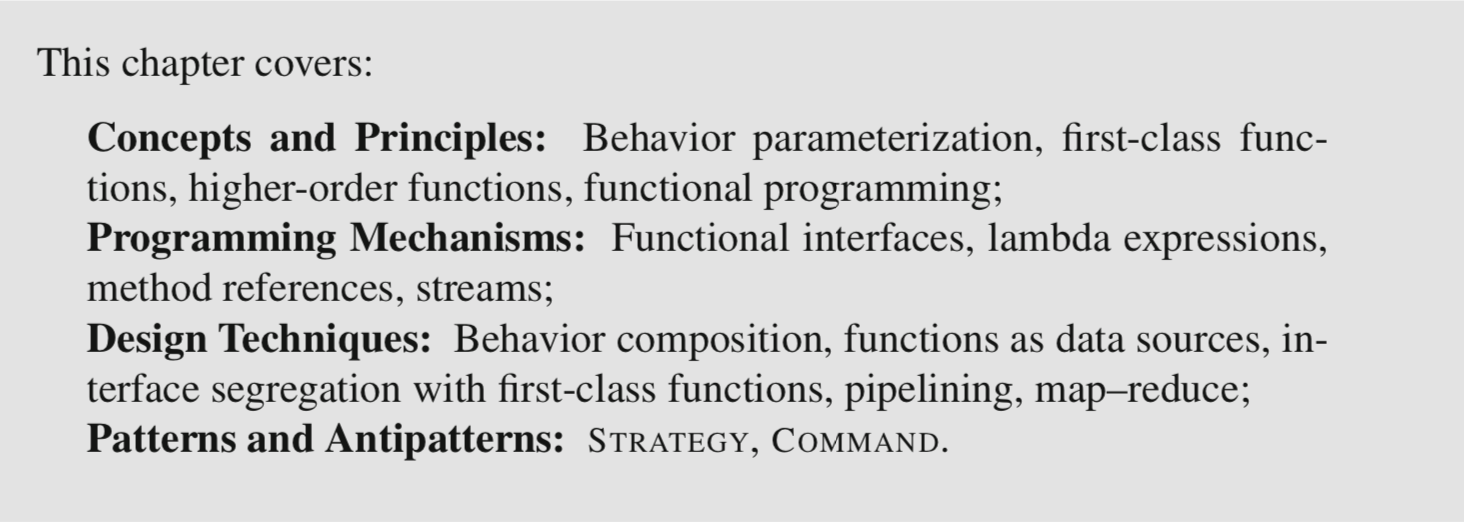
Chapter 9

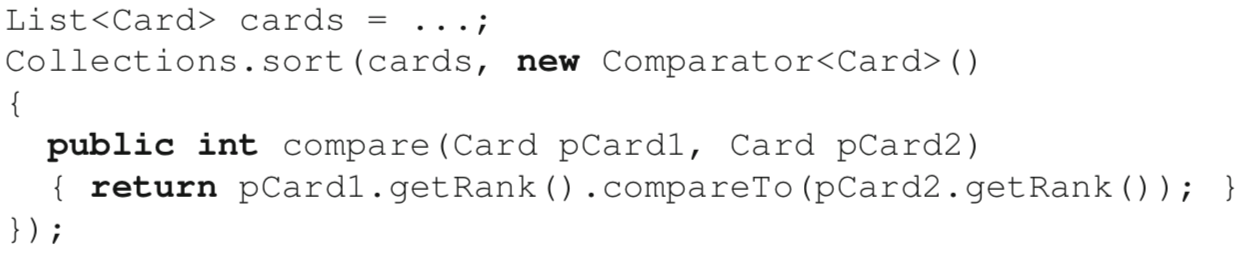
Functional Design



9.1 First-Class Functions

[Old solution]

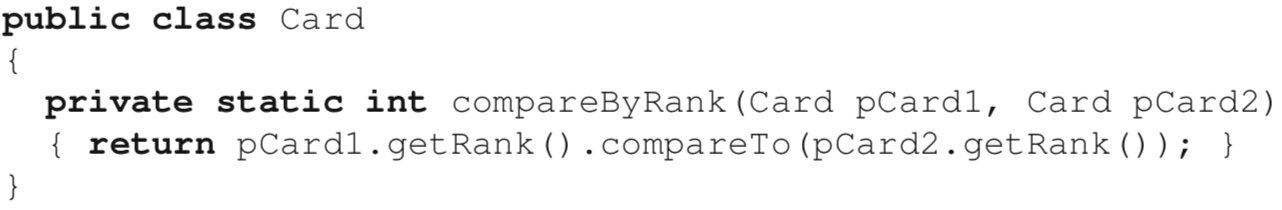
The design goal is to parameterize the behavior of the sort method, and the mechanism we use to do this is to pass a reference to an object.



What would be a better fit, would be for the sort method to take in as input the desired sorting function directly.

Providing functions as input to other functions, however, requires the programming language to allow this by supporting first-class functions. This essentially means *treating functions as values that can be passed as argument, stored in variables, and returned by other functions.*

e.g. we could define a function in class Card that compares two cards by rank:



and supply a reference to this function as the second argument to method sort:



This code, which compiles and does what we want, is actually syntactic sugar that gives the illusion of first-class functions but actually converts the method reference Card::compare into an instance of Comparator<Card>.

With first-class functions, it becomes possible to design functions that take other functions as arguments. Such functions are called higher-order functions. In a way, when considering the above code from a functional point of view, we can say that Collections.sort is a higher-order function. In some contexts, it is possible to build entire applications from the principled use of higher-order functions. In such cases, we would say that the application is designed in the functional programming paradigm.

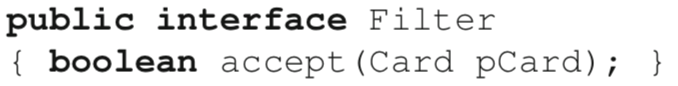
9.2 Functional Interfaces, Lambda Expressions, and Method References

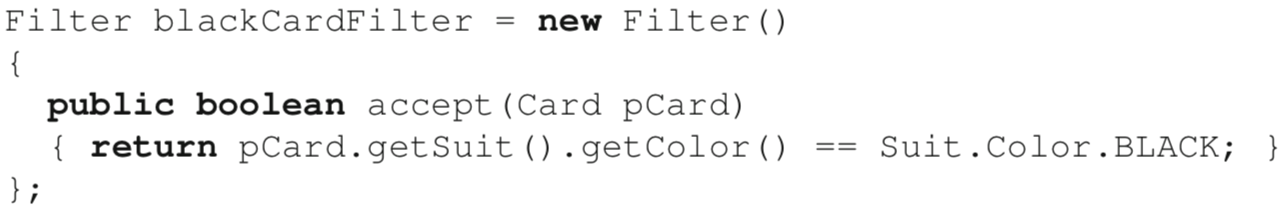
**Functional Interfaces**

In Java, a functional interface is an interface type that declares a **single abstract** method.

🡪 equivalent to a function type, enables fist-class functions in Java

e.g. we could define an interface to represent “filtering” behavior for a collection of cards:





Functional interfaces define a function type.

If we forget about the implicit parameter for a second, we can consider method accept of interface Filter to be a function that takes as parameter a Card instance and returns a boolean. Thus, we have a function of type Card → boolean. Now, because our Filter interface only defines a single abstract method, implementing this interface amounts to supplying the implementation for this single function. With a bit of imagination, we can consider that obtaining an instance of Filter is equivalent to obtaining an implementation of a method that takes as argument a reference to a Card instance and returns a boolean. Hence, functional interfaces can play the role of function types.

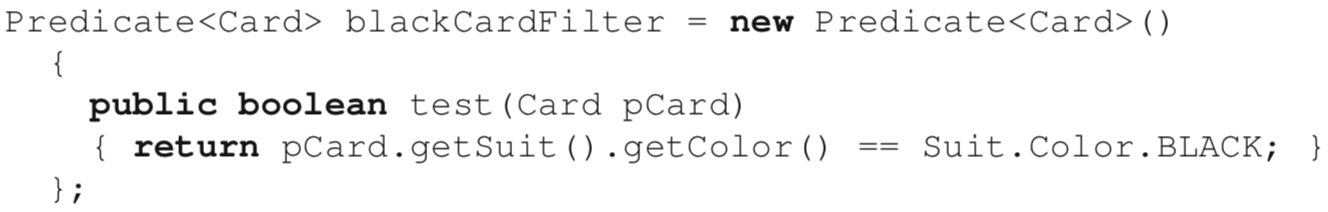
The use of the word abstract in the definition of a functional interface is important.

Starting with version 8 of the language, interfaces in Java can define static and default methods. Because an implementation for such methods is provided directly in the interface, implementing types are not required to provide one. Static and default methods are thus, by definition, not abstract. This means that an interface can define multiple methods, and still qualify as a functional interface if only one of them is abstract.

A good example of such an interface is Comparator<T>. The Comparator<T> interface defines numerous static and default methods, whose purpose is going to become clear later in this chapter. However, the interface defines a single abstract method: int compare(T,T) (where T is a type parameter). For this reason, Comparator is a functional interface that defines the function type (T, T) → int. The implication for functional-style programming is that we are able to treat instances of Comparator<T> as first-class functions.

With functional-style programming, Java 8 introduced a library of convenient functional interfaces, located in package java.util.function. These interfaces provide the most common function types, such as Function<T,R>, a generic type that can represent the type of any unary function between reference types. The interface has a single method apply.

To use a library type instead of our custom Filter interface, we use the functional interface Predicate<T>, which represents the type of a function with a single argument of type T and returns a boolean. The name of the abstract method for Predicate<T> is test(T).



**Lambda Expressions**

The use of the new keyword in the definition of the behavior of our predicate betrays the fact that we are still *creating an object*. To more directly express our design in terms of a first-class function, we can define the implementation of a functional interface as a lambda expression. In Java, lambda expressions are basically anonymous functions.

[Syntax of lambda expressions]

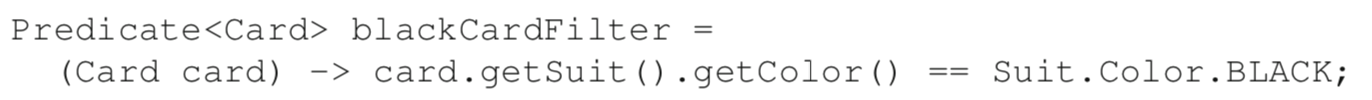
1. a list of parameters; () if empty

2. a right arrow (the characters ->)

3. a body: two forms

initialize blackCardFilter with *behavior (a function)* as opposed to *data (an object)*

• a single expression (e.g., a == 1).



Note: How expressing the body of a lambda as an expression does not require a semicolon after the expression; the final semicolon terminates the entire assignment statement, not the lambda expression.

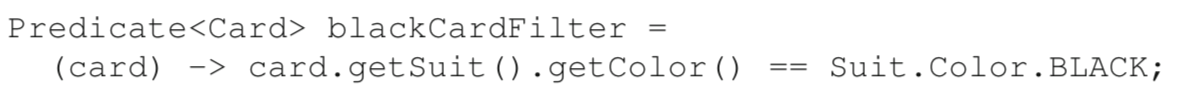
• a block of one or more statements (e.g., {return a == 1;}).



Compiler checks:

* The type of the variable is a functional interface;
* The parameter types of the lambda expression are compatible with those of the functional interface;
* The type of the value returned by the body of the lambda expression is compatible with the one of the abstract method of the functional interface.

Because the types of the parameters of the function implemented by the lambda expression are already encoded in the definition of the abstract method in the corresponding functional interface, it is not necessary to repeat them in the declaration of the lambda expression. To make our code more compact, we could simply omit the optional declaration of parameter type Card:



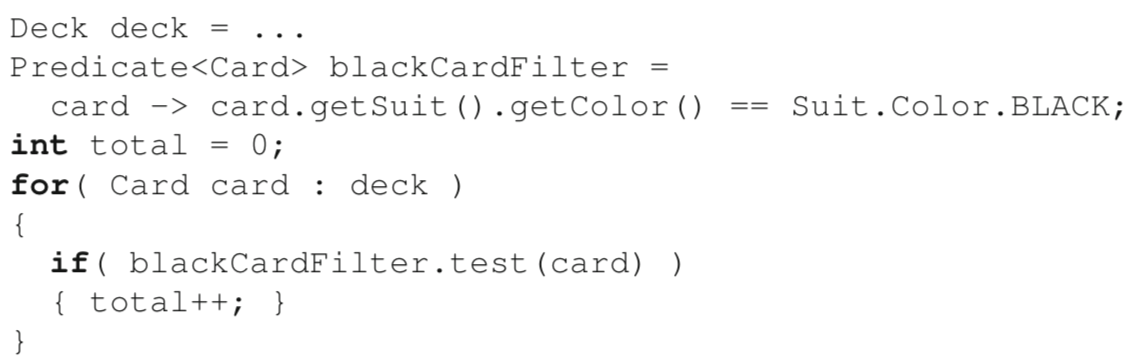
In fact, if the function type takes a single parameter, we can even omit the parentheses around the parameter:



Whether or not to include parameter types in the declaration of a lambda expression is a matter of style. However, it is good to keep in mind that they can help make the code more readable. When types are provided, a compact variable name becomes more acceptable. For example, we could rewrite the above as:



How to call a lambda expression



Lambda expressions are also a good match for providing behavior in-place when required by library or application functions. For example, the method removeIf of class ArrayList takes a single argument of type Predicate<T> and removes all elements in the ArrayList for which the predicate is true. Given an ArrayList of Card, we can remove all black cards from the list with a single call:

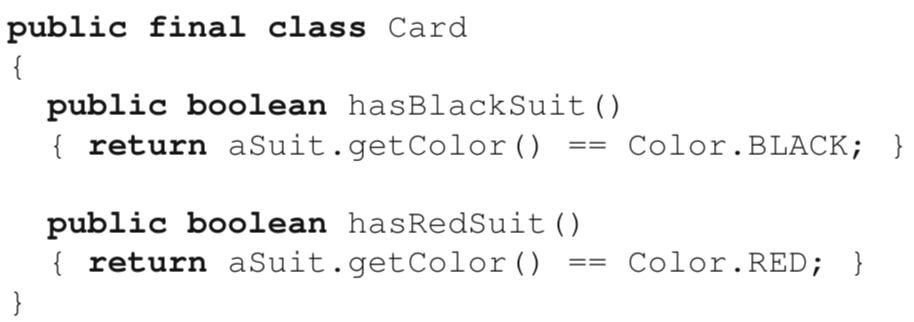


**Model Reference**

In Java, method references are indicated with a double colon expression C::m where m refers to the method of interest and C the class in which it is defined.

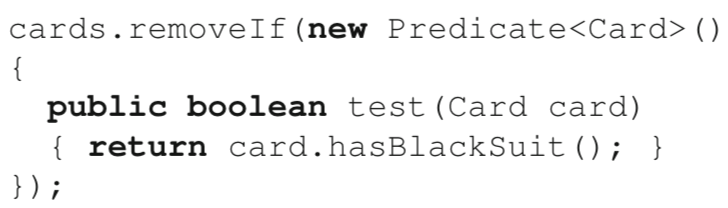
Method references support using both static and instance methods as first-class functions.

(1) Instance method

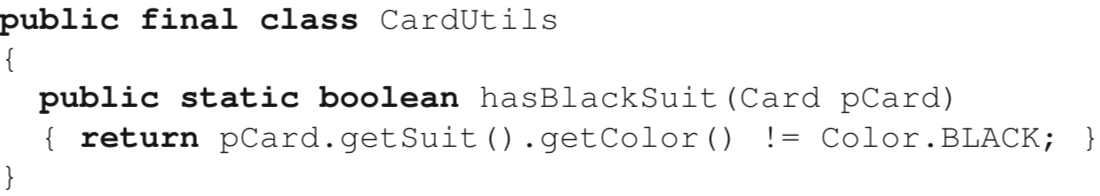




In this case, the method reference is interpreted as (the argument of the call to the method of the functional interface is bound to the implicit parameter of the method reference)

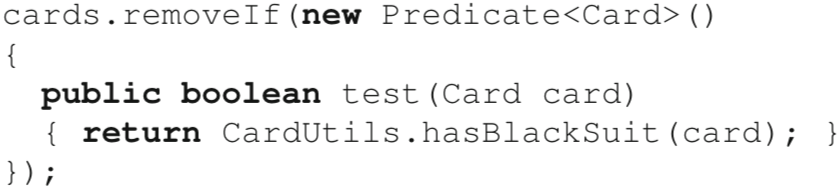


(2) Static method



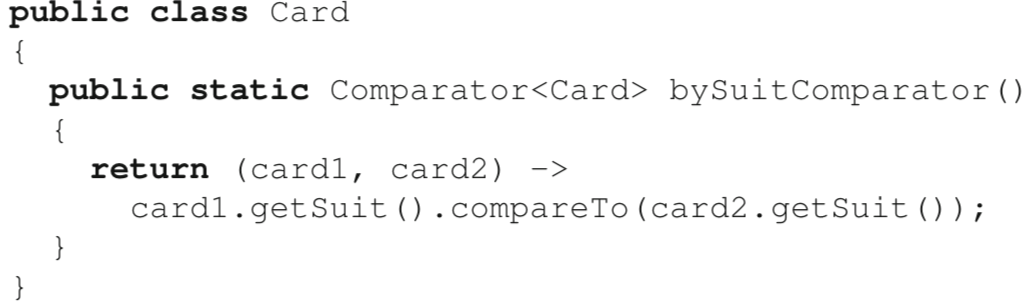


In this case, the method reference is interpreted as (the argument is bound to the explicit parameter of the method reference)



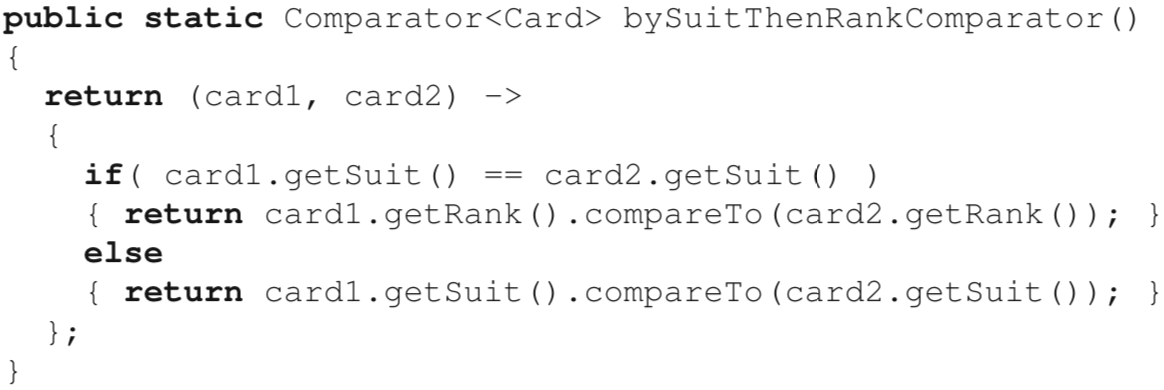
Comment: Both Card::hasBlackSuit and CardUtils::hasBlackSuit return a boolean and take as input a single parameter of type Card. In the case of the instance method, the parameter is the implicit parameter of the method, whereas in the case of the static method, the parameter is the formal parameter of the method. Either way, both implement the function type Card → boolean and can thus be assigned to a variable of type Predicate<Card>.

9.3 Using Functions to Compose Behavior

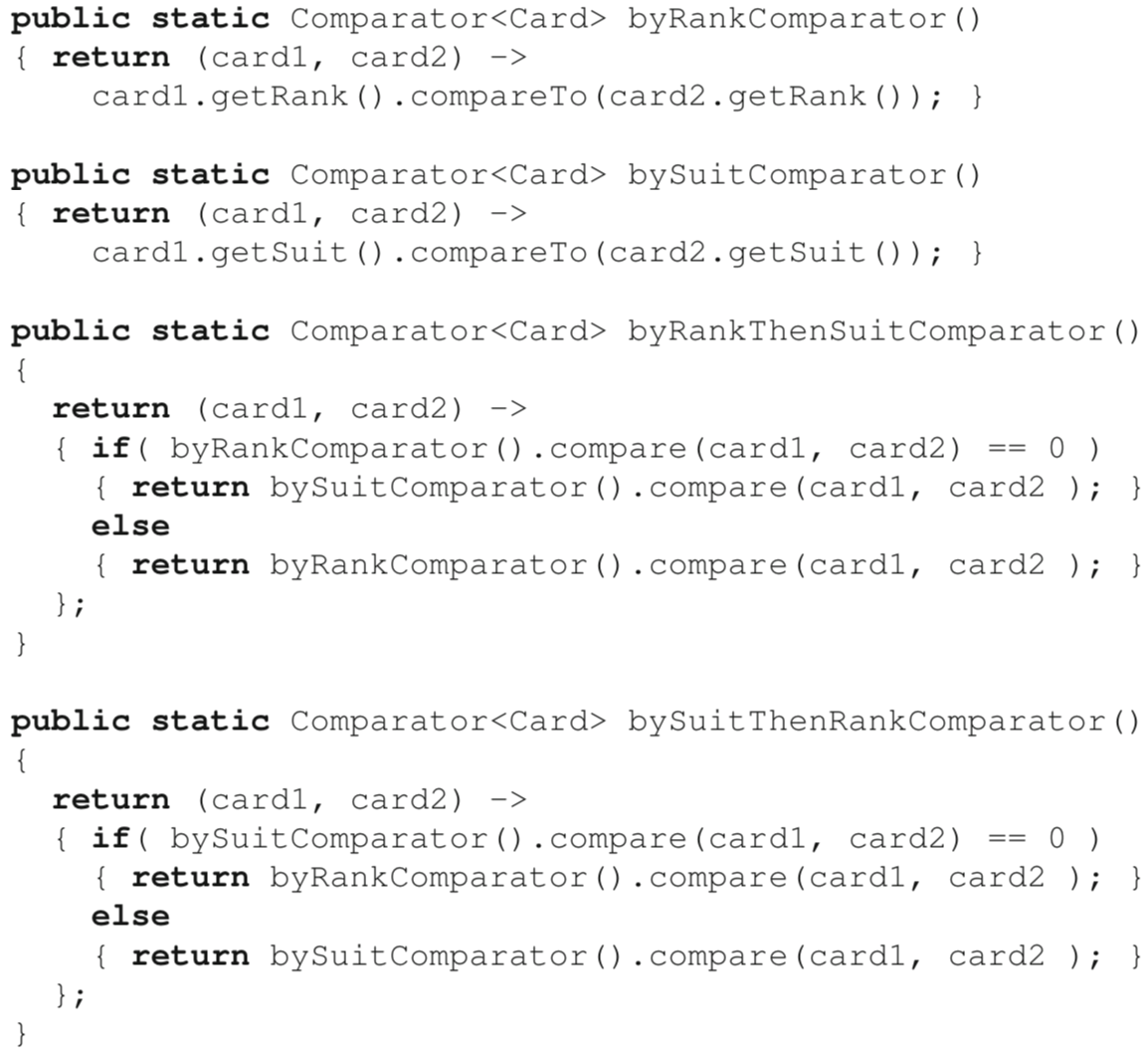


This design uses a static factory method to return a comparator object that compares two cards in terms of their suit, as defined by the suit ordering in the enumerated type Suit. Because we use a lambda expression, the code expresses the solution more in terms of a first-class function than a function object. However, this solution is incomplete because if two cards have the same suit, their relative order is undefined, which is not ideal for many card sorting contexts. To complete the solution, we need to specify a secondary comparison order by rank.

Method 1: extend the code of lambda expression



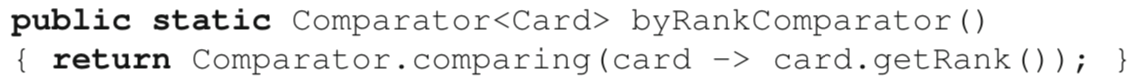
Method 2: compose two single-level comparator



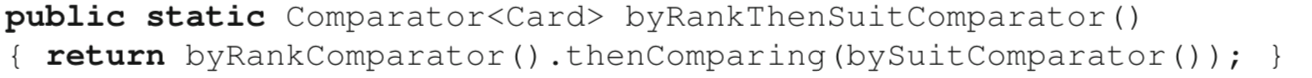
Unfortunately, without extra help, this idea does not really mitigate the complexity of the composite function (and does not even cover the option to reverse the order of either suit- or rank-based ordering).

Solution: Use functions to do the composition

comparing(...)creates a comparator by building on a function that extracts a comparable from its input argument. For example, we could rewrite byRankComparator() as:

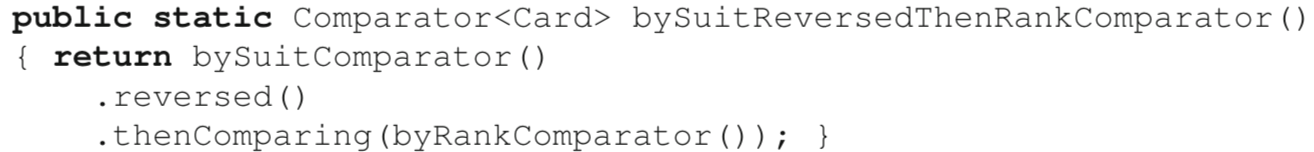


thenComparing(...) is a default method called on a comparator that takes as input another comparator for the same type. 🡪 cascade comparisons (for example to compare by suit if the rank is the same, or vice versa)

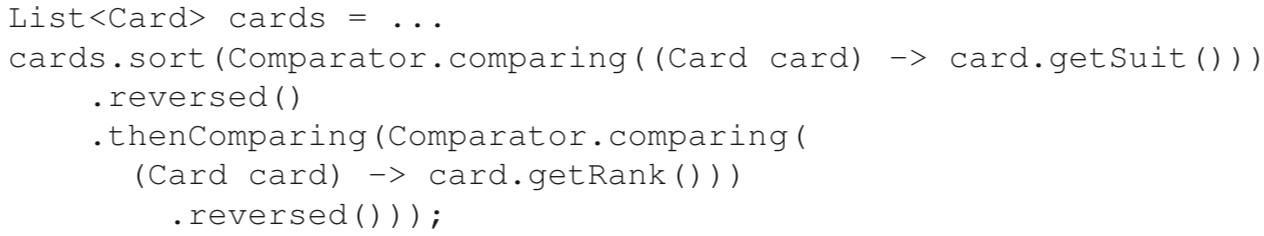


reversed() creates a new comparator that orders elements using the reverse of the order used by the implicit argument of reversed().

e.g. sort by descending suit, then ascending rank

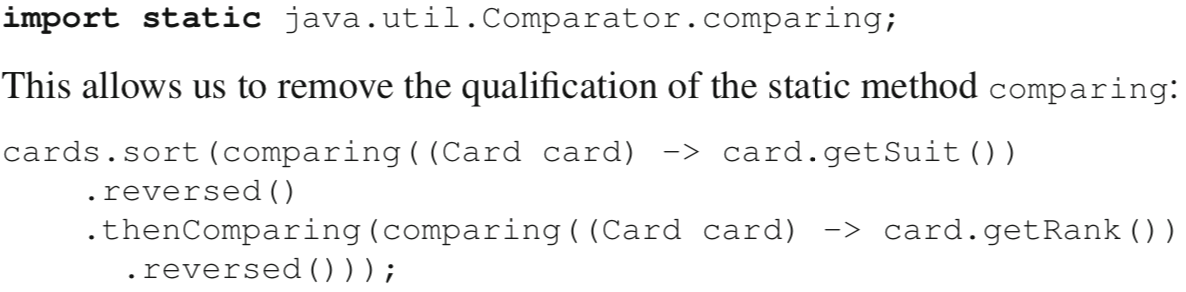


Because the only part of the Card interface needed to define the comparison behavior is already available through the getter methods getSuit() and getRank(), the factory methods are not strictly necessary.

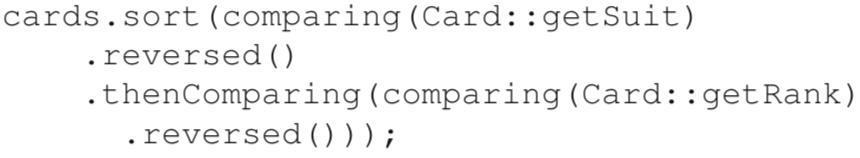


Improvement

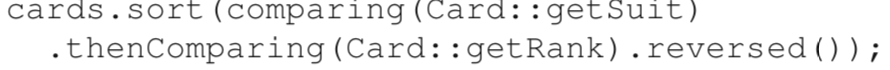
1. Use Java’s static import feature to eliminate the need to qualify the static methods:



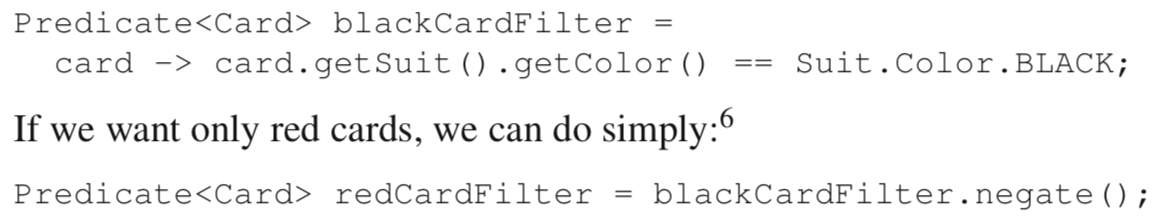
2. Use method references to refer to getSuit() and getRank() instead of redefining a lambda expression that simply calls them.



3. We can observe that class Comparator has an overloaded version of thenComparing that combines the behavior of comparing and thenComparing by directly taking a function that returns the value of the key we wish to use for comparison. In this case we can move the reversal of the comparison to the final comparator.



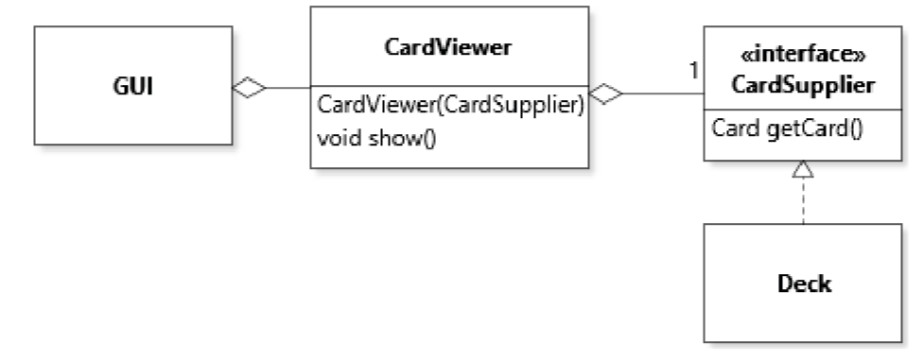
Other helper methods in java.util.function



9.4 Using Functions as Suppliers of Data

Supplier<T>

[context] We are designing a class CardViewer to show a graphical representation of the card at the top of a pile in a card game. The CardViewer must be able to show the card of interest whenever necessary by calling a method show().



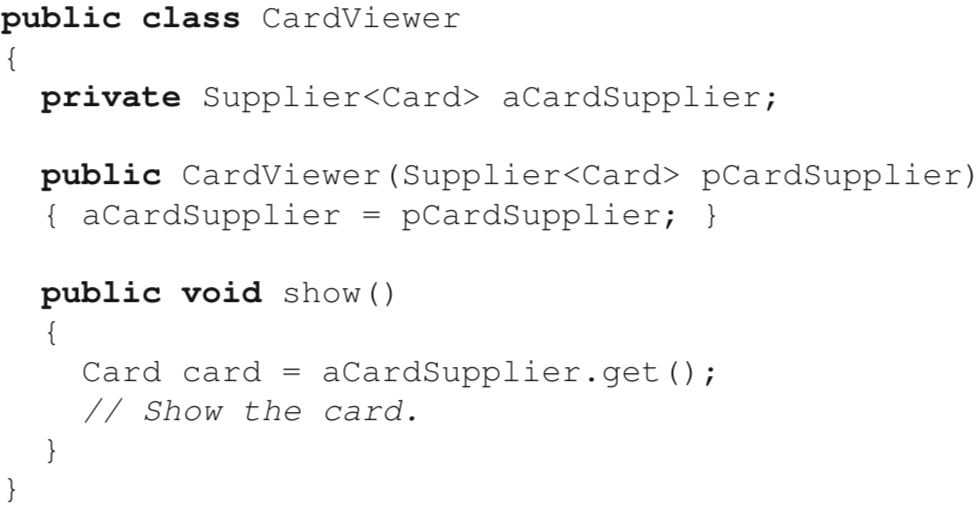
OBSERVER(pull)

Define a new interface CardSupplier with

a single method getCard(),

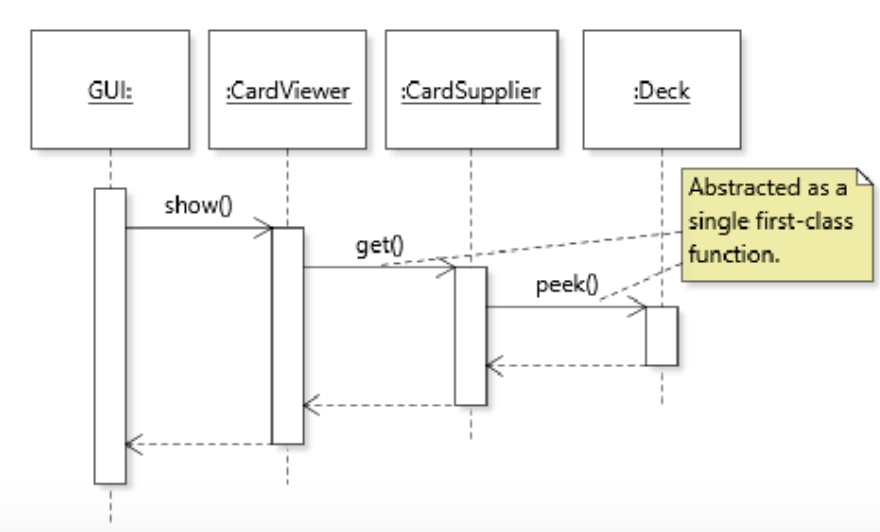
and make implement this interface.

Interface Supplier<T> defines a single method get() that returns a value of type T. Thus, we can avoid defining a new interface for this purpose, and use Java’s Supplier<T>:



With this code, we can initialize a CardViewer with any first-class function that takes no argument and returns a value of type Card. In a code context where we want to create an instance of CardViewer and we already have an instance of Deck, we could initialize the CardViewer with its supplier as follows:





Advantages

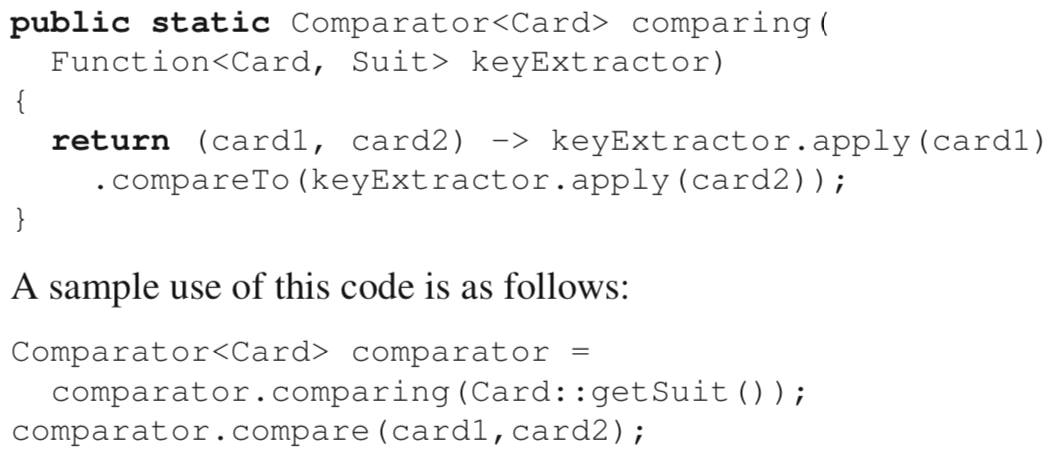
1. When passing supplier functions to objects instead of actual data, objects can access the information on-demand instead of storing it as a piece of data they have to manage.

2. To narrow the interfaces that objects use to exchange data. Passing supplier functions to objects can take the interface segregation principle to its optimal point, by allowing objects to request precisely the information they need through a set of supplier interfaces, as opposed to aggregating interfaces that may include services they do not need.

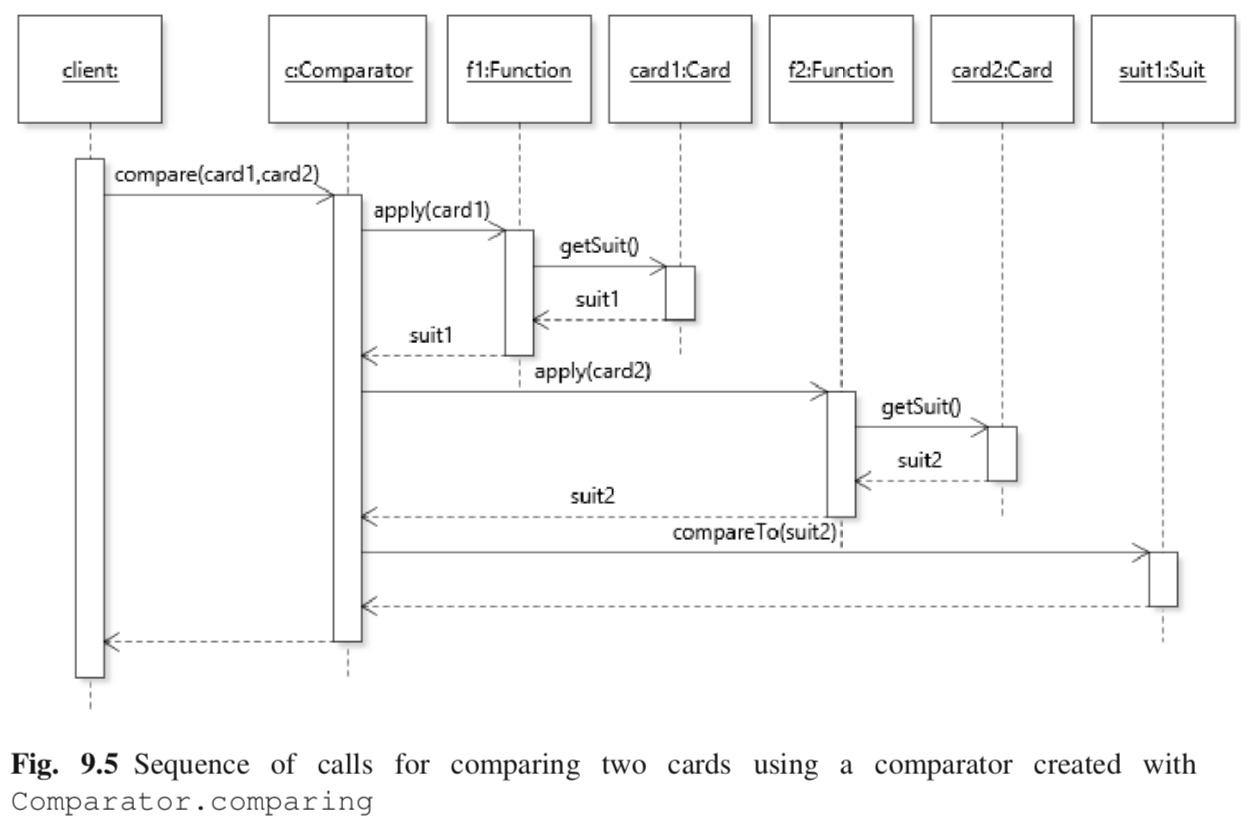
Comparator.Comparing(...)

If we want to build a comparator that compares Card objects based on their suit, we do:

The argument is an instance of Fucntion<Card,Suit>, a functional interface whose method (apply) takes an argument of type Card and returns a reference to an object of type Suit. This means that that the code that implements method comparing will have way to extract an instance of Suit from an instance of Card whenever necessary in the logic of the method’s implementation.



When comparing is called, it creates a new function object that binds Card:: getSuit to keyExtractor, but without calling either apply or its delegate getSuit. This indirection is necessary because comparing uses the supplier function as a building block when creating a new function, as opposed to simply using the supplier (as in our example above). When method compare is called, only then is apply called, this time twice, once for each card. Because apply redirects to getSuit, at that point the suit value is obtained from the card and used in the comparison.



A final note concerning method comparing is that, in contrast to the CardViewer example, its “supplier” function keyExtractor actually takes an argument. In this sense, the function is more a data “extractor” than a pure supplier. Notwithstanding this distinction, the mechanism involved is the same: comparing will obtain a reference to a function that it can call whenever it needs some data to work with.

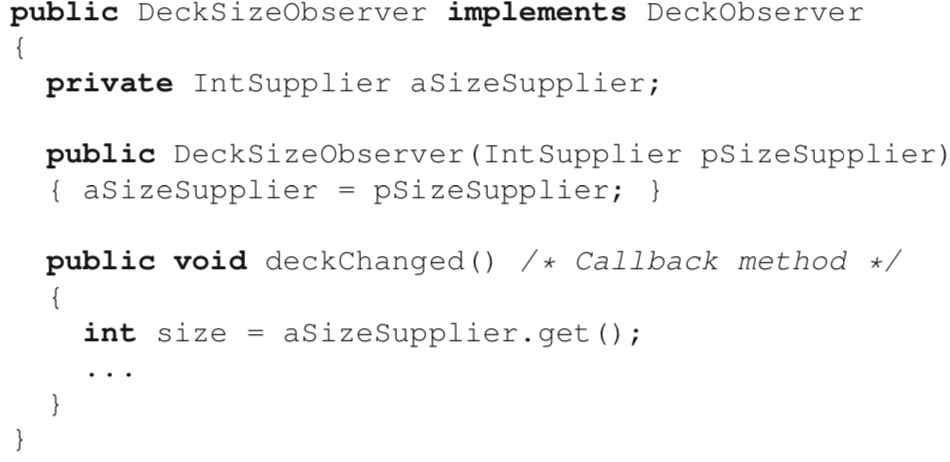
**Supplier Functions and the OBSERVER Pattern**

When implementing the OBSERVER pattern, supplier functions can offer a flexible way to support the pull data-flow strategy.

e.g. One concrete observer needs the size of the deck, one observer needs the top card, and one observer needs to iterate through all cards in the deck. If we define an interface DeckData



then all observers will get access to the three pieces of information, even though they only need one each.



IntSupplier is a functional interface of the Java library that defines a single int get() method. With this design we can thus connect the observer to an instance of Deck stored in a deck variable as such:

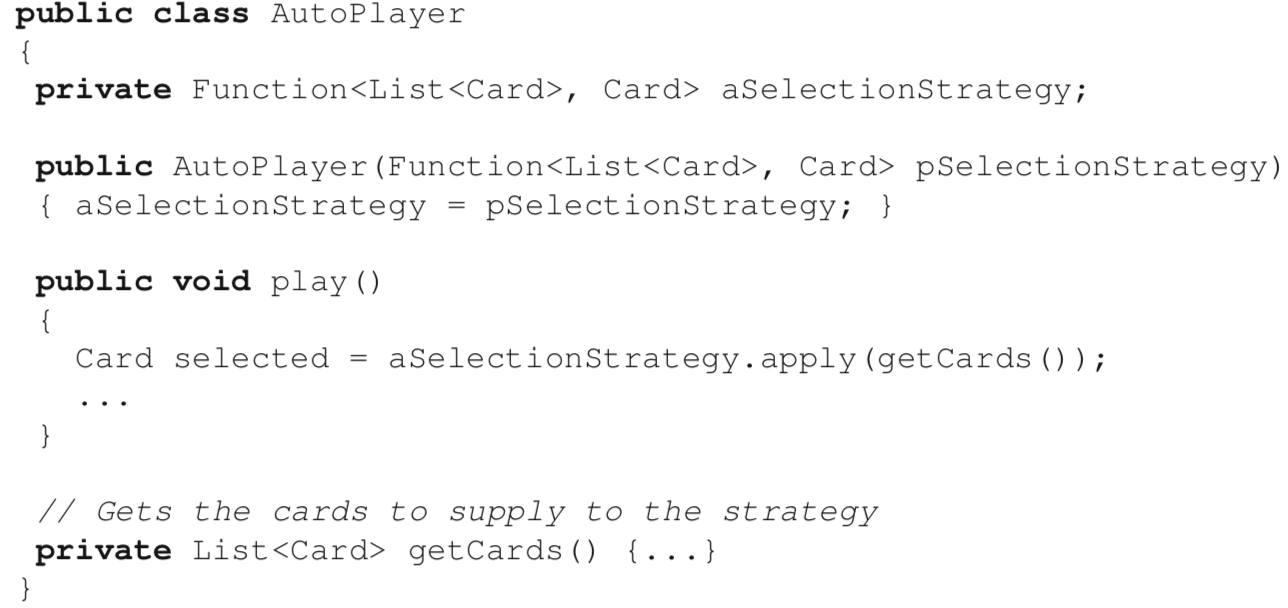


9.5 First-Class Functions and Design Patterns

*Instead of creating objects of different classes and enabling polymorphism through a common supertype, we can define families of functions whose type is compatible and invoke them interchangeably.*

**Functional-Style STRATEGY**

Let us consider a hypothetical context where client code may want to use different strategies for selecting a card in a list. Then, strategies are simply implementations of method apply of interface Function<List<Card>, Card>, which becomes the abstract strategy. In our case, method apply takes as input a list of cards and returns a single card.

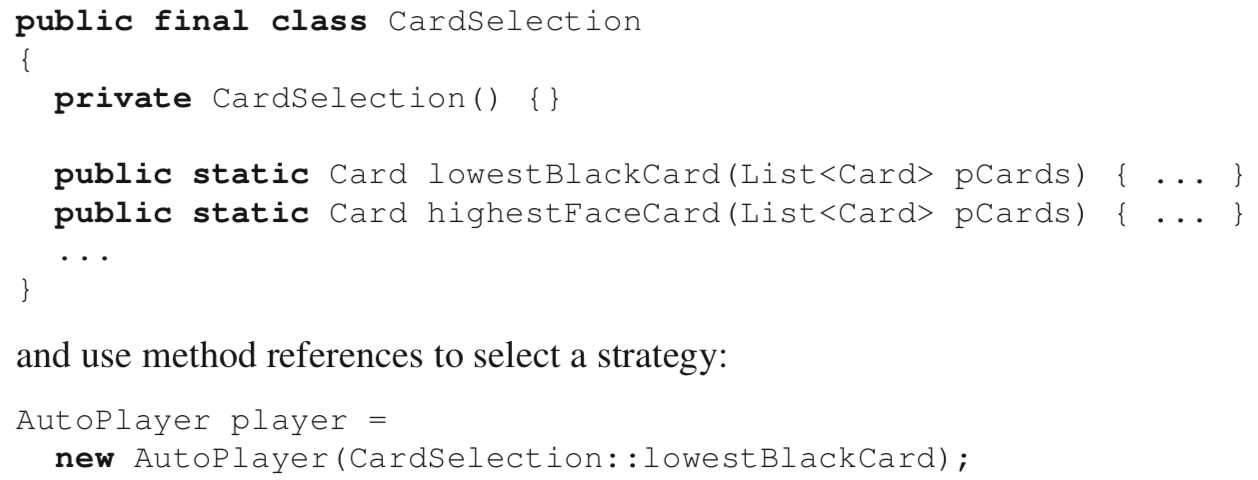


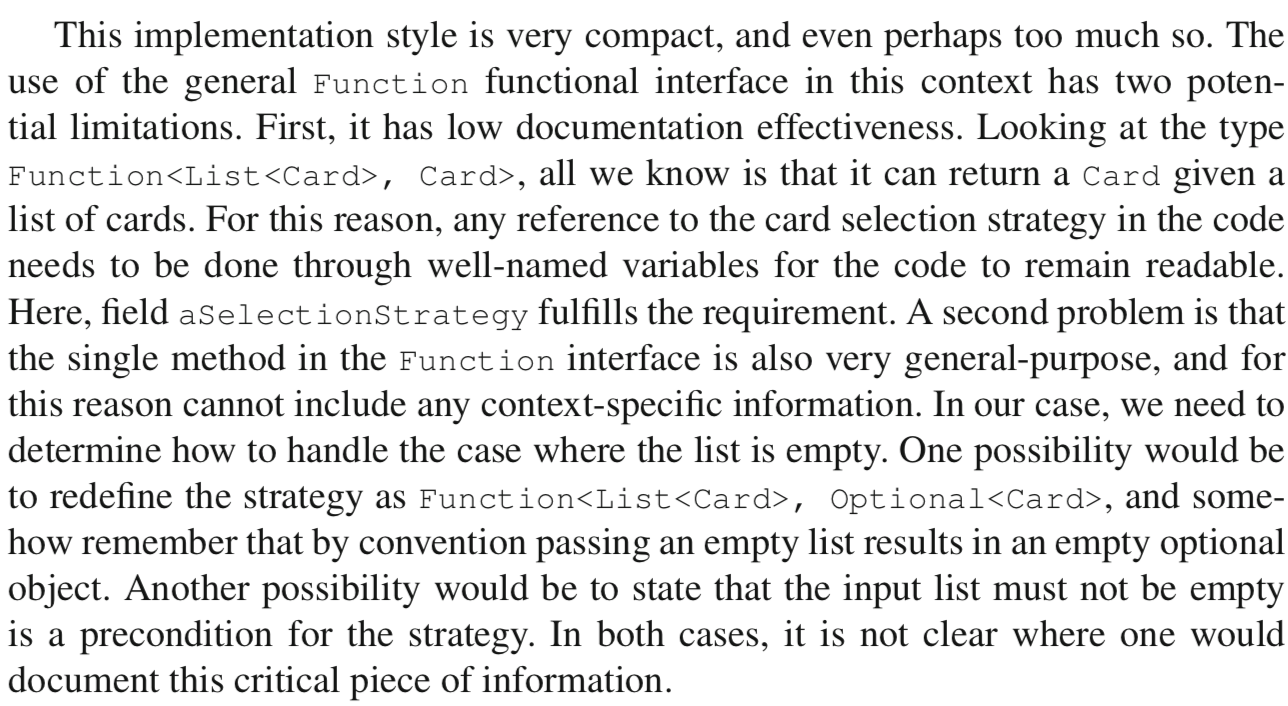
Because the strategy is modeled as a first-class function, defining it involves defining the behavior of this function at any convenient point in the code.

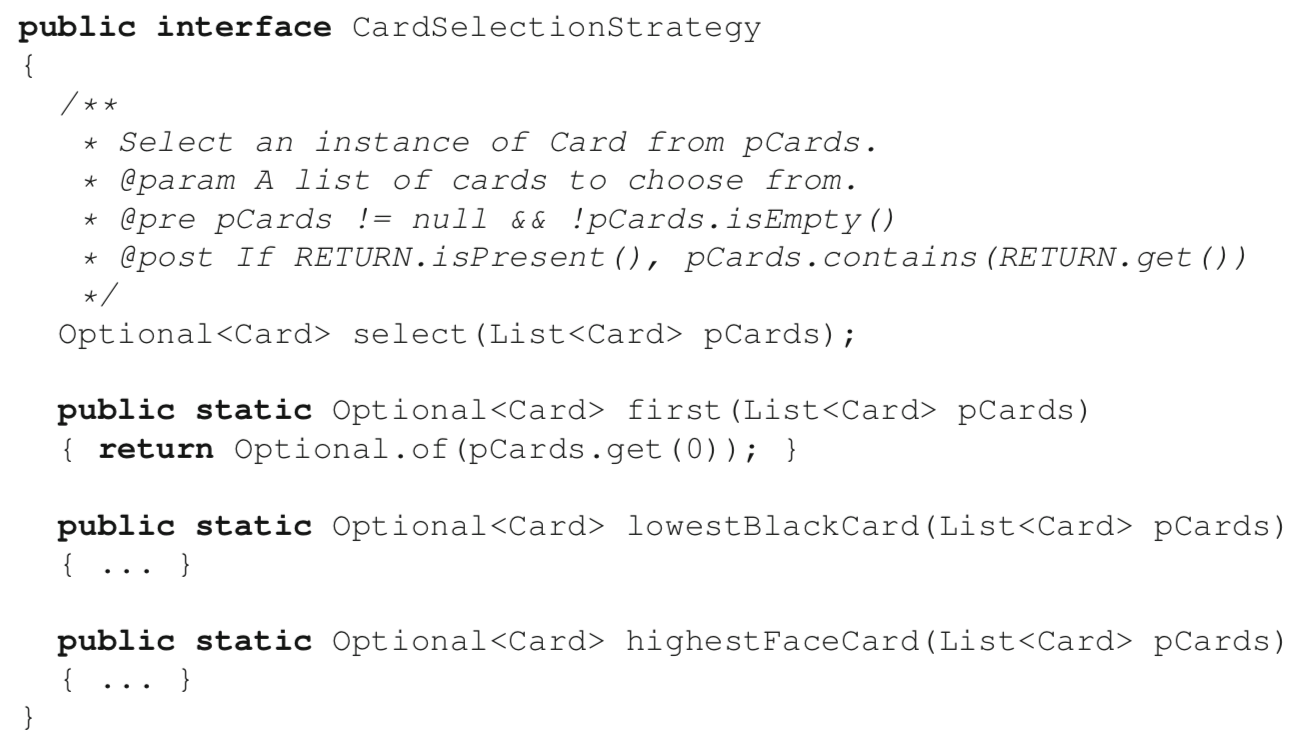
* One option could be to define it on the fly at the location where the instance of AutoPlayer is created. For example, a strategy to always select the first card would be:



* For more elaborate strategies, another option could be to define a collection of common strategies in a utility class:







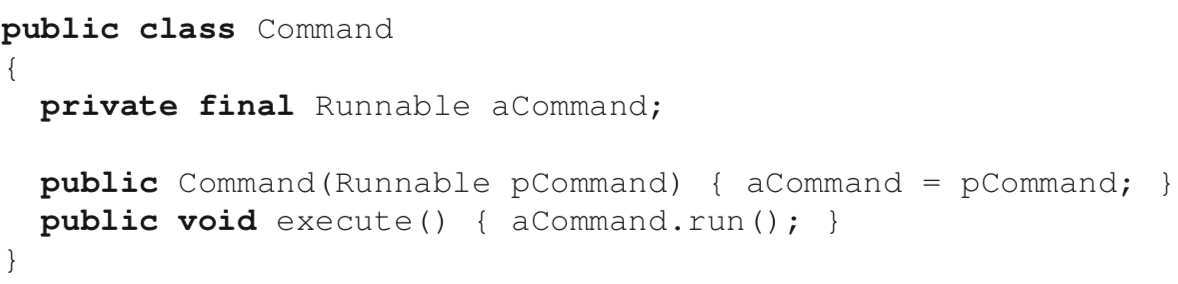
uses design by contract to guard against the case of selecting from an empty list;

uses the Optional type to guard against the case where a strategy yields no card.

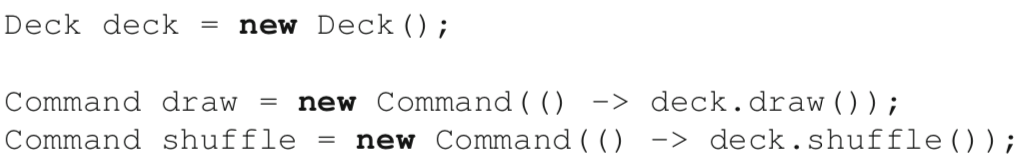
**Functional-Style COMMAND**

Similarly to the STRATEGY pattern, the central idea of the COMMAND pattern is one of behavior parameterization. In the classic implementation of the pattern, the concrete behavior of a command is achieved by defining different classes with a common supertype, where each class represents a type of command (see Section 6.8). *With first-class functions, we can parameterize the behavior of a single concrete command class by defining a field that stores a function that is called when executing the command.*

Let us consider again the original example of COMMAND introduced in Section 6.8, to support a number of commands to modify the state of a Deck object. In this case, we could apply the pattern by designing a single command class that serves as both the abstract and concrete command:



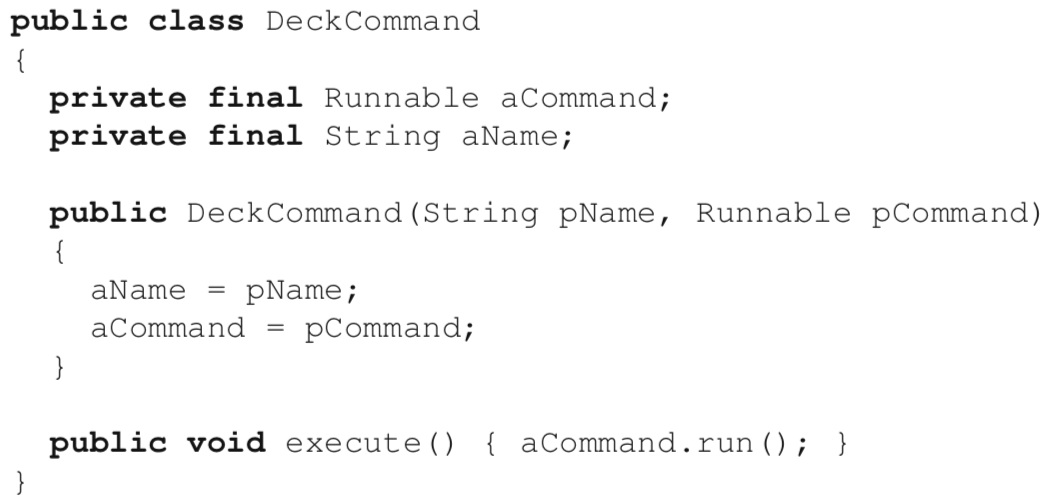
This minimalist application of the pattern uses the library functional interface Runnable, which declares a single method run() that takes no parameter and returns no value.



Although workable, this design suffers from limitations caused by the minimalism of the implementation [similar to what STRATEGY faces]

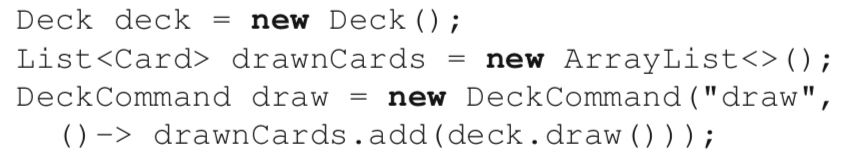
* Command objects are not self-documenting. Once a command object is created, we need to rely on external means (such as variable names) to keep track of it;
* The Runnable interface is very general: we cannot arbitrarily add constraints to the declaration of its run() method, for example to prevent executing commands on an empty deck;
* If some commands require an outflow of data, we need a mechanism to support this data flow.

First, we can make commands more self-documenting by storing the name of the command in the command object and naming the command class more specifically:



Second, we can define our own functional interface and use it instead of Runnable to qualify the type of the first-class function that represents the command. However, here because the first-class functions that will represent commands can only be called by being first passed as values to the constructor of DeckCommand, we have a more obvious single point for documenting the preconditions for executing the function. It is possible to make our design constraint explicit by appropriately documenting the constructor of the DeckCommand class, its execute() method, or both.

Finally, the choice of how to support collecting the result of a command is, just as in the object-oriented version of the pattern, an open-ended decision that should be informed by the design context in which the pattern is applied. The more flexible option is to store the result in a data structure accessed through a closure, e.g.,



Another option is to replace Runnable with a functional interface that returns a value (e.g., of type Card) and stores this value in the command object, as in the original application of the pattern.