Chapter 9
Functional Design

This chapter covers:

Concepts and Principles: Behavior parameterization, first-class functions, higher-order functions, functional programming;
Programming Mechanisms: Functional interfaces, lambda expressions, method references, streams;
Design Techniques: Behavior composition, functions as data sources, interface segregation with first-class functions, pipelining, map–reduce;
Patterns and Antipatterns: STRATEGY, COMMAND.

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.

```
List<Card> cards = ...;
Collections.sort(cards, new Comparator<Card>()
{
  public int compare(Card pCard1, Card pCard2)
  { return pCard1.getRank().compareTo(pCard2.getRank()); }
});
```

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:

```
public class Card
{
  private static int compareByRank(Card pCard1, Card pCard2)
  { return pCard1.getRank().compareTo(pCard2.getRank()); }
}
```

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

```
Collections.sort(cards, Card::compareByRank);
```

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:

```
public interface Filter
{ boolean accept(Card pCard); }

Filter blackCardFilter = new Filter()
{
  public boolean accept(Card pCard)
  { return pCard.getSuit().getColor() == Suit.Color.BLACK; }
};
```

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)`.

```
Predicate<Card> blackCardFilter = new Predicate<Card>()
  {
    public boolean test(Card pCard)
    { return pCard.getSuit().getColor() == Suit.Color.BLACK; }
  };
```

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
    - a single expression (e.g., a == 1).

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

```
Predicate<Card> blackCardFilter =
  (Card card) -> card.getSuit().getColor() == Suit.Color.BLACK;
```

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;}).

```
Predicate<Card> blackCards =
  (Card card) ->
    { return card.getSuit().getColor() == Suit.Color.BLACK; };
```

<u>Compiler checks:</u>
- 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`:

```
Predicate<Card> blackCardFilter =
  (card) -> card.getSuit().getColor() == Suit.Color.BLACK;
```

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

```
Predicate<Card> blackCardFilter =
  card -> card.getSuit().getColor() == Suit.Color.BLACK;
```

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:

```
Predicate<Card> blackCardFilter =
  (Card c) -> c.getSuit().getColor() == Suit.Color.BLACK;
```

How to call a lambda expression

```
Deck deck = ...
Predicate<Card> blackCardFilter =
  card -> card.getSuit().getColor() == Suit.Color.BLACK;
int total = 0;
for( Card card : deck )
{
  if( blackCardFilter.test(card) )
  { total++; }
}
```

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:

```
ArrayList<Card> cards = ...
cards.removeIf(
  card -> card.getSuit().getColor() == Suit.Color.BLACK);
```

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

```
public final class Card
{
  public boolean hasBlackSuit()
  { return aSuit.getColor() == Color.BLACK; }

  public boolean hasRedSuit()
  { return aSuit.getColor() == Color.RED; }
}
cards.removeIf(Card::hasBlackSuit);
```

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)

```
cards.removeIf(new Predicate<Card>()
{
  public boolean test(Card card)
  { return card.hasBlackSuit(); }
});
```

(2) Static method

```
public final class CardUtils
{
  public static boolean hasBlackSuit(Card pCard)
  { return pCard.getSuit().getColor() != Color.BLACK; }
}
cards.removeIf(CardUtils::hasBlackSuit);
```

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

```
cards.removeIf(new Predicate<Card>()
{
  public boolean test(Card card)
  { return CardUtils.hasBlackSuit(card); }
});
```

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

```
public class Card
{
  public static Comparator<Card> bySuitComparator()
  {
    return (card1, card2) ->
      card1.getSuit().compareTo(card2.getSuit());
  }
}
```

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

```
public static Comparator<Card> bySuitThenRankComparator()
{
  return (card1, card2) ->
  {
    if( card1.getSuit() == card2.getSuit() )
    { return card1.getRank().compareTo(card2.getRank()); }
    else
    { return card1.getSuit().compareTo(card2.getSuit()); }
  };
}
```

Method 2: compose two single-level comparator

```
public static Comparator<Card> byRankComparator()
{ return (card1, card2) ->
    card1.getRank().compareTo(card2.getRank()); }

public static Comparator<Card> bySuitComparator()
{ return (card1, card2) ->
    card1.getSuit().compareTo(card2.getSuit()); }

public static Comparator<Card> byRankThenSuitComparator()
{
  return (card1, card2) ->
  { if( byRankComparator().compare(card1, card2) == 0 )
    { return bySuitComparator().compare(card1, card2 ); }
    else
    { return byRankComparator().compare(card1, card2 ); }
  };
}

public static Comparator<Card> bySuitThenRankComparator()
{
  return (card1, card2) ->
  { if( bySuitComparator().compare(card1, card2) == 0 )
    { return byRankComparator().compare(card1, card2 ); }
    else
    { return bySuitComparator().compare(card1, card2 ); }
  };
}
```

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:

```
public static Comparator<Card> byRankComparator()
{ return Comparator.comparing(card -> card.getRank()); }
```

`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)

```
public static Comparator<Card> byRankThenSuitComparator()
{ return byRankComparator().thenComparing(bySuitComparator()); }
```

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

```
public static Comparator<Card> bySuitReversedThenRankComparator()
{ return bySuitComparator()
      .reversed()
      .thenComparing(byRankComparator()); }
```

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.

```
List<Card> cards = ...
cards.sort(Comparator.comparing((Card card) -> card.getSuit()))
     .reversed()
     .thenComparing(Comparator.comparing(
       (Card card) -> card.getRank()))
         .reversed()));
```

Improvement

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

```
import static java.util.Comparator.comparing;
```

This allows us to remove the qualification of the static method `comparing`:

```
cards.sort(comparing((Card card) -> card.getSuit())
     .reversed()
     .thenComparing(comparing((Card card) -> card.getRank())
       .reversed()));
```

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

```
cards.sort(comparing(Card::getSuit)
    .reversed()
    .thenComparing(comparing(Card::getRank)
      .reversed()));
```

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.

```
cards.sort(comparing(Card::getSuit)
  .thenComparing(Card::getRank).reversed());
```

Other helper methods in java.util.function

```
Predicate<Card> blackCardFilter =
  card -> card.getSuit().getColor() == Suit.Color.BLACK;
```

If we want only red cards, we can do simply:[6]

```
Predicate<Card> redCardFilter = blackCardFilter.negate();
```

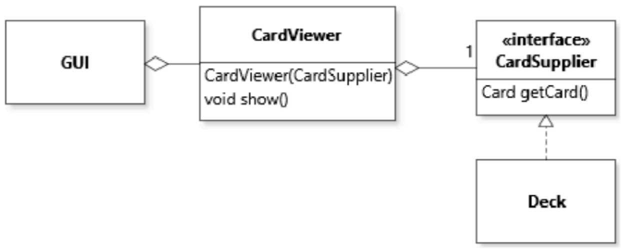
9.4 Using Functions as Suppliers of Data

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>:

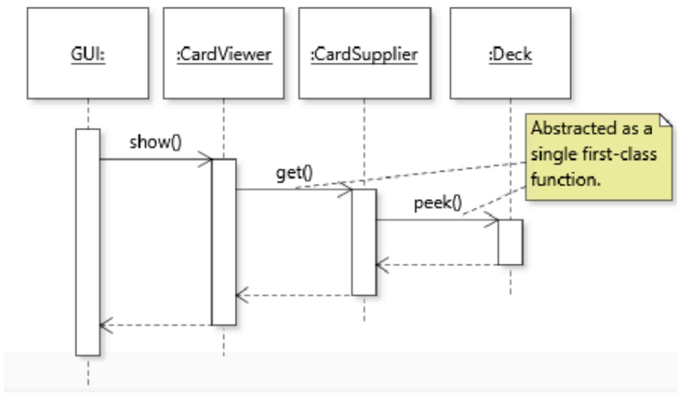
```
public class CardViewer
{
  private Supplier<Card> aCardSupplier;

  public CardViewer(Supplier<Card> pCardSupplier)
  { aCardSupplier = pCardSupplier; }

  public void show()
  {
    Card card = aCardSupplier.get();
    // Show the card.
  }
}
```

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:

```
 Deck deck = ...
 CardViewer viewer = new CardViewer( () -> deck.peek() );
```



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:

```
Comparator<Card> bySuit = Comparator.comparing(Card::getSuit);
```

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.

```
public static Comparator<Card> comparing(
  Function<Card, Suit> keyExtractor)
{
  return (card1, card2) -> keyExtractor.apply(card1)
    .compareTo(keyExtractor.apply(card2));
}
```

A sample use of this code is as follows:

```
Comparator<Card> comparator =
  comparator.comparing(Card::getSuit());
comparator.compare(card1,card2);
```

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.

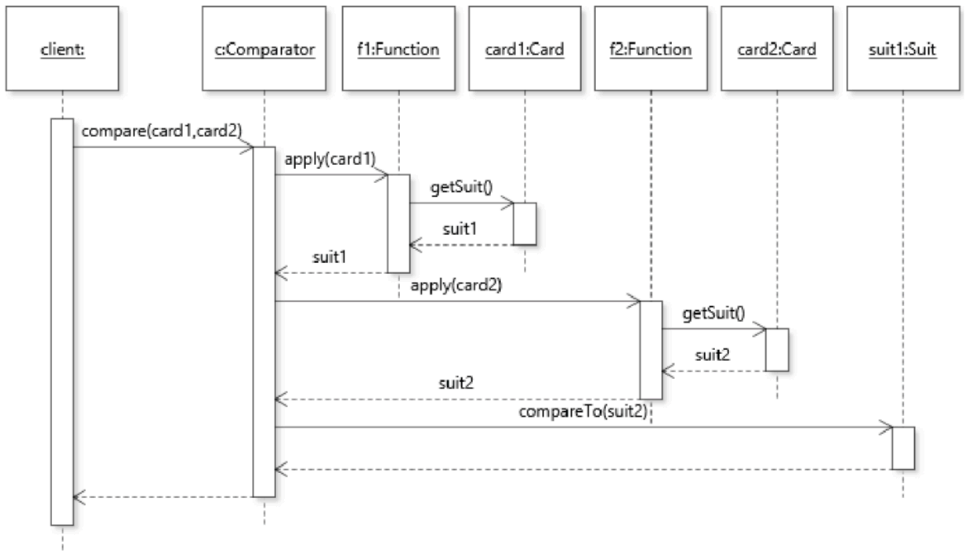


Fig. 9.5 Sequence of calls for comparing two cards using a comparator created with `Comparator.comparing`

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`

```
public interface DeckData extends Iterable<Card>
{
  int size();
  Card peek();
}
```

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

```
public DeckSizeObserver implements DeckObserver
{
  private IntSupplier aSizeSupplier;

  public DeckSizeObserver(IntSupplier pSizeSupplier)
  { aSizeSupplier = pSizeSupplier; }

  public void deckChanged() /* Callback method */
  {
    int size = aSizeSupplier.get();
    ...
  }
}
```

`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:

```
DeckSizeObserver observer = new DeckSizeObserver(()->deck.size());
```

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.

```
public class AutoPlayer
{
  private Function<List<Card>, Card> aSelectionStrategy;

  public AutoPlayer(Function<List<Card>, Card> pSelectionStrategy)
  { aSelectionStrategy = pSelectionStrategy; }

  public void play()
  {
    Card selected = aSelectionStrategy.apply(getCards());
    ...
  }

  // Gets the cards to supply to the strategy
  private List<Card> getCards() {...}
}
```

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:

  ```
  AutoPlayer player = new AutoPlayer(cards -> cards.get(0));
  ```

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

  ```
  public final class CardSelection
  {
    private CardSelection() {}

    public static Card lowestBlackCard(List<Card> pCards) { ... }
    public static Card highestFaceCard(List<Card> pCards) { ... }
    ...
  }
  ```

  and use method references to select a strategy:

  ```
  AutoPlayer player =
    new AutoPlayer(CardSelection::lowestBlackCard);
  ```

This implementation style is very compact, and even perhaps too much so. The use of the general `Function` functional interface in this context has two potential limitations. First, it has low documentation effectiveness. Looking at the type `Function<List<Card>, Card>`, all we know is that it can return a `Card` given a list of cards. For this reason, any reference to the card selection strategy in the code needs to be done through well-named variables for the code to remain readable. Here, field `aSelectionStrategy` fulfills the requirement. A second problem is that the single method in the `Function` interface is also very general-purpose, and for this reason cannot include any context-specific information. In our case, we need to determine how to handle the case where the list is empty. One possibility would be to redefine the strategy as `Function<List<Card>, Optional<Card>`, and somehow remember that by convention passing an empty list results in an empty optional object. Another possibility would be to state that the input list must not be empty is a precondition for the strategy. In both cases, it is not clear where one would document this critical piece of information.

```
public interface CardSelectionStrategy
{
  /**
   * Select an instance of Card from pCards.
   * @param A list of cards to choose from.
   * @pre pCards != null && !pCards.isEmpty()
   * @post If RETURN.isPresent(), pCards.contains(RETURN.get())
   */
  Optional<Card> select(List<Card> pCards);

  public static Optional<Card> first(List<Card> pCards)
  { return Optional.of(pCards.get(0)); }

  public static Optional<Card> lowestBlackCard(List<Card> pCards)
  { ... }

  public static Optional<Card> highestFaceCard(List<Card> pCards)
  { ... }
}
```

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:

```
public class Command
{
  private final Runnable aCommand;

  public Command(Runnable pCommand) { aCommand = pCommand; }
  public void execute() { aCommand.run(); }
}
```

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.

```
Deck deck = new Deck();

Command draw = new Command(() -> deck.draw());
Command shuffle = new Command(() -> deck.shuffle());
```

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:

```
public class DeckCommand
{
  private final Runnable aCommand;
  private final String aName;

  public DeckCommand(String pName, Runnable pCommand)
  {
    aName = pName;
    aCommand = pCommand;
  }

  public void execute() { aCommand.run(); }
}
```

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.,

```
Deck deck = new Deck();
List<Card> drawnCards = new ArrayList<>();
DeckCommand draw = new DeckCommand("draw",
  ()-> drawnCards.add(deck.draw()));
```

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.

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.

9.6 Functional-Style Data Processing

Up to now, the design ideas presented in this chapter involve including functional elements into an otherwise object-oriented design. In some cases, the design context motivates solutions that have a much stronger flavor of functional-style programming. One scenario where functional-style programming shines is to structure code responsible for processing a collection of data. In a way, most of what software does is to process data, so here we can tighten the definition and consider that functional-style programming provides good support for design problems that involve applying transformations to large sequences of data elements. An example of data processing that meets this definition is counting the number of acronyms in a body of text. In this case, the input is a sequence of words, and the transformations are to filter the input for acronyms, and then to compute the total number of instances found.

Functional-style design is a good match for this type of data processing because it naturally calls for the use of behavior parameterization and higher-order functions. Higher-order functions implement the general data-processing strategies, which are then parameterized for a particular context with first-class functions. In the text processing example above, the general strategy is to check whether each input element (a word) matches a certain predicate (acronym or not). Although the general strategy of filtering over a predicate is likely to apply to many different problems, the predicate itself (acronym detection) is specific to the particular design context. In other cases we might want to write code that detects short words, proper nouns, etc. This idiom can be illustrated by the statement:

```
data.higherOrderFunction(firstClassFunction);
```

Applied to our current example, this would mean:

```
listOfWords.filter(isAcronym);
```

Functional-style data processing is a major topic in software design. This section provides a basic overview of the main concepts and techniques that underlie this design style, and how to realize them in Java.

Data as a Stream

The main concept that enables functional-style data processing in Java and similar technologies is that of a *stream*. Simply stated, a stream is a sequence of data elements, a bit like a collection. However, the major conceptual difference between a stream and a collection is that a collection represents a *store* of data whereas a stream represents a *flow* of data. This distinction is similar to the difference between storing music as a file vs. playing music via an on-line streaming service. For software design, the distinction between collections and streams has many practical implications:

- Elements in a collection have to exist before they are added to the collection, but elements in a stream can be computed on-demand.
- Although collections can only store a finite number of elements, streams can technically be infinite. For example, although it is not possible to define a list that contains all the even numbers, it is possible to create a stream that produces this data.
- Collections can be traversed multiple times, but the traversal code is located outside the collection, for example in a **for** loop or iterator class. In contrast, streams can be traversed once: their elements are *consumed* as part of the traversal. However, the traversal code is hidden within the higher-order functions provided by the stream's interface.
- Streams are amenable to being parallelized, mainly because the traversal of their elements is hidden as part of the stream abstraction.

An additional, more pragmatic, difference relates to the evolution of the Java language. Collection classes (`List`, `Set`, etc.) were released before the language had explicit support for first-class functions, so collections provide almost no support for higher-order functions.[8] In contrast, Java 8 provides support for first-class functions (in the form of method references and lambda expressions) and includes a powerful `Stream` API designed to support functional-style design. The remainder of this section shows how to design functional-style data processing in Java using the `Stream` API.

In Java, the simplest way to obtain a stream is to call the `stream()` method on an instance of a collection class. For example, let us modify the design of the `Deck` class illustrated in Figure 9.6 to support streaming cards using an instance of `Deck` as supplier of data.

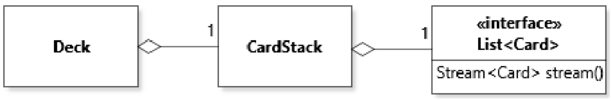


Fig. 9.6 Initial design of the `Deck` class, which can be modified to add support for streaming `Card` instances from a `Deck` instance

Making an instance of `Deck` able to stream cards simply requires propagating the `Stream` instance obtainable from the list to the interface of `Deck`:

[8] The notable exception is the default method `forEach` available on the `Iterable<T>` interface since Java 8.

```
public class CardStack implements Iterable<Card>
{
  private final List<Card> aCards;
  public Stream<Card>   stream() { return aCards.stream(); }
  ...
}

public class Deck implements CardSource , Iterable<Card>
{
  private CardStack aCards;
  public Stream<Card> stream() { return aCards.stream(); }
  ...
}
```

By themselves, streams already support many useful (non-higher-order) functions. For example, we can easily count the elements in the stream:

```
Stream<Card> cards = new Deck().stream();
long total = cards.count();
```

Streams also support operations that take as input a stream as its implicit parameter and output a different stream, a process known as *pipelining*. For example, the `sorted` method returns the elements of the original stream in sorted order. Because method `sorted()` requires the instances in the stream to be subtypes of `Comparable`, the code below assumes the version of class `Card` used implements `Comparable`:

```
Stream<Card> sortedCards = cards.stream().sorted();
```

Pipelining also makes it easy to combine operations on streams. For example, method `limit(int max)` returns up to `max` elements from the stream. To obtain the first ten cards in sorted order, we can thus write:

```
Stream<Card> sortedCards = cards.stream().sorted().limit(10);
```

It is also possible to combine multiple streams. For example, to assemble all the cards from two decks and sort them, we can do:

```
Stream<Card> cards =
  Stream.concat(new Deck().stream(), new Deck().stream());
```

To revert to a single deck one option is to remove the duplicates using `distinct()`:

```
Stream<Card> withDuplicates =
  Stream.concat(new Deck().stream(), new Deck().stream());
Stream<Card> withoutDuplicates = withDuplicates.distinct();
```

Applying Higher-Order Functions to Streams

The main way that streams support functional-style programming is that they define a number of higher-order functions. The essential higher-order function for streams is `forEach`, which applies an input function to all elements of the stream. For example, to print all cards in a stream in a functional way, we could do the following:

```
new Deck().stream().forEach(card -> System.out.println(card));
```

The method `forEach` takes an argument of type `Consumer<? super T>`, which means we can supply it a reference to a function that defines a single parameter of type `Card` (or any supertype of `Card`). In the example above this reference is supplied in the form of a lambda expression.[9] Because `forEach` is not guaranteed to respect the order in which elements are encountered in the stream, a second version, `forEachOrdered`, can be used if ordering is important. Because it does not return a stream, the outcome of the `forEach` function (either variant) cannot be further transformed as part of a pipeline. Generally speaking, stream functions that do not produce a stream of results that can be further processed as part of a pipeline are called *terminal* operations. Another example of a terminal operation on streams is the `count` function seen above.

Other types of terminal higher-order functions that can be applied to streams include searching functions such as `allMatch`, `anyMatch`, and `noneMatch`, which take as argument a predicate on the stream element type and return a Boolean value that indicates respectively whether all, any, or none of the elements in the stream evaluate the predicate to **`true`**. For example, to determine if all cards in a list are in the clubs suit, we would do:

```
List<Card> cards = ...;
boolean allClubs = cards.stream()
  .allMatch(card -> card.getSuit() == Suit.CLUBS );
```

Filtering Streams

The `sorted()` stream function, mentioned above, shows how we can define *intermediate* operations to create a pipeline of transformations on a stream. An intermediate operation thus has a stream as an implicit argument, and returns a stream. An important function in this pipelining process is the `filter` method, which takes a `Predicate` and returns a stream that consists of all the elements of the original stream for which the predicate evaluates to true. For example, assuming we want to count the face cards in a list of cards:

```
long numberOfFaceCards = cards.stream()
  .filter(card -> card.getRank().ordinal() >=
    Rank.JACK.ordinal()).count();
```

To leverage the benefits of both object-orientation and functional-style programming, predicates such as the one above are best captured as instance methods:

```
public final class Card
{
  public boolean isFaceCard()
  { return getRank().ordinal() >= Rank.JACK.ordinal(); }
```

[9] An equivalent expression would be to pass the method reference `System.out::println` directly, but this uses a reference to a method applied to a specific instance, a technique that is outside the scope of this chapter.

This allows us to use method references and make the functional code self-explanatory:

```
long numberOfFaceCards = cards.stream()
  .filter(Card::isFaceCard)
  .count();
```

At first glance, capturing predicates in dedicated methods may seem like an obstacle to the creation of compound predicates. For example, what if we want to count only face cards in the clubs suit? Do we have to revert to our original lambda expression?

```
long result = cards.stream()
  .filter(card -> card.getRank().ordinal() >= Rank.JACK.ordinal()
    && card.getSuit()==Suit.CLUBS).count();
```

A key insight to avoid ugly code like this is to observe that filters, being an intermediate operation, can also be pipelined:

```
long result = cards.stream()
  .filter(Card::isFaceCard)
  .filter(card -> card.getSuit() == Suit.CLUBS)
  .count();
```

At this point, it should become apparent that our functional-style code is starting to look much more like a set of high-level rules for processing data than a set of instructions telling a program how to operate on inputs. Indeed, one major advantage of writing data processing code in a functional style is that the result is more declarative than imperative, and thus better conveys the intent behind the code. Here, for example, a single glance at the statement shows that we wish to only consider face cards, then further restrict the data to only consider cards in the clubs suit, and then finally count the data elements. It is also worth noting how the code is formatted, with each stream operation indented and prefixed with its period starting the visible part of a line. This coding style is a usual convention for formatting stream operations in Java. Its benefit is that it emphasizes the declarative nature of the code.

Mapping Data Elements

There are often situations in data processing where we need to transform all data elements in a stream into a derived value. In this case, we leverage the idea of *mapping* objects to their desired value. Here, the word “mapping” is employed in the mathematical sense synonymous to a function. For example, consider how the function that computes the square of a number x, usually denoted x^2, simply *maps* a number x to its square $x \cdot x$.

Many programming languages that support some form of functional-style processing provide a mechanism to apply a map (i.e., a function) to every element in a data collection. In Java the `Stream` class defines a `map` method that takes as input a parameter of type `Function<? super T, ? extends R>`. In other words, the argument to the `map` function is another function that takes as input an object of type `T` and returns an object of type `R`.[10] This means that the `map` function will transform a stream of objects of one type into an stream of objects of a different type.

As a simple example, let us consider a function that maps an instance of `Card` to an instance of an enumerated type `Color` that represents the color of the card's suit. We can apply this function systematically to all elements in a stream using the `map` method:

```
cards.stream().map(card -> card.getSuit().getColor() );
```

If this expression is evaluated on a shuffled deck, the resulting stream will be a random interleaving of the values `Color.BLACK` and `Color.RED`. Because the result is also a stream, we can pipeline the result of a mapping operation as for any other stream. For example, to count the number of black cards in a collection, we could write:

```
long result = cards.stream()
  .map(card -> card.getSuit().getColor() )
  .filter( color -> color == Color.BLACK )
  .count();
```

Although the same result can achieved more directly by using `filter` with a lambda expression that retrieves the card's color, the example above illustrates how we can use `map` to unpack an object and use only the part of the object that is of interest for a given computation.

Mapping, however, can accomplish much more than simply extracting data from an input element. Let us consider a second example, where we want to compute the score that a card represents. In some games, cards are assigned a point value that corresponds to their rank (e.g., three of clubs is worth three points), except for face cards which are all worth ten points. With a mapping process, we can convert a stream of card objects into a stream of integers that correspond to the score of each card in the original stream:

```
cards.stream()
  .map(card -> Math.min(10, card.getRank().ordinal() + 1));
```

The result of this expression will be a stream of `Integer` objects that represent the score of each card. As usual, whether to encapsulate the score computation in an instance method of class `Card` is a context-sensitive design decision. If the score value is used in multiple calling contexts, then it would make sense to add a method `getScore()` to the interface of class `Card`. Otherwise, the lambda expression will suffice.

When mapping to numerical values, as in this case, it is useful to know that the language provides specialized support for streams of numbers in the form of classes such as `IntStream` and `DoubleStream`. These types of streams work just like other streams, but they define additional operations that only make sense when processing numbers, such as summing the elements in the stream. To adapt our scoring example

[10] Technically, `T` or one of its supertypes, and `R` or one of its subtypes.

to get the total score, the code needs to explicitly map to an `IntStream`, and then call the `sum` terminal operation:

```
int total = cards.stream()
  .mapToInt(card -> Math.min(10, card.getRank().ordinal() + 1))
  .sum();
```

As an alternative and more declarative way to specify this computation, we could also do:

```
int total = cards.stream()
  .map(Card::getRank)
  .mapToInt(Rank::ordinal)
  .map(ordinal -> Math.min(10, ordinal + 1))
  .sum();
```

Reducing Streams

When processing streams of data, a common scenario is that we want not only to inspect each data element, but also do something with the data as a whole. Typically, this means either:

- Aggregating the effect of the operations into a single result. Terminal operations such as `count()` and `sum()` are good examples of data aggregation in that sense;
- Collecting the individual results of the operations into a stored data structure. In Java this would typically mean a `List` or similar collection type.

Although they seem different, both of these alternatives actually have in common that conceptually they represent *reducing* a stream to a single entity. In the second case, the entity may be a collection of many elements, but conceptually it is nevertheless a single, stored structure as opposed to a stream. The advantage of generalizing all types of data aggregation as a single high-level operation, reduction, introduces a clear distinction between intermediate operations, namely mapping,[11] and terminal operations, namely reducing. In fact programming systems where computation is expressed as a series of mapping operations followed by a reduction operation are commonly known as the map–reduce programming model. Although the term “map–reduce” is mostly used in the context of cluster computing, the basic model itself is directly applicable to functional-style programming with streams.

In Java, reduction is supported through various overloaded versions of the `reduce` method available in `Stream` classes. Implementing a reduction from scratch is a bit tricky, and the complete details are outside the scope of this book. However, the general idea is to provide the `reduce` function with an *accumulator* object that can incrementally update the reduced version of the input every time an element is encountered. For example, to implement the `sum` operation on an `IntStream` using the `reduce` method, the following code is used:

[11] We can consider that filtering is a type of mapping without loss of generality.

```
IntStream numbers = ...;
int sum = numbers.reduce(0, (a, b) -> a+b);
```

In practice, a summing reduction uses 0 as the base case and accumulates results by iteratively adding elements. The code below shows a simplified mock-up implementation of the `reduce` method for `IntStream`

```
public int reduce(int pBase, IntBinaryOperator operator)
{
  int result = pBase;
  Iterator<Integer> iterator = this.iterator();
  while( iterator.hasNext() )
  {
    int number = iterator.next();
    result = operator.applyAsInt(result, number);
  }
  return result;
}
```

Initially, the result of the reduction operation is set to the base provided, in our case 0. Then, for each element in the stream, the binary operator provided as input to `reduce` is applied using, as argument, the current result and the next element in the stream. In our case the binary operator was specified using the lambda expression `(a,b) -> a + b`. Thus, for each element in the stream, the value of the current reduction will be assigned to itself plus the value of the next element.

This being said, because most common reduction operations (`min`, `max`, `count`, `sum`, etc.) are directly supported by the stream classes, it is possible to get started with streams and actually get quite far without mastering the art of writing reductions.

Reductions that serve to accumulate data in a structure are a bit of a special case. Let us say we wish to collect all face cards in a list of cards in a separate list. One quick solution would be to use the `forEach` method to store the elements of the stream in the target list:

```
List<Card> result = new ArrayList<>();
cards.stream()
  .filter(Card::isFaceCard)
  .forEach(card -> result.add(card));
```

Although workable, this design loses some of the properties of declarative, functional-style expressions of a computation, because the first-class function that simulates the reduction is implemented using explicit list manipulation operations. As an alternative that supports a more functional style, the Java libraries provide helper methods to create a type of reduction called a *collector*. A collector is a reduction that accumulates elements into a collection. With a collector, the code above can be re-written as:

```
List<Card> result = cards.stream()
  .filter(Card::isFaceCard)
  .collect(Collectors.toList());
```