(**double** amount) {  **if**(amount>0) {  account.withdraw(amount);  }  }  ...  } | **public** **static** **void** testDeposit() {  Account a = **new** Account(2000.00);  Person p = **new** Person("Vaugn", a);  p.deposit(1000.0);  System.***out***.println(p);  }  **public** **static** **void** testApplyInterest() {  Account a = **new** Account(2000.00);  Person p = **new** Person("Vaugn", a);  a = p.getAccount();  a.applyInterest(0.1);  System.***out***.println(p);  } |

These design choices depend on a number of factors, one of which is *class cohesion*. Ideally, we want to class to do **one thing**, in which case the class would have *strong* cohesion. If the *Person* class above had many other responsibilities that relate to it being a person (get name, age, address, beneficiary, *etc.*) and a number of responsibilities related to managing the account, we would say that the *Person* class has *weaker cohesion*. In general, we want classes that have strong cohesion. When classes start to take on too many responsibilities, we should break them into separate classes that are associated. This is a skill that takes experience and time to develop. Mostly, in this class, I tell you what classes are needed. In subsequent classes, it will be up to you to design the classes.

## Exercises

1. (Solution in *exercise\_person\_car2* package) Consider the *Person* class from previous Exercise. Add the following convenience methods to the *Person* class: (a) *driveCar* method that accepts a time and rate and delegates to the *Car* classes *drive* method, (b) a *getDriveDistance* method that returns the total distance the car has driven. Write some sample code to use this class.
2. (Solution in *exercise\_person\_account1* package) Consider the two classes, *Person* and *Account* shown below. Make the following change to the *Account* class and write code to test:

* Keep a running total of all deposits to the associated account (not the number of deposits, the value of all deposits). Hint: introduce a new instance variable, *depositTotal* and every time *deposit* is called, add the amount to this variable (and of course continue to update the balance). Also, write a getter for this variable, *getTotalDeposits*
* Modify the *toString* method so that it also shows the running total of deposits.

Make the following change to the *Person* class and write code to test:

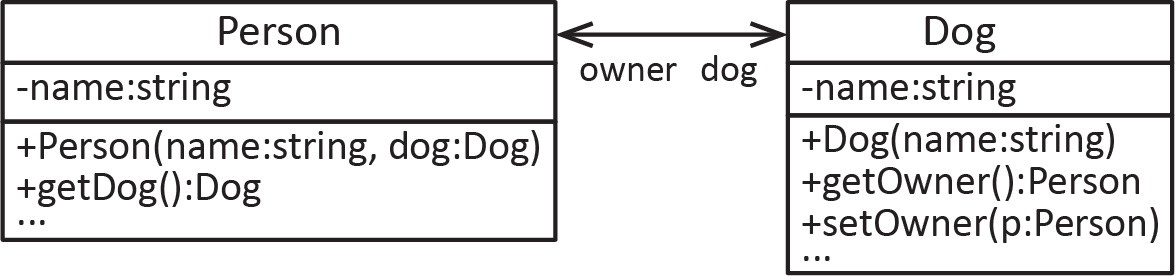
* Provide a convenience method, *getTotalDeposits,* to return the total of deposits. Hint: this method simply delegates to the *Account* class’s *getTotalDeposits* method.

|  |  |
| --- | --- |
| ***Account* Class** | ***Person* Class** |
| **public** **class** Account {  **private** **double** balance;  **public** Account() { balance = 0.0; }  **public** Account(**double** balance) {  **this**.balance = balance;  }  **public** **void** applyInterest(**double** intRate) {  balance \*= (1+intRate);  // Or, balance = balance + intRate\*balance;  }  **public** **double** getBalance() { **return** balance; }    **public** **void** deposit(**double** amount) {  balance += amount;  }  **public** **void** withdraw(**double** amount) {  balance -= amount;  }  **public** String toString() {  **return** "Account: balance=" + balance;  }  } | **public** **class** Person {  **private** String name;  **private** Account account;  **public** Person(String name, Account account) {  **this**.name = name;  **this**.account = account;  }  **public** Account getAccount() {  **return** account;  }  **public** String getName() {  **return** name;  }  @Override  **public** String toString() {  **return** "name=" + name + ", account: " + account;  }  } |

## Two-way Navigability

The code for the example below is in the *example\_person\_dog\_2\_way\_navigability* package.

*Two-way navigability* refers to the situation where each end of the association knows about the other. As shown in the class diagram below, not only does the *Person* know who his *Dog* is; the *Dog* knows who its *owner* is. One way to model this is to add an *owner* instance variable to the *Dog* class and provide a getter and setter as shown below.



For example, the code below shows how to create the two objects and connect them, and how things are represented in memory.

| **Code** | **Memory** |
| --- | --- |
| // Create dog  Dog d = **new** Dog("Mocho");  // Create person with dog  Person p = **new** Person("Xavier", d); | Object diagram showing a person object that has a link to a dog. |
| // Connect dog to its owner  d.setOwner(p); | Object diagram showing a person object that has a link to a dog and that the dog has a link back to the person. |
| // If have a person, can always get dog  d = p.getDog();  // If have a dog, can always get person  p = d.getOwner(); |  |

The implementation is shown below.

|  |  |
| --- | --- |
| **public** **class** Person {  **private** String name;  **private** Dog dog;  **public** Person(String name, Dog dog) {  **this**.name = name;  **this**.dog = dog;  }  **public** Dog getDog() {  **return** dog;  }  ...  } | **public** **class** Dog {  **private** String name;  **private** Person owner;  **public** Dog(String name) {  **this**.name = name;  }  **public** Person getOwner() {  **return** owner;  }  **public** **void** setOwner(Person owner) {  **this**.owner = owner;  }  ...  } |

When do we implement two-way navigability? The short answer is: you will know when you need it. We consider this in a bit more detail in a later section.

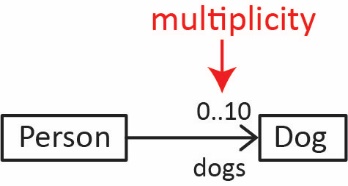
## Exercises

1. (Solution in *exercise\_person\_account2* package) Consider the *Account* class from the preceding Exercise. (a) Modify it so that it implements two-way navigability, *i.e.* so that an *Account* knows who its owner is. (b) Write code to test. (c) Draw a class diagram.

# One-to-Many Relationships

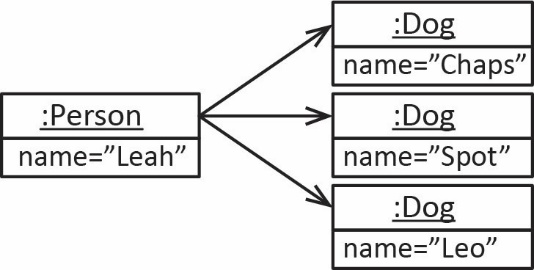
Next, we consider the one-to-many relationship between classes.

## UML for One-to-Many

Consider the class diagram on the right which shows that a *Person* can have up to 10 dogs. This is an example of a *1-to-many* association between two classes. Notice the *role name* is *dogs*; this is the name of the instance variable in the *Person* class that holds the dogs. For now, that will be an array:

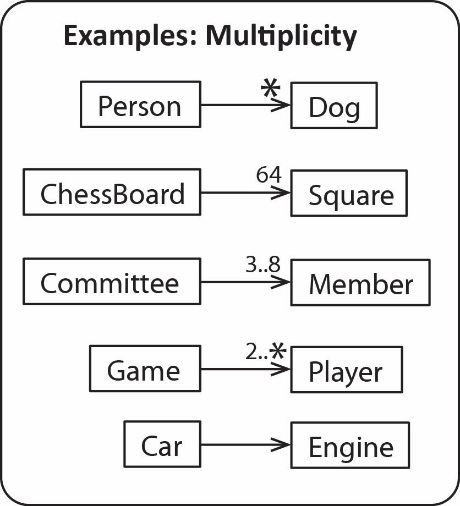
**private** Dog[] dogs = **new** Dog[10];

For example, if a person has 3 (of the possible 10) dogs, we could illustrate that with an object diagram:



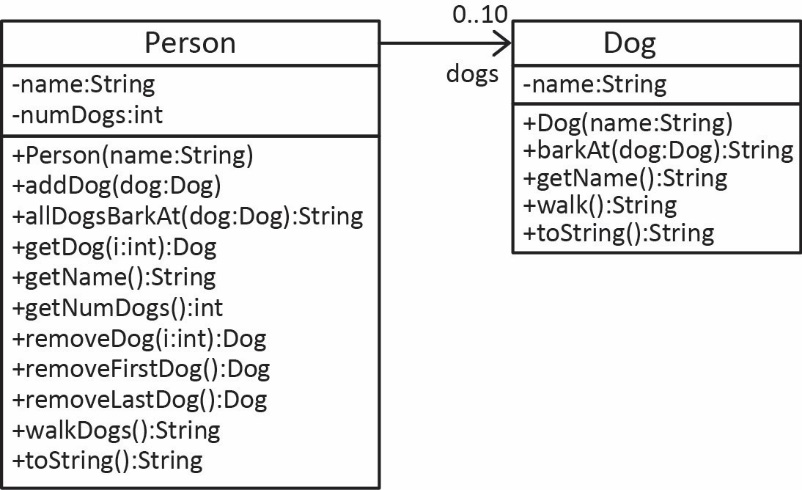
The *multiplicity* of an association can take on any of the values shown on the right.

| **Multiplicity** | **Meaning** |
| --- | --- |
| n | Exacltly *n* instances |
| a..b | Any value from *a* to *b* instances |
| 0..n | Any value from zero to n instances |
| \* | Many (zero or more) instances |
| a..\* | At least *a* instances |
| (blank) | Zero or one instance |

Consider the examples on the right and note the following:

* Each *Person* has many *Dogs*.
* Each *ChessBoard* has 64 *Squares.*
* Each *Committee* has 3-8 *Members.*
* Each *Games* has at least 2 *Players.*
* A *Car* has an *Engine.*

## Example: Implementing One-to-Many

The example below is in the *example\_person\_dogs* package. In this section, we look closely at an example where a *Person* has many *Dogs*. Consider the class diagram on the right. Below, we step through the implementation of the entire *Person* class.

The *Dog* class should be straightforward. However, we show two of the methods we haven’t considered so far:

**public** String barkAt(Dog d) {

**return** name + "barks at " + d.name;

}

**public** String walk() {

**return** name + " is walking";

}

The *Person* class needs a way to keep track of the (up to) 10 dogs. We could have 10 instance variables:

Dog dog1; Dog dog2; ... Dog dog10;

However, this would be very unwieldly. We would need getters and setters for each one, and other issues. And, what if we wanted to change the number of dogs we allow, say, to 15. We would have to add a lot of code. A better solution (from the things we currently know) is to use an array of *Dog*s as the instance variable:

**private** Dog[] dogs = **new** Dog[10];

When we implement 1-many with an array, we will need an instance variable to keep track of exactly how many dogs we have, as well as a way to retrieve this value. We will also have *addDog* which will add a *Dog* to the array and increment the number of dogs.

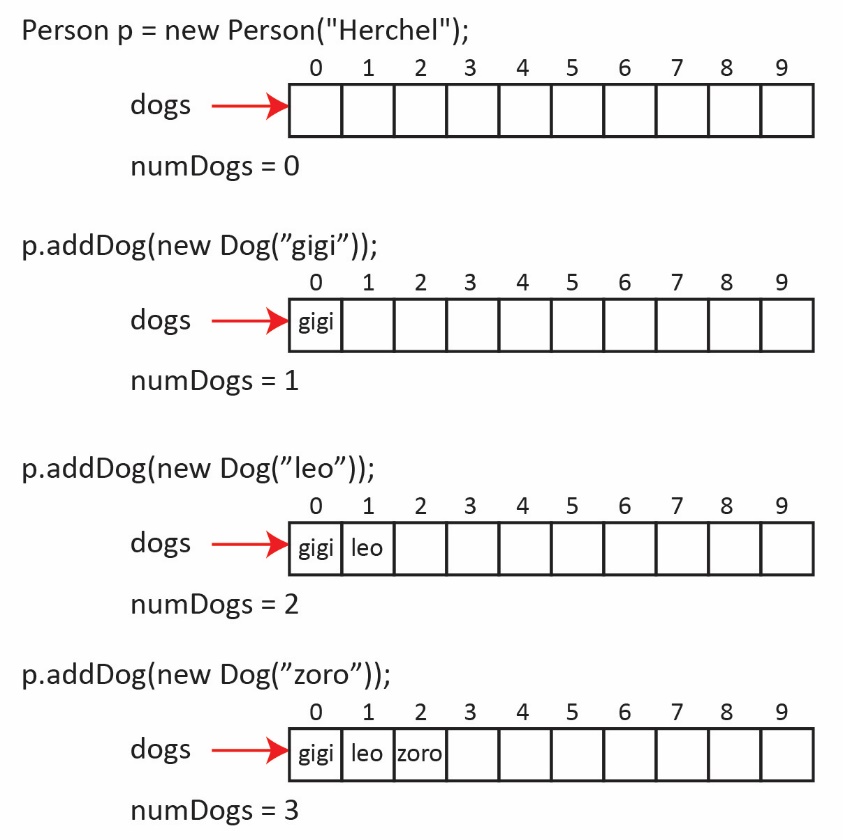
|  |  |  |
| --- | --- | --- |
| **Instance Variable** | **Getter** | **addDog** |
| **private** **int** numDogs = 0; | **public** **int** getNumDogs() {  **return** numDogs;  } | **public** **void** addDog(Dog dog) {  // We’ll see this code shortly  numDogs++;  } |

Note:

* The methods in the class *manage numDogs,* where appropriate. For example, when a *Dog* is added (removed) from *Person*, then we make sure our code increments (decrements) *numDogs.* Thus, we never “count” how many dogs there are in the array; *numDogs* will always contain the exact number of *Dogs* that *Person* has.
* Many of the methods we write below will depend on *numDogs*.

We begin the *Person* class with the code below and then we will add the other methods below.

|  |  |
| --- | --- |
| **public** **class** Person {  **private** String name;  **private** **int** numDogs;  **private** Dog[] dogs = **new** Dog[10];  **public** Person(String name) {  **this**.name = name;  numDogs = 0;  } | **public** String getName() {  **return** name;  }    **public** **int** getNumDogs() {  **return** numDogs;  }  } |

In a one-to-many relationship, I find it useful to refer to the “one” class as a *manager* class. In our example, the *Person* class is a manager class; it *manages the dogs*. A *manager* class will usually have an “add” method to add items. Thus, we provide an *addDog* method. The usual approach is to store the dogs sequentially in the array, starting at position 0. In the example on the right, notice that the first *Dog* added is added at index 0, the next at index 1, *etc.* And, each time we add a *Dog, numDogs* is incremented.

Consider the *addDog* method:

**public** **void** addDog(Dog dog) {

**if**(numDogs<dogs.length) {

dogs[numDogs] = dog;

numDogs++;

}

}

Study the *addDog* method carefully; afterwards, note the following:

* It first makes sure the array is not full.
* The *numDogs* instance variable not only stores how many dogs we currently have, it also is the position in the array where the next dog will go. Make sure you see this. Initially, there are 0 dogs. So, where does the first dog that is added go? It goes in position 0.
* Finally, we increment *numDogs.*

And, as stated above, the getter for *numDogs* simply has to return the instance variable.

**public** **int** getNumDogs() {

**return** numDogs;

}

I emphasize this as I have seen students many times implement the getter by writing a loop that (tries to) count all the non-null elements in the array. For example:

**public** **int** getNumDogs() {

**int** count = 0;

**for**(**int** i=0; i<dogs.length; i++) {

**if**(dogs[i]!=**null**) {

count++;

}

}

**return** count;

}

This is a very poor choice when all we have to do is increment *numDogs* whenever we modify the array and then the getter simply returns *numDogs.* Further, this method can be incorrect once we add *removeDog* methods (later).

We provide a *getDog* method that accepts an index for the *Dog* to return. Note that we require that the index be valid: it must be between 0 and the number of dogs that have been added.

**public** Dog getDog(**int** i) {

**if**(i>=0 && i<numDogs) {

**return** dogs[i];

}

**return** **null**;

}

A common mistake is to specify a valid index as one that is less than the length of the array. This is incorrect because the array is not necessarily full.

|  |  |
| --- | --- |
| **Correct** | **Incorrect** |
| i>=0 && i<numDogs | i>=0 && i<dogs.length |

We might like to be able to access dogs by their name instead of an index. This is something we consider later in the course.

We provide a way for a *Person* to walk all their dogs:

**public** String walkDogs() {

String msg = "";

**for**(**int** i=0; i<numDogs; i++) {

Dog d = dogs[i];

msg += d.walk() + "\n";

}

**return** msg;

}

Notice that the stopping condition for the loop above (highlighted) involves the *numDogs* instance variable, not the length of the array. We want to iterate over just the dogs that we have. A common mistake is to use the length of the *dogs* array, which is incorrect because the array is not necessarily full and a runtime error would result if it is not.

|  |  |
| --- | --- |
| **Correct** | **Incorrect** |
| i<numDogs | i<dogs.length |

Notice also that we cannot use a *for-each* loop as it will traverse the entire array:

|  |  |
| --- | --- |
| **Correct** | **Incorrect** |
| **for**(**int** i=0; i<numDogs; i++) {  Dog d = dogs[i];  msg += d.walk() + "\n";  } | **for**(Dog d : dogs) {  msg += d.walk() + "\n";  } |

We provide a way for all the dogs to bark at another dog:

**public** String allDogsBarkAt(Dog d) {

String msg = "";

**for**(**int** i=0; i<numDogs; i++) {

Dog dog = dogs[i];

msg += dog.barkAt(d) + "\n";

}

**return** msg;

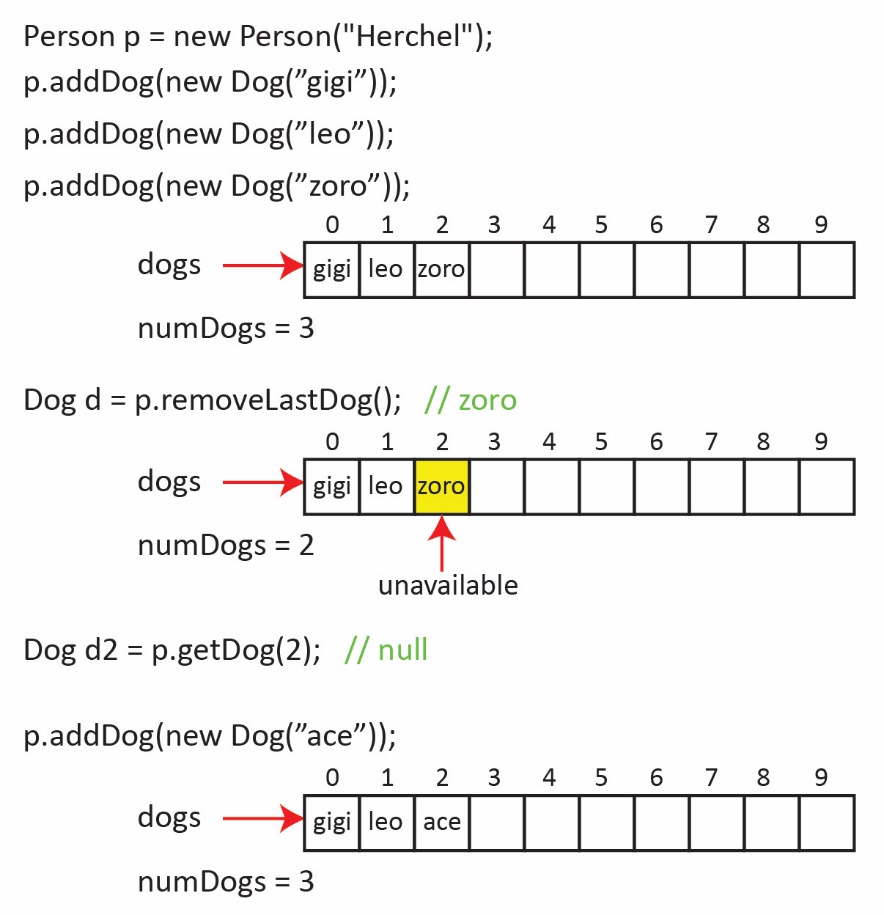
}

An *iterative method* is one that iterates (loops) over the elements in an array (collection) and does something to or with each one. For example, the *walkDogs* and *allDogsBarkAt* methods above are both iterative methods. **Creating an iterative method is a must-have skill.**

Next, we consider removing a dog from a person’s array of dogs. We will do this incrementally, by first considering the easiest case, removing the last dog (*removeLastDog*). After this, we consider, *removeFirstDog*. Finally, we consider the more general, *removeDog(position)*.

We provide a method, *removeLastDog,* to remove the last dog and return it (typically, *remove* methods not only remove the item, but also return it). Look carefully at the code below. Where is the last dog located? We know that *numDogs* contains the position where the *next* dog would be added. Thus, if we subtract 1 from *numDogs,* that will be the location of the last dog.

|  |  |
| --- | --- |
| **Method** | **Alternate Version** |
| **public** Dog removeLastDog() {  **if**(numDogs>0) {  numDogs--;  **return** dogs[numDogs];  }  **return** **null**;  } | **public** Dog removeLastDog() {  **if**(numDogs>0) {  **return** dogs[--numDogs];  }  **return** **null**;  } |

Now, follow the example on the right and correlate it with the *removeLastDog* method above. In the example, *numDogs* is 3, then we remove the last dog. However, we didn’t actually remove the last dog, we just decremented *numDogs* to 2. Thus, even though the dog we removed is still at index 2, it is unavailable because *addDog*, *getDog*, *etc* all depend on *numDogs*. For example, after removing the last dog, consider executing this line of code: d=p.getDog(2). Now go back and look at the *getDog* method and see that it will return *null*. Why? Because the argument, 2, is only valid:

**if**(i>=0 && i<numDogs)

Finally, if we add another dog, as usual, it goes in the next available spot, which overwrites (the inaccessible) *zoro*.

Next, we provide a method, *removeFirstDog*, to remove the first dog and return it. It is standard to not have an “holes” in the array. So, this means that when we remove the first dog, we need to move all the other dogs to the right, over one position to the left. The algorithm:

1. Get a reference to the first dog
2. Move all the other dogs over one position to the left
3. Decrement the number of dogs
4. Return the first dog

Study the code and example carefully, making sure you understand the loop and how it implements step 2 above. Notice that the last dog, *zoro* in the example below, appears twice, once in its new position (index=1), and once in its original position. However, as we noted above, it is unavailable because *numDogs=2*.

|  |  |
| --- | --- |
| **Method** | **Example** |
| **public** Dog removeFirstDog() {  **if**(numDogs>0) {  Dog returnDog = dogs[0];  **for**(**int** i=1; i<numDogs; i++) {  dogs[i-1] = dogs[i];  }  numDogs--;  **return** returnDog;  }  **return** **null**;  } | Shows the dogs array with 3 dogs. Then, shows how dogs at indices 1 & 2 are shifted to the left one position so that they now are located at indices 0 & 1. |

Finally, we provide a method, *removeDog(index:int)* to remove the dog at a particular index and return it. Similar to *removeFirst*, all the dogs to the right of the removed dog, must be moved over one position to the left. An algorithm:

1. If the index is valid
   1. Get a reference to the dog at the index
   2. Loop over the dogs to the right of the one at the index

* Move current dog one position to the left.
  1. Decrement the number of dogs
  2. Return the dog that was at the index.

1. Else, return null.

Study the code and example carefully, making sure you understand the loop and how it implements step 1b above.

|  |  |
| --- | --- |
| **Method** | **Example** |
| **public** Dog removeDog(**int** i) {  **if**(i>=0 && i<numDogs) {  Dog returnDog = dogs[i];  **for**(**int** j=i+1; j<numDogs; j++) {  dogs[j-1] = dogs[j];  }  numDogs--;  **return** returnDog;  }  **return** **null**;  } | Shows the dogs array with 4 dogs. The dogs at indices 2 & 3 are moved to indices 1 & 2 after the removal of the dog at index 1. |

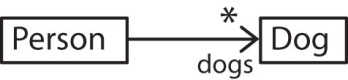
Note that *removeLast* and *removeFirst* can be made much simpler by simply calling *removeDog*:

|  |  |
| --- | --- |
| **public** Dog removeLastDog() {  **return** removeDog(numDogs-1);  } | **public** Dog removeFirstDog() {  **return** removeDog(0);  } |

Also, note that these two methods are *convenience methods*. To illustrate this: how would you use *removeDog(i:int)* to remove the first dog? The last dog?

Dog dFirst = p.removeDog(0);

Dog dLast = p.removeDog(p.getNumDogs()-1);

If we wanted to allow for any number of dogs, as shown in the class diagram on the right, then we could start with an array of some size, say 10 and when we try to add the 11th, we simple create a larger array. For example,

**public** **void** addDog2(Dog dog) {

// If array is full

**if**(numDogs>=dogs.length) {

// Create new array twice the size of current one

Dog[] temp = **new** Dog[dogs.length\*2];

// Copy current array to new array

System.*arraycopy*(dogs, 0, temp, 0, dogs.length);

// Make the instance variable point to the new array

dogs=temp;

}

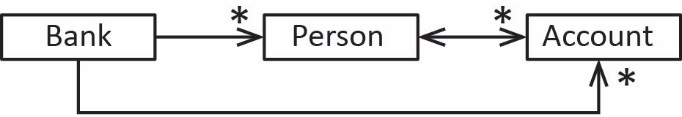
dogs[numDogs++] = dog;

}

Returning to the question of when do we implement two-way navigability, consider this example where a *Bank* has many *Persons* and each *Person* has many *Accounts*. Perhaps, in this situation, only one-way navigability is needed between *Person* and *Account.*

Class diagram showing a bank class that has many persons, and each person has many accounts.

However, if the *Bank* also maintains a list of all *Accounts,* then it might be useful to be able to navigate from *Account* to *Person.*



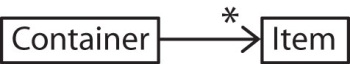
## Exercises

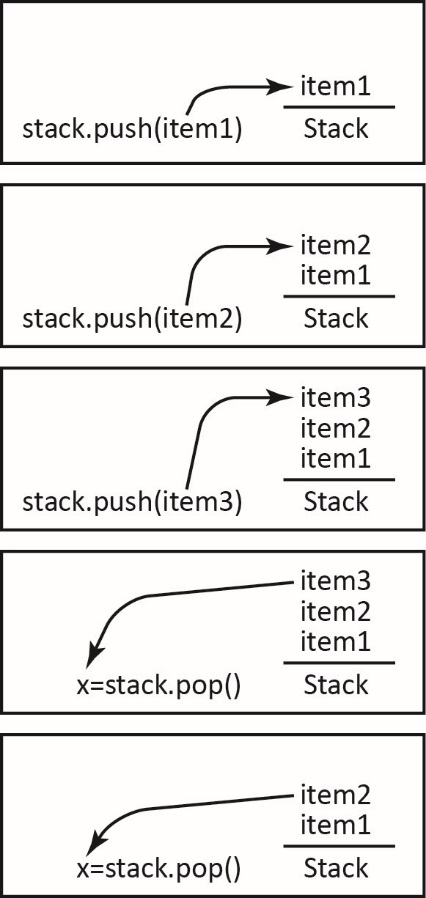
1. (Solution in *exercise\_person\_dogs* package) Consider the example for this section. Add a method to the *Person* class, *getDogWithShortestName* that returns the *Dog* with the shortest length name. Note: (a) this is an iterative method, (b) the code for the *Person* and *Dog* classes is in the *example\_person\_dogs* package.
2. (Solution in *exercise\_person\_accounts* package) Modify the *Person* and *Account* classes from a previous Exercise (where two-way navigability was NOT used, *i.e.* from the classes in the *exercise\_person\_account1* package) to fulfill these requirements:

* A *Person* can have up to 10 *SavingsAccounts.* Provide a way to get an account, add an account, remove an account, and get the number of accounts
* *getTotalBalance* – returns the sum of the balances of all the accounts.
* *applyInterest(rate:double*) – Applies the *rate* to all accounts.
* *getSmallestAccount –* Returns the account with the least balance
* *removeAccount(index:int):Account* – Removes and returns the account at *index*, provide the *index* is valid.
* *toString* – returns a nicely formatted string that contain the name of the person, and a list of all their account balances.

1. Draw a UML class diagram to document your design.
2. Draw an object diagram to depict the case where a person has 3 accounts.
3. Write the *Person* class.
4. Write a *PersonTest* class.

# Implementing a Stack

The code for the example in this section is in the *example\_stack\_of\_integers* package.

In computing, a *collection* (also called a *container* or *manager*)is the generic name for a class that has a1-many relationship with another class. A *collection* class *manages* a group of items that it contains. An array is a collection in the general sense. Later in the semester we will study some other types of collections. Here, we study a *stack* which is a type of collection. A *stack* is like a stack of trays in a cafeteria, all operations occur at the *top* of the stack. For example:

* When you approach the stack of trays, you take a tray from the top of the stack.
* Later, an attendant places clean trays on the top of the stack.

A stack is a very useful data structure that we will consider several more times this semester, and in subsequent classes.

A collection must have a way to add and remove elements. A stack uses the terminology: *push* and *pop* to referring to adding and removing, respectively of an element. You *push* items onto the top of a stack and you *pop* the stack to remove and return the item at the top. This is illustrated in the figures on the right.

A stack is a *last in – first out* (LIFO) data structure: the last item pushed is the next item that is popped, *i.e.* the first one to come out.

The JVM utilizes a stack as it executes your program. When method A starts to execute, all its variables are pushed onto the stack. When A calls method B, all the variables in method B are pushed onto a stack. Then, when method B calls method C, method C’s variables are pushed onto the stack. When method C ends, the JVM pops the stack, removing C’s variables, leaving method B’s variables on the top of the stack, and continues running. Again, we will see this in detail in another chapter.

Standard stack operations:

|  |  |
| --- | --- |
| **Method** | **Description** |
| isEmpty():boolean | Returns true if there are no items in stack, false otherwise. |
| peek():Item | Returns the item at the top of the stack, but does not remove it. |
| push(item:Item):void | Puts the item on the top of the stack. |
| pop():Item | Removes and returns the top item on the stack |
| getSize():int | Returns the number of items in the stack. |

Suppose we want to design a *Stack* class. This is very similar to the 1-many relationship we have considered so far. The only real difference, besides the change in the names of the methods (*push* instead of *add*, and *pop* instead of *remove*), is that the remove method simply removes the last element.

Example – Write a class, *StackOfIntegers* to implement a stack that stores integers.

1. Code

**public** **class** StackOfIntegers {

**private** **int**[] vals; // holds the integers in stack

**private** **int** size = 0;

**public** StackOfIntegers(**int** capacity) {

vals = **new** **int**[capacity];

}

**public** **void** push(**int** val) {

**if**(size<vals.length) {

vals[size++] = val;

}

}

**public** **int** pop() {

**if**(!isEmpty()) {

**return** vals[--size];

}

**return** Integer.***MAX\_VALUE***;

}

**public** **int** peek() {

**if**(!isEmpty()) {

**return** vals[size-1];

}

**return** Integer.***MAX\_VALUE***;

}

**public** **boolean** isEmpty() {

**return** size == 0;

}

**public** **int** getSize() {

**return** size;

}

}

1. Test Code
2. Create an instance of *StackOfIntegers* with a capacity of 10 integers.

StackOfIntegers stack = **new** StackOfIntegers(10);

1. Add 4 integers to the stack.

stack.push(12);

stack.push(8);

stack.push(16);

stack.push(11);

1. Write a statement to print whether the stack is empty and to print its size.

System.***out***.println("stack.isEmpty()=" + stack.isEmpty()); // false

System.***out***.println("stack.getSize()=" + stack.getSize()); // 4

1. Write a statement to remove and return the top value on the stack.

**int** val = stack.pop();

System.***out***.println("stack.pop()=" + val); // 11

System.***out***.println("stack.getSize()=" + stack.getSize()); // 3

1. Write a statement to obtain the top value on the stack without removing it:

val = stack.peek();

System.***out***.println("stack.peek()=" + val); // 16

System.***out***.println("stack.getSize()=" + stack.getSize()); // 3

1. Write a snippet of code to completely empty the stack:

**while**(!stack.isEmpty() ) {

System.***out***.println("stack.pop()=" + stack.pop());

}

System.***out***.println("stack.isEmpty()=" + stack.isEmpty()); // true

System.***out***.println("stack.getSize()=" + stack.getSize()); // 0

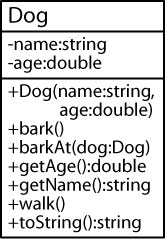
1. Consider the use case for 2 vi above (completely empty the stack). Add a method, *clear* to the *StackOfIntegers* class to achieve this. Hint: you do not need the loop, you can do it with a single line of code:

**public** **void** clear() {

size = 0;

}

## Exercises

1. ****(Solution in *exercise\_stack\_of\_dogs*) Consider the *Dog* class shown on the right.
2. Write a class, *StackOfDogs* to model a stack of *Dog* objects. Hint: look back at the *StackOfIntegers* class. Just a few things need to be changed to create the *StackOfDogs* class. For example: the *vals* array changes type to *Dog[]*, the *push* method doesn’t accept an integer, it accepts a *Dog*, *etc.*
3. Write a *StackOfDogsTest* classto test this class.

# Object Oriented Programming

There are two aspects to OO programming: *writing* classes and *using* classes. For instance, suppose we are developing some banking software. First, we write classes like *Account*, *Loan*, *Customer*, *etc*. that model the *domain* of the problem. Next, we *use* those classes to solve the specific problem we have been given. For example, the system may need to be able to add and remove customers, accounts; display customers (or accounts) sorted on name, or account number, *etc.* To write such a system, we would use the classes above, and we would have to introduce a number of more classes. Mostly, for this class, you will simply write the domain classes.

A major focus in writing classes is designing classes that are easy to use and versatile. In general, these classes are not solving any specific problem; rather, they are modeling the context/domain of the problem (as discussed above). Identifying and designing these classes is a skill that takes time to develop. The more problems you solve, the better you will get at it. If you want practice at this, think about something you are interested in (a sport, music, some aspect of a job you have, anything) and try to develop a class diagram of the domain, just the classes and their associations, without the methods.

The interface (API, public interface) for a class is the set of public members (public fields and methods). In other words, the interface specifies exactly what messages can be sent (methods called) to an object from the outside. **Thus, a programmer who is using a class is usually only interested in the public interface for the class because those are the only things it can use.** In other words, to the person *using* classes, the classes themselves are black boxes. For example, for any particular method, the person doesn’t know how the code is implemented, they just care about *what* the class does. The term *encapsulation* refers to this idea of hiding the details of a class by providing an easy to use interface.

1. A *use case* is simply a statement of a single goal that an end user wants to be able to accomplish with a system. For example:

   * A person can walk their dog.
   * A person can obtain their balance from an ATM.

   [↑](#footnote-ref-1)