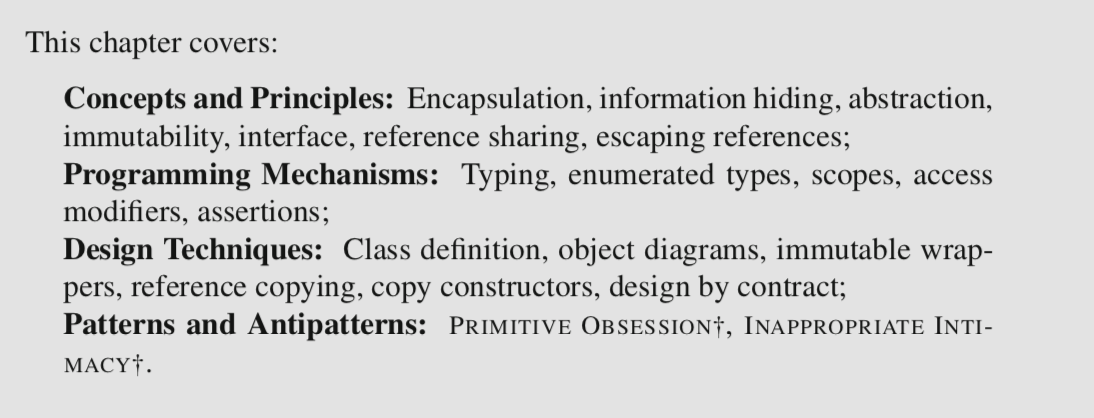
Chapter 2

Encapsulation



[Solitaire Game]

SUIT(Hearts ♥, Spades ♠, Diamonds ♦, Clubs ♣)

RANK(Ace, 2, 3, ..., 10, Jack, Queen, King)

2.1 Encapsulation and Information Hiding

The idea of **encapsulation** is to enclose something as if it were in a capsule, *to hide the internal implementation of an abstraction behind an interface that tightly controls how an abstraction can be used.* In software design we encapsulate both data and computation to limit the number of contact points between different parts of the code.

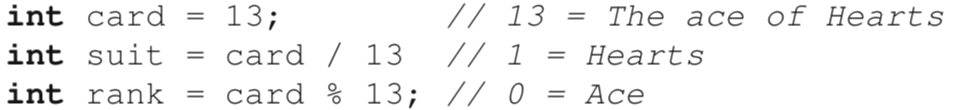
Encapsulation in software design is related to the principle of **information hiding**: *Only reveal the minimum amount of information that is necessary to use them, and hide the rest.*

* It makes it easier to understand a piece of code in isolation
* It makes the use of the isolated part by the rest of the code less error-prone
* It makes it easier to change one part of the code without breaking anything

e.g. A typical example of information hiding is an implementation of a stack abstract data type (ADT) whose interface only provides push and pop operations. This minimal interface allows client code to make full use of the stack structure, but low-level design decisions on how to store elements in the stack remain hidden from the code that uses the stack.

2.2 Encoding Abstractions as Types

Use a single integer between 0 and 51 where the value of the integer somehow represents the card according to a given convention



[Disadvantages]

* The representation of a card does not map to the corresponding domain concept.

To facilitate code understanding and help avoid programming errors, the representation of values should ideally be tied to the concept they represent. For example, the general type int maps to the concept of an integer (a type of number), not that of a playing card. We could define a variable of type int intended to store a playing card, and unwittingly put a value that represents a different entity in it (e.g., the number of cards in the deck). This will not be noticed as an error by the compiler, yet it is likely to lead to intense confusion when executing the code.

* The representation of a card is coupled to its implementation.

If our design decision is that cards should be represented as integers, any location in the code that must store a value that represents a card will refer to an integer. Changing this encoding to something else (e.g., int[] card = {1,0}) will require discovering and changing every single location where an int variable is used to store a card, and all the code that works with cards as integers.

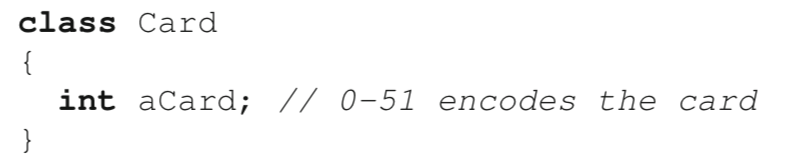
* It is easy to corrupt a variable that stores a value that represents a card.

In Java a variable of type int can take 232 distinct values. To represent a playing card we only need a tiny subset of these (52). Consequently, the overwhelming majority of values we can store in an int variable (232 − 52) does not represent any valid information. This opens the door very wide to errors.

It is generally a bad idea to try to shoehorn domain concepts into basic general types like int, String, and so on. **Ideally, these types should only be used to hold values that are proper values of the type.** The tendency to *use primitive types to represent other abstractions* is a common problem in software design that has been captured by the antipattern PRIMITIVE OBSESSION†.

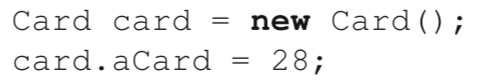
To apply the principle of information hiding, we instead organize our code to *hide* the decision of how exactly we represent a card.

This class defines a single instance variable **aCard** of type **int**. The name of the instance variable includes the prefix **a** as part of a coding convention



Although using a class somewhat links the value a bit better to the domain concept of a card, the other problems are still present.

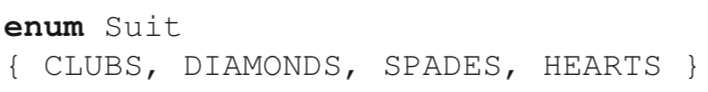
* It is still possible to corrupt the representation of a card.
* The decision to represent this value as an int is not exactly hidden, given that client code would be accessing the variable directly through a dereference of the instance variable.



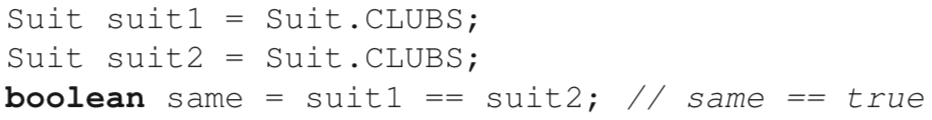
dereference

Define abstractions for ranks and suits

Notice that the rank of a playing card can only be one of 13 distinct values, which are known in advance and can be enumerated. In the case of suits, the number of values is even smaller (four).

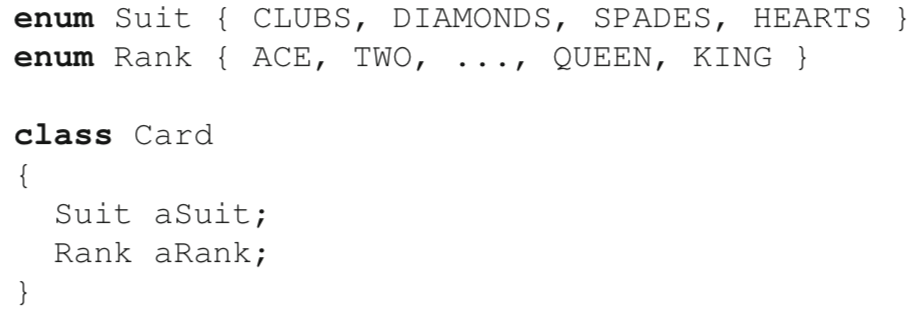


In Java, enumerated types are a special kind of class declaration*. The identifiers listed in the declaration of the enumerated type are globally available constants. These constants store a unique reference to an object of the class that corresponds to the enumerated value.*



They meet all our design requirements, because variables of type Suit and Rank are directly tied to their corresponding concept of rank and suit, and variables of these types can only take values that are meaningful for the type. [With the exception of null. ]

It assumes that each enumerated type is defined in its own file



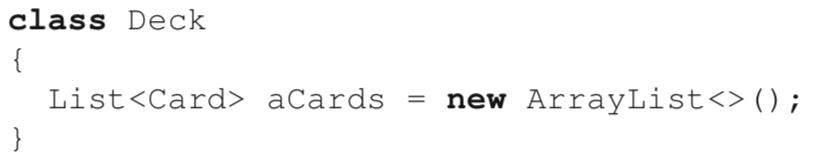
Because a deck is just a collection of cards, we could represent a deck of cards as a List of Cards



[Disadvantages (same as representing a playing card as an int)]

* A list of cards is not strongly tied to the concept of a deck. It could represent any list of cards, e.g., the cards in one of the piles created while playing Solitaire, the cards discarded as part of the game, etc.
* Using a list of cards ties the representation of a deck in the program with its implementation. If we decide later to replace the list by, say, an array, we would have to change all the corresponding source code locations.
* The structure can easily be corrupted: A simple deck of cards can hold a maximum of 52 cards, without duplicates. A list allows one to put any number of cards in the structure, including duplicates.

A better way to approach the representation of a deck of cards in our code is to also define a proper type for it, hiding the decision of how the cards are stored.



2.3 Scopes and Visibility Restrictions

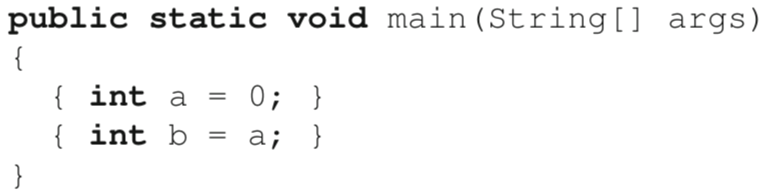
An object is a mechanism to *group variables together and access their values through the process of dereferencing.*

Access modifiersare Java keywords that *control what parts of the code can see, or access, certain program elements (e.g., classes, fields, methods).*

[Dual Purposes]

* They express the **intent** of the developer about how certain structures are meant to be used.
* They support the automatic enforcement of the stated intent through compilation.

The idea of restricting access to fields is very similar to that of visibility and scope for local variables. In most programming languages, *a scope is a lexical region that acts as a sort of boundary for variables.* In Java, scopes are defined using curly braces.

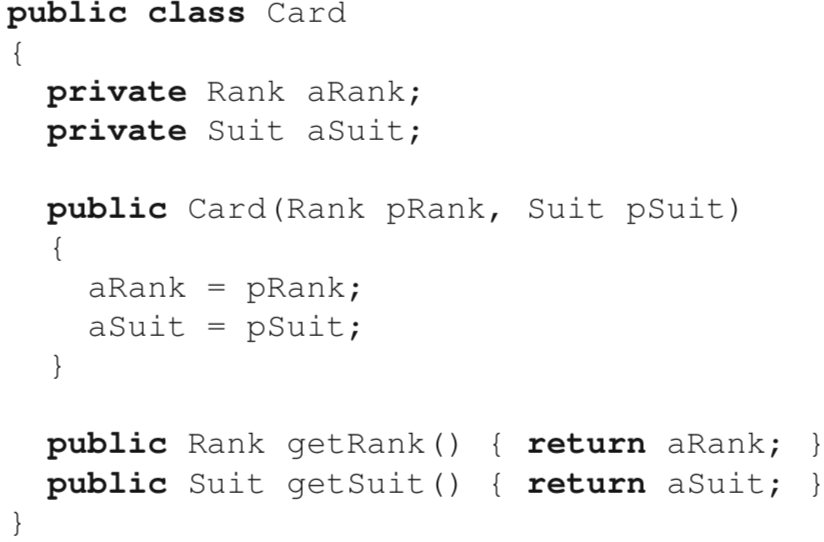


compile error

**public**: visible anywhere in the code.

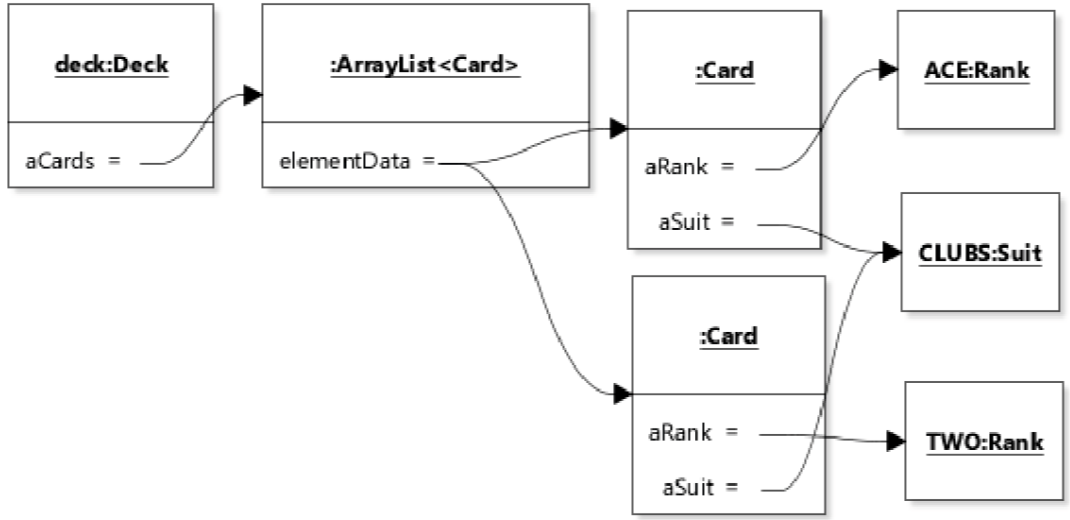
**private:** only visible within the scope of the class.

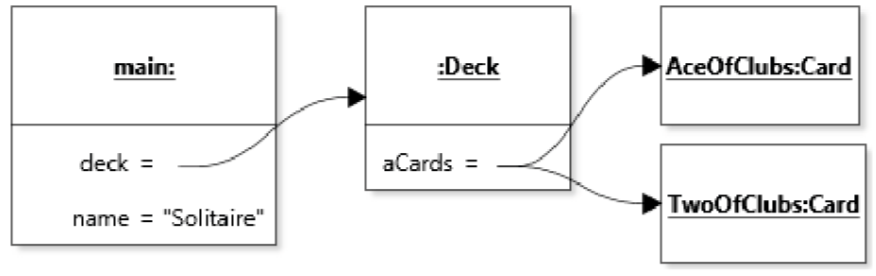
A general principle for achieving good encapsulation is to use the narrowest possible scope for class members. Thus, instance variables should almost always be private. Also, public methods should reveal as little as possible about implementation decisions meant to be encapsulated.



2.4 Object Diagrams

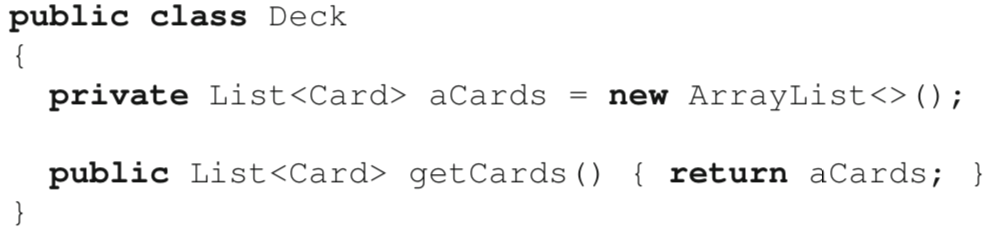
In an object diagram, a rectangle represents an object, with its name and type indicated as name:type. Both name and type information are **optional**, but in general it is useful to have at least one of the two. By convention, in UML diagrams the name of objects (as opposed to classes) are underlined. Objects can contain fields, which are just like fields in a Java program. Fields can contain a value of a primitive type or a value of a reference. In the latter case, the value is represented as a directed arrow to the object.



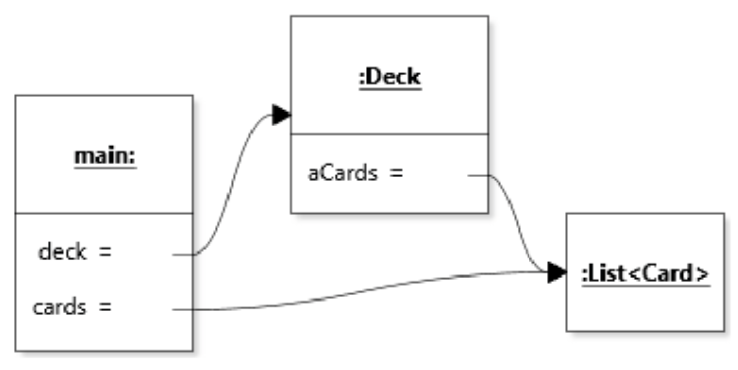


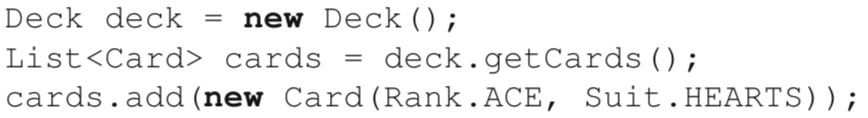
2.5 Escaping Reference

1. Returning a reference to an internal object



It allows a reference to the private internal list of cards to *escape* the scope of the class, thus granting access to internal elements of the class from outside the class.

Client code:



Having a class that is mostly accessed through getters and setters points to a design weakness, because the abstraction the object represents is not effective. This problem is also known as the INAPPROPRIATE INTIMACY† antipattern, because its symptom is that classes “spend too much time delving in each others’ private parts”. To the extent possible, objects should interact with each other using methods that involve abstractions above individual instance variables. In the case of the Deck class, this means prohibiting access to the internal list of cards, which constitutes a “private part”.

2. Storing an external reference internally

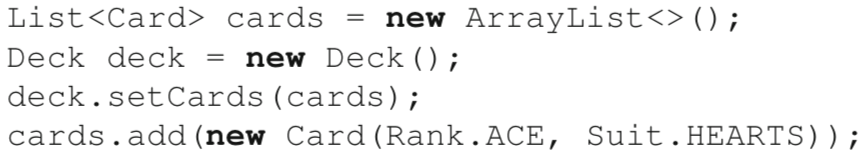
* Setter method



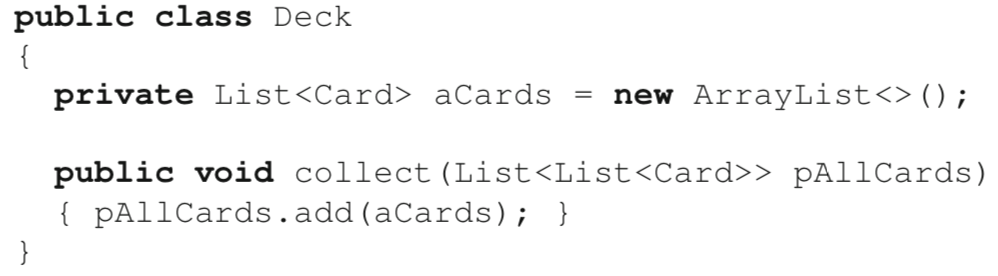
* Constructor

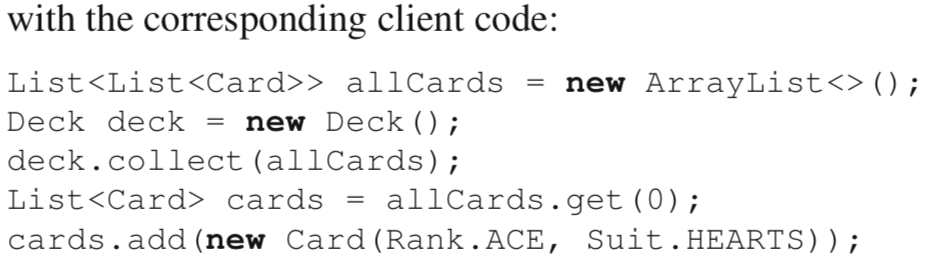


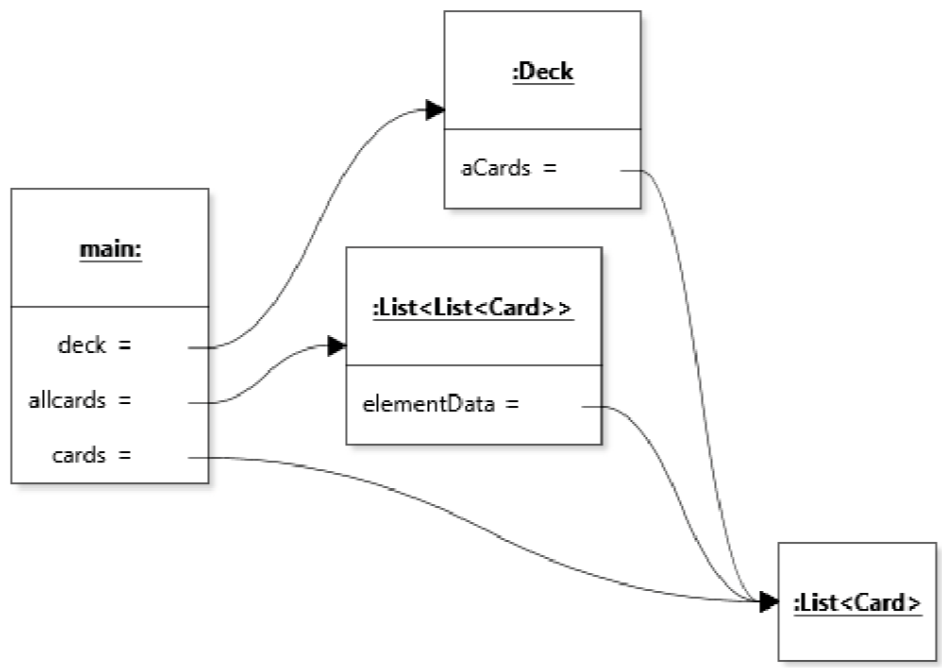
Here, we can corrupt the state of the deck from the scope of the client code, for example by adding an ace of hearts.



3. Leaking references through shared structures





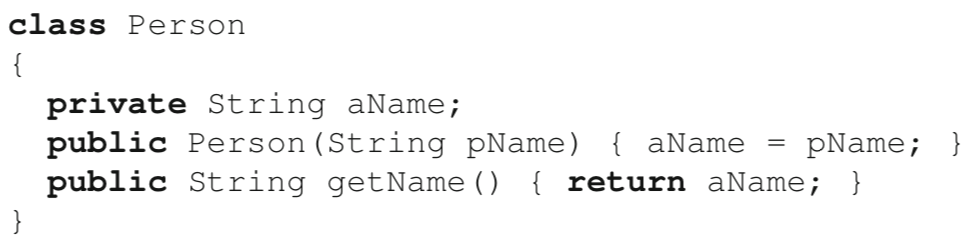


The issue of escaping references is complex because references can escape through any number of shared structures, which may not always be obvious.

2.6 Immutability

Good encapsulation: *It should not be possible to modify the internal state of an object without going through its methods.*

Section 2.5 discussed the issue of escaping references, and how they threaten encapsulation. There is, however, one situation where leaking a reference to an internal object is harmless: when the object is immutable (i.e., impossible to change).

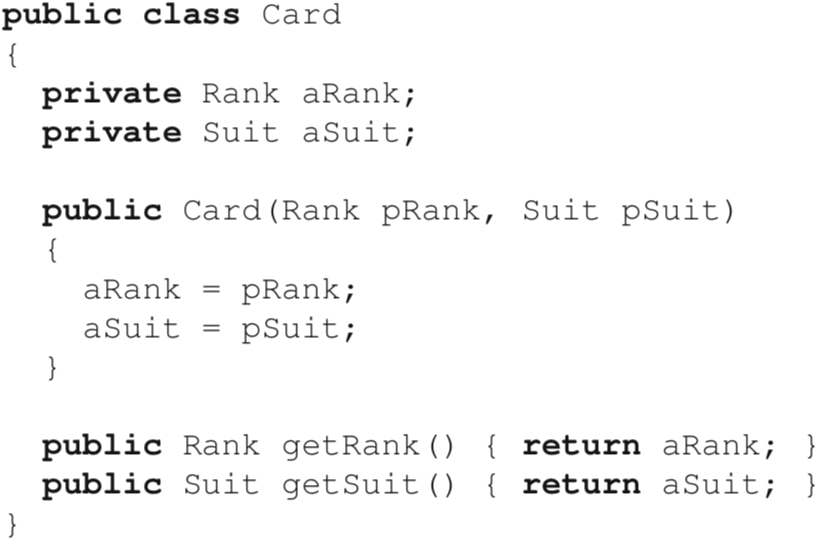


*Objects are immutable if their class provides no way to change the internal state of the object after initialization*

🡪 Support sharing information encapsulated in an object without breaking encapsulation

[Define class Card to be immutable]

First, we rely on our two **enumerated** types Rank and Suit which we assume to be **immutable.**



In this definition of the class, the only way to set the values of the two instance variables is through the constructor call which, by definition, is only executed once for each object. The fields are private, so they cannot be accessed from outside the class. There are only two methods. Although they are public, neither mutates (i.e., changes) the state of the object. Finally, although the methods return a reference to the content of a field, the type of these fields is immutable, so it will not be possible to change the state of the referenced objects in any case. The class is thus immutable. [To make the immutability of the class even more explicit, the keyword final could be placed in front of the class keyword, as well as before the type of both fields.]

2.7 Exposing Internal Data

In many cases the objects of the classes we define will need to expose part of the information they encapsulate to other objects but without breaking encapsulation.

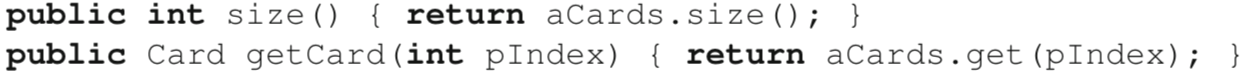
[Design our Deck class so that it is possible to find out what cards are in a deck]



Adding a getter method that simply returns aCards is out of the question, as this allows code outside the class Deck to modify the internal representation of a Deck instance.

1. Extended interface

Extend the interface of the class to include access methods that only return references to immutable objects.

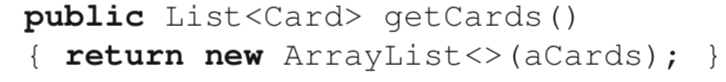


However, it is somewhat inelegant if client code typically needs to **access all the cards** in the deck. In such a situation, the code would become cluttered with calls to size() and for loops going over all indexes. Code might also need to be written to check that the argument to getCard is not out of bounds, and so on.

2. Returning copies

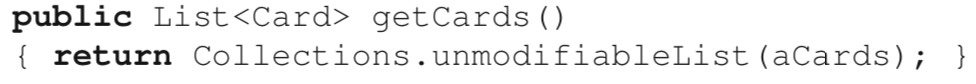
Return a *copy* of the list stored in aCards.

Assum that Card objects were immutable, so it was sufficient to perform a **shallow copy**. *A shallow copy of a list is a copy of the list with shared references to the elements in the original list (i.e., the elements are not copied).*

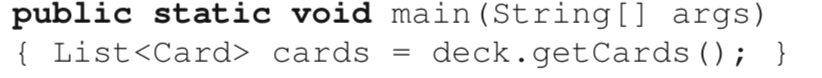


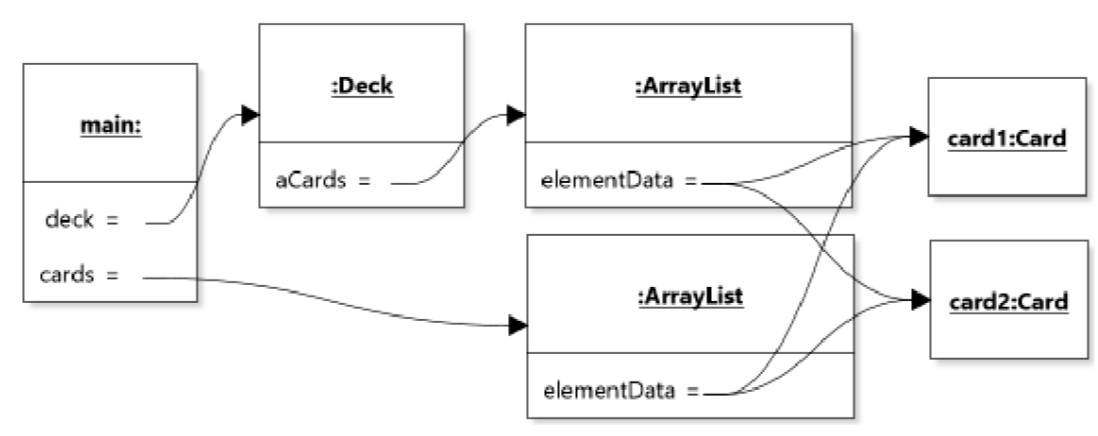
This code relies on the behavior of the constructor ArrayList(Collection), which creates a new ArrayList and initializes this list with all the elements in the collection, in the same order. Thus, a client would receive a reference to a **different** list of cards, with the same cards.

OR

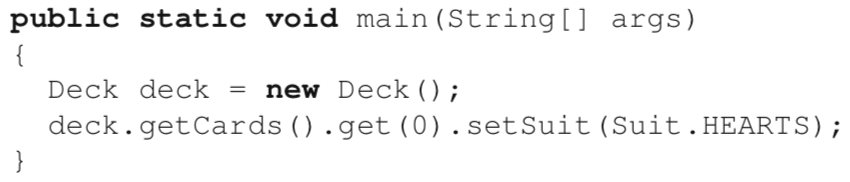


Client code:



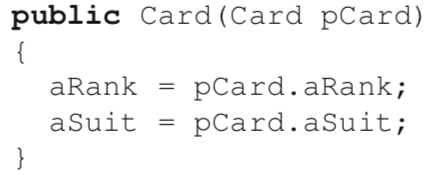


Card objects were mutable, it would become possible to change the state of a deck without going through its interface.

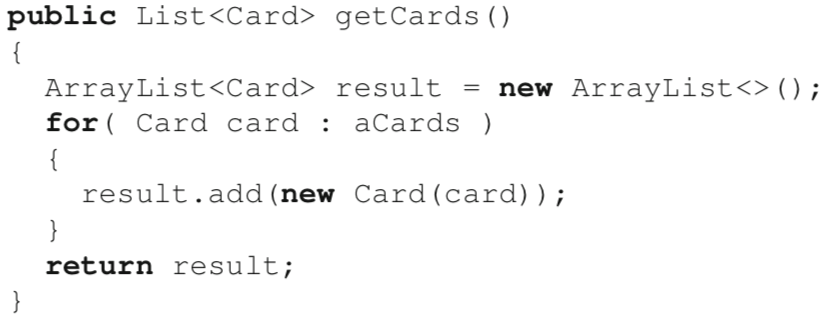


*🡪 Copy Constructor*

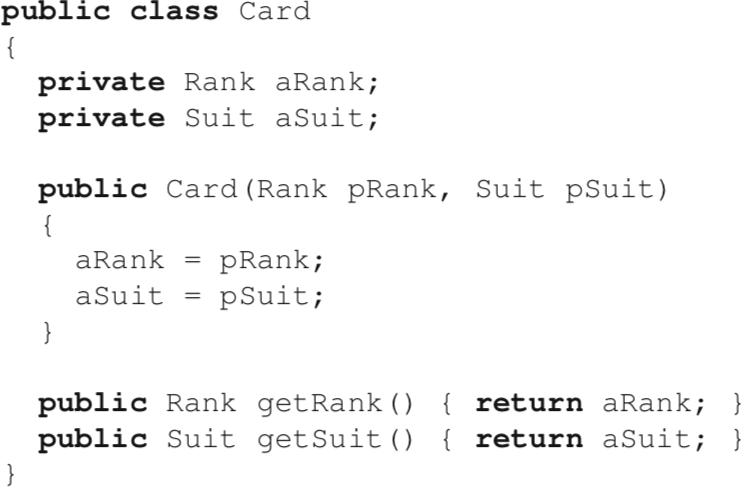
The idea is to design a constructor that takes as argument an object of the same class, and (usually) to copy matching field values:



new ArrayList<>(aCards), is an example of a copy constructor for ArrayList.



2.8 Design by contrast



The encapsulation provided by this class is very good, but there remains a crack in the “shell” it provides: It is possible to create a new card with a null reference:

Card card = **new** Card(**null**, Suit.CLUBS);

The interface to the class is ambiguous: it is not clear whether client code should not use null as an argument to the constructor, or whether it should simply be careful not to use a card for which either the rank or suit is null.

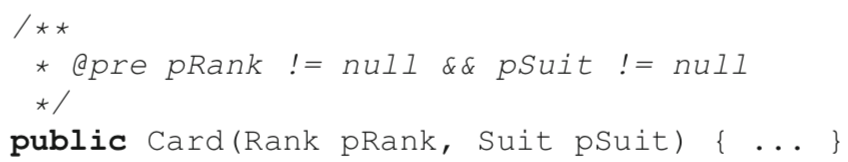
The main idea of design by contract is *for method signatures (and related documentation) to provide a sort of contract between the client (the caller method) and the server (the method being called).* This contract takes the form of a set of **preconditions** and a set of **postconditions**.

A precondition is a predicate that must be true when a method starts executing. The predicate typically involves the value of the method’s arguments, including the state of the target object upon which the method is called.

Similarly, postconditions are predicates that must be true when the execution of the method is completed.

Given preconditions and postconditions, the contract is basically that the method can only be expected to conform to the postconditions if the caller conforms to the preconditions. If a client calls a method without respecting the preconditions, the behavior of the method is undefined. In practice, design by contract is a great way to force us to think about all possible ways to use a method.

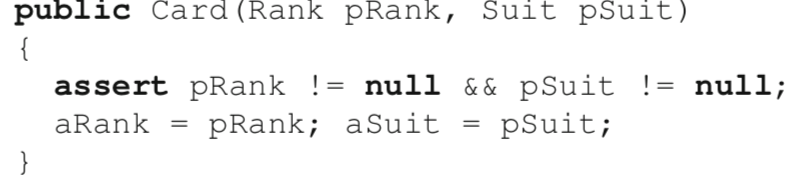
Preconditions are specified using Java statements in the comments using the Javadoc @pre tag and postconditions are specified using the tag @post.



It is possible to make pre- and postconditions (and any other predicates) checkable in Java using the assert statement. The **assert** statement *evaluates the predicate in the statement and raises an* AssertionError *if the predicate evaluates to false*.

[Assertion checking is disabled by default in Java, so to use this properly it is necessary to add

-ea (enable assertions) as a VM parameter when running Java programs.]



Correctly implemented, design by contract helps prevent the tedious idiom of *defensive programming* where corner cases (such as null references) are checked for everywhere in the code. Additionally, the technique supports rapid *blame assignment* while debugging: If a precondition check fails, the client (caller method) is to blame. If a postcondition check fails, the actual method being called is to blame.

A final note about design by contract is that the addition of preconditions to a method’s interface actually relieves us of the requirement to handle the condition. So the code below is not properly designed because it both states that null references are not a valid input and handles them in a consistent way (by raising an exception). If a method checks for a certain type of input (like null references) and produces a well-defined behavior as the result, then this is part of the method’s **interface specification**. When designing method interfaces, it is important to decide whether the method will be in charge of rejecting illegal values, or whether these will simply be specified as invalid. These are two different design choices.

