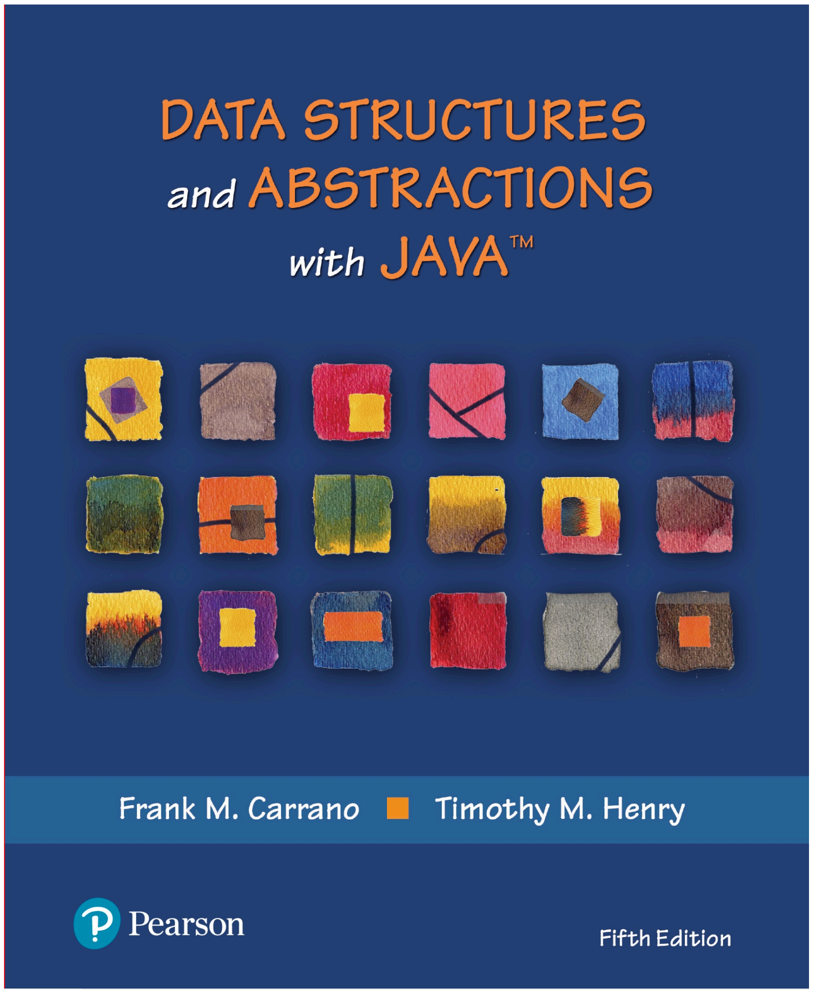
Solutions for Selected Exercises



Frank M. Carrano
University of Rhode Island

Timothy M. Henry
New England Institute of Technology

Charles Hoot
Oklahoma City University

Please send comments or errors to carrano@acm.org or timhenry@acm.org

Contents

(Click on any entry below to locate the solutions for that chapter.)

Prelude: Designing Classes

1. Consider the interface `NameInterface` defined in Segment P.13. We provided comments for only two of the methods. Write comments in `javadoc` style for each of the other methods.

```java
/** Sets the first and last names.
    @param firstName  A string that is the desired first name.
    @param lastName   A string that is the desired last name. */
public void setName(String firstName, String lastName);

/** Gets the full name.
    @return  A string containing the first and last names. */
public String getName();

/** Sets the first name.
    @param firstName  A string that is the desired first name. */
public void setFirst(String firstName);

/** Gets the first name.
    @return  A string containing the first name. */
public String getFirst();

/** Sets the last name.
    @param lastName  A string that is the desired last name. */
public void setLast(String lastName);

/** Gets the last name.
    @return  A string containing the last name. */
public String getLast();

/** Changes the last name of the given Name object to the last name of this Name object.
    @param aName  A given Name object whose last name is to be changed. */
public void giveLastNameTo(NameInterface aName);

/** Gets the full name.
    @return  A string containing the first and last names. */
public String toString();
```

2. Consider the class `Circle` and the interface `Circular`, as given in Segments P.16 and P17.
   a. Is the client or the method `setRadius` responsible for ensuring that the circle's radius is positive?
   b. Write a precondition and a postcondition for the method `setRadius`.
   c. Write comments for the method `setRadius` in a style suitable for `javadoc`.
   d. Revise the method `setRadius` and its precondition and postcondition to change the responsibility mentioned in your answer to Part *a*.

a. The client is responsible for guaranteeing that the argument to the setRadius method is positive.
b. Precondition: `newRadius` >= 0. Postcondition: The radius has been set to `newRadius`.
c.
```java
/** Sets the radius.
    @param newRadius  A non-negative real number. */
```

d. Precondition: `newRadius` is the radius. Postcondition: The radius has been set to `newRadius` if `newRadius` >= 0.

```java
/** Sets the radius.
    @param newRadius  A real number.
    @throws  ArithmeticException if newRadius < 0. */
public void setRadius(double newRadius) throws ArithmeticException
{
   if (newRadius < 0)
      throw new ArithmeticException("Radius was negative");
   else
      radius = newRadius;
} // end setRadius
```

3. Write a CRC card and a class diagram for a proposed class called `Counter`. An object of this class will be used to count things, so it will record a count that is a nonnegative whole number. Include methods to set the counter to a given integer, to increase the count by 1, and to decrease the count by 1. Also include a method that returns the current count as an integer, a method `toString` that returns the current count as a string suitable for display on the screen, and a method that tests whether the current count is zero.

```
Counter

Responsibilities
  Set the counter to a value
  Add 1 to the counter
  Subtract 1 from the counter
  Get the value of the counter as an integer
  Get the value of the counter as a string
  Test whether the counter is zero

Collaborations
```

```
Counter
-------------------------------------
-count: integer
-------------------------------------
+setCounter(theCount:integer): void
+incrementCount(): void
+decrementCount(): void
+getCurrentCount(): integer
+toString(): String
+isZero(): boolean
```

4. Suppose you want to design software for a restaurant. Give use cases for placing an order and settling the bill. Identify a list of possible classes. Pick two of these classes, and write CRC cards for them.

System: Orders
Use case: Place an Order
Actor: Waitress
Steps:
1. Waitress starts a new order.
2. The waitress enters a table number.
3. Waitress chooses a menu item and adds it to the order.
   a. If there are more items, return to step 3.
4. The order is forwarded to the kitchen.

System: Orders
Use case: Settle Bill
Actor: Cashier
Steps:
1. The cashier enters the order id.
2. The system displays the total.
3. The customer makes a payment to the cashier.
4. The system computes any change due.
5. The cashier gives the customer a receipt.

Possible classes for this system are: Restaurant, Waitress, Cashier, Menu, MenuItem, Order, OrderItem, and Payment.

Chapter 1: Bags

1. Specify each method of the class `PiggyBank`, as given in Listing 1-3, by stating the method's purpose; by describing its parameters; and by writing preconditions, postconditions, and a pseudocode version of its header. Then write a Java interface for these methods that includes `javadoc`-style comments.

Purpose: Adds a given coin to this piggy bank.
Parameter: aCoin - a given coin
Precondition: None.
Postcondition: Either the coin has been added to the bank and the method returns true,
or the method returns false because the coin could not be added to the bank.
`public boolean add(aCoin)`

Purpose: Removes a coin from this piggy bank.
Precondition: None.
Postcondition: The method returns either the removed coin or null in case the bank
was empty before the method began execution.
`public Coin remove()`

Purpose: Detects whether this piggy bank is empty.
Precondition: None.
Postcondition: The method returns either true if the bank is empty or
false if it is not empty.
`public boolean isEmpty()`

```java
/**
    An interface that describes the operations of a piggy bank.
    @author Frank M. Carrano
    @version 4.0
*/
public interface PiggyBankInterface
{
   /** Adds a given coin to this piggy bank.
       @param aCoin  A given coin.
       @return  Either true if the coin has been added to the bank,
                or false if it has not been added. */
   public boolean add(Coin aCoin);

   /** Removes a coin from this piggy bank.
       @return  Either true if a coin has been removed from the bank,
                or false if it has not been removed. */
   public Coin remove();

   /** Detects whether this piggy bank is empty.
       @return  Either true if the bank is empty, or false if it not empty. */
   public boolean isEmpty();
} // end PiggyBankInterface
```

2. Suppose that `groceryBag` is a bag filled to its capacity with 10 strings that name various groceries. Write Java statements that remove and count all occurrences of `"soup"` in `groceryBag`. Do not remove any other strings from the bag. Report the number of times that `"soup"` occurred in the bag. Accommodate the possibility that `groceryBag` does not contain any occurrence of `"soup"`.

```java
int soupCount = 0;
while (bag.remove("soup"))
   soupCount++;
System.out.println("Removed " + soupCount + " cans of soup.");
```

3. Given `groceryBag`, as described in Exercise 2, what effect does the operation `groceryBag.toArray()` have on `groceryBag`?

No effect; `groceryBag` is unchanged by the operation.

4. Given `groceryBag`, as described in Exercise 2, write some Java statements that create an array of the distinct strings that are in this bag. That is, if `"soup"` occurs three times in `groceryBag`, it should only appear once in your array. After you have finished creating this array, the contents of `groceryBag` should be unchanged.

```
Object[] items = groceryBag.toArray();
BagInterface<String> tempBag = new Bag<>(items.length);
for (Object anItem: items)
{
   String aString = anItem.toString();
   if (!tempBag.contains(aString))
      tempBag.add(aString);
} // end for
items = tempBag.toArray();
```

5. The *union* of two collections consists of their contents combined into a new collection. Add a method `union` to the interface `BagInterface` for the ADT bag that returns as a new bag the union of the bag receiving the call to the method and the bag that is the method's one argument. Include sufficient comments to fully specify the method.

   Note that the union of two bags might contain duplicate items. For example, if object *x* occurs five times in one bag and twice in another, the union of these bags contains *x* seven times. Specifically, suppose that `bag1` and `bag2` are `Bag` objects, where `Bag` implements `BagInterface`; `bag1` contains the `String` objects `a`, `b`, and `c`; and `bag2` contains the `String` objects `b`, `b`, `d`, and `e`. After the statement

```
BagInterface<String> everything = bag1.union(bag2);
```

executes, the bag `everything` contains the strings a, b, b, b, c, d, and e. Note that union does not affect the contents of `bag1` and `bag2`.

```
/** Creates a new bag that combines the contents of this bag and a
    second given bag without affecting the original two bags.
    @param anotherBag  The given bag.
    @return  A bag that is the union of the two bags. */
public BagInterface<T> union(BagInterface<T> anotherBag);
```

6. The *intersection* of two collections is a new collection of the entries that occur in both collections. That is, it contains the overlapping entries. Add a method `intersection` to the interface `BagInterface` for the ADT bag that returns as a new bag the intersection of the bag receiving the call to the method and the bag that is the method's one argument. Include sufficient comments to fully specify the method.

   Note that the intersection of two bags might contain duplicate items. For example, if object *x* occurs five times in one bag and twice in another, the intersection of these bags contains *x* twice. Specifically, suppose that `bag1` and `bag2` are `Bag` objects, where `Bag` implements `BagInterface`; `bag1` contains the `String` objects `a`, `b`, and `c`; and `bag2` contains the `String` objects `b`, `b`, `d`, and `e`. After the statement

```
BagInterface<String> commonItems = bag1.intersection(bag2);
```

executes, the bag `commonItems` contains only the string `b`. If `b` had occurred in `bag1` twice, `commonItems` would have contained two occurrences of `b`, since `bag2` also contains two occurrences of `b`. Note that `intersection` does not affect the contents of `bag1` and `bag2`.

```
/** Creates a new bag that contains those objects that occur in both this
    bag and a second given bag without affecting the original two bags.
    @param anotherBag  The given bag.
    @return  A bag that is the intersection of the two bags. */
public BagInterface<T> intersection(BagInterface<T> anotherBag);
```

7. The *difference* of two collections is a new collection of the entries that would be left in one collection after removing those that also occur in the second. Add a method `difference` to the interface `BagInterface` for the ADT bag that returns as a new bag the difference of the bag receiving the call to the method and the bag that is the method's one argument. Include sufficient comments to fully specify the method.

Note that the difference of two bags might contain duplicate items. For example, if object *x* occurs five times in one bag and twice in another, the difference of these bags contains *x* three times. Specifically, suppose that `bag1` and `bag2` are `Bag` objects, where `Bag` implements `BagInterface`; `bag1` contains the `String` objects `a`, `b`, and `c`; and `bag2` contains the `String` objects `b`, `b`, `d`, and `e`. After the statement

```
BagInterface leftOver1 = bag1.difference(bag2);
```

executes, the bag `leftOver1` contains the strings `a` and `c`. After the statement

```
BagInterface leftOver2 = bag2.difference(bag1);
```

executes, the bag `leftOver2` contains the strings `b`, `d`, and `e`. Note that `difference` does not affect the contents of `bag1` and `bag2`.

```
/** Creates a new bag of objects that would be left in this bag
    after removing those that also occur in a second given bag
    without affecting the original two bags.
    @param anotherBag  The given bag.
    @return  A bag that is the difference of the two bags. */
public BagInterface<T> difference(BagInterface<T> anotherBag);
```

8. Write code that accomplishes the following tasks: Consider two bags that can hold strings. One bag is named letters and contains several one-letter strings. The other bag is empty and is named vowels. One at a time, remove a string from letters. If the string contains a vowel, place it into the bag vowels; otherwise, discard the string. After you have checked all of the strings in letters, report the number of vowels in the bag vowels and the number of times each vowel appears in the bag.

```
BagInterface<String> allVowels = new Bag<>();
allVowels.add("a");
allVowels.add("e");
allVowels.add("i");
allVowels.add("o");
allVowels.add("u");
BagInterface<String> vowels = new Bag<>();

while (!letters.isEmpty())
{
   String aLetter = letters.remove();
   if (allVowels.contains(aLetter))
      vowels.add(aLetter);
} // end while

System.out.println("There are " + vowels.getCurrentSize() + " vowels in the bag.");
String[] vowelsArray = {"a", "e", "i", "o", "u"};
for (int index = 0; index < vowelsArray.length; index++)
{
   int count = vowels.getFrequencyOf(vowelsArray[index]);
   System.out.println(vowelsArray[index] + " occurs " + count + " times.");
} // end for
```

9. Write code that accomplishes the following tasks: Consider three bags that can hold strings. One bag is named `letters` and contains several one-letter strings. Another bag is named `vowels` and contains five strings, one for each vowel. The third bag is empty and is named `consonants`. One at a time, remove a string from `letters`. Check whether the string is in the bag `vowels`. If it is, discard the string. Otherwise, place it into the bag `consonants`. After you have checked all of the strings in `letters`, report the number of consonants in the bag `consonants` and the number of times each consonant appears in the bag.

```java
while (!letters.isEmpty())
{
   String aLetter = letters.remove();
   if (!vowels.contains(aLetter))
      consonants.add(aLetter);
} // end while

System.out.println("There are " + consonants.getCurrentSize() +
                   " consonants in the bag.");
final String[] CONSONANTS = {"a", "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
                             "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z"};
for (int index = 0; index < CONSONANTS.length; index++)
{
   int count = consonants.getFrequencyOf(CONSONANTS[index]);
   System.out.println(CONSONANTS[index] + " occurs " + count + " times.");
} // end for
```