Algorithms and Data Structures 1
CS 0445

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

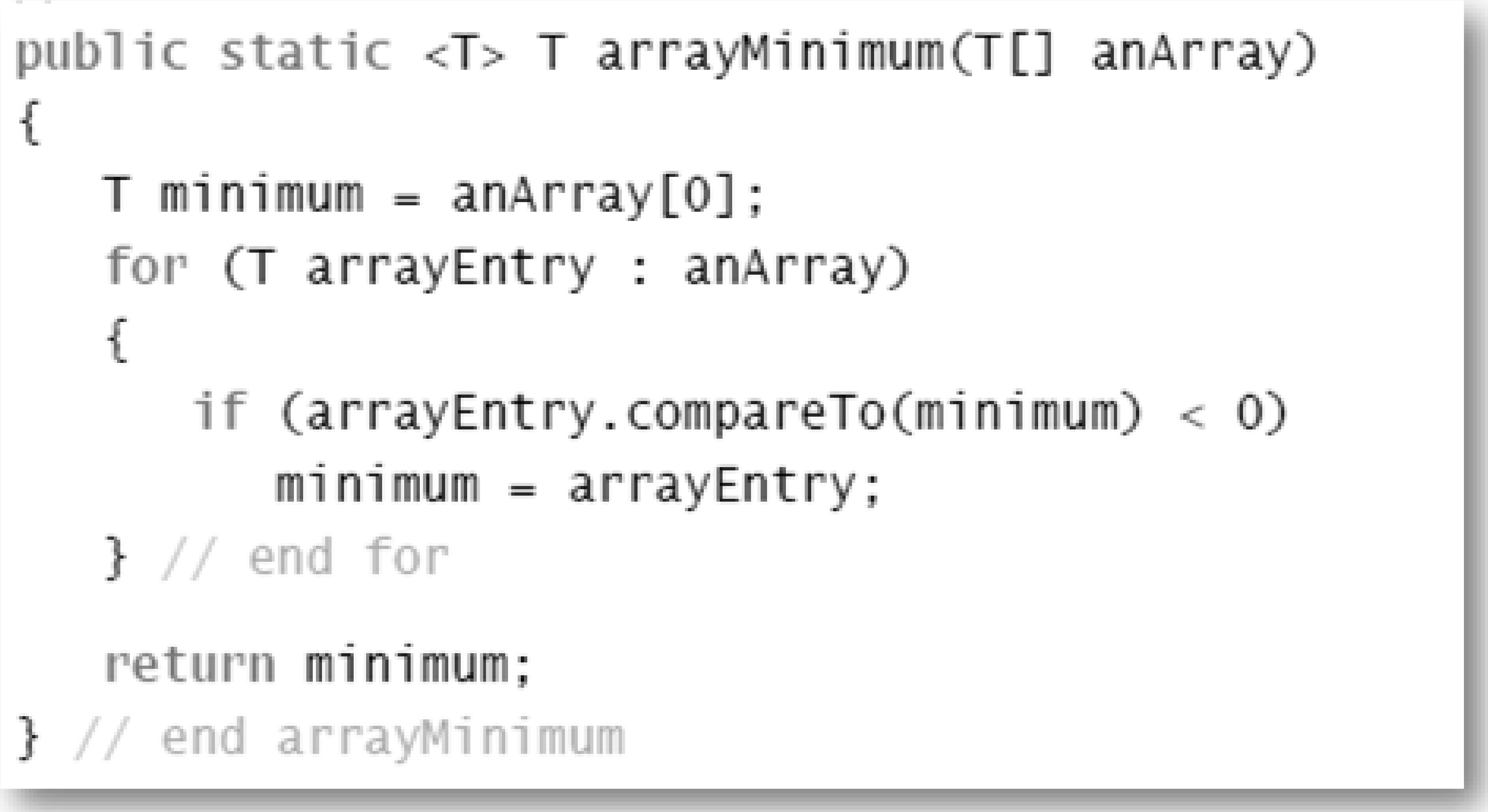
- Recitations start this week
- Homework 1 due this Friday
- Draft slides and handouts available on Canvas

Today’s Agenda

- Abstract Data Types
  - Generics
- File Operations
- ArrayBag

Bounded Type Parameters

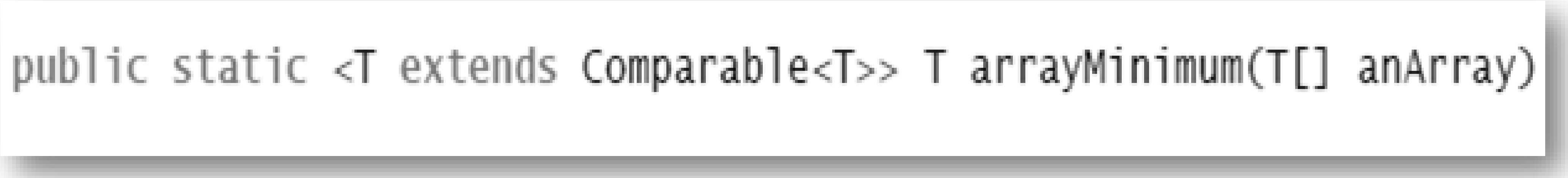
- Imagine that we want to write a static method that returns the smallest object in an array. Suppose that we wrote our method shown above



```
public static <T> T arrayMinimum(T[] anArray)
{
   T minimum = anArray[0];
   for (T arrayEntry : anArray)
   {
      if (arrayEntry.compareTo(minimum) < 0)
         minimum = arrayEntry;
   } // end for

   return minimum;
} // end arrayMinimum
```

Bounded Type Parameters

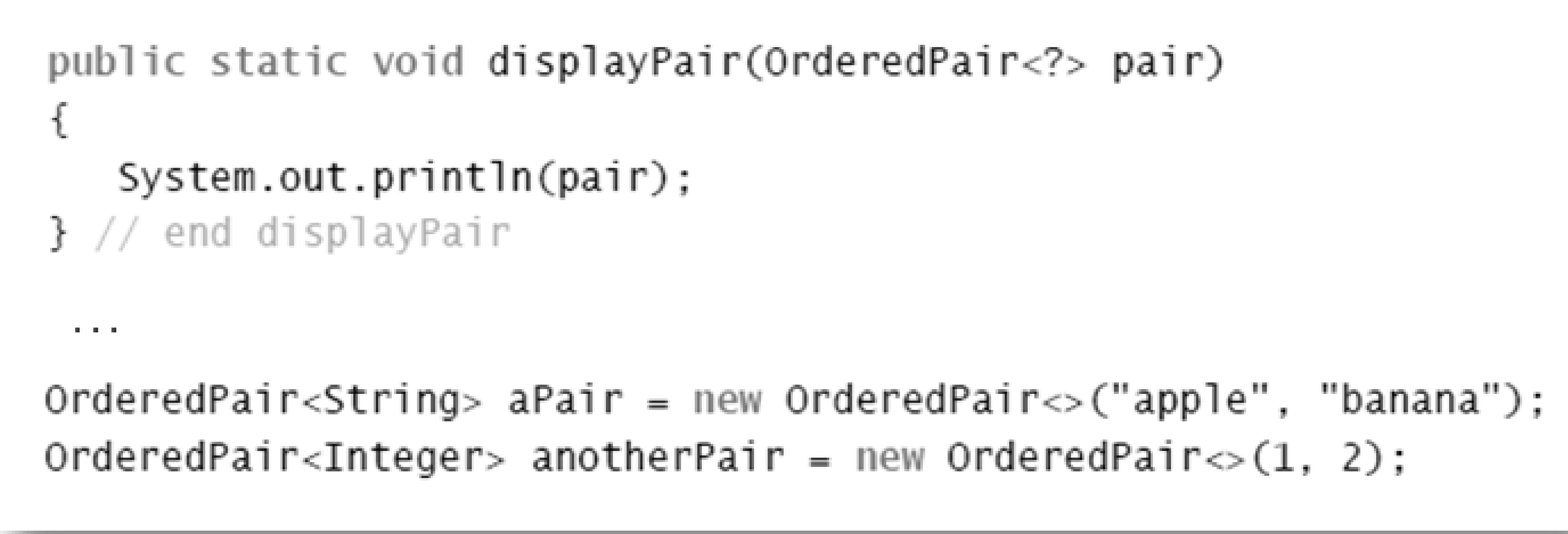
- Header really should be as shown



```
public static <T extends Comparable<T>> T arrayMinimum(T[] anArray)
```

Wildcards

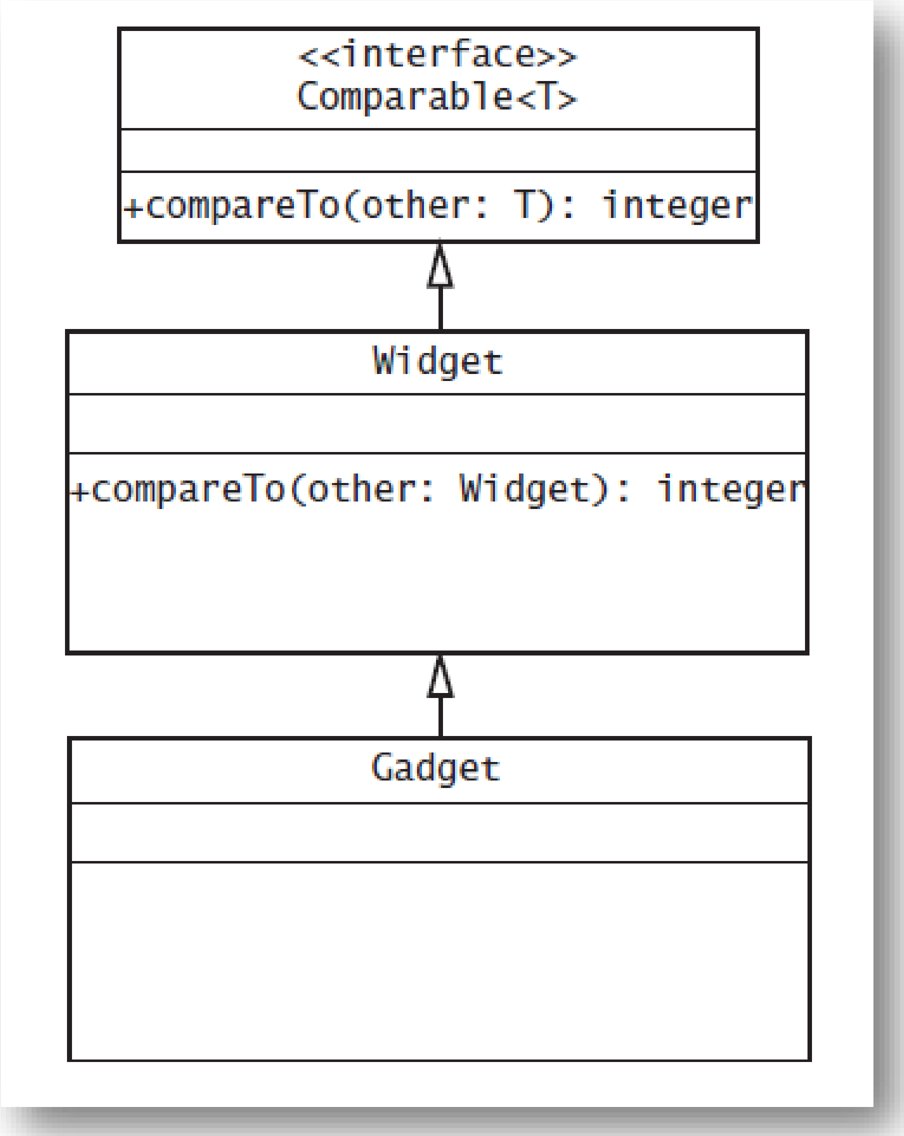
- Question mark, ?, is used to represent an unknown class type
    - Referred to as a wildcard
- Method **displayPair** will accept as an argument a pair of objects whose data type is any one class



```
public static void displayPair(OrderedPair<?> pair)
{
   System.out.println(pair);
} // end displayPair

 ...

OrderedPair<String> aPair = new OrderedPair<>("apple", "banana");
OrderedPair<Integer> anotherPair = new OrderedPair<>(1, 2);
```

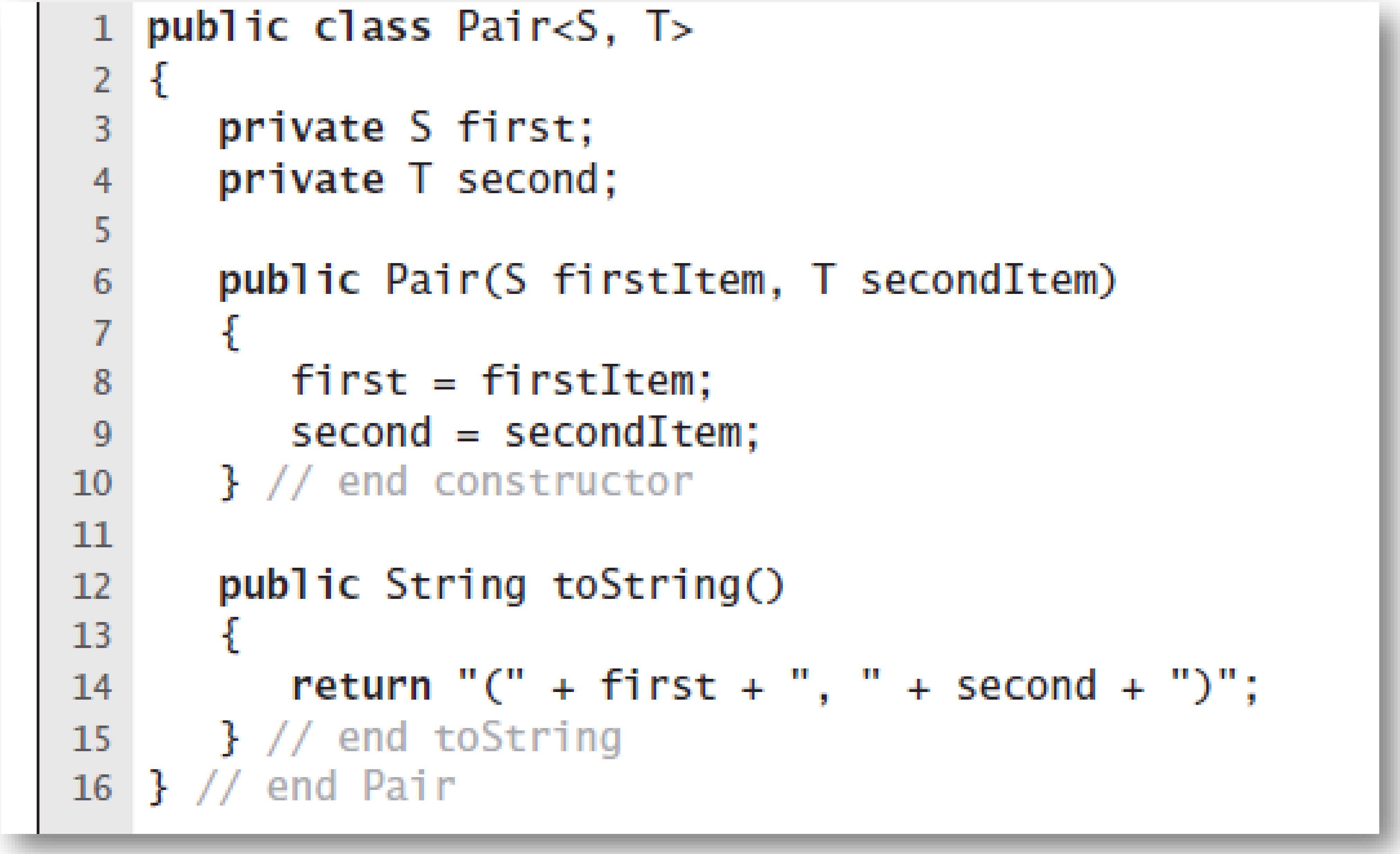
Bounded Wildcards

- The class **`Gadget`** is derived from the class **`Widget`**, which implements the interface **`Comparable`**
- **`UML diargam`**



<<interface>>
Comparable<T>
+compareTo(other: T): integer
Widget
+compareTo(other: Widget): integer
Gadget

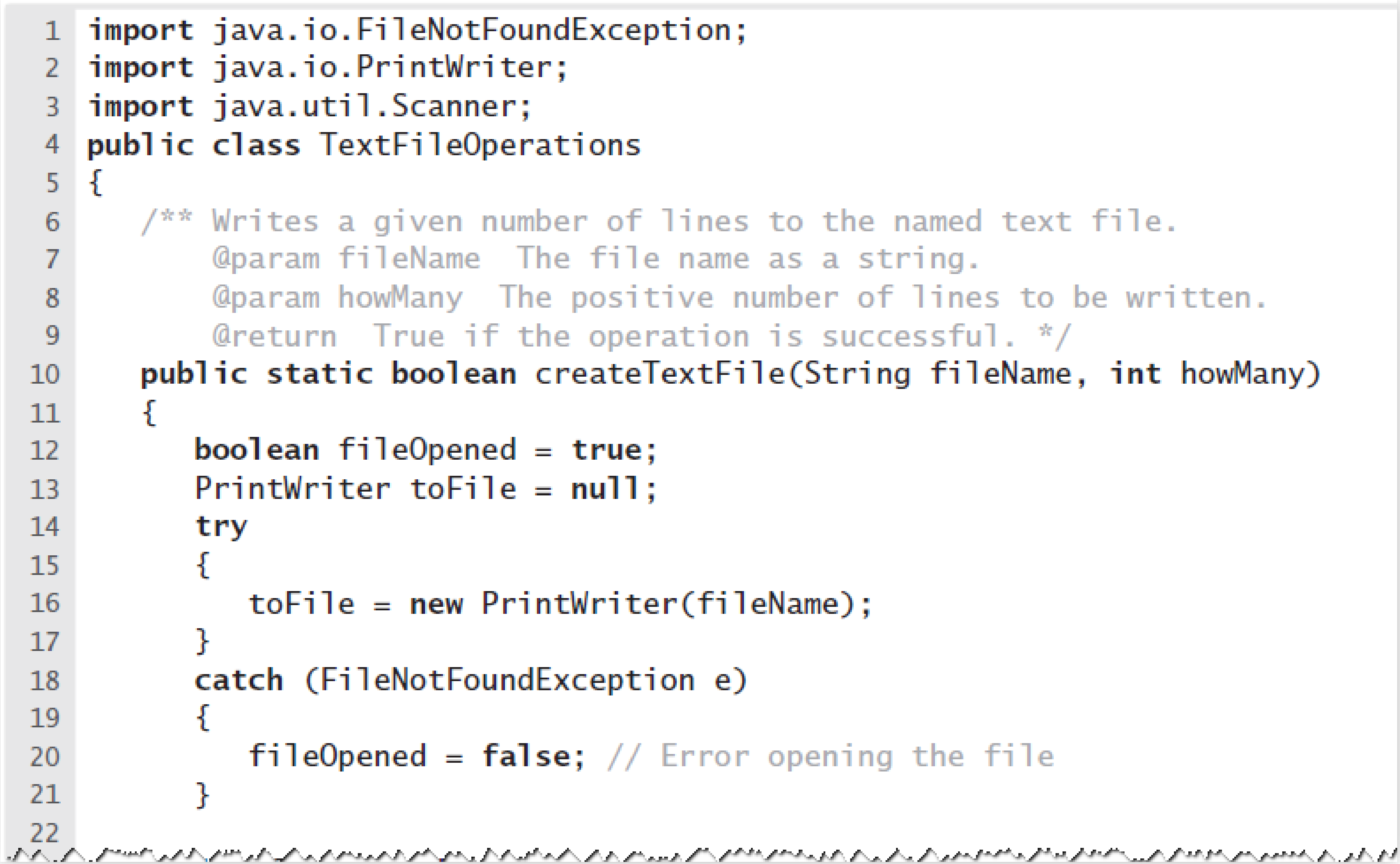
More Than One Generic Type



```java
public class Pair<S, T>
{
    private S first;
    private T second;

    public Pair(S firstItem, T secondItem)
    {
        first = firstItem;
        second = secondItem;
    } // end constructor

    public String toString()
    {
        return "(" + first + ", " + second + ")";
    } // end toString
} // end Pair
```

Writing to a Text File

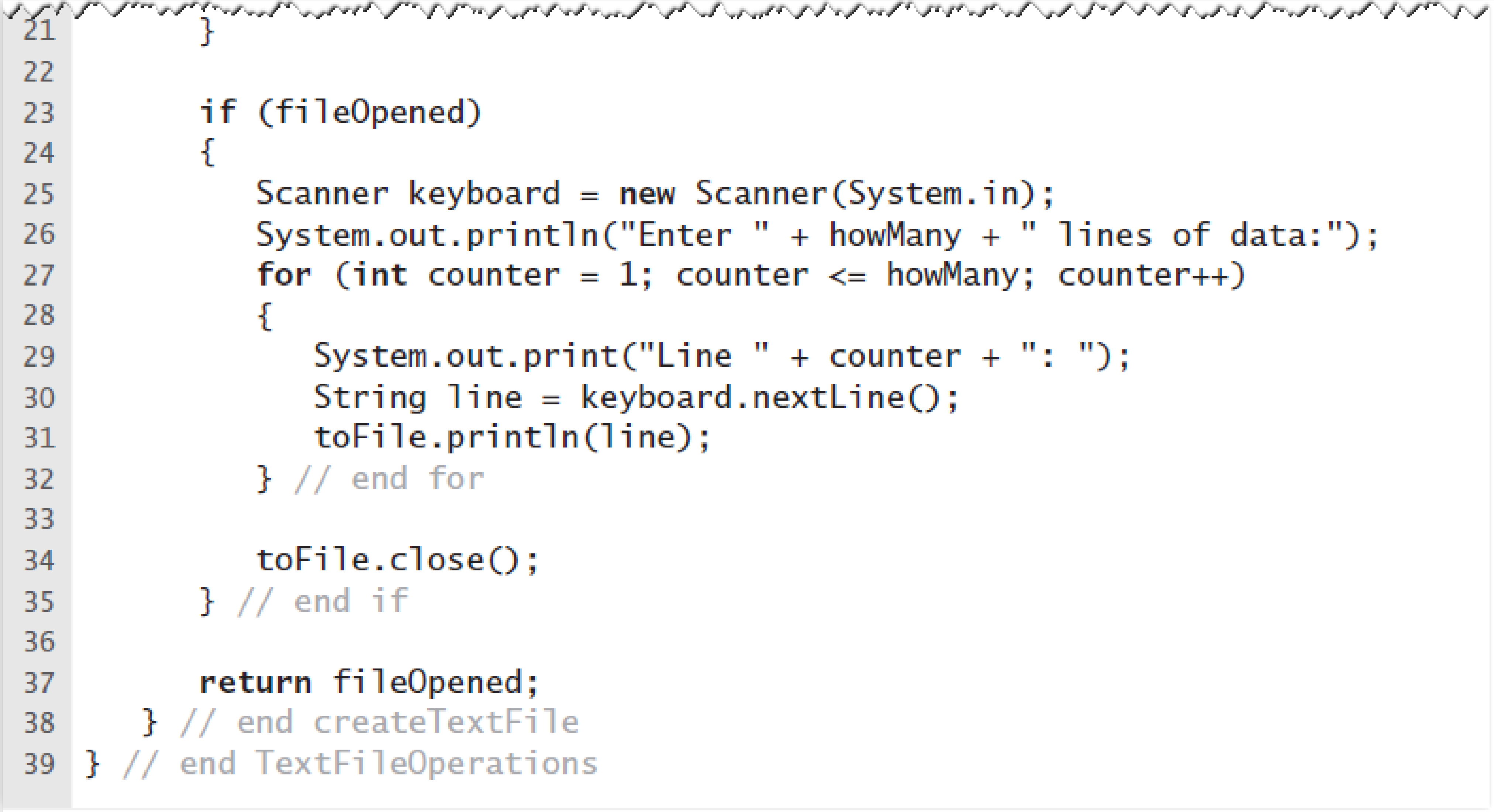
- **Using java.io.PrintWriter**



```
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;
public class TextFileOperations
{
   /** Writes a given number of lines to the named text file.
       @param fileName  The file name as a string.
       @param howMany  The positive number of lines to be written.
       @return  True if the operation is successful. */
   public static boolean createTextFile(String fileName, int howMany)
   {
      boolean fileOpened = true;
      PrintWriter toFile = null;
      try
      {
         toFile = new PrintWriter(fileName);
      }
      catch (FileNotFoundException e)
      {
         fileOpened = false; // Error opening the file
      }

```

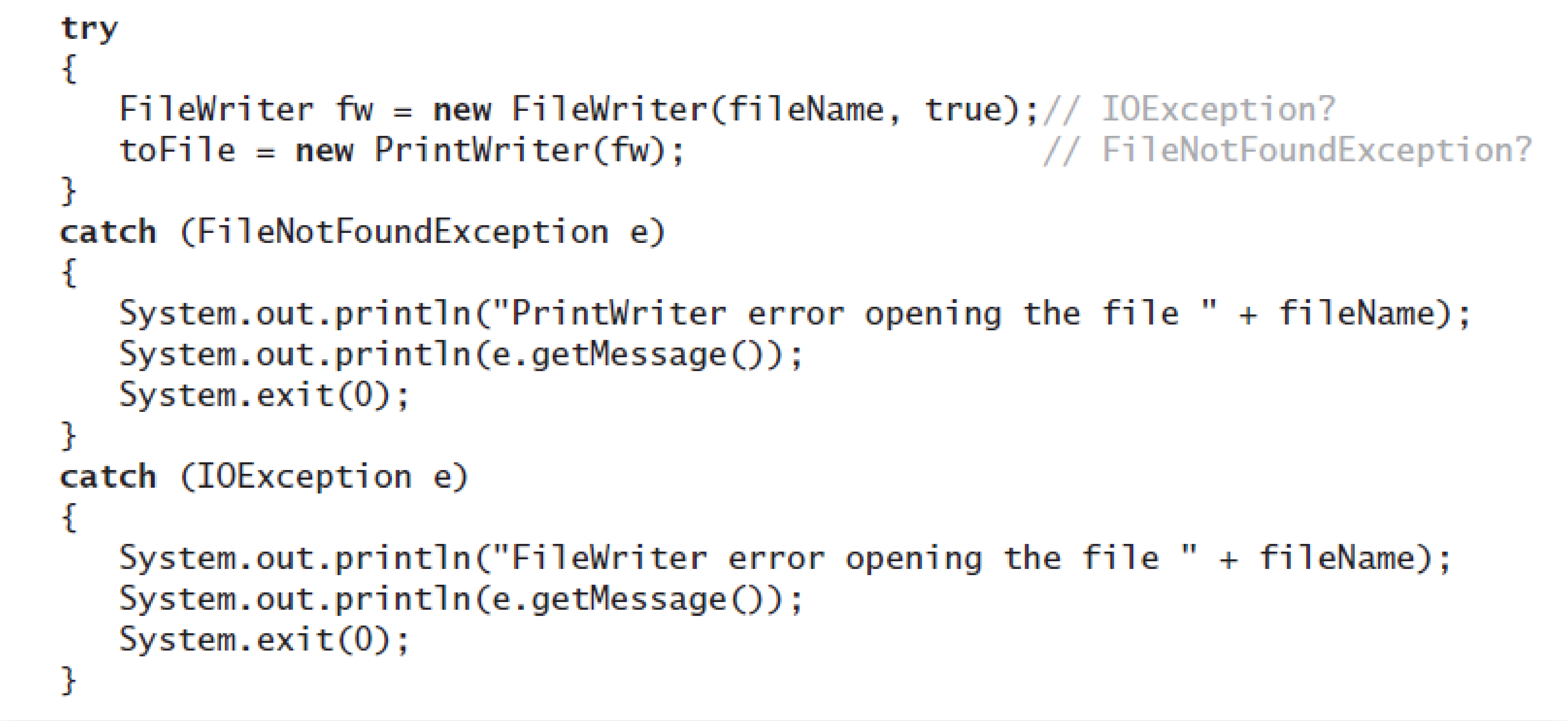
Writing to a Text File

- **Using java.io.PrintWriter.println**



```
      }

      if (fileOpened)
      {
         Scanner keyboard = new Scanner(System.in);
         System.out.println("Enter " + howMany + " lines of data:");
         for (int counter = 1; counter <= howMany; counter++)
         {
            System.out.print("Line " + counter + ": ");
            String line = keyboard.nextLine();
            toFile.println(line);
         } // end for

         toFile.close();
      } // end if

      return fileOpened;
   } // end createTextFile
} // end TextFileOperations
```

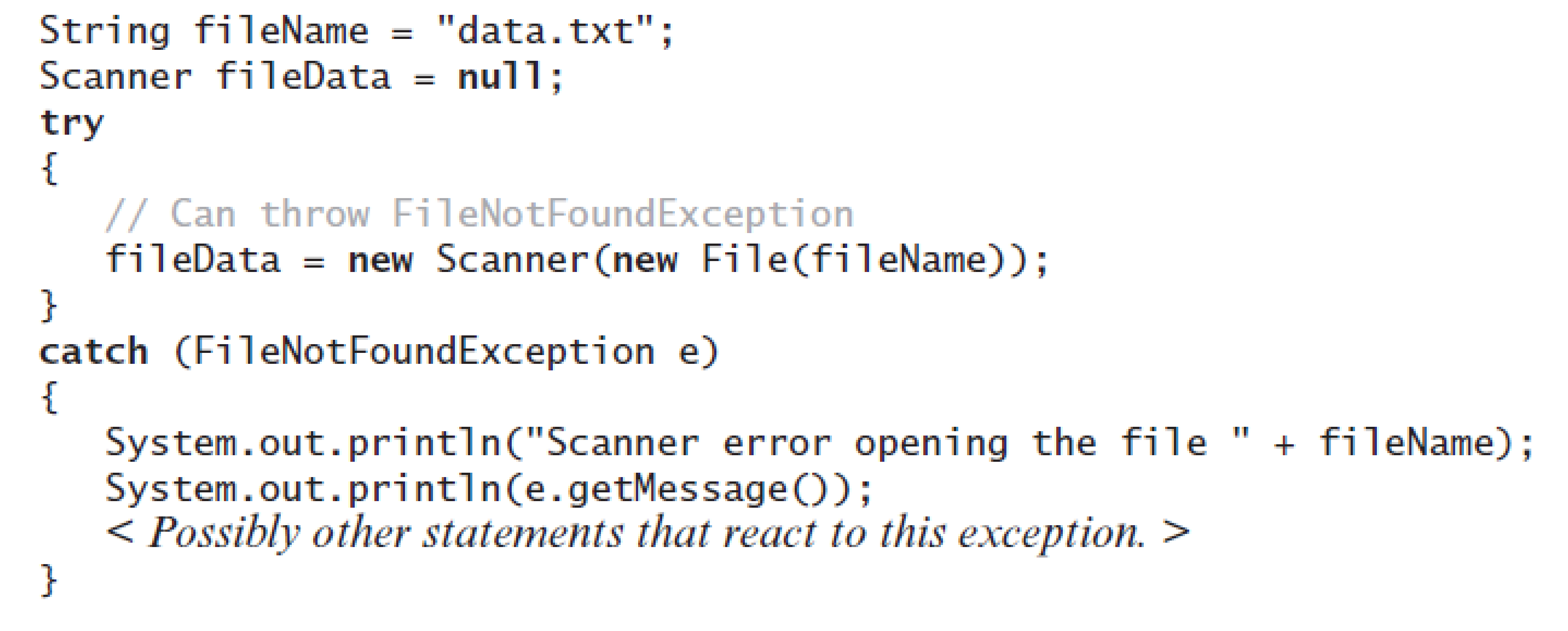
FileWriter vs. PrintWriter (Appending)



```
try
{
   FileWriter fw = new FileWriter(fileName, true);// IOException?
   toFile = new PrintWriter(fw);                  // FileNotFoundException?
}
catch (FileNotFoundException e)
{
   System.out.println("PrintWriter error opening the file " + fileName);
   System.out.println(e.getMessage());
   System.exit(0);
}
catch (IOException e)
{
   System.out.println("FileWriter error opening the file " + fileName);
   System.out.println(e.getMessage());
   System.exit(0);
}
```

Reading a Text File

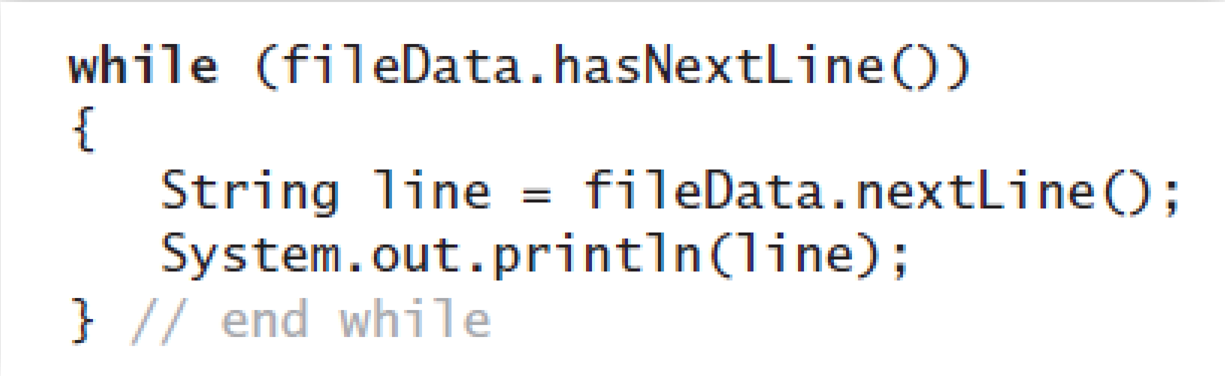
- Opening the text file named **data.txt** for input



```
String fileName = "data.txt";
Scanner fileData = null;
try
{
   // Can throw FileNotFoundException
   fileData = new Scanner(new File(fileName));
}
catch (FileNotFoundException e)
{
   System.out.println("Scanner error opening the file " + fileName);
   System.out.println(e.getMessage());
   < Possibly other statements that react to this exception. >
}
```

Reading a Text File

- If you do not know format of the data in file,
  - Use the **Scanner** method **nextLine** to read it line by line.



```
while (fileData.hasNextLine())
{
   String line = fileData.nextLine();
   System.out.println(line);
} // end while
```

Bag ADT

- The Bag
  - Think of a real bag in which we can place things
  - No rule about how many items to put in
  - No rule about the order of the items
  - No rule about duplicate items
  - No rule about what type of items to put in
    - However, we will make it homogeneous by requiring the items to be the same class or subclass of a specific Java type
  - Let’s look at the interface
    - See BagInterface.java

ADT Bag

- Note what is NOT in the interface:
  - Any specification of the data for the collection
    - We will leave this to the implementation
    - The interface specifies the behaviors only
      - However, the implementation is at least partially implied
      - Must be some type of collection
  - Any implementation of the methods
- Note that other things are not explicitly in the interface but maybe should be
  - Ex: What the method should do
  - Ex: How special cases should be handled
  - We typically have to handle these via comments

ADT Bag

- Ex: `public boolean add(T newEntry)`
  - We want to consider specifications from two points of view:
    1) What is the purpose / effect of the operation in the normal case?
    2) What unusual / erroneous situations can occur and how do we handle them?
  - The first point can be handled via preconditions and postconditions
    - Preconditions indicate what is assumed to be the state of the ADT prior to the method's execution
    - Postconditions indicate what is the state of the ADT after the method's execution
    - From the two we can infer the method's effect

ADT Bag

- Ex: for add(newEntry) we might have:

  Precondition:

  Bag is in a valid state containing N items

  Postconditions:

  Bag is in a valid state containing N+1 items

  newEntry is now contained in the Bag

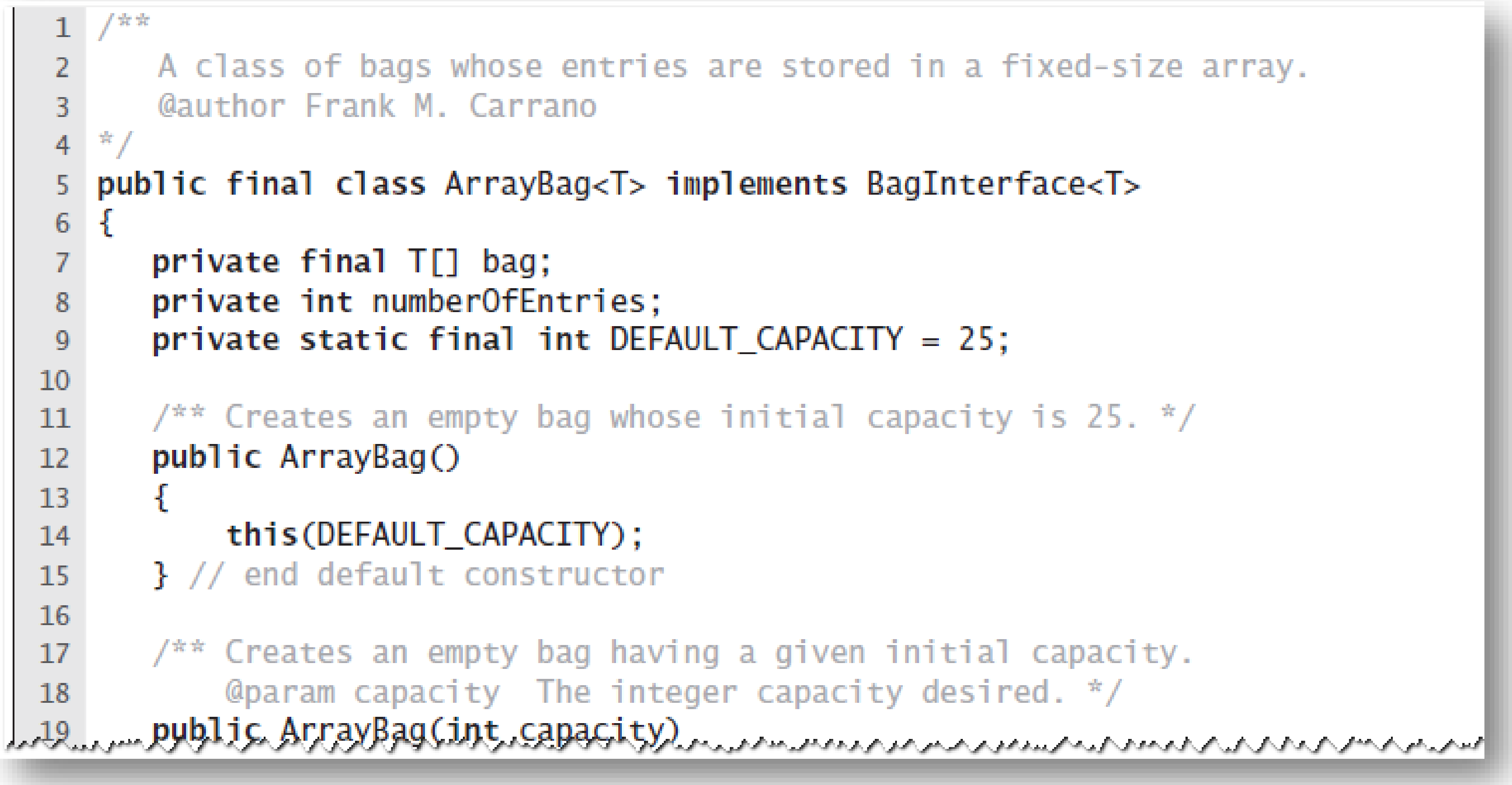
- This is somewhat mathematical, so many ADTs also have operation descriptions explaining the operation in plainer terms
    - More complex operations may also have more complex conditions
  - However, pre and postconditions can be very important for verifying correctness of methods

ADT Bag

- The second point (abnormal cases) is often trickier to handle
    - Sometimes the unusual / erroneous circumstances are not obvious
    - Often they can be handled in more than one way
    - Ex: for add(newEntry) we might have
        - Bag is not valid to begin with due to a previous error
        - newEntry is not a valid object
    - Assuming we detect the problem, we could handle it by
        - Doing a "no op"
        - Returning a false boolean value
        - Throwing an exception
    - We need to make these clear to the user of the ADT so they know what to expect

Fixed-Size Array

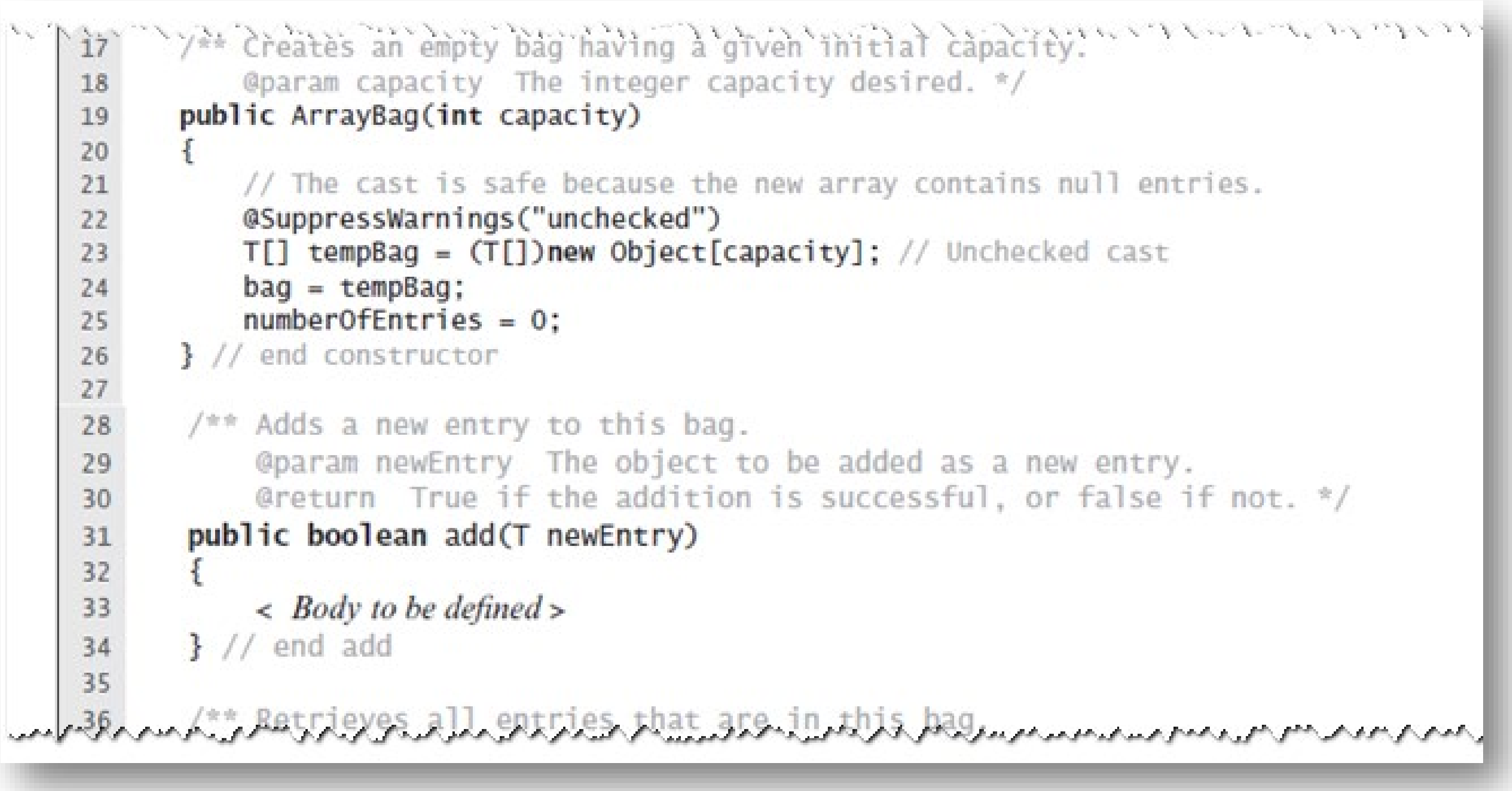
An outline of the class `ArrayBag`



```java
/**
   A class of bags whose entries are stored in a fixed-size array.
   @author Frank M. Carrano
*/
public final class ArrayBag<T> implements BagInterface<T>
{
   private final T[] bag;
   private int numberOfEntries;
   private static final int DEFAULT_CAPACITY = 25;

   /** Creates an empty bag whose initial capacity is 25. */
   public ArrayBag()
   {
      this(DEFAULT_CAPACITY);
   } // end default constructor

   /** Creates an empty bag having a given initial capacity.
       @param capacity  The integer capacity desired. */
   public ArrayBag(int capacity)
```

Fixed-Size Array

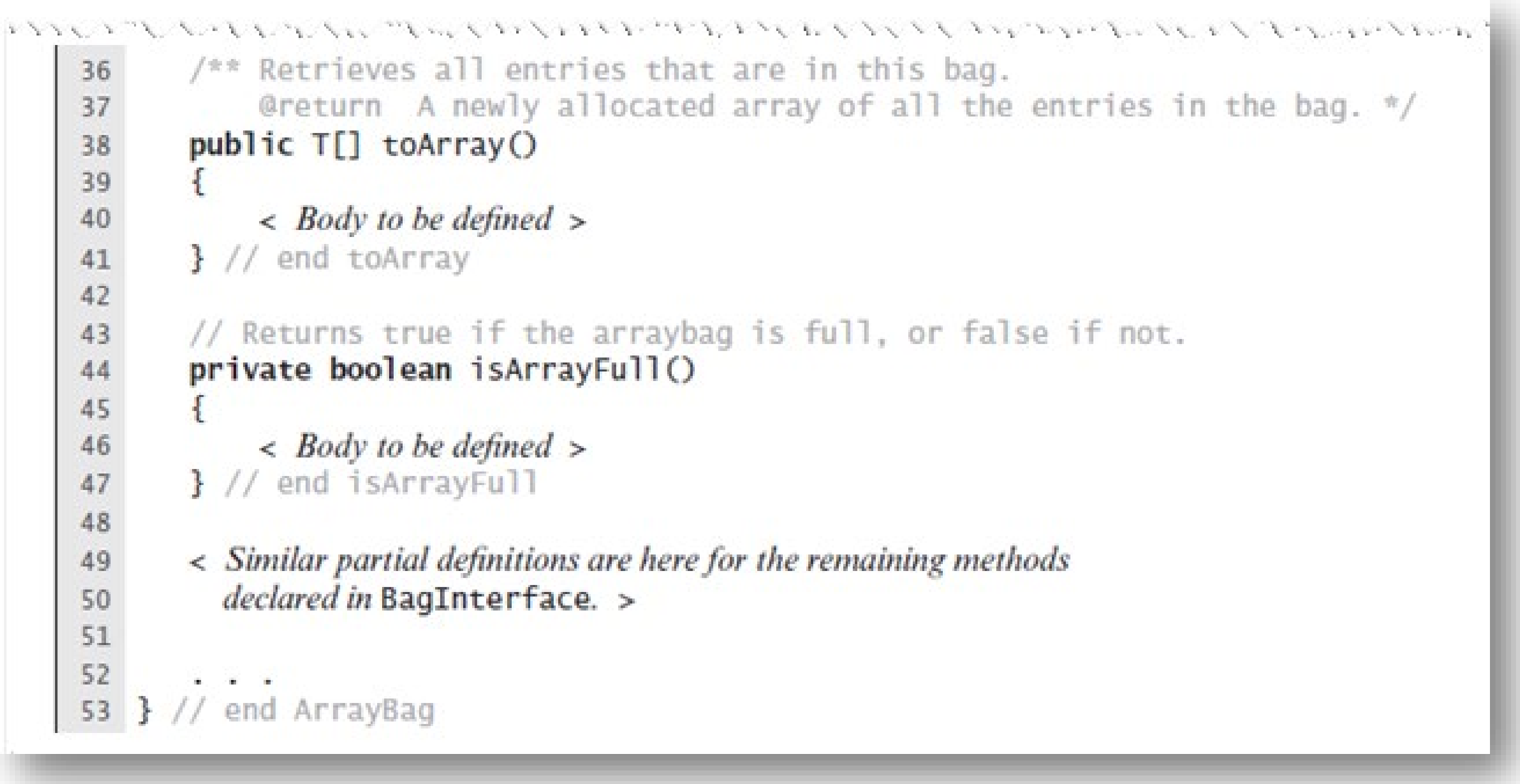
An outline of the class **ArrayBag**



```
/** Creates an empty bag having a given initial capacity.
    @param capacity  The integer capacity desired. */
public ArrayBag(int capacity)
{
    // The cast is safe because the new array contains null entries.
    @SuppressWarnings("unchecked")
    T[] tempBag = (T[])new Object[capacity]; // Unchecked cast
    bag = tempBag;
    numberOfEntries = 0;
} // end constructor

/** Adds a new entry to this bag.
    @param newEntry  The object to be added as a new entry.
    @return  True if the addition is successful, or false if not. */
public boolean add(T newEntry)
{
    < Body to be defined >
} // end add

/** Retrieves all entries that are in this bag.
```

Fixed-Size Array

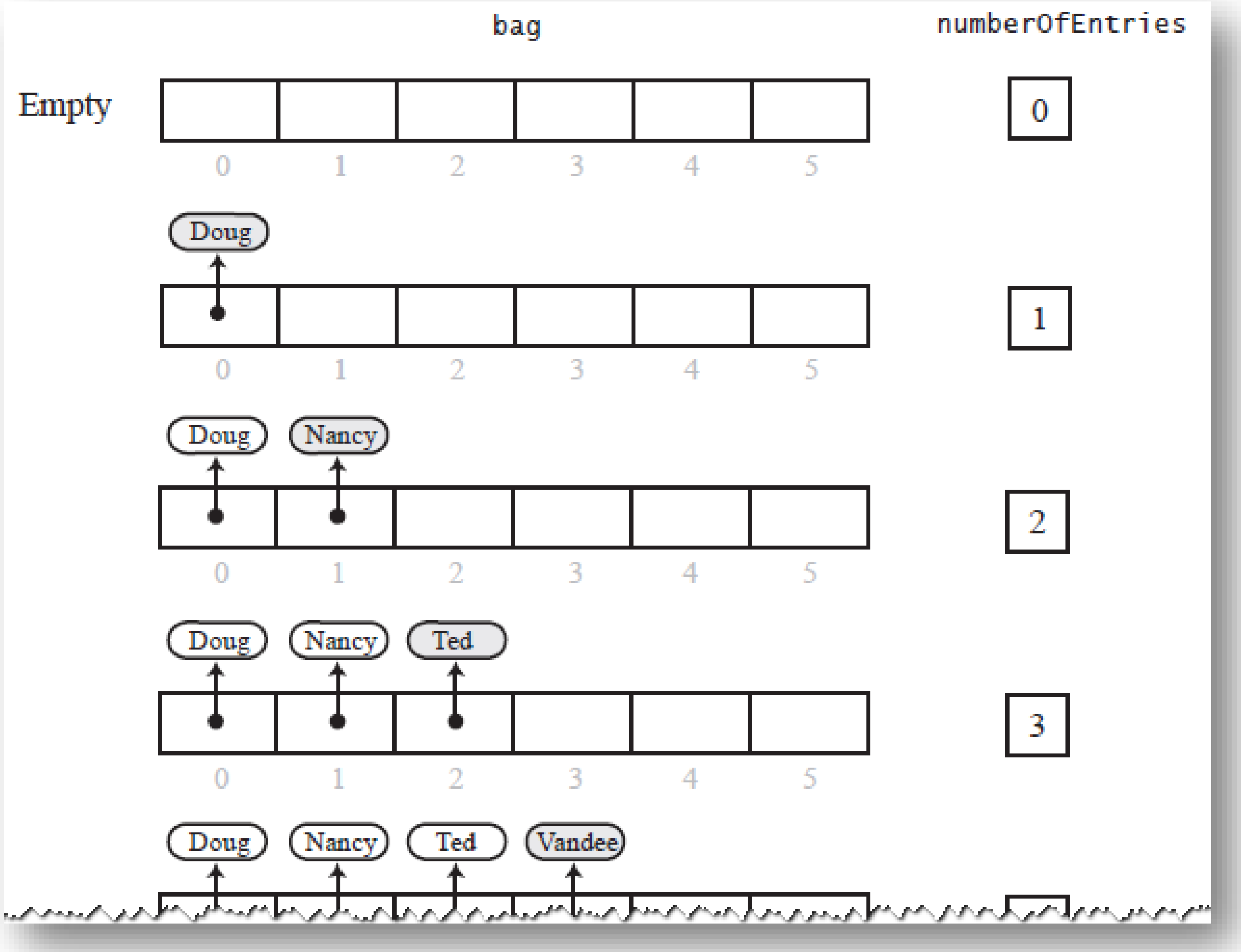
An outline of the class **ArrayBag**



```
  /** Retrieves all entries that are in this bag.
      @return  A newly allocated array of all the entries in the bag. */
  public T[] toArray()
  {
      < Body to be defined >
  } // end toArray

  // Returns true if the arraybag is full, or false if not.
  private boolean isArrayFull()
  {
      < Body to be defined >
  } // end isArrayFull

  < Similar partial definitions are here for the remaining methods
   declared in BagInterface. >

  . . .
} // end ArrayBag
```

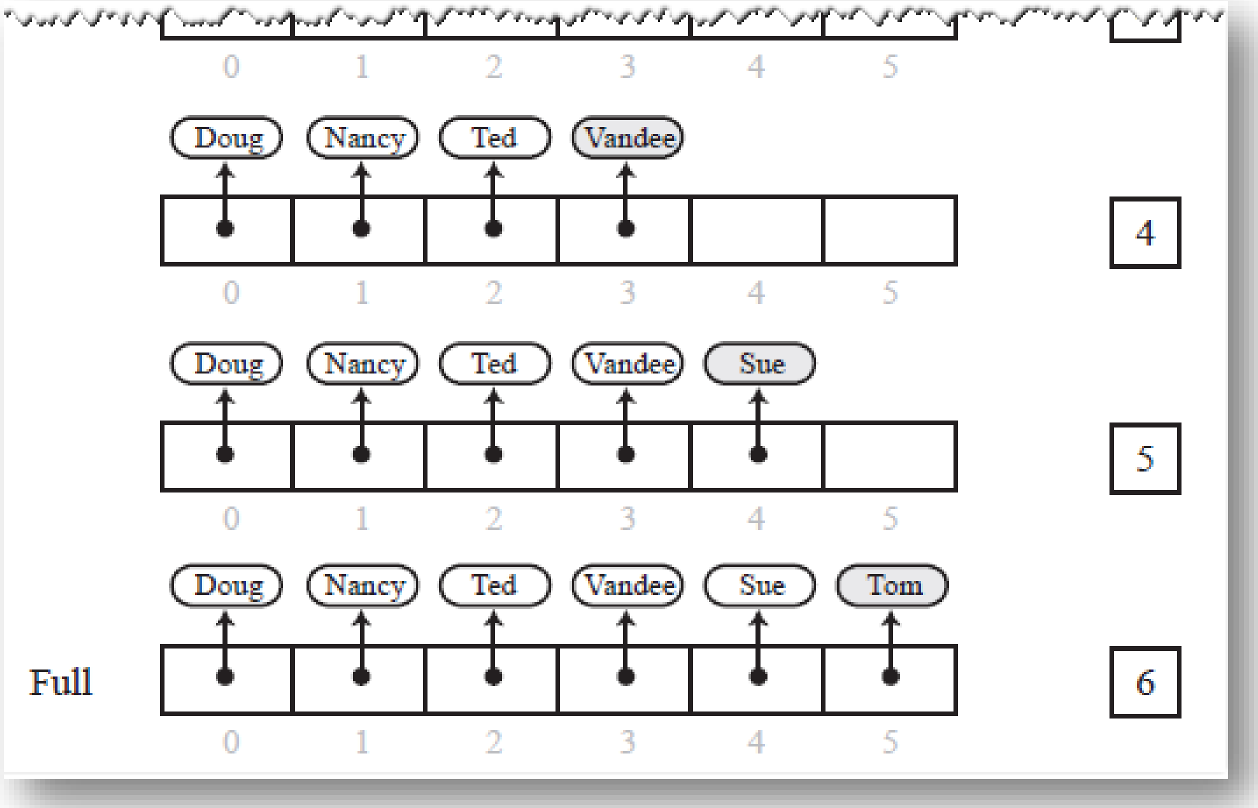
Fixed-Size Array

Adding entries to an array that represents a bag, whose capacity is six, until it becomes full



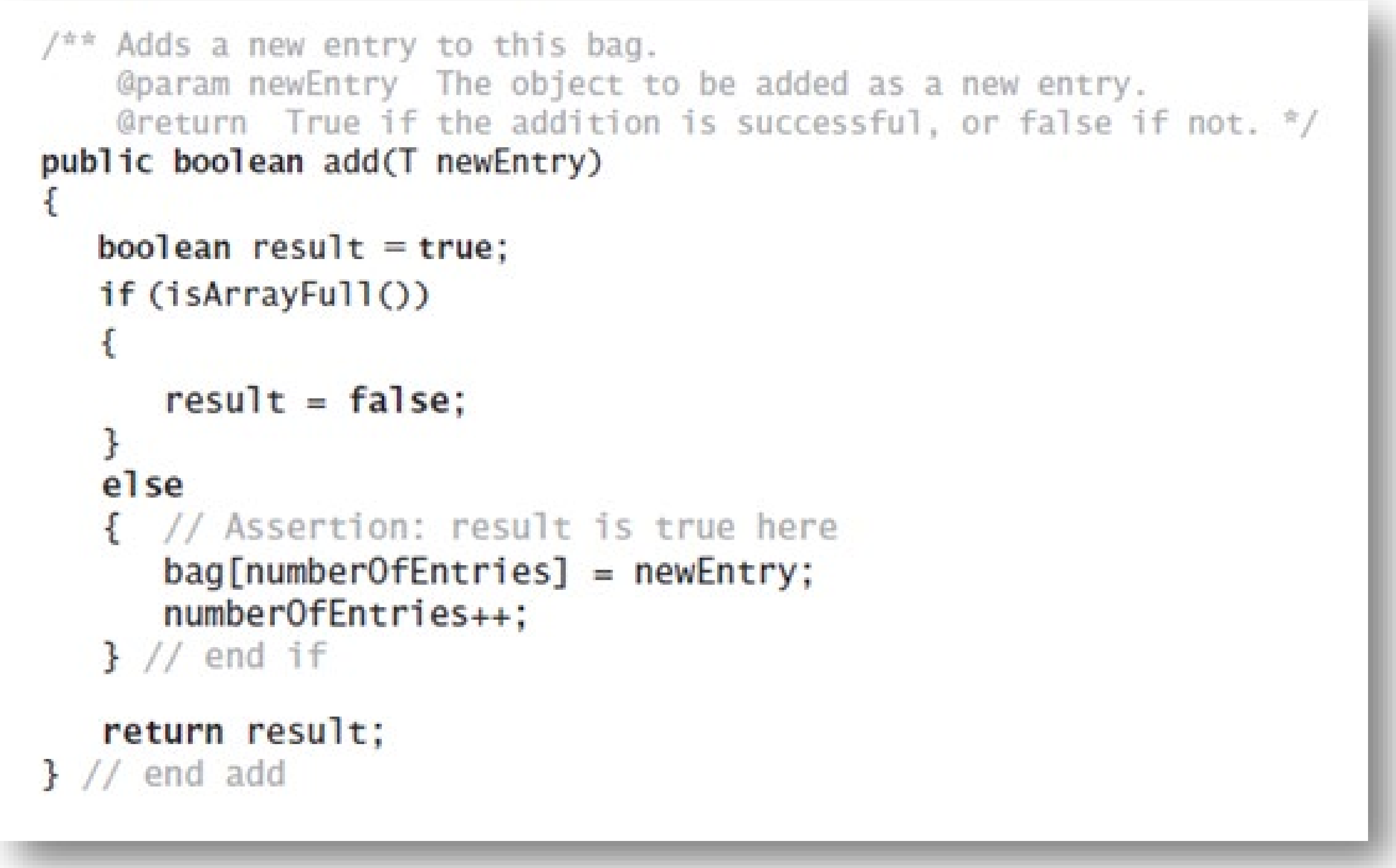
Fixed-Size Array

Adding entries to an array that represents a bag, whose capacity is six, until it becomes full



Fixed-Size Array

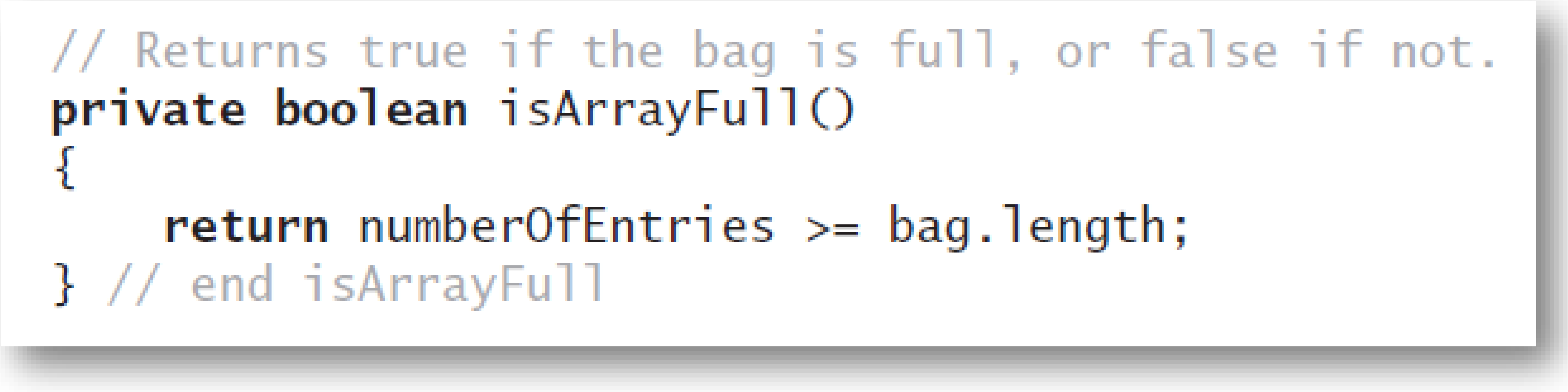
Method add



```
/** Adds a new entry to this bag.
    @param newEntry  The object to be added as a new entry.
    @return  True if the addition is successful, or false if not. */
public boolean add(T newEntry)
{
   boolean result = true;
   if (isArrayFull())
   {
      result = false;
   }
   else
   {  // Assertion: result is true here
      bag[numberOfEntries] = newEntry;
      numberOfEntries++;
   } // end if

   return result;
} // end add
```

Fixed-Size Array

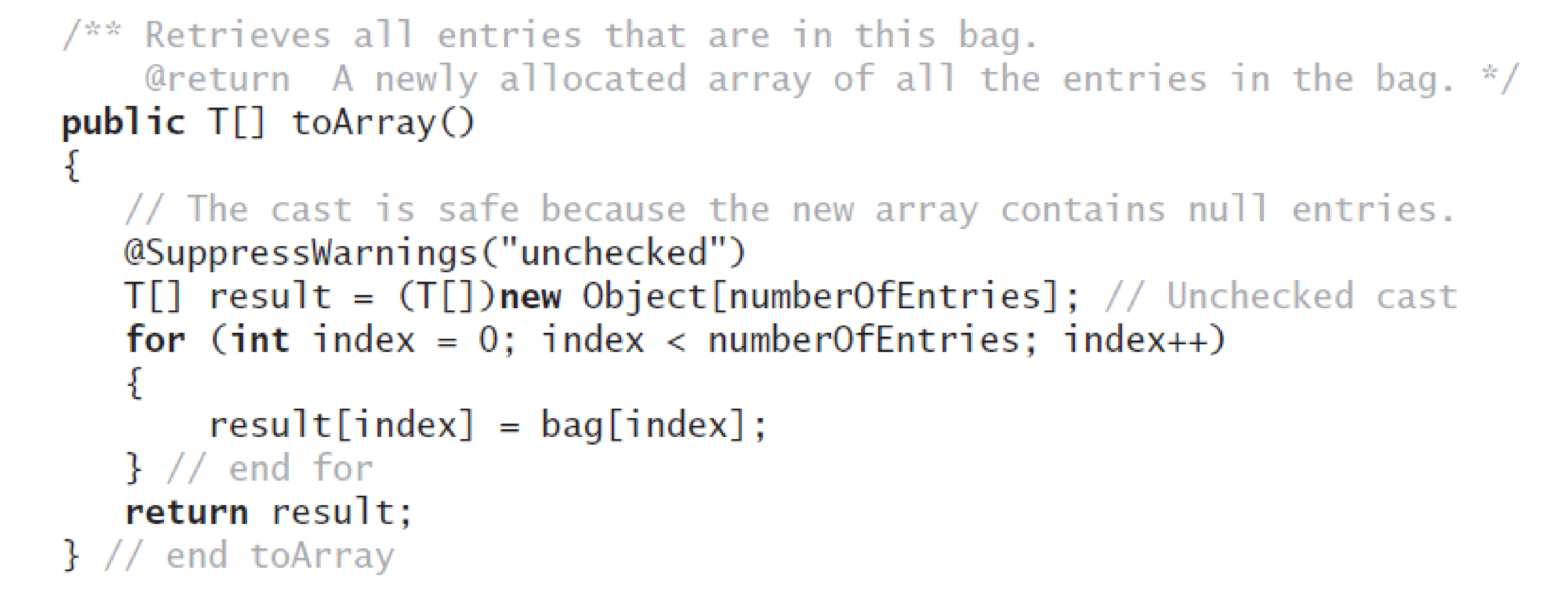
Method `isArrayFull`



```
// Returns true if the bag is full, or false if not.
private boolean isArrayFull()
{
    return numberOfEntries >= bag.length;
} // end isArrayFull
```

Fixed-Size Array

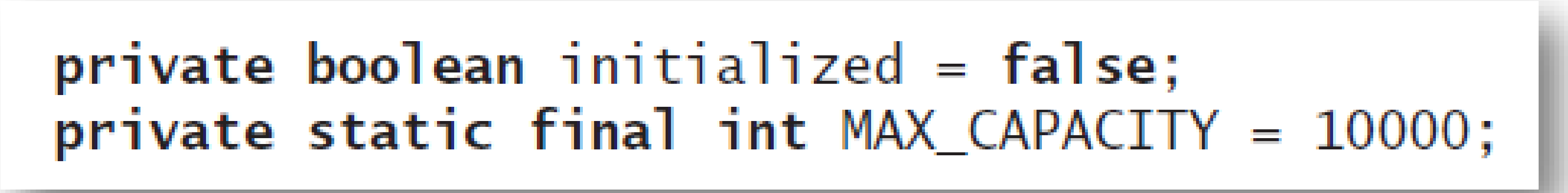
Method `toArray`



```
/** Retrieves all entries that are in this bag.
    @return  A newly allocated array of all the entries in the bag. */
public T[] toArray()
{
   // The cast is safe because the new array contains null entries.
   @SuppressWarnings("unchecked")
   T[] result = (T[])new Object[numberOfEntries]; // Unchecked cast
   for (int index = 0; index < numberOfEntries; index++)
   {
       result[index] = bag[index];
   } // end for
   return result;
} // end toArray
```

Making the Implementation Secure

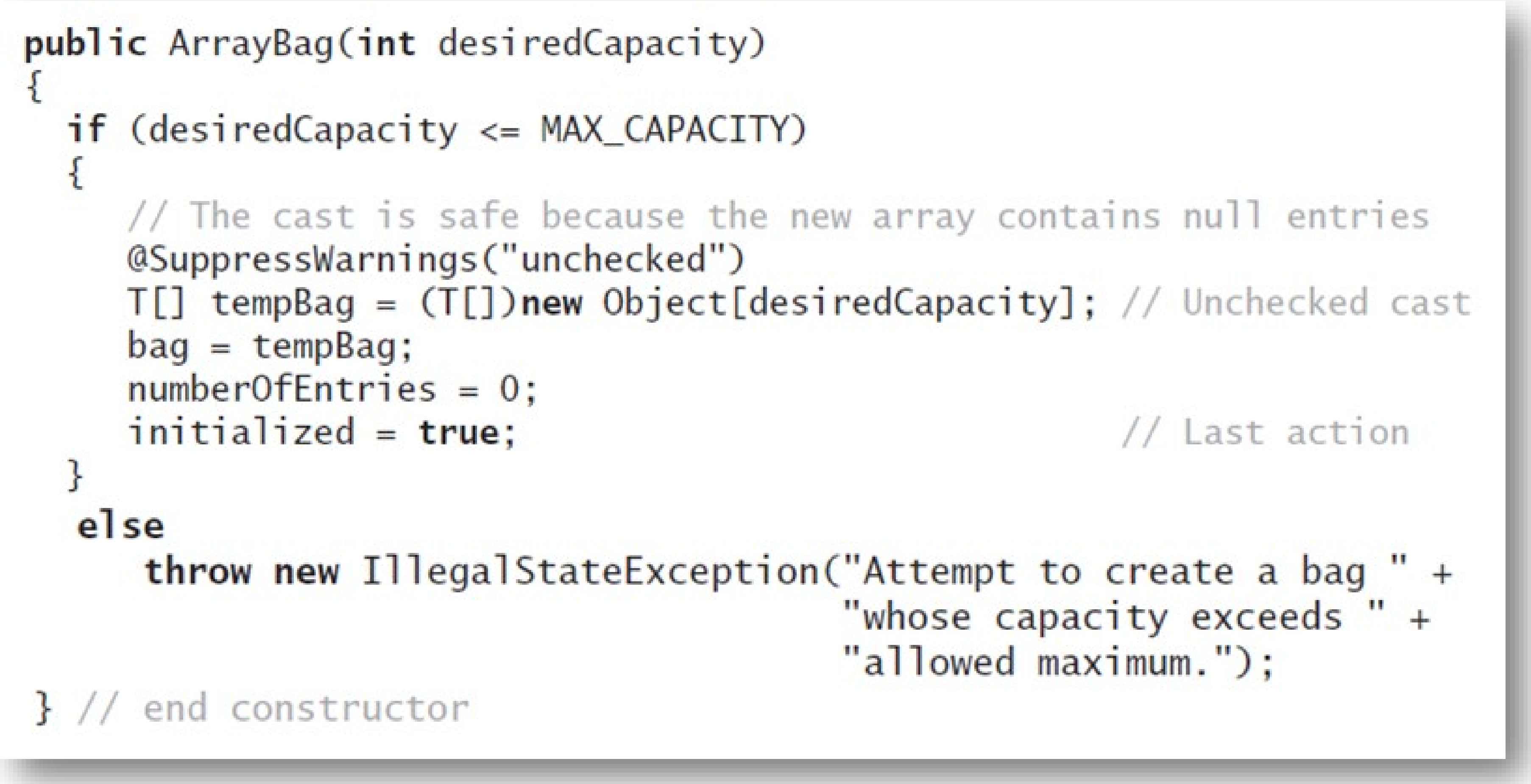
- Practice fail-safe programming by including checks for anticipated errors
- Validate input data and arguments to a method
- Refine incomplete implementation of `ArrayBag` to make code more secure by adding the following two data fields



```
private boolean initialized = false;
private static final int MAX_CAPACITY = 10000;
```

Making the Implementation Secure

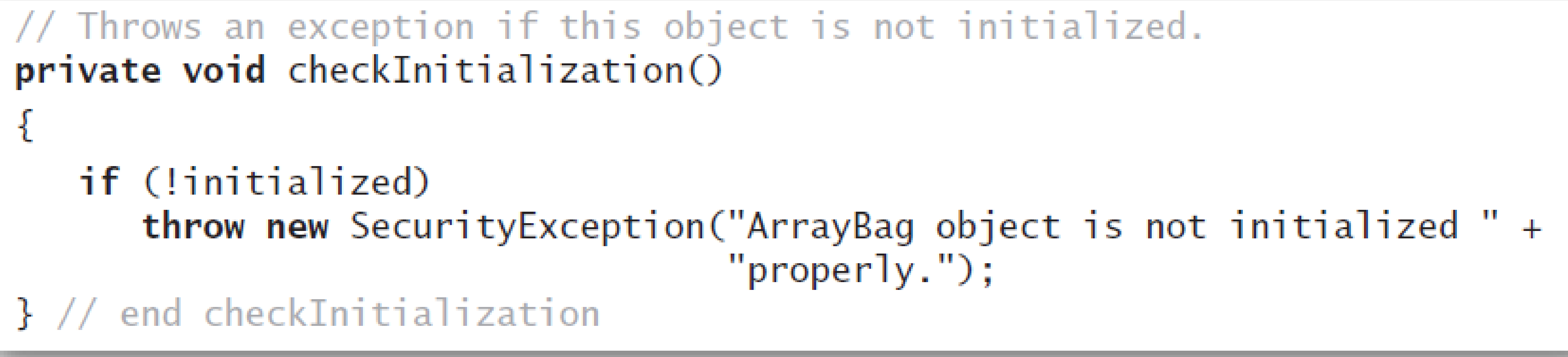
Revised constructor



```
public ArrayBag(int desiredCapacity)
{
  if (desiredCapacity <= MAX_CAPACITY)
  {
     // The cast is safe because the new array contains null entries
     @SuppressWarnings("unchecked")
     T[] tempBag = (T[])new Object[desiredCapacity]; // Unchecked cast
     bag = tempBag;
     numberOfEntries = 0;
     initialized = true;                              // Last action
  }
  else
      throw new IllegalStateException("Attempt to create a bag " +
                                      "whose capacity exceeds " +
                                      "allowed maximum.");
} // end constructor
```

Making the Implementation Secure

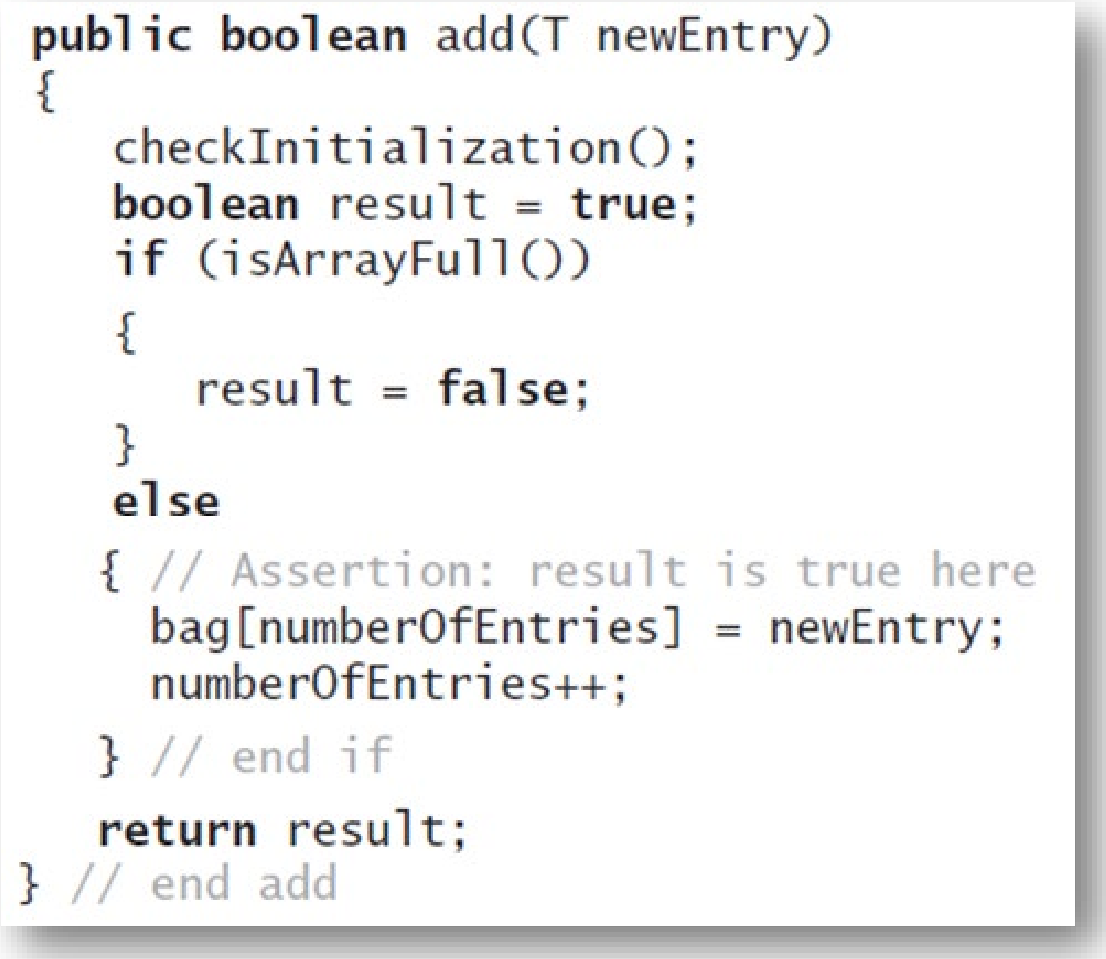
Method to check initialization



```
// Throws an exception if this object is not initialized.
private void checkInitialization()
{
   if (!initialized)
      throw new SecurityException("ArrayBag object is not initialized " +
                                  "properly.");
} // end checkInitialization
```

Making the Implementation Secure

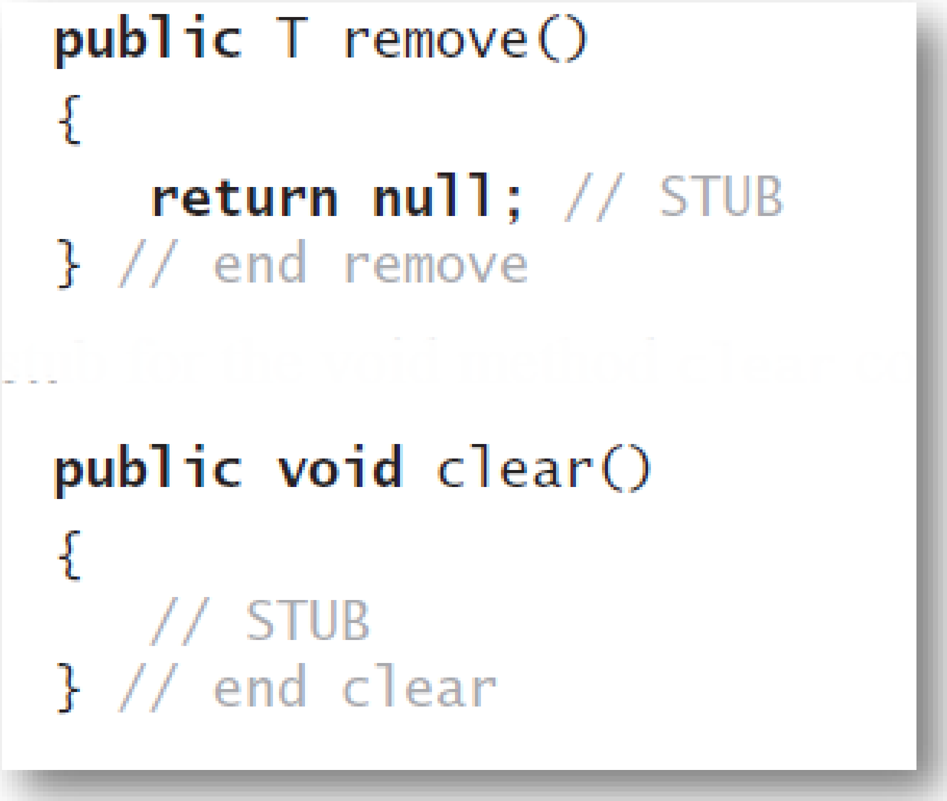
Revise the method **add**



```
public boolean add(T newEntry)
{
   checkInitialization();
   boolean result = true;
   if (isArrayFull())
   {
      result = false;
   }
   else
  { // Assertion: result is true here
     bag[numberOfEntries] = newEntry;
     numberOfEntries++;
  } // end if
  return result;
} // end add
```

Testing the Core Methods

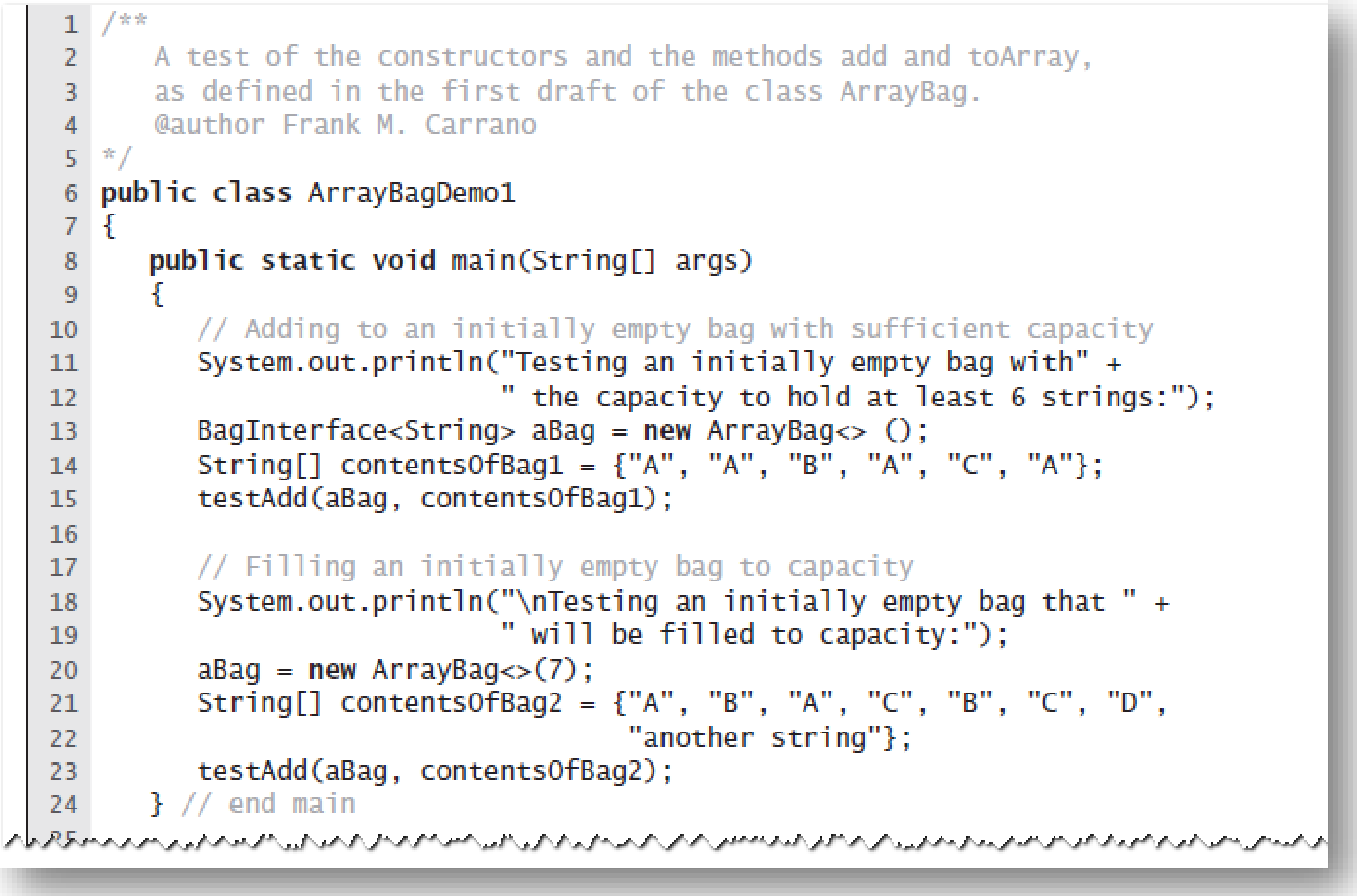
Stubs for `remove` and `clear`



```
public T remove()
{
   return null; // STUB
} // end remove

public void clear()
{
   // STUB
} // end clear
```

Testing the Core Methods

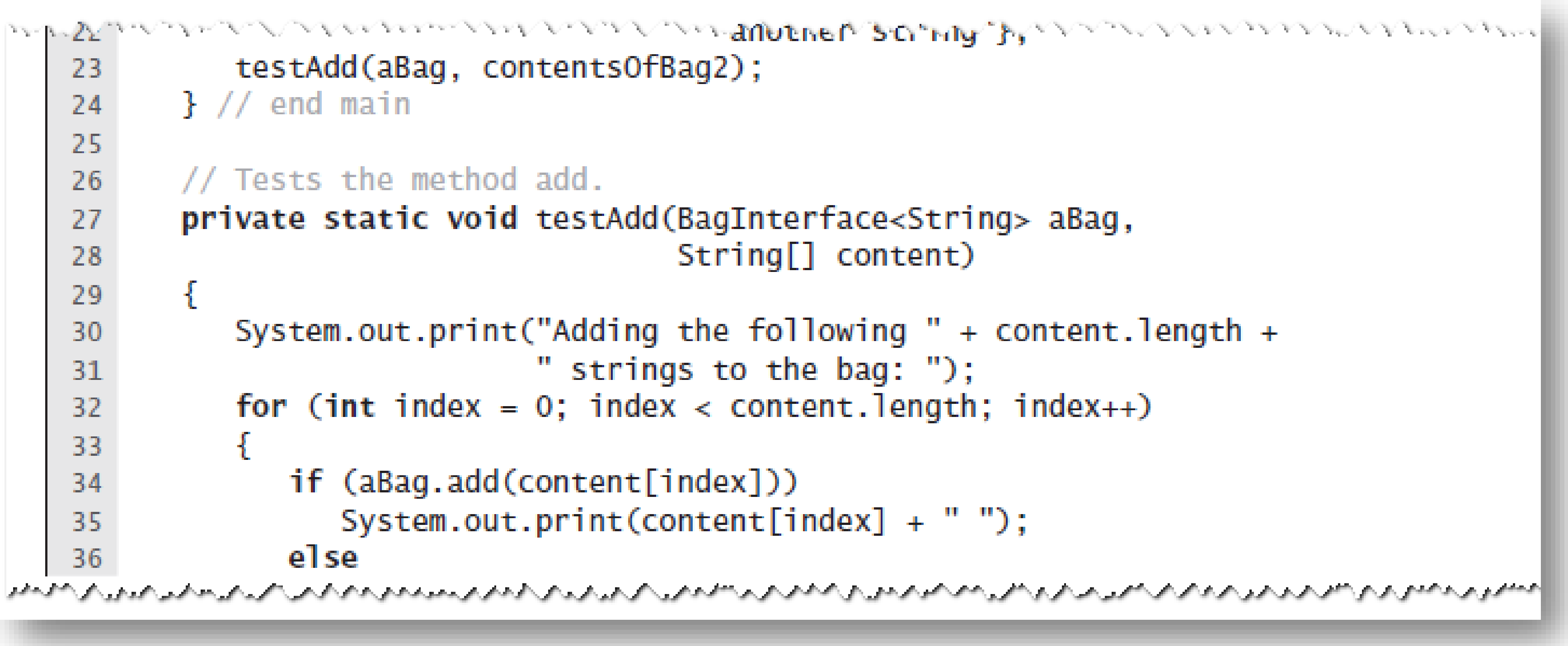
A program that tests core methods
of the class **ArrayBag**



```
/**
   A test of the constructors and the methods add and toArray,
   as defined in the first draft of the class ArrayBag.
   @author Frank M. Carrano
*/
public class ArrayBagDemo1
{
   public static void main(String[] args)
   {
      // Adding to an initially empty bag with sufficient capacity
      System.out.println("Testing an initially empty bag with" +
                         " the capacity to hold at least 6 strings:");
      BagInterface<String> aBag = new ArrayBag<> ();
      String[] contentsOfBag1 = {"A", "A", "B", "A", "C", "A"};
      testAdd(aBag, contentsOfBag1);

      // Filling an initially empty bag to capacity
      System.out.println("\nTesting an initially empty bag that " +
                         " will be filled to capacity:");
      aBag = new ArrayBag<>(7);
      String[] contentsOfBag2 = {"A", "B", "A", "C", "B", "C", "D",
                                 "another string"};
      testAdd(aBag, contentsOfBag2);
   } // end main
```

Testing the Core Methods

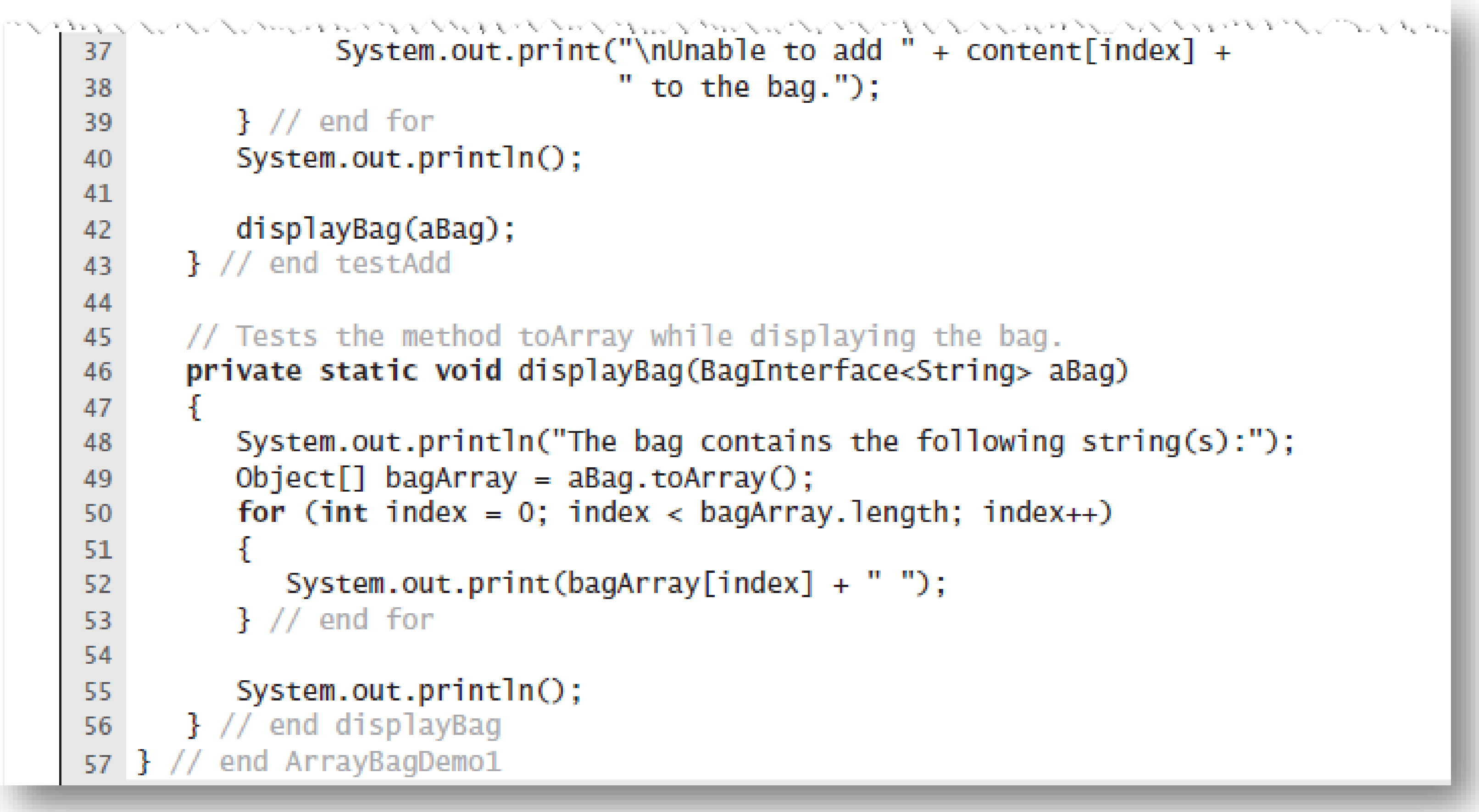
A program that tests core methods
of the class **`ArrayBag`**



```
                                 "another string" };
      testAdd(aBag, contentsOfBag2);
   } // end main

   // Tests the method add.
   private static void testAdd(BagInterface<String> aBag,
                               String[] content)
   {
      System.out.print("Adding the following " + content.length +
                       " strings to the bag: ");
      for (int index = 0; index < content.length; index++)
      {
         if (aBag.add(content[index]))
            System.out.print(content[index] + " ");
         else
```

Testing the Core Methods

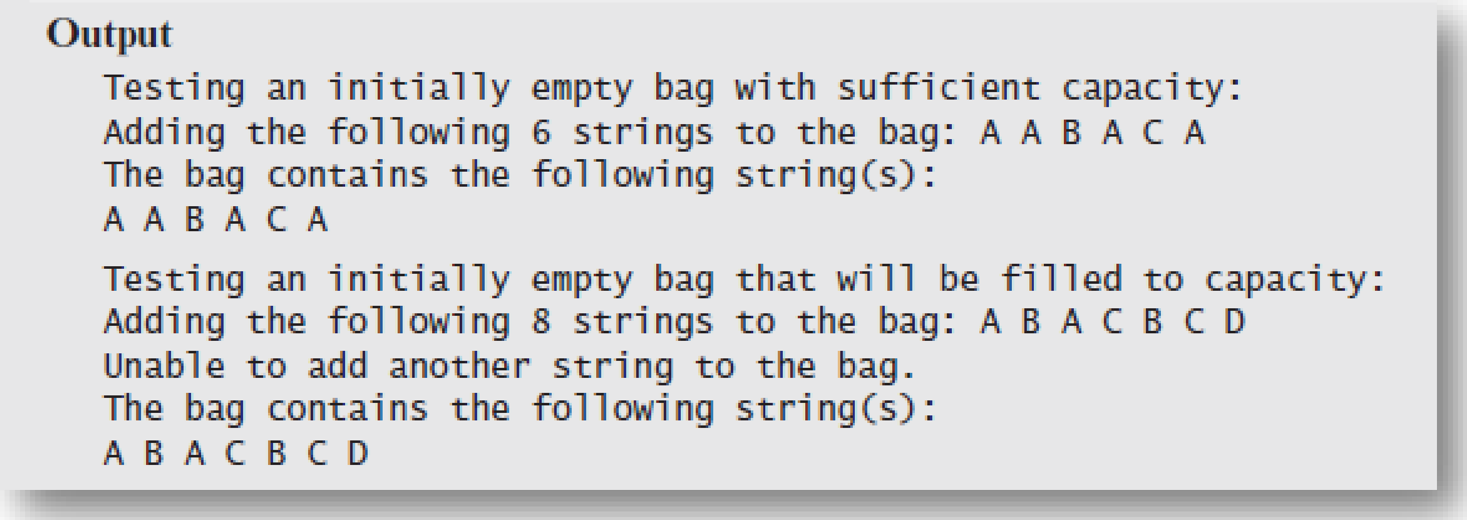
A program that tests core methods of the class **ArrayBag**



```java
            System.out.print("\nUnable to add " + content[index] +
                             " to the bag.");
      } // end for
      System.out.println();

      displayBag(aBag);
   } // end testAdd

   // Tests the method toArray while displaying the bag.
   private static void displayBag(BagInterface<String> aBag)
   {
      System.out.println("The bag contains the following string(s):");
      Object[] bagArray = aBag.toArray();
      for (int index = 0; index < bagArray.length; index++)
      {
         System.out.print(bagArray[index] + " ");
      } // end for

      System.out.println();
   } // end displayBag
} // end ArrayBagDemo1
```

Testing the Core Methods

A program that tests core methods
of the class **ArrayBag**

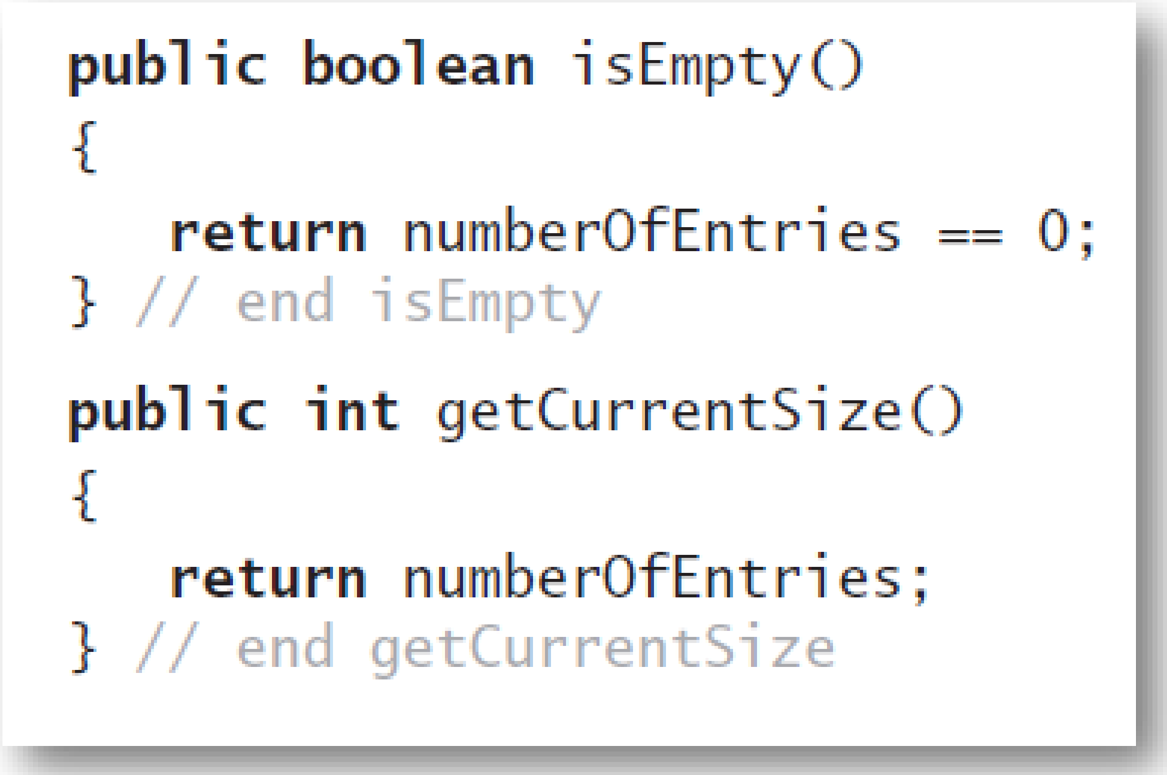


Output

```
Testing an initially empty bag with sufficient capacity:
Adding the following 6 strings to the bag: A A B A C A
The bag contains the following string(s):
A A B A C A

Testing an initially empty bag that will be filled to capacity:
Adding the following 8 strings to the bag: A B A C B C D
Unable to add another string to the bag.
The bag contains the following string(s):
A B A C B C D
```

Implementing More Methods

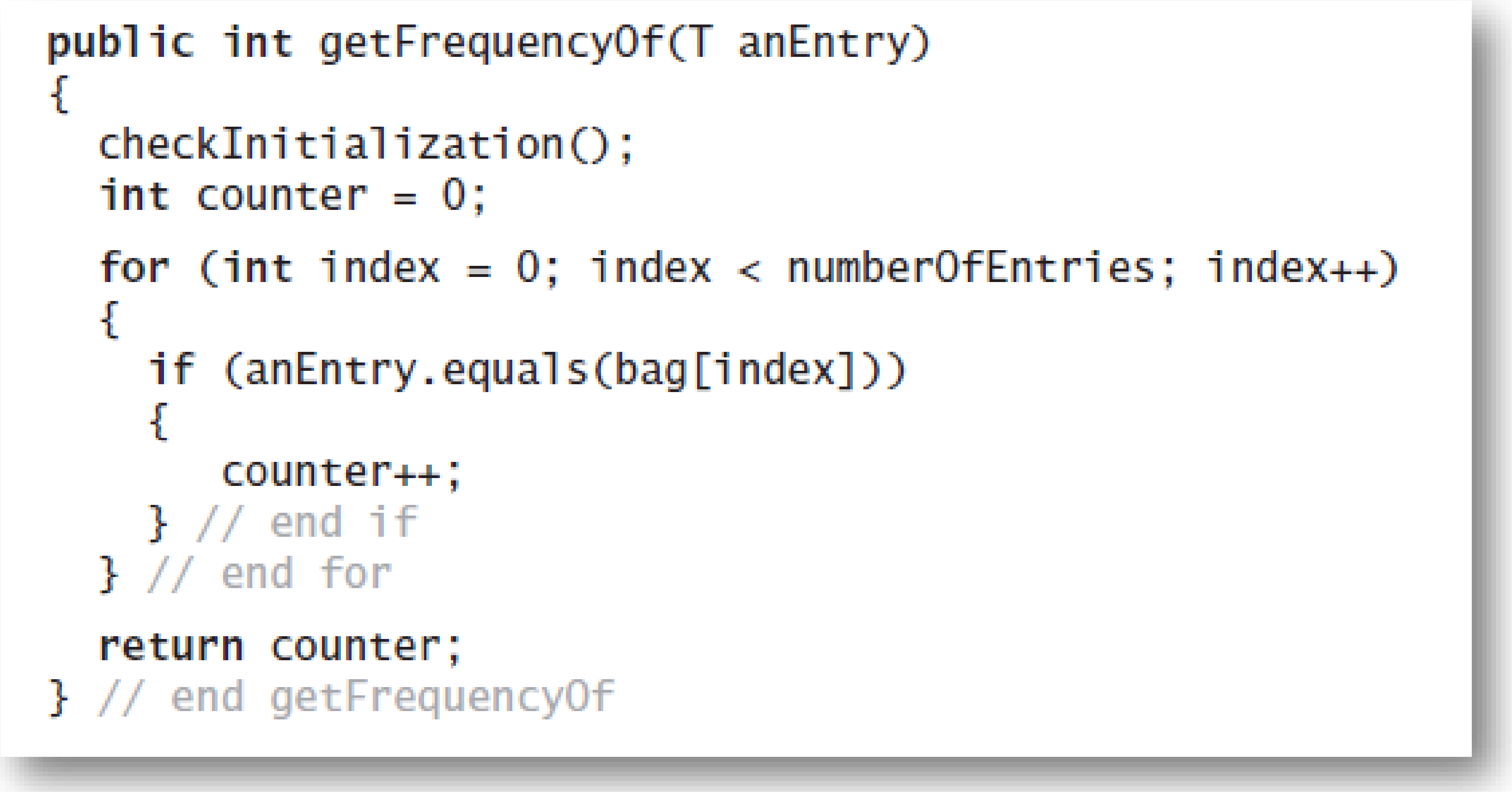
Methods `isEmpty` and `getCurrentSize`



```java
public boolean isEmpty()
{
   return numberOfEntries == 0;
} // end isEmpty

public int getCurrentSize()
{
   return numberOfEntries;
} // end getCurrentSize
```

Implementing More Methods

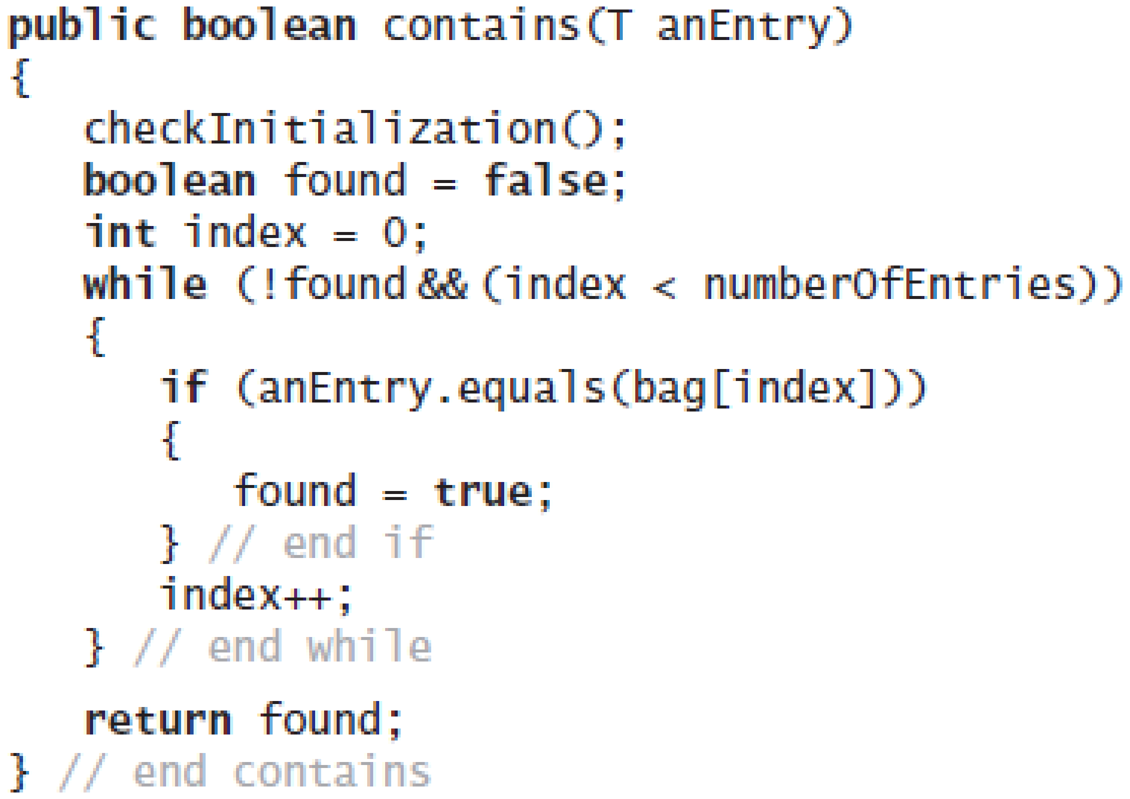
Method getFrequencyOf



```
public int getFrequencyOf(T anEntry)
{
   checkInitialization();
   int counter = 0;

   for (int index = 0; index < numberOfEntries; index++)
   {
      if (anEntry.equals(bag[index]))
      {
          counter++;
      } // end if
   } // end for

   return counter;
} // end getFrequencyOf
```

Implementing More Methods

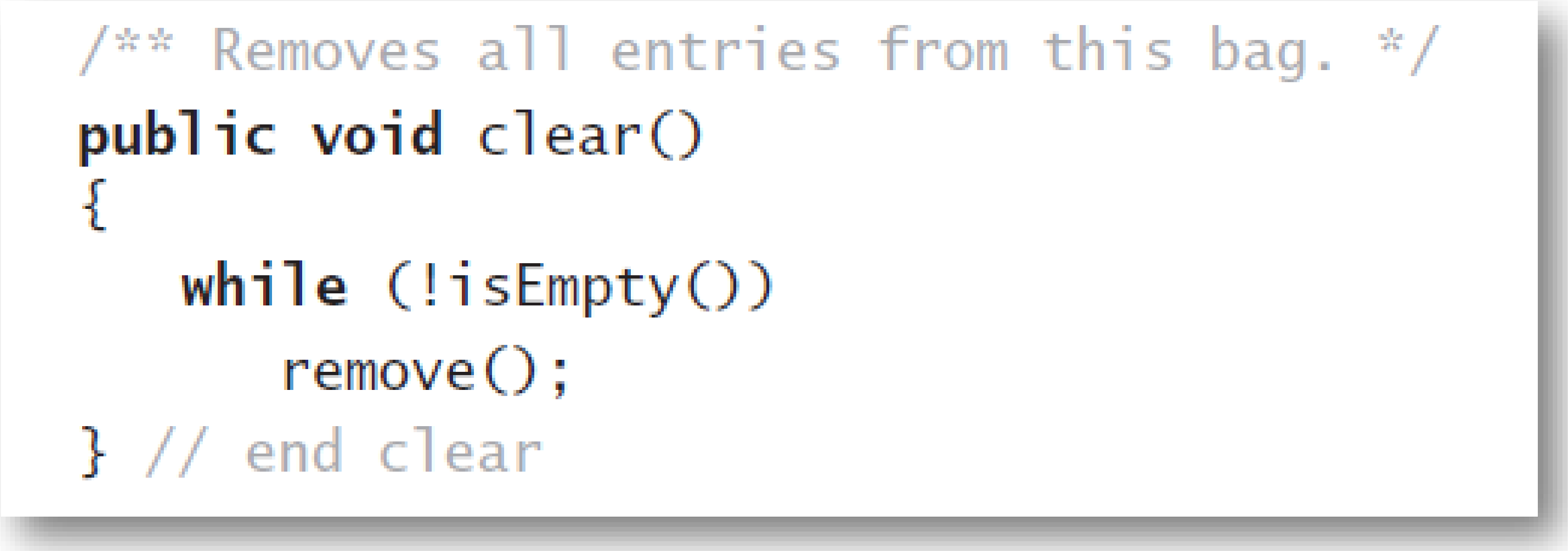
Method `contains`



```
public boolean contains(T anEntry)
{
   checkInitialization();
   boolean found = false;
   int index = 0;
   while (!found && (index < numberOfEntries))
   {
      if (anEntry.equals(bag[index]))
      {
         found = true;
      } // end if
      index++;
   } // end while

   return found;
} // end contains
```

Methods That Remove Entries

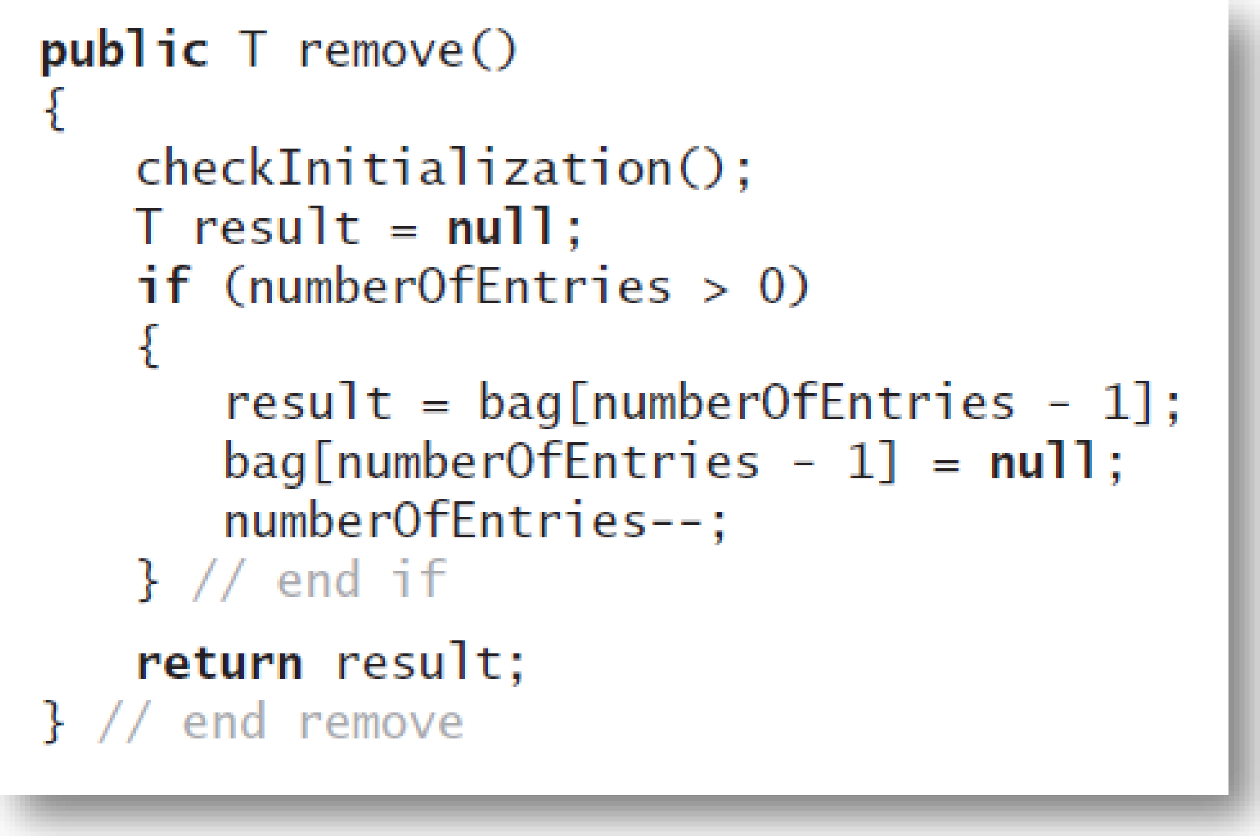
The method `clear`



```
/** Removes all entries from this bag. */
public void clear()
{
   while (!isEmpty())
      remove();
} // end clear
```

Methods That Remove Entries

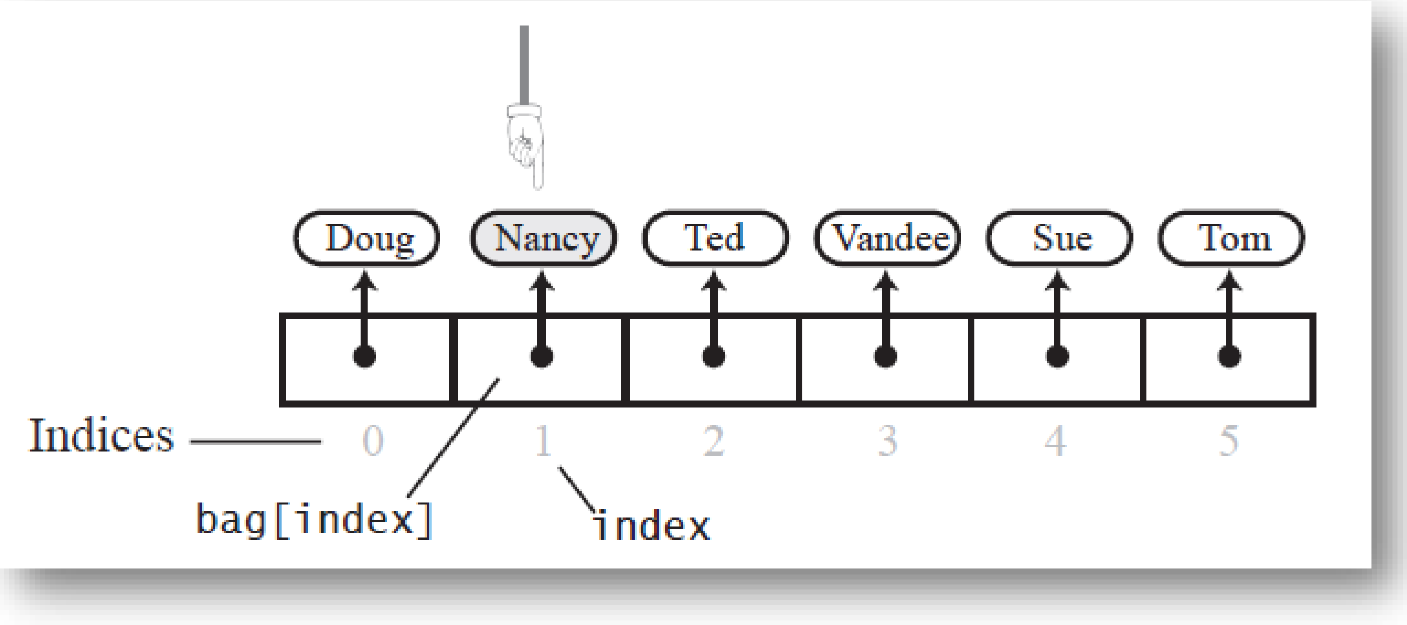
The method **remove**



```
public T remove()
{
   checkInitialization();
   T result = null;
   if (numberOfEntries > 0)
   {
      result = bag[numberOfEntries - 1];
      bag[numberOfEntries - 1] = null;
      numberOfEntries--;
   } // end if

   return result;
} // end remove
```

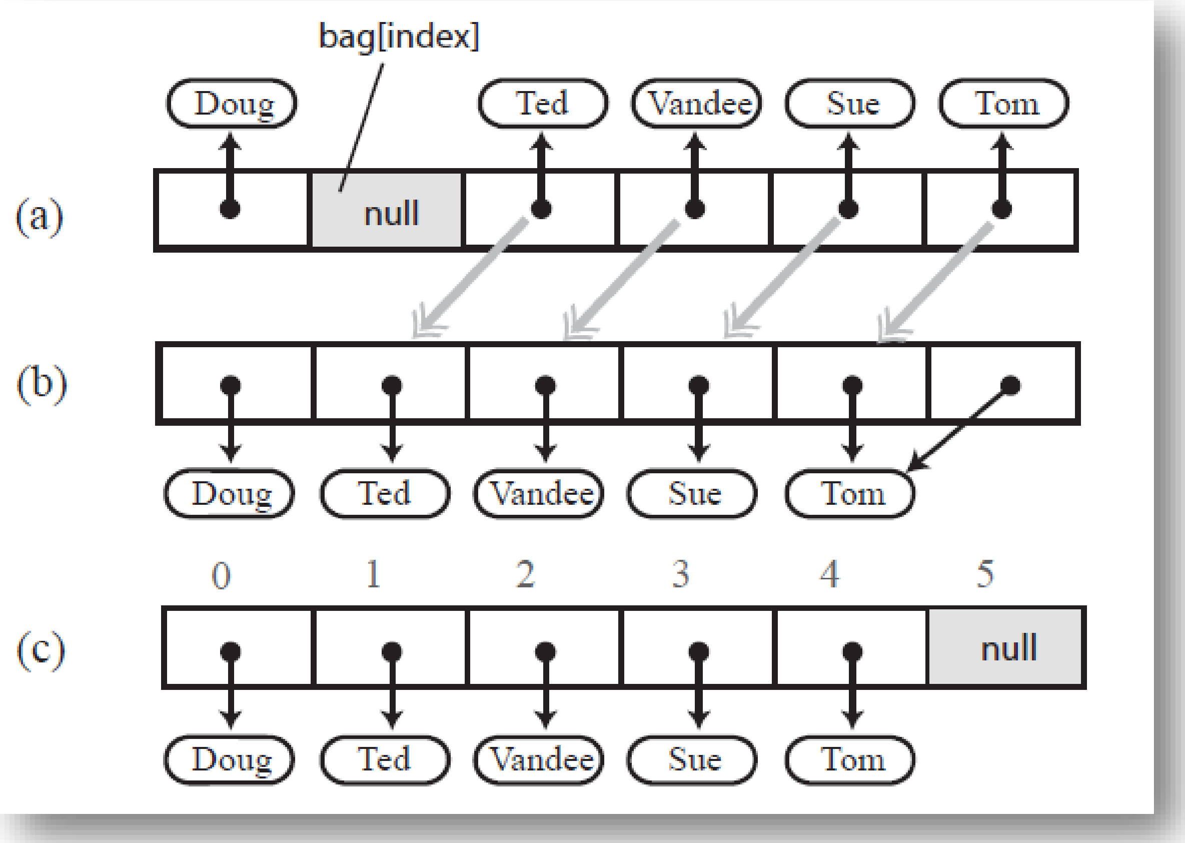
Methods That Remove Entries

The array bag after a successful search for the string “Nancy"



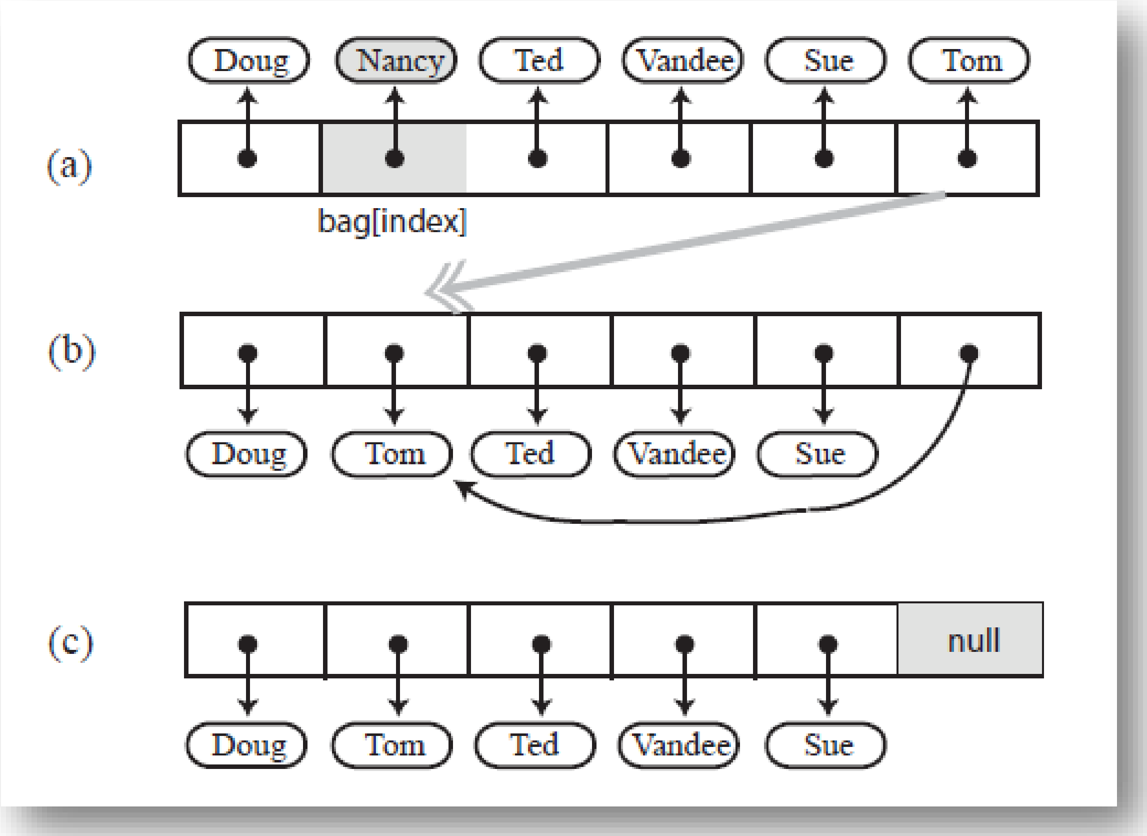
Methods That Remove Entries

- (a) A gap in the array bag after setting the entry in `bag[index]` to `null`
- (b-c) the array after shifting subsequent entries to avoid a gap



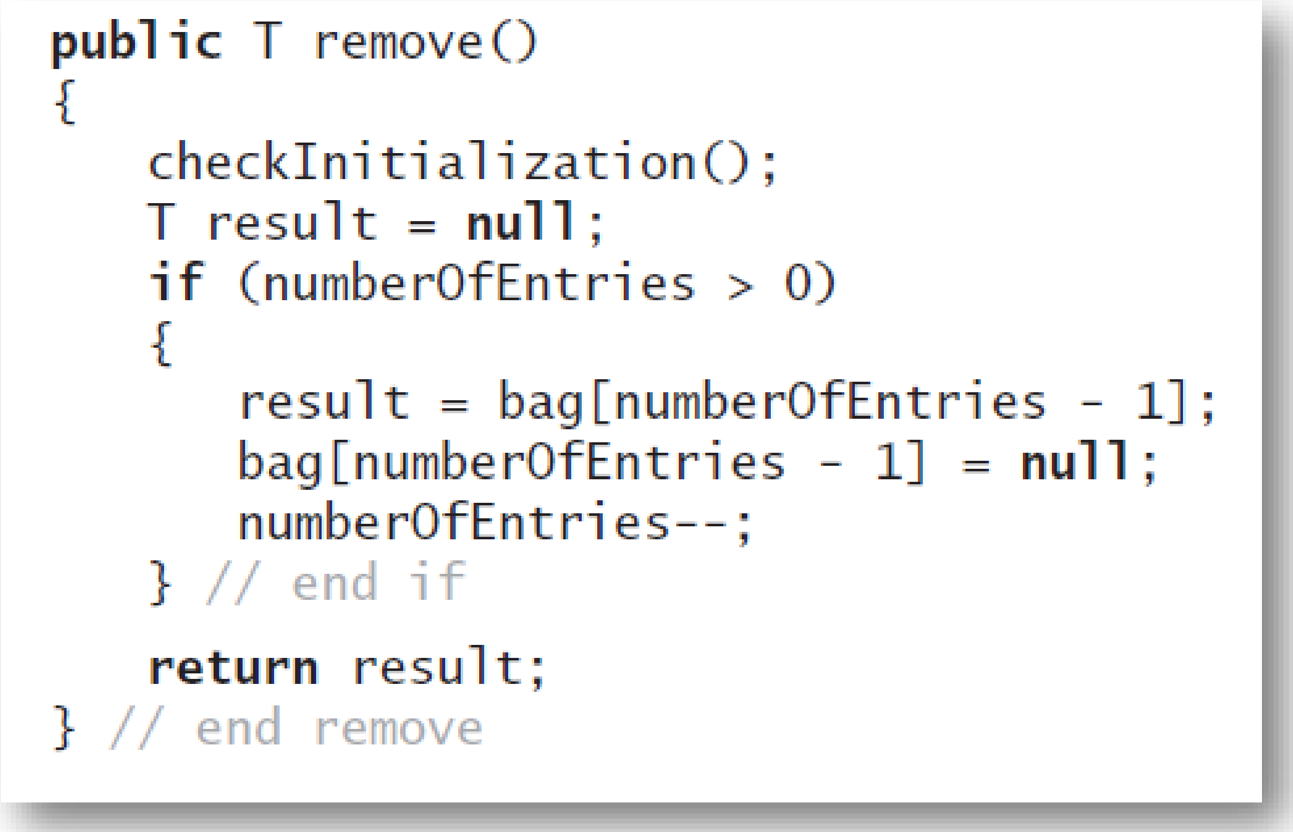
Methods That Remove Entries

Avoiding a gap in the array while removing an entry



Methods That Remove Entries

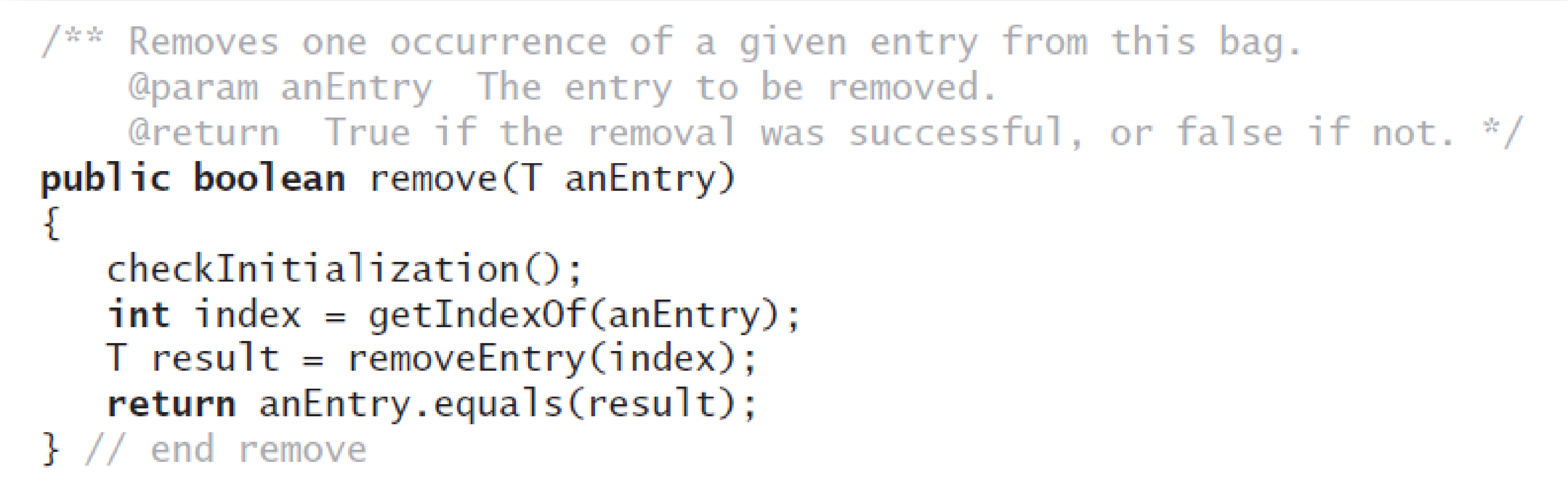
New definition of `remove`



```
public T remove()
{
   checkInitialization();
   T result = null;
   if (numberOfEntries > 0)
   {
      result = bag[numberOfEntries - 1];
      bag[numberOfEntries - 1] = null;
      numberOfEntries--;
   } // end if

   return result;
} // end remove
```

Methods That Remove Entries

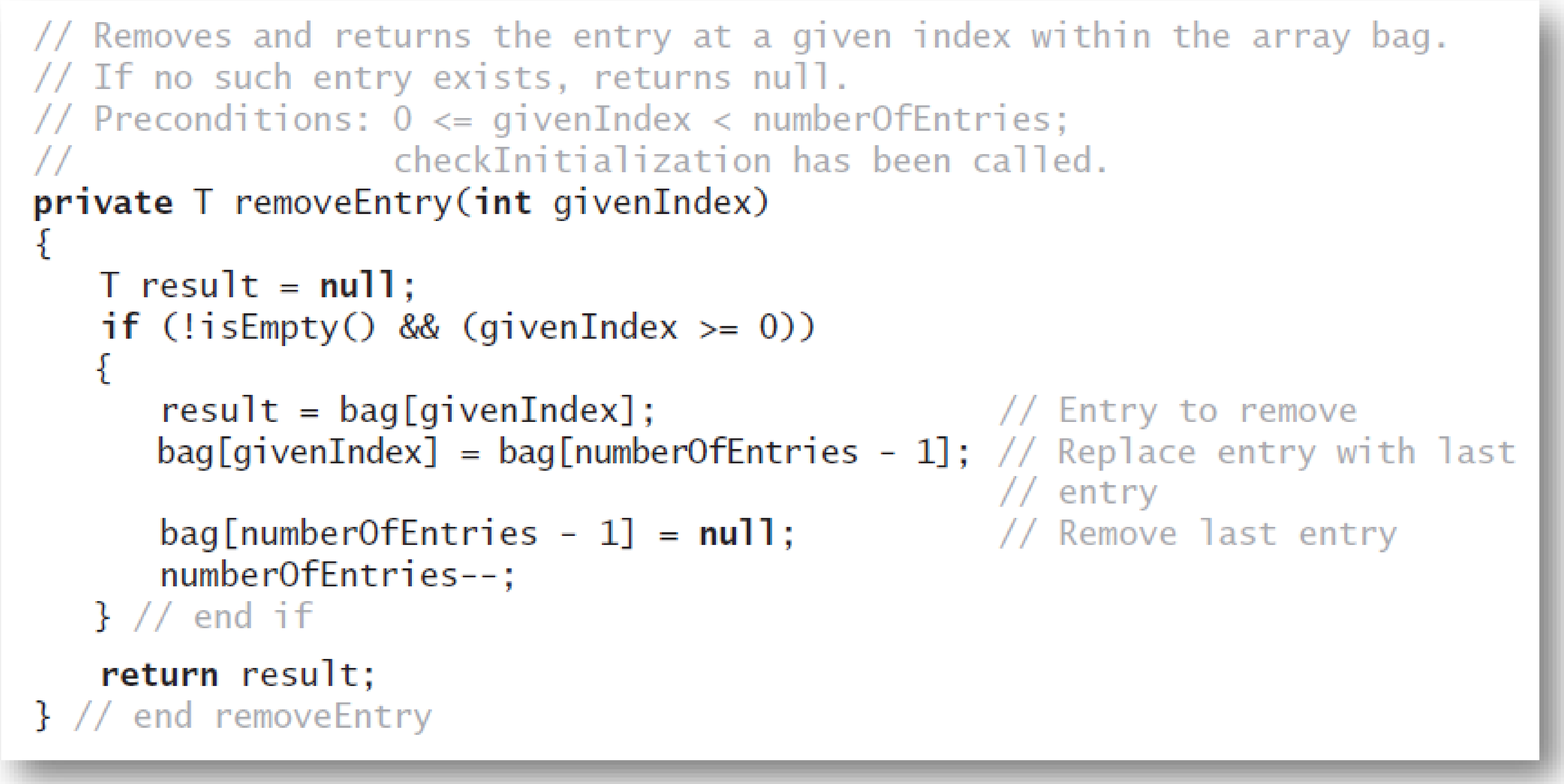
The second **`remove`** method



```java
/** Removes one occurrence of a given entry from this bag.
    @param anEntry  The entry to be removed.
    @return  True if the removal was successful, or false if not. */
public boolean remove(T anEntry)
{
   checkInitialization();
   int index = getIndexOf(anEntry);
   T result = removeEntry(index);
   return anEntry.equals(result);
} // end remove
```

Methods That Remove Entries

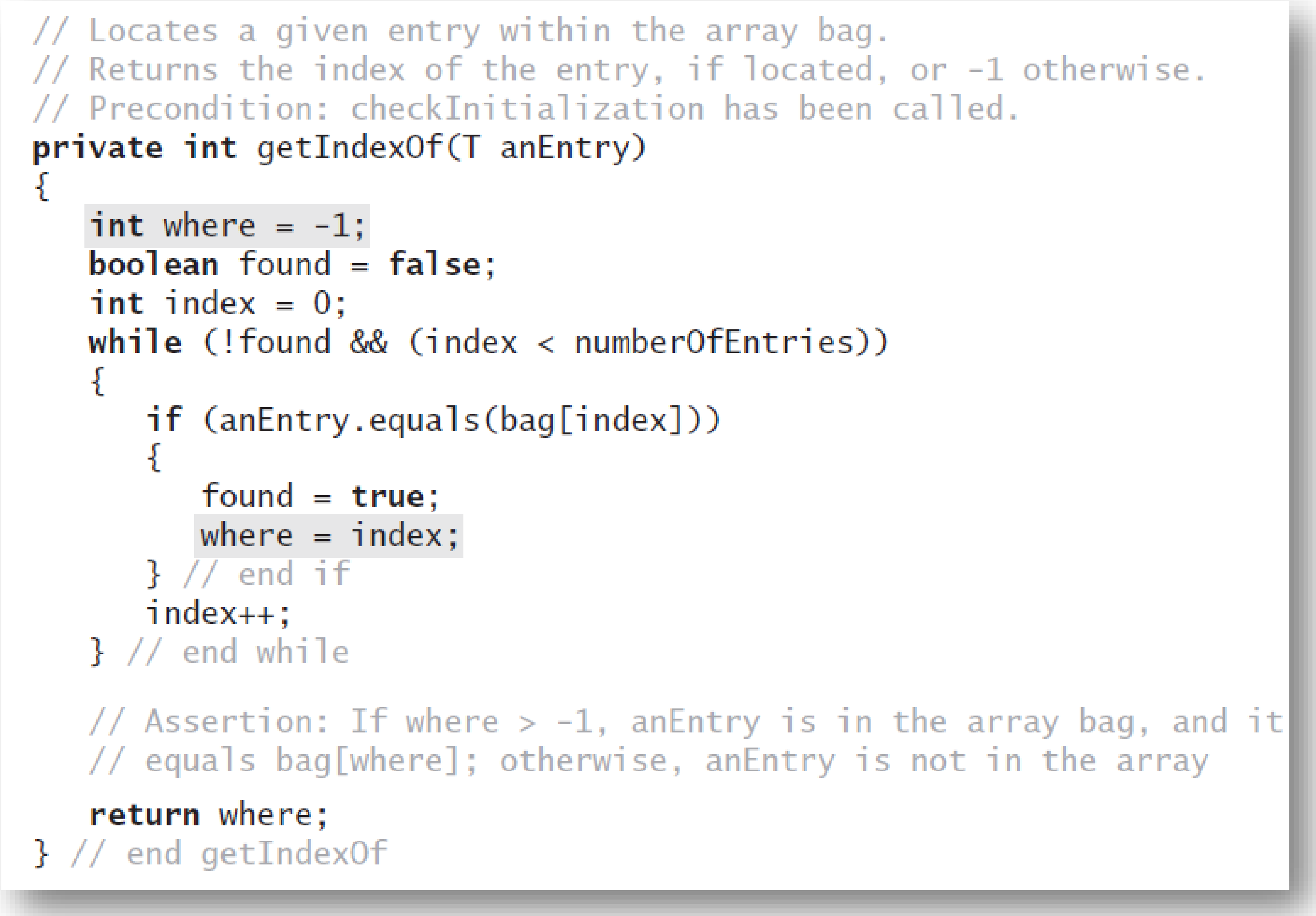
The `removeEntry` method



```
// Removes and returns the entry at a given index within the array bag.
// If no such entry exists, returns null.
// Preconditions: 0 <= givenIndex < numberOfEntries;
//                checkInitialization has been called.
private T removeEntry(int givenIndex)
{
   T result = null;
   if (!isEmpty() && (givenIndex >= 0))
   {
      result = bag[givenIndex];                 // Entry to remove
      bag[givenIndex] = bag[numberOfEntries - 1]; // Replace entry with last
                                                // entry
      bag[numberOfEntries - 1] = null;          // Remove last entry
      numberOfEntries--;
   } // end if

   return result;
} // end removeEntry
```

Methods That Remove Entries

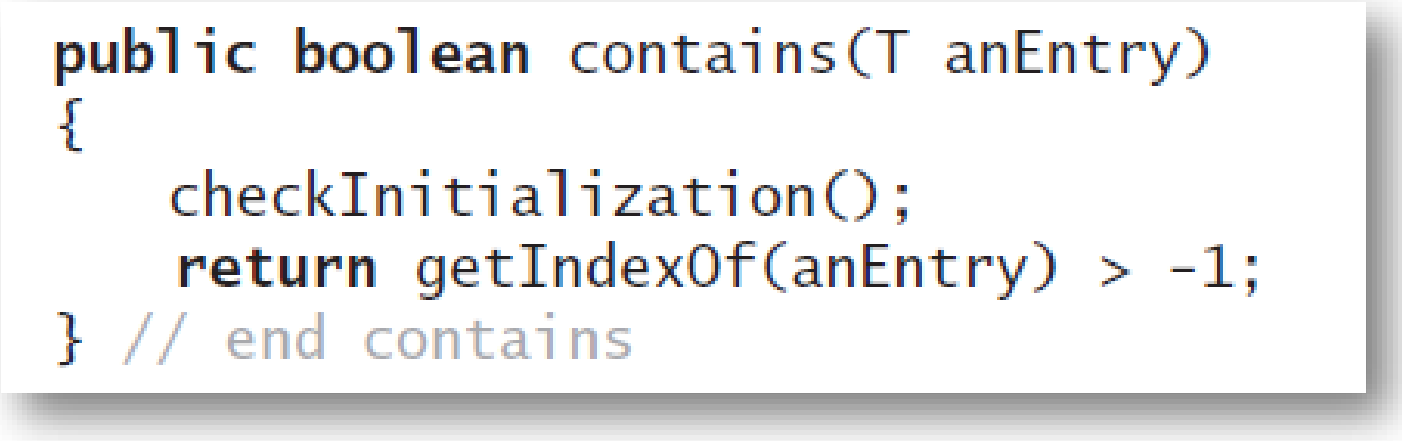
Definition for the method getIndexOf



```
// Locates a given entry within the array bag.
// Returns the index of the entry, if located, or -1 otherwise.
// Precondition: checkInitialization has been called.
private int getIndexOf(T anEntry)
{
   int where = -1;
   boolean found = false;
   int index = 0;
   while (!found && (index < numberOfEntries))
   {
      if (anEntry.equals(bag[index]))
      {
         found = true;
         where = index;
      } // end if
      index++;
   } // end while

   // Assertion: If where > -1, anEntry is in the array bag, and it
   // equals bag[where]; otherwise, anEntry is not in the array

   return where;
} // end getIndexOf
```

Methods That Remove Entries

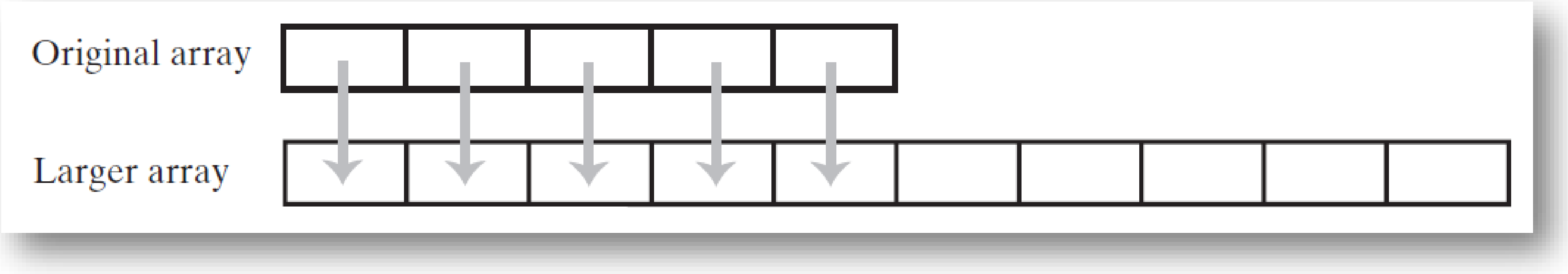
Revised definition for the method `contains`



```
public boolean contains(T anEntry)
{
    checkInitialization();
    return getIndexOf(anEntry) > -1;
} // end contains
```

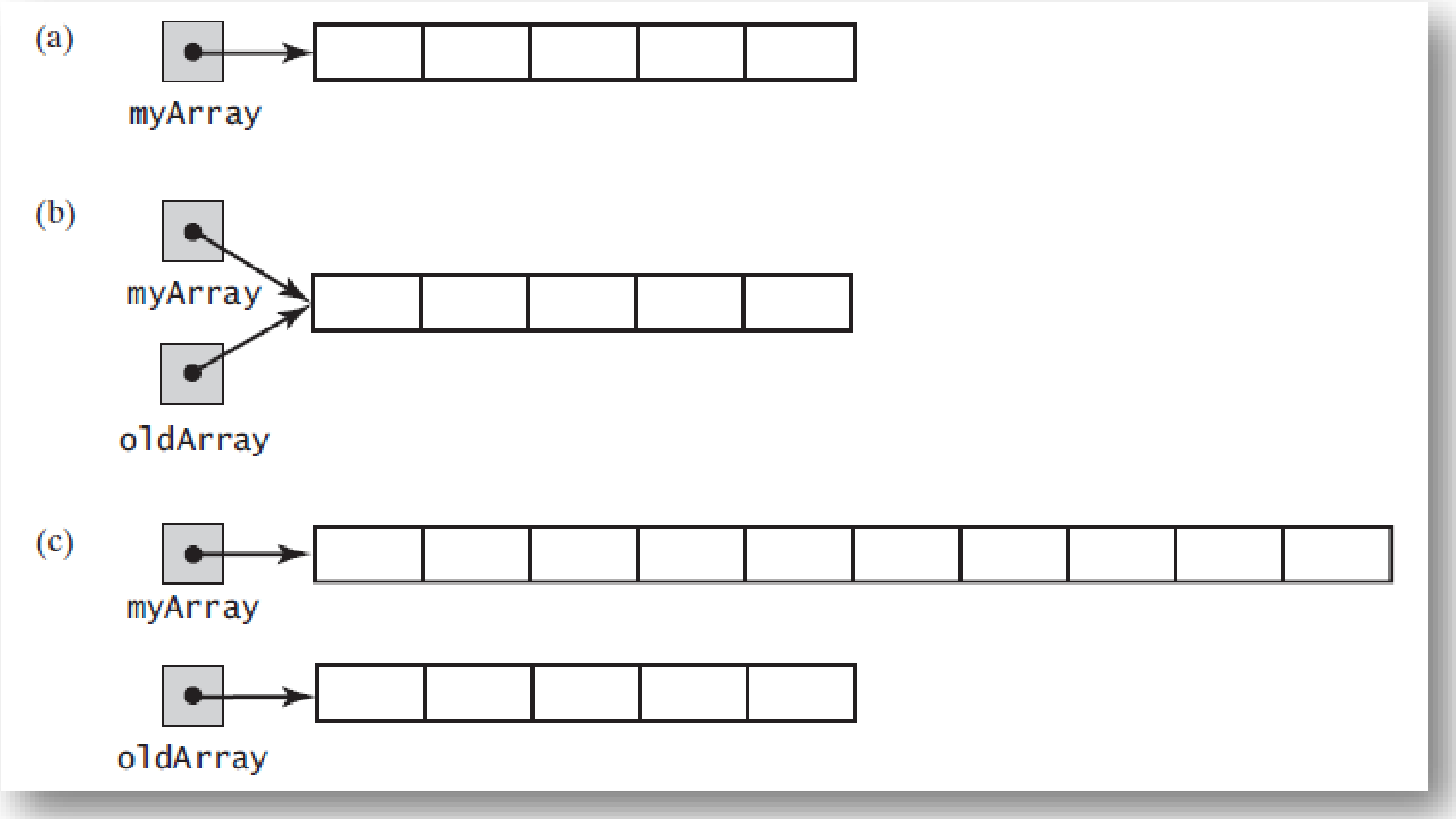
Using Array Resizing

Resizing an array copies its contents to a larger second array



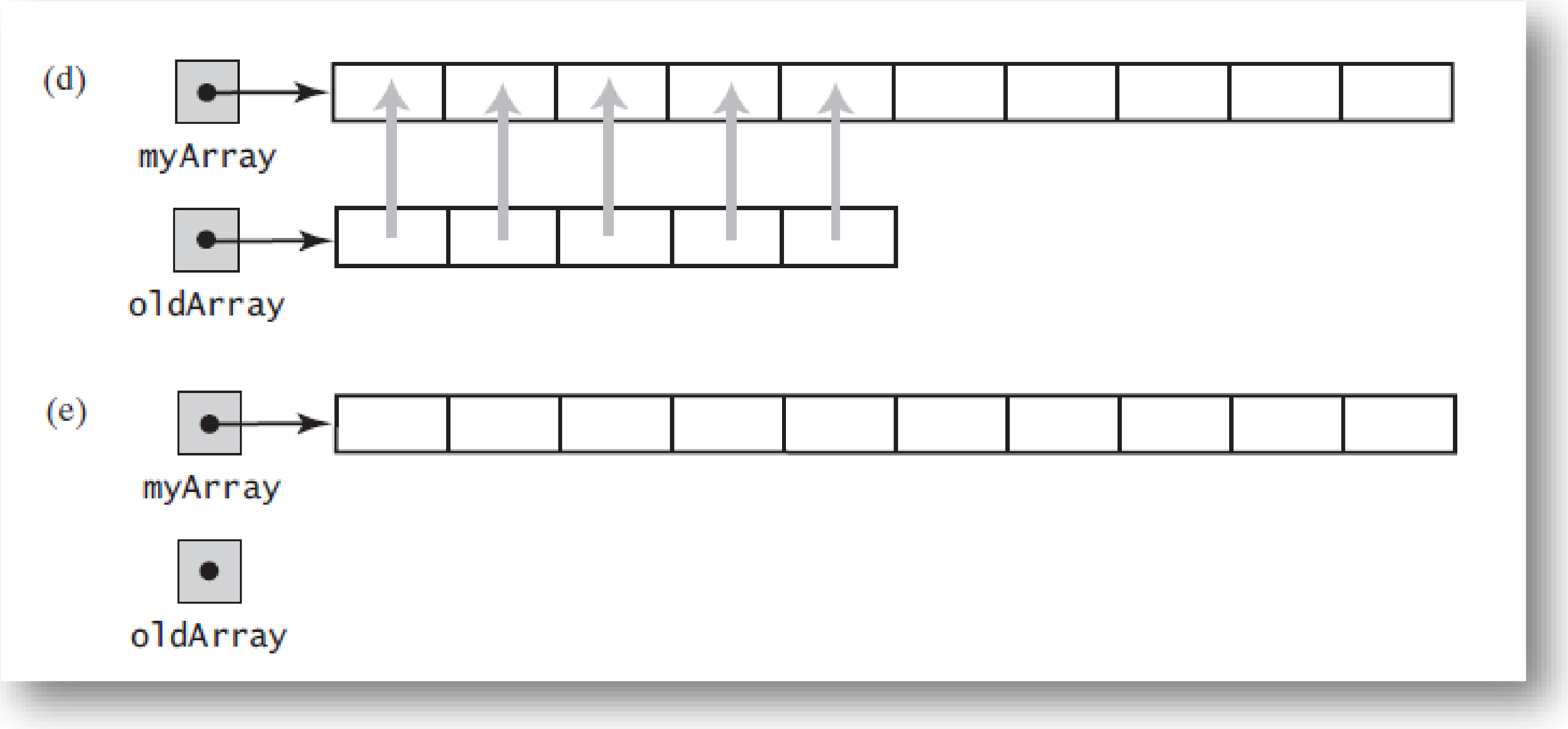
Using Array Resizing

(a) An array; (b) two references to the same array; (c) the original array variable now references a new, larger array;



Using Array Resizing

(d) the entries in the original array are copied to the new array; (e) the original array is discarded

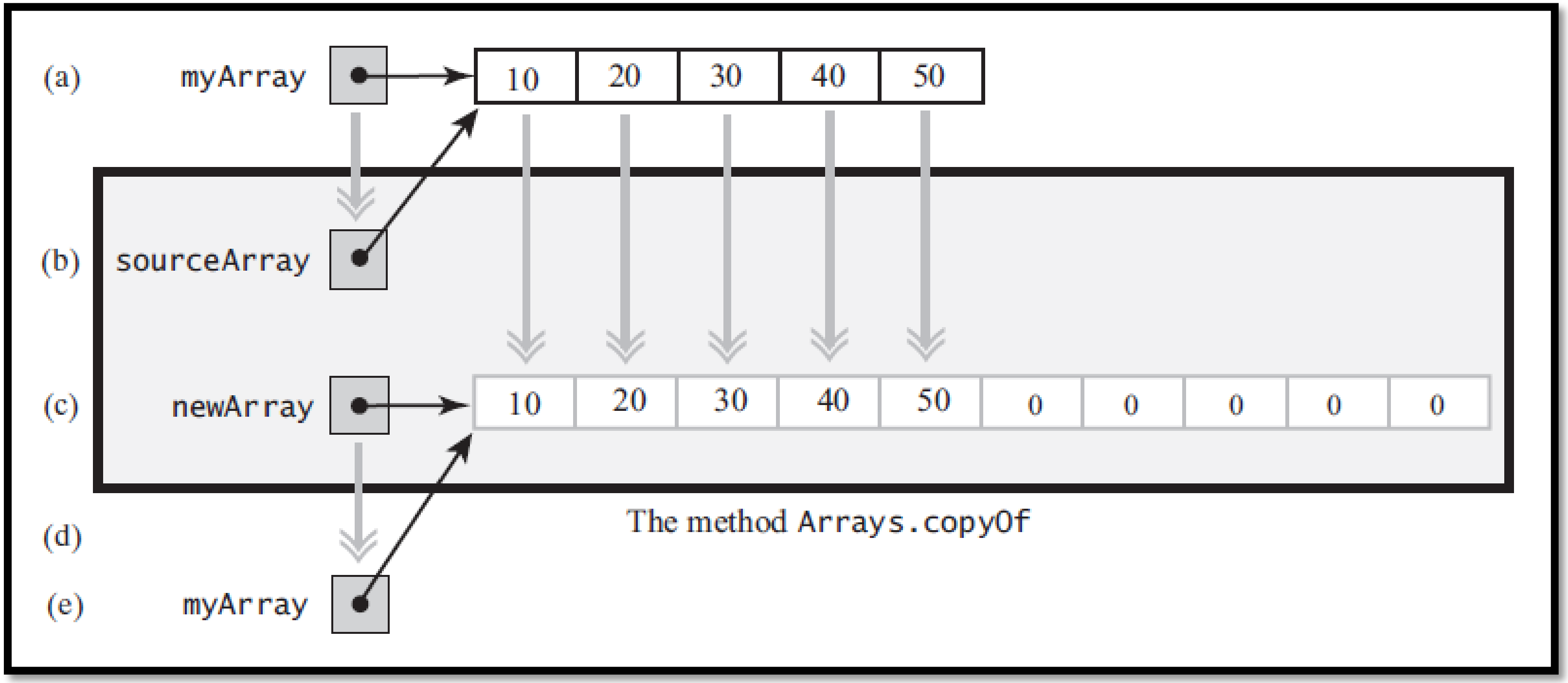


Using Array Resizing

The effect of the statement

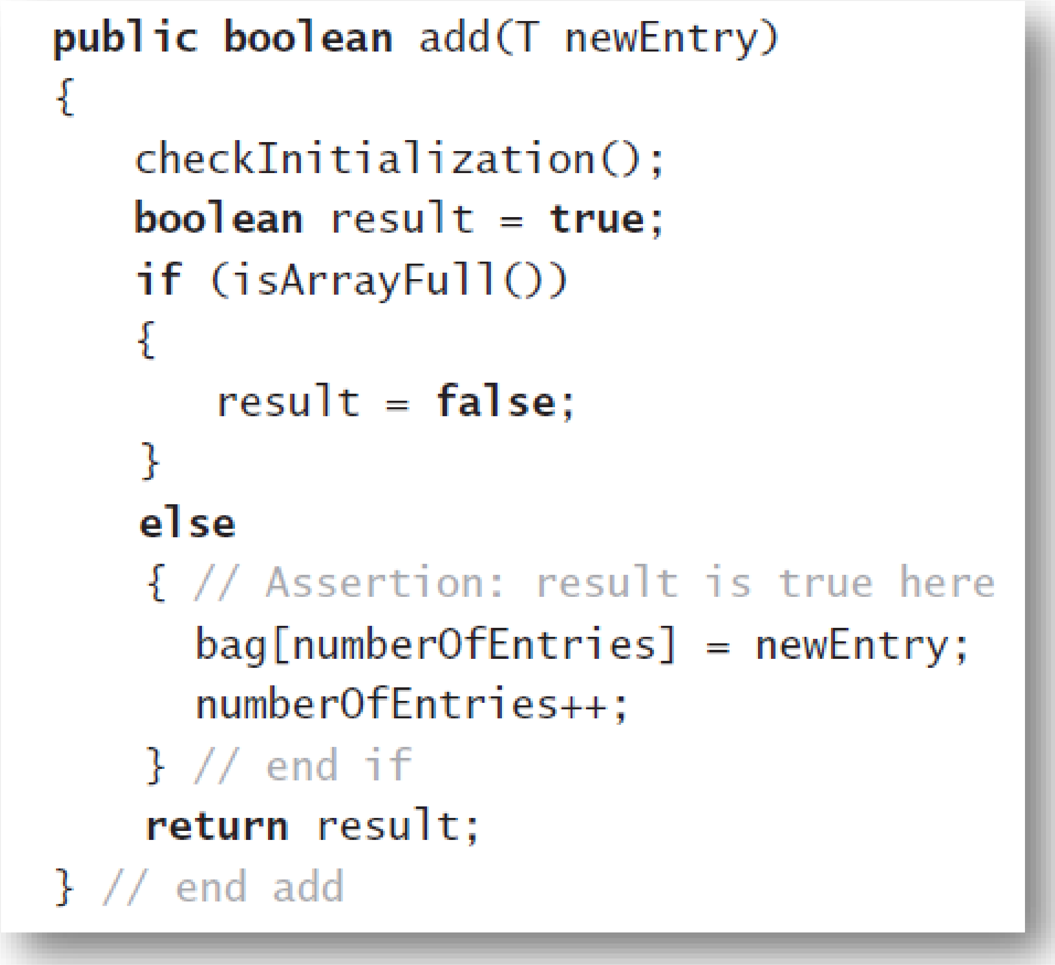
```
   myArray = Arrays.copyOf(myArray, 2 * myArray.length);
```

(a) The argument array; (b) the parameter that references the argument array; (c) a new, larger array that gets the contents of the argument array; (d) the return value that references the new array; (e) the argument variable is assigned the return value



New Implementation of a Bag

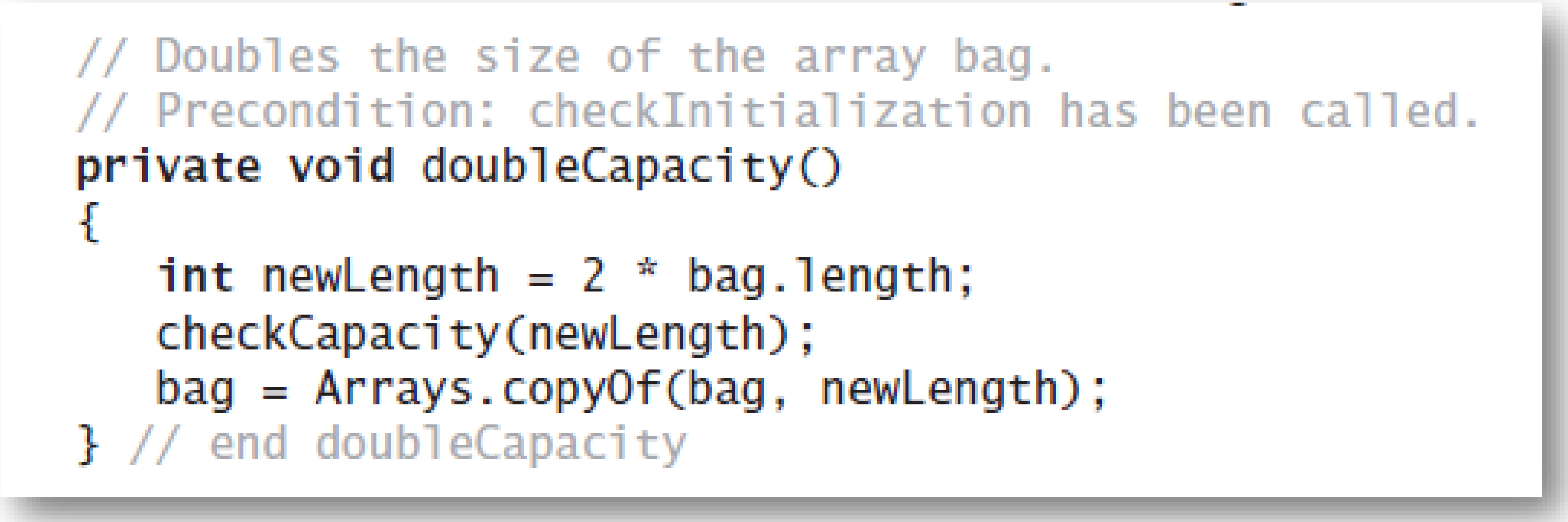
Previous definition of method `add`



```
public boolean add(T newEntry)
{
    checkInitialization();
    boolean result = true;
    if (isArrayFull())
    {
        result = false;
    }
    else
    { // Assertion: result is true here
       bag[numberOfEntries] = newEntry;
       numberOfEntries++;
    } // end if
    return result;
} // end add
```

New Implementation of a Bag

The method **doubleCapacity**



```
// Doubles the size of the array bag.
// Precondition: checkInitialization has been called.
private void doubleCapacity()
{
   int newLength = 2 * bag.length;
   checkCapacity(newLength);
   bag = Arrays.copyOf(bag, newLength);
} // end doubleCapacity
```

Using a Bag

A Bag is a simple ADT, but it can still be useful

- See examples in text
- Here is another simple one
  - A number of players “shout” Snap! each with a certain probability.
  - Add the player number to a Bag if she shouts.
  - Count the number of shouts in the Bag.

Pros and Cons of Using an Array

- Adding an entry to the bag is fast
- Removing an unspecified entry is fast
- Removing a particular entry requires time to locate the entry
- Increasing the size of the array requires time to copy its entries

Problems with Array Implementation

- Array has fixed size
- May become full
- Alternatively may have wasted space
- Resizing is possible but requires time overhead