4 Linear Data Storage

Linear Collections. What is a good way to store a sequence of elements? Fixed-size arrays are great, but they cannot efficiently support growing and shrinking.

Earlier in the book, we saw how a fixed-size array can be used to store a sequence of values. The fixed-size arrays fall short when the sequence we are storing can dynamically grow and shrink. This chapter explores two fundamental ideas for storing sequences while supporting dynamic size changes: linked lists and array-based lists. As we go through this chapter, it is vital to understand the strengths and limitations of each of these options.

4.1 A Linked Data Structure

Fixed-size arrays are extremely useful but they are limiting in at least one significant way: once created, they remain at that size and are costly to grow or shrink. Indeed, an important limitation of the fixed arrays is that resizing almost always means creating a new array and copying the data, which is expensive. We will now look at using references to build resizable collections—one where adding a new element will be easy and inexpensive.

Our initial goal will be to store a list of `int`s. We name the class an `IntNode` class since we refer to the holder of each data item as a node. Here is a basic structure, which is sufficient for the task but not handy to use:

```
class IntNode {
    int head;
    IntNode next;
}
```

In declaring this class, we are working with a mental model where a list is made up of a head element—stored in the `head` attribute—and has a link to the remainder of the list—given by the `next` attribute. This view can be seen as a recursive definition of a list: with `null` denoting the empty list, a list is either empty or a head followed by a list (i.e., the rest of the list).

We will now go ahead and build a list [7, 11, 9], like so:

```
IntNode list = new IntNode();
list.head = 7;
list.next = new IntNode();
list.next.head = 11;
list.next.next = new IntNode();
list.next.next.head = 9;
```

It helps to run this in a visualizer. The end result? Figure 4.1 shows a schematic depiction of the structure we have just created.

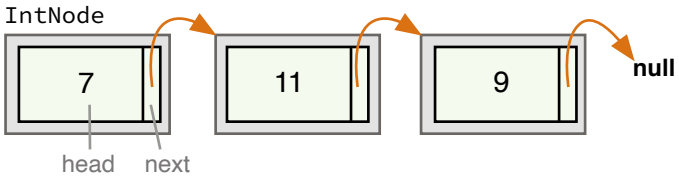


Figure 4.1: Bare-bones list of integers illustrated.

Notice that this is fundamentally a different organization of data from a (fixed) array. Whereas a fixed array stores data in consecutive memory, a linked structure forms a chain—each data item (in this case, an int) is kept in a container (known as a node) and we make a list by chaining them together, forming longer and longer chains, as lengthy as we wish.

Adding a constructor

Using the class in the current form is rather awkward. We will make it a little nicer by adding a custom constructor to the `IntNode` so that both the `head` and `next` variables can be conveniently set at creation.

What is a constructor, may you ask? Remember that the constructor of a class is called when an object of a class is created, so we can initialize the internal variables. Examples will make this abstract concept more concrete.

We will add a constructor to the `IntNode` class, like so:

```
public IntNode(int head, IntNode next) {
    this.head = head;
    this.next = next;
}
```

It is useful to note that in Java, a class's constructor is a method with the same name as the class itself and can bear as many parameters as one wants. In our case, we make our constructor take in two arguments: the value for `head` and the value for `next`. While it is possible to have fancy logic in the constructor, our constructor for `IntNode` simply sets the attributes `head` and `next` to the supplied values.

With this constructor, it is most natural to create the list backwards. If we think about it, when we `new` an `IntNode`, we do not know what comes next unless we build it from the back. Literally, this starts with the empty list and the list becomes progressively longer as one more item is tagged on to the front. Hence, we can write:

```
IntNode list = null;
list = new IntNode(9, list);
list = new IntNode(11, list);
list = new IntNode(7, list);
```

It helps to run this code in a visualizer. The end result is the same as before; however, this time we are building the list starting from the back.

Most useful lists can display its elements and know its size, so we will implement the following methods together:

- `public String toString()` returns the contents of the list. For example, at the end of the above example, `list.toString()` should return `"9, 11, 7"`.
- `public int size()` returns the length of the list (i.e., how many elements the list has).

When we say the list, we are referring to the list given by the object we are at. This has to be said because if we rewrite the above example more explicitly, we end up with 3 lists (each a sublist of another):

```
IntNode list0 = null;
IntNode list1 = new IntNode(9, list0);
IntNode list2 = new IntNode(11, list1);
IntNode list3 = new IntNode(7, list2);
```

Notice that `list3`'s `next` points to `list2`, which points to `list1` and, in turn, to `list0`. Therefore, the meanings of `toString` and `size` change with where we call them. For example, `list3.toString()` should return `"7, 11, 9"` but `list2.toString()` should be `"11, 9"`. Likewise, `list3.size()` should return 3 and `list1.size()` should be 1.

We will approach the implementation of these methods in two ways—recursively and iteratively.

Recursive Traversal

At the core of this endeavor is the ability to walk the whole list. It is often handy to think about this list recursively. We will begin by asking ourselves: what is a list? In this case, one convenient answer—as has been foreshadowed—is the following simple characterization:

> A list either **(a)** stores a single number (`int`) or **(b)** is a number (`int`) followed by a list (i.e., the rest of the list).

In our representation, a singleton list is one where `next` is `null` (e.g., `list1`). Otherwise, it is a number (stored in `head`) followed by a non-`null` list (as pointed to by `next`).

An Implementation of `toString`

In this view, the `toString` method has two cases:

- A base case of a singleton list: The list contains no more (i.e., `next` is `null`), so `toString` should just return the number at the head.
- Otherwise, we should return the number at the front, followed by the numbers in the rest of the list (How can we obtain this?)

One can use `Integer.toString(n)` to turn an integer `n` into a corresponding `String`.

```
public String toString() {
    if (this.next == null)
        return Integer.toString(this.head);
    else
        return Integer.toString(this.head) +
           ", " + this.next.toString();
}
```

An Implementation of size

To implement the `size` method, one can handle it in the same way as `toString`. There are two cases to consider. What should we return in each case?

- A base case of a singleton list: This clearly has size 1.
- Otherwise, the size is `1 +` the size of the rest of the list (which can be determined by calling its `size()` method on the rest.)

Hence, we have (only the `size` method is shown):

```
public int size() {
    if (this.next == null)
        return 1;
    else
        return 1 + this.next.size();
}
```

Iterative Traversal

The code so far has been recursive. Not that it is bad but we sometimes have reasons to write it in a nonrecursive manner. Here is a general template for walking such a list. For explicitness, we will use the while loop, though admittedly, one can clean it up a little bit using the for loop.

Using the while loop, there are 3 ingredients in walking such a list:

- a variable to keep track of where we currently are in the list (initially, this is set to the front of the list).
- a condition that checks whether we are at the end of the list.
- logic to move to the next node of the list.

In code:

```
// current stores where we currently are, it initially points
// to the start of the list.
IntNode current = ...;

// current is not null (i.e., not yet an empty list)
while (current != null) {
    // logic to do something with the current node
    // ...
    // ...

    // now, advance current to the next node
    current = current.next;
}
```

Compared to a standard index-based loop, the variable `current` is analogous to the index variable, e.g., `int i`. The while-condition is analogous to boundary checking, e.g., `i < array.length`. The logic to move to the next node is analogous to incrementing the index, e.g., `i++`.

Using this template (with a while loop), the `size` method, for instance, can be implemented as follows:

```
public int iterativeSize() {
    IntNode current = this;
    int totalSize = 0;

    while (current != null) {
        totalSize++;
        current = current.next;
    }

    return totalSize;
}
```

4.2 Evolving Rustic Lists

The list data structure we have just developed feels very primitive: there are no convenient functions, so using it requires understanding and manipulating the inside of the data structure ourselves. We will now improve upon it in several ways, both in terms of usability and performance. Ultimately, we will take it apart and put the pieces back together!

Improvement I: Encapsulation, Wrapping it Inside a Package

An important goal in object-oriented design is to prevent data from being accessed by the code outside the shield of a class. As a useful byproduct, this usually helps hide unnecessary details from the users and make our package more user-friendly.

Following this thinking, the first improvement we will bring to the table is that of repackaging the implementation, capturing it in a new class that hopefully will be easier to use and less error-prone when the users use it.

As shown in Code 4.1, this quick reorganization wraps the actual data storage node into a class. Instead of working directly with data storage nodes, users are provided with methods such as `addFirst` and `getFirst`. These changes make a few things easier:

- Creating a new list is easy: `new SLList()` and `new SLList(3)` will give you an empty list and a singleton list, respectively.
- The users do not have to manipulate the list themselves. This makes it much more natural to use. For example:

```
SLList L = new SLList(19);
L.addFirst(12);
L.addFirst(3);
int x = L.getFirst();
```

A little discussion is in order. Our upgrade created a new class that wraps inside it convenient methods. But then, why not just add an `addFirst` method to the `IntNode` class? It turns out there is no efficient way to do this. Importantly, an `IntNode` has no efficient way of knowing where the front of the list is. Hence, it is not practically possible to update the front of the list in this design.

Constructor Overloading. Java allows a class to have multiple constructors, known as constructor overloading. It is permitted as long as they have different signatures. This makes it possible to cater to different ways to creating an instance of the class.

Code 4.1: Encapsulating the nodes inside a class.

```
1  // wrap IntNode inside a new class
2  public class SLList {
3      // the front of the list, so users doesn't have to
4      // update this themselves. remember: IntNode was the
5      // class we wrote previously.
6      IntNode first;
7
8      // two constructors to make initializing a list easier
9      public SLList() { first = null; }
10     public SLList(int x) { first = new IntNode(x, null); }
11
12
13     // so that users don't have to manage the inner
14     // bookkeeping themselves
15     public void addFirst(int x) {
16         first = new IntNode(x, first);
17     }
18
19     public int getFirst() {
20         return first.head;
21     }
22 }
```

In summary, our first upgrade dealt with creating a "wrapper" that acts as a middleman between the users and the raw data structure.

Improvement II: More Access Control

We can further hide details from our users, reducing the contact surface where things can go wrong. In the implementation so far, someone can still write `L.first.next = L.first`* and bad things tend to follow after that. We can prevent such uncouth behaviors by limiting access.

By declaring a method or variable **`private`**, we prevent code in other classes from using that member (or constructor) of a class. We often do this for several reasons:

- There is less for the user of class to understand.
- It is safer for the class owner to change private (i.e., internal) methods

A New Trick: Nested Classes

While we are at it, we can tidy up the implementation a little. When a class does not stand on its own (e.g., an obvious subordinate of another class), it makes sense for aesthetic reasons and otherwise to nest it inside another class—and potentially hide it by making it private.

*This will cause the list to wraps back to itself like a twisted pretzel!

Code 4.2: Further access control via nesting and the keyword `private`.

```
public class SLList {

    private static class IntNode {
        int head;
        IntNode next;

        public IntNode(int h, IntNode r) {
            this.head = h; this.next = r;
        }
    }
    // the front of the list, so users doesn't have to
    // update this themselves. remember: IntNode was the
    // class we wrote previously.
    private IntNode first;

    // two constructors to make initializing a list easier
    public SLList() { first = null; }
    public SLList(int x) { first = new IntNode(x, null); }


    // so that users don't have to manage the inner
    // bookkeeping themselves
    public void addFirst(int x) {
        first = new IntNode(x, first);
    }

    public int getFirst() {
        return first.head;
    }
}
```

As an example, we will move our `IntNode` into `SLList` and hide it too because clearly no one else has to care about our "raw" data structure. This will also mean that no one else can manipulate our internal storage without our knowledge.

There is one more trick we can play. If the nested class never uses any instance variables or methods of the outer class, declare it **`static`**. By declaring it so, the class cannot access outer class's instance variables or methods. This results in a minor savings of memory. Code 4.2 shows what we have after these modifications.

Improvement III: A Faster `size()`

We will now bring back the `size` method. Previously, to determine the size of a list, we essentially walk the whole list (either recursively or iteratively). But

how efficient is `size` really? Suppose `size()` takes 2 seconds on a list of size 1,000. How long will it take on a list of size 1,000,000?

The trouble is, if the list has n elements, we expect the program to spend time proportional to n because we have to step on every element of the list. *Is there a better way?*

The crux that enables this is our airtight packaging (encapsulation). We know precisely when someone adds a new element: they have to call our `addFirst` method. This means we can just keep track of size as a separate property to avoid computing it every time it is asked.

The main idea is as follows:

- Keep a `size` instance variable, initially set to `0`.
- For every `addFirst`, increment it by one.

Hence, the `size()` method can simply return the stored size. Note that it no longer needs to walk the length of the list and will return instantaneously.

Improvement IV: More Functionality—`addLast`

We will now try to add more functionality: a new method `addLast`, which adds an item at the end of the list.

The main challenge is, in order to put a new item at the end of our list, we need to (a) know where the end of the list is and (b) change it to store an additional node. A natural idea is to walk the length of the list and stick a new node there, like so:

```
public void addLast(int x) {
    size += 1; // update the size property

    // This is to handle when first is null
    if (first == null) {
        first = new IntNode(x, null);
        return;
    }

    IntNode p = first;
    while (p.next != null) {
        p = p.next;
    }

    p.next = new IntNode(x, null);
}
```

But this is ugly! Why do we need a special `if` for the case when the list is empty. On the one hand, this is necessary because when the list is empty, adding to the end of the list is the same as adding to the front, which requires updating the variable `first`. On the other hand, though, imagine how this would be for more complex data structures. Could we possibly eliminate special cases? Now this should be our goal in life because we only have so much working memory. As you can relate, simple is better than complicated.

A Sentinel Trick

One common trick in programming linked data structures is the use of *sentinel nodes*. According to dictionary.com, a *sentinel* (**sen**-tn-l) is

> a person or thing that watches or stands as if watching.
>
> a soldier stationed as a guard to challenge all comers and prevent a surprise attack:*to stand sentinel*.

In a more technical sense, a sentinel node does not hold any meaningful data; it is there so that we can avoid corner cases.

Tips

Special cases are generally not that special. Eliminate special cases as much as possible. We only have so much working memory. Hence, simple is better than complicated.

The root of evilness in the above implementation is that the empty list is `null`, not even a proper `IntNode` object; therefore, accessing `first.next` causes an error. To fix this:

- We will make a special object that is always there. So then, the empty list is a list with just this special object (the sentinel node).
- The true first is what this sentinel node points to.

To implement this idea, we'll rename `first` to `sentinel` because it is no longer pointing to the true first. These are the relevant bits to implement the sentinel idea:

```
// COMMENTED OUT: Our previous "first"
// private IntNode first;

// Put in the sentinel node (instead of first).
private IntNode sentinel;

// the sentinel node doesn't store any meaningful value.
// we use -1 for an arbitrary/dummy value.
public SLList() { sentinel = new IntNode(-1, null); size = 0; }
public SLList(int x) {
    IntNode first = new IntNode(x, null);
    sentinel = new IntNode(-1, first);
    size = 1;
}

// UPGRADED addLast
public void addLast(int x) {
    size += 1;

    IntNode p = sentinel;
    while (p.next != null) {
      p = p.next;
    }
```

```
24
25      p.next = new IntNode(x, null);
26  }
```

As we move toward using a sentinel design, notice that `addFirst` has to be updated as well. We leave it as an exercise to the reader, see Exercise 4.2.

Improvement V: Faster `addLast`

As it stands, inserting an element at the back of the list takes much longer than inserting one at the front. This is pretty easy to see because while `addFirst` simply puts in a new node, `addLast` has to walk the whole length of the list before putting in the new element.

How can we modify our list data structure so that `addLast` *is also fast?*

One natural idea is to keep a reference pointing to the last node. If we had this, perhaps we would not need to walk the whole length anymore. (Readers should try this idea before reading on.)

However, there is a catch. We also plan to support the entire crew of `addFirst`, `getFirst`, `removeFirst`, `addLast`, `getLast`, `removeLast`. It is not difficult to convince ourselves that maintaining a reference to the last node alone does not allow us to easily remove the thing at the rear end.

Importantly, we need *an ability to walk backwards.*

Idea: Keep Forward and Backward Pointers

We introduce a back reference for each node, allowing us to walk both forward and backward. This design is known as a *doubly-linked list*.

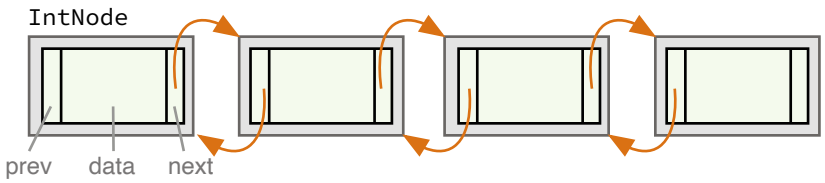


Figure 4.2: Doubly-linked list with pointers forward and backward.

In code, we can update the `IntNode` with forward and backward pointers as follows (only the member variables are shown).

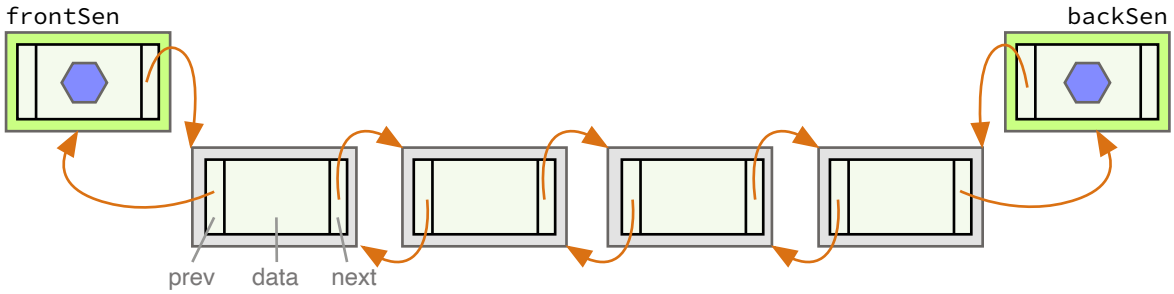
```
class IntNode {
    int data;
    IntNode next, prev;
}
```

Notice that in the process, the attributes have also been renamed: `data` stores the node's data; the pointers `next` and `prev` keep the pointers to the next and previous nodes, respectively.

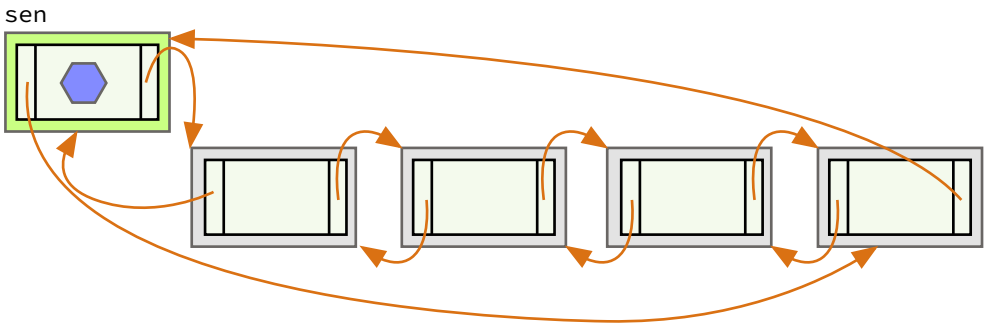
Exercise 4.8. will explore doubly-linked lists in greater detail. When implementing it, we will find that it has an annoying special case: `last`

sometimes points at the sentinel, and it sometimes points at a "real" node. Here are some ideas to help eliminate special cases and help with our sanity:

Two Sentinels. We can avoid this ambiguity (`last` sometimes pointing a real node and some other time pointing to the sentinel) by keeping two sentinels: one in the front (`frontSen`), one in the back (`backSen`). In this case, the back sentinel replaces `last`. Actual data lie between them. This means that the code will strive to maintain the following: as we walk from `frontSen` forward, we will hit all the data entries and eventually `backSen`. On the other hand, if we start from `backSen` and move backward, we will encounter all the data entries (in reverse) and eventually `frontSen`. This will look as follows:



Circular Sentinel. We can alternatively keep a ring. In this proposal, there is only one sentinel node (`sen`). Now `sen.next` points to the first real data node and `sen.prev` points to the last node. Note that if the list is empty, `sen.next` points to `sen` itself and `sen.prev` also points to `sen`. The main idea is that if we start walking forward from `sen`, we will hit every node in the list order and finally come back to `sen`. And if we start walking backward from `sen`, we will hit every node in reversed order and finally come back to `sen`. This will look as follows:



Generic Lists

Our list thus far only supports storing integers; it is not possible to store `Strings` or double values. There is a nice feature of Java, however, that will

remedy this: Java allows us to defer type selection until we use the class. For a concrete example, we will go back the very basic implementation before all the improvements and turn that into one that supports a generic type.

Code 4.3: Using Java generics to defer choosing an element type.

```
// the type T is a parameter, allowing us to defer type
// selection
public class SLList<T> {

    private class Node {
        // use T instead of the type int, so head stores
        // data of type T
        T head;
        Node next;

        public Node(T h, Node r) {
            this.head = h; this.next = r;
        }
    }
    // the front of the list, so users don't have to update
    // this themselves. remember: IntNode was the class we
    // wrote previously.
    private Node first;

    // two constructors to make initializing a list easier
    public SLList() { first = null; }
    public SLList(T x) { first = new Node(x, null); }


    // so that users don't have to manage the inner
    // bookkeeping themselves
    public void addFirst(T x) {
        first = new Node(x, first);
    }

    public T getFirst() {
        return first.head;
    }
}
```

Code 4.3 shows the adapted implementation. When we declare `SLList<T>`, that `T` is a type parameter. The type parameter can be thought of as a type variable. We can choose whatever name we want, but single capital letters are common. After that, whenever we want to refer to this type, just use `T`. For example, we write `T head;`—meaning declare a variable named `head` with type `T`, the type given by the parameter `T`.

We will now look at how to specify the parameter `T`. When we instantiate a class, do the following:

```
SLList<Double> list1 = new SLList<>();
```

That is, write out the desired type using type declaration—and use the empty diamond operator (i.e.,<>) during instantiation.

For technical reasons, the type parameter has to be a reference type. Fortunately, each of the primitive types has a corresponding reference type:

```
int -> Integer        double -> Double    char -> Character
boolean -> Boolean    long   -> Long      byte -> Byte
short  -> Short       float  -> Float
```

Java's Built-In `LinkedList`

It is worth mentioning that Java has a built-in linked list implementation, which is internally implemented as a doubly-linked list. This is the linked list class `LinkedList<T>`, which leaves a generic type `T` for us to specify its element type. As a doubly-linked list, operations at either end of the list tends to be constant time. But as we can intuitively see, working with elements elsewhere (e.g., in the middle) requires walking to that location, which is most likely proportional to how far the walk is. The built-in class offers many convenience methods and can be used in conjunction with utility functions from the `Collections` class.

4.3 A Resizable Array-Based Structure

Limitation of Linked Chains. We have just developed a list that supports adding to, getting, and removing the front or the back of the list efficiently—in time that is constant, independent of how long the list is. However, for the kind of linked lists we studied, getting or setting the i-th position can take a long time—we intuitively expect it to take time proportional to how far the element is from the closer end.

This is a fundamental limitation of a design that chains data elements together in a list. Specifically, only elements with direct pointers to them can be efficiently accessed. The other elements require some amount of "walking" in the list.

Supporting Fast Random Access. In an attempt to sidestep this limitation, we will look at a different list data structure that will allow for getting and setting data at any position in constant time. However, there is no free lunch; there are other features that we have to give up.

We may remember that the reason `get(i)` was slow for long lists is because it has to walk the list to the desired position. How can we fix this? At one end of the spectrum, we have the linked list structure, which arranges all the data elements in a chain. At the other end of the spectrum, we have the array structure, which lays out the data elements in consecutive memory.

For this reason, accessing any position of an array is extremely fast and is independent of the array size. That is, if `a` is an array, then `a[i]` can be retrieved in constant time, independent of how long `a` is.

Code 4.4: Basic implementation of an array-based list.

```
1  public class ArrayIntList {
2      private int[] items;
3      private int size;
4
5      public ArrayIntList() {
6          // Q: How big of an array to create?
7          items = new int[10];
8          size = 0;
9      }
10
11     public void addLast(int x) {
12         items[size] = x;
13         size++;
14     }
15
16     public int get(int i) {
17         return items[i];
18     }
19
20     public int size() {
21         return size;
22     }
23 }
```

How do we use the array to implement a list? At the core, Java, as well as most other modern languages, provides fixed-size arrays, upon which our implementation will be based.

Naïve Array List

Our goal is to build an array list out of the fixed-size array. Code 4.4 shows a basic implementation with limited use. For ease, we'll make an `int` array list for now. The same trick with generics we learned earlier can be readily applied to support arbitrary types. This code has several features and design points that should be discussed:

- The array `items` is used to store data items.
- For reasons that will be apparent, we have decided to keep the capacity of `items` separate from the size of our list. For this, we keep another attribute called `size`.

A General Outline. Together, we seek to maintain the following invariants, summarized in the figure below:

- `size` indicates the number of actual data items. More precisely, `items[0]`, `items[1]`, ..., `items[size-1]` are the actual data entries.
- This means, if `addLast` is called, it should go into position `items[size]`.

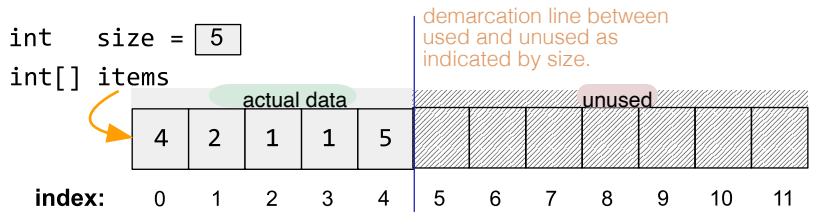


Figure 4.3: Schematic depiction of invariants in an array-based list.

Supporting `getFirst` and `getLast`. Where is the front of the list? Where is the last of the list? Do we know which index into `items` is the front? How about the last? Let's turn this into code for `getFirst` and `getLast`:

```
public int getFirst() {
    return items[0];
}
public int getLast() {
    return items[size - 1];
}
```

Supporting `removeLast`. How do we remove the last item? Let's try this out on your own. (Hint: adjust `size`.) After a moment's thoughts, here is a way to implement `removeLast` that respects the invariants laid out above:

```
public int removeLast() {
    int itemToRemove = items[size - 1];
    // good habit to clean up the unused spot
    items[size - 1] = 0;
    size--;

    return itemToRemove
}
```

Initial Array Size and Resizing

How big should `items` be? If we keep adding more and more items, the array will run out of space. We'll need to grow it. But by how much?

First thing first, how do we know the `items` array is already full? That is, we have no more room for more items. This is pretty easy to check:

```
if (size == items.length) {
    // it's full
}
```

More interestingly, perhaps, what to do when when the array is full and we wish to add more items? we need to make room for them. This is also pretty easy. The steps are:

- Allocate a new array (but how big?).
- Copy the items from the old array into the new array.

Writing this in code is straightforward. We will dedicate a function `grow` to it. We would call grow with a new capacity, one that is at least as large as before.

```java
private void grow(int capacity) {
    int[] expandedItems = new int[capacity];
    System.arraycopy(items, 0, expandedItems, 0, size);
    items = expandedItems;
}
```

In this implementation, `System.arraycopy` is a lower-level routine than `Arrays.copyOfRange`. It gives us more control; in many Java implementations, `copyOfRange` internally calls `System.arraycopy`.

The more difficult question is, *what capacity should it be resized to?*

Option I: Increase the capacity by 1. Let us start with `items` of capacity 0 and when it is full, grow the capacity by 1—that is, `grow(size+1)`. As a thought experiment, we ask: With this option, if 500 `addLast` starting from an empty list takes 2 seconds, how long do you think 5000 `addLast` will take?

Actually running a program can give a definitive answer. But mathematical analysis can also come in to help us here. Every time we `grow` to size k, we create a new array size k, copy $k-1$ items into this new piece, and deposit 1 new item (from `addLast`). The cost of doing all these is approximately $c \cdot k$ for some constant c, which depends on the machine, Java internals, etc. etc.

This means if we start with an empty list, calling `addLast` n times will require a total of

$$c \cdot 1 + c \cdot 2 + \ldots c \cdot n = c \cdot \frac{n(n+1)}{2},$$

where we have used the summation formula $1+2+3+\cdots+n = n(n+1)/2$.

The important thing to note at this point is that on average, the cost per `addLast` is

$$\frac{1}{n} \cdot \frac{c \cdot n(n+1)}{2} \approx c \cdot n/2.$$

It suffices to say, this is not constant—it takes longer on a longer list than a shorter one. More precisely, it would be prohibitively slow to call multiple `addLasts` in succession. This stands in sharp contrast with `addLast` in a doubly-linked list. But don't despair yet!

Option II: Double the Capacity. We will double the capacity every time the array `items` is full. That is, if we start with 1, it will grow to 2, to 4, to 8, etc. We note here that this is how Python and Java implement its list (Python's list and Java's `ArrayList`). Let us attempt to understand this mathematically. *How good is the doubling trick?*

Say we start with an array of capacity 1 and call `addLast` n times. For simplicity, suppose n is a power of 2, i.e., $n = 2^k$. The following observation is easy to see:

- We only `grow` the array at capacity $1, 2, 2^2, 2^3, \ldots, 2^k - 1$.
- Each time we grow from size 2^i to 2^{i+1}, it takes time around $c \cdot 2^{i+1}$ for some constant c (by a reasoning similar to what we did for Option I).

Hence, the total time spent is

$$\begin{aligned} & c \cdot 2^1 + c \cdot 2^2 + c \cdot 2^3 + \cdots + c \cdot 2^k \\ & \quad = 2c(1 + 2^1 + \cdots + 2^{k-1}) \\ & \quad = 2c(2^k - 1) \\ & \quad \leqslant 2cn, \end{aligned}$$

where we have used the geometric sum formula $1 + 2^1 + 2^2 + \cdots + 2^t = 2^{t+1} - 1$. If we mentally "redistribute" time evenly to all the operations, this means each `addLast` only costs about 2c, which is constant. This sense of redistribution of time is known as amortized time. Hence, the amortized time of `addLast` does not grow with the length of the array, effectively meaning each time we call `addLast`, we can think of it as taking constant time.

Discussion

In an array list, adding to the front (`addFirst`) will inevitably be slow. If we stick with this layout, it intuitively requires moving elements in the entire array to make room for the first slot. Note that so far the capacity tracks the number of elements rather closely. We can show that it is always within a factor of 2.

There is an additional tricky bit. How about `removeLast`? If we keep deleting elements from the array list, at one point we will be left with so much excess capacity. But should we "scale down" when we are not using most of the array? The short answer is yes. One strategy is the following: if less than 25% of the array is used, we halve the capacity. Further lessons on algorithms and data structures will give us a framework to reason about this and make clear why this is a good idea. Java has a built-in `ArrayList<T>` class, which uses this resizing strategy.

Exercises

Exercise 4.1. Consider the "rustic" `IntNode` class discussed at the beginning of the chapter. Write a method

```
public IntNode incrList(int delta)
```

that returns a new `IntNode`-list identical to this list, but with all values incremented by `delta`. Note that your method must not change the original list. Use either recursion or iteration.

Exercise 4.2. When we upgraded our `SLList` to use a sentinel, we left `addFirst` as an exercise to the reader. Update `addFirst` to work correctly with the sentinel design.

Exercise 4.3. Our discussion of singly-linked lists (`SLList`) contains several improvements to the basic implementation. It is time to use the fragments described earlier to put together a working `SLList` that uses a sentinel. Your `SLList` class will not keep a reference to the last node. It should be complete with the following constructors/methods:

- Two constructors: `SLList()` to construct an empty list, and `SLList(int x)` to construct a list with with an `int`, namely `x`.
- All of the following methods: `addFirst`, `addLast`, `getFirst`, `getLast`, `size`. The `size` method should be fast.

Exercise 4.4. Add a `public String toString()` method to the `SLList` class above. Feel free to change the internal `IntNode` class. Be reminded that it is possible to implement this without touching `IntNode` at all.

Exercise 4.5. Add a `public void removeFirst()` method to the `SLList` class above. This method removes the element at the front of the list. If the list is empty, it does nothing. How does it affect how we maintain size?

Exercise 4.6. Add two methods

```
public int get(int index);
public void set(int index, int newValue);
```

to the `SLList` class above. The `get` method returns the item at index `index` in the list (the front element has index is 0). The `set` method sets the item at index `index` in the list to a new value of `newValue`. For ease, assume that the index is valid.

Exercise 4.7. Add a `public void insert(int newValue, int k)` method to the `SLList` class above. The method `insert(newValue, k)` inserts `newValue` into the list at position `k`. This means, for example, `insert(x, 0)` will insert `x` at the front of the list. Because `insert` adds a new entry to the list, if the list's size was n prior, it will be $n + 1$ after.

Exercise 4.8. Implement a doubly-linked list by piecing together ideas described earlier. Name your class `DLList<T>` with a generic type parameter `T`. Your implement should follow the design of our `SLList`, which encapsulates internal details within the class. Make sure to use one of the two sentinel designs discussed earlier. Your class should be complete with the following constructors/methods:

- Two constructors: `DLList()` to construct an empty list, and `DLList(T x)` to construct a list with one element `x`.
- All of the following methods: `addFirst`, `addLast`, `getFirst`, `getLast`, `size`. Each of these should run in constant time, i.e., independent of how long the list is.

Exercise 4.9. Add support for adding an item to the front of our int array list by writing a public method:

```
public void addFirst(int x)
```

How long do you you think it takes to run this? Does it change with the length of the the list?

Exercise 4.10. Implement an array list class `class AList<T>` capable of storing elements of a generic type `T`, similar to what we did for the linked list. Your implementation should be complete with the same set of convenience methods. The general idea is to change `int[]` items to a generic `T[]` items. But there are some technical concerns.

How to make an array of generics? It is not possible to write new `T[2]`. Java simply doesn't allow that. To sidestep Java's craziness, you'll create an `Object` array and type-cast it as `T[]`. This will cause Java to give you a warning, but you can safely ignore it. In effect, if you want to create an array of type `T` of size `n` and store it as items, you'll write:

```
items = (T[]) new Object[n];
```

The joy of garbage collection. When you implement `removeLast`, it is important that you set the removed item to `null`, like so:

```
public T removeLast() {
    T itemToRemove = items[size - 1];
    items[size - 1] = null; // important: see below
    size--;

    return itemToRemove
}
```

While zeroing out the removed position was optional in the `int` array list implementation, it is crucial in the generic implementation. Here are a few things to understand: We don't explicitly free unused memory in Java. Java destroys unused objects automatically—and it knows this because the last reference to an object has been lost. Therefore, if you keep a reference to an otherwise unused object, Java will continue to keep this object around. We need to "free " this reference, so Java can properly collect garbage!

Chapter Notes

For an alternative exposition of linked- and array- list design and implementations, check out Goodrich et al.'s book [GTG14]. This chapter also touches on class design principles and tricks. Read Joshua Bloch's *Effective Java* [Blo08] and Herbert Schildt's *Java: A Beginner's Guide* [Sch18a] for further lessons. In an array-based list, proper resizing is key to efficiency. Learn about amortized analysis and the standard doubling trick that make each add and remove practically free over time [Cor+09; Eri19].

Interlude: Java Extras

This interlude describes extra Java features that will prove useful in implementing data structures and algorithms down the road. We will take a look at auto- boxing and unboxing (i.e., the ability to seamlessly go back and forth between primitive types and wrapped reference types), and more discussion of generics, including how one can put bounds on the type parameters. We close with another useful implementation technique: higher-order functions—passing a function as input to another function—and subtyping.

Autoboxing

We already knew that for each primitive type in Java, there is a corresponding reference type. For example, the `int` has an `Integer` counterpart. Why, then, is it possible to write the following code?

```
Integer iob = 100; // instead of = new Integer(100);
int i = iob; // instead of = iob.intValue();
```

Prior to Java 5, the process of wrapping a value inside an object (known as boxing) and extracting a wrapped value (known as unboxing) has to be carried out manually. Gladly, in modern Java, we no longer have to do that. Converting between the primitive types and their type-wrapper classes is more or less seamless.

However, this has some important implications for programmers like ourselves:

Unboxing `null`. Our type wrapper objects can be `null`. Unboxing a `null` value leads to a `NullPointerException` exception where we least expect it. Here are some examples:

```
static double triple(double x) { return 3*x; }
```

```
public static void main(String[] args) {
    Double x = null;
    double y = 2.0 + triple(x);
    System.out.println(y);
    System.out.println((x - 32)/9);
}
```

Try running this piece of code and we will be met with a surprise! The line `double y = 2.0 + triple(x);` looks so uninteresting that when Java informs us that there is a `NullPointerException` on that line, we could be puzzled. The issue is when `triple` is called, the actual static method expects a primitive `double`. Now Java, through auto-unboxing, will gladly unbox the reference `Double` variable `x`, except the value of `x` is `null`. Boom!

Boxed Objects Aren't Readily Computable. Java cannot perform computation directly on the wrapped objects, so if `iob` is an `Integer` object, the statement `ob++` involves more steps that it appears:

- unboxing,
- incrementing, and
- boxing
- (plus, throwing the original boxed object away).

The extra, but hidden, steps can lead to important performance bugs. The two loops below may seem to achieve the same effect, but they take vastly different amounts of time to run. Readers are encouraged to try them out:

```
final int N = 1_000_000_000;
for (Integer iob=0;iob<N;iob++) {} // loop, do nothing inside
for (int i=0;i<N;i++) {} // loop, do nothing inside
```

More Generics

In Java, generics essentially means type parameters, enabling us to write a program in which the type of data is left as a parameter. Let us quickly recap the basics of generics.

Generic Parameter(s) for Classes and Interfaces. To declare a class with generic parameters, write

```
class A<T>
class B<T, U>
class C<Key, Value, Time>
```

Modifiers such as `public`, `private`, etc. apply as usual. Then, the type parameters can be used inside the class that we declare. The same goes for interfaces—for example, we would write `interface I<T>` and `interface ExampleInterface<S, T>`.

To indicate that a generic class implements a generic inteface, write

```
class C<T> implements I<T> { ... }
```

where we mentally note that `T` is a type parameter of class `C`, which is then passed on to interface `I`.

Every reference type qualifies as a generic parameter. The same does not work with primitive types, but then we have type wrappers. For further illustration:

```
List<Integer> list = new ArrayList<>(); // OK
List<int> badList = new ArrayList<>(); // Nope
```

The Diamond Operator <>. Starting in Java 7, Java has support for the diamond operator <>, which adds type inference and enables programmers to save quite a bit of typing. We start with a basic example:

```
// without the diamond operator
List<Integer> listA = new ArrayList<Integer>();
// with the diamond operator
List<Integer> listB = new ArrayList<>();
```

From the declaration that we provide, Java can infer the most suitable type to be used for constructing the `ArrayList`. Hence, we can drop the explicit type declaration, reducing verbosity.

Generics in Methods and Constructors. It is possible to have generics for static methods, constructors, and (regular) methods (in addition to what the class has already declared). Below is an example:

```
class MyClass<T> {
    <ItemT> MyClass(ItemT arg) { ... }  // constructor
    static<X> staticMethod(X arg) { ... } // static method
    <X> regularMethod(X arg) { ... } // regular method
}
```

Notice that additional type parameters are declared at the start of the expression that declares that particular construct.

Bounded Types and Wildcards

Bounded Types. We often wish to limit the types that can be passed to a type parameter. For example, we would like a generic type `T` to be only numbers, not just any type. Java allows us to do this: write

```
<T extends superClass>
```

to indicate that only `T` that is a `superClass` will be permitted. Here is a concrete example:

```
class NumberFun<T extends Number> {
    T number;
    NumberFun(T num) { number = num; }
    double getFractional() {
        return number.doubleValue() - number.intValue();
    }
}
```

In the example above, `NumberFun` will accept any type `T` as long as `T` has `Number` as a superclass. By this virtue, we know that the type `T` inside this class will have all the methods/properties of `Number`. This is how we are able to write code that uses the methods `doubleValue` and `intValue`, which are guaranteed to exist in every class that is a `Number`.

Before moving on, we will note that each type parameter can have its own type bound.

Wildcards. Suppose we wish to extend the class `NumberFun` above with a method `integralEqual(that)` that compares whether the integral part of this object is that same as the integral part of that object. Our first attempt might be to write

```
boolean integralEqual(NumberFun<T> that) {
    return this.number.intValue() == that.number.intValue();
}
```

This only works in a limited sense: we can only compare `NumberFun<T>` of the same underlying number type `T`. That is to say, we cannot compare a `NumberFun<Double>` against a `NumberFun<Integer>`.

It is possible to fix this, however. The body of our code above, in fact, does not care what kind of number it is as long as it is a `Number` and hence has a method `intValue`. The trick is to use the concept of a wildcard. We will rewrite it as follows:

```
boolean integralEqual(NumberFun<?> that) { ... }
```

The body of this method stays the same. But with the question mark (`?`), `NumberFun<?>` matches any underlying type.

Higher-Order Functions and Subtyping

To motivate this discussion, we will start with a case study of writing a generic function `maxIndex` that finds the index that stores the largest element in an array. This is such a common routine that should only need to be written once and used essentially everywhere. Below is a straightforward implementation for the `int[]` array:

```
int maxIndex(int[] items) {
    if (items.length == 0)
        return -1;
    int maxIndex = 0;
    for (int index=0;index<items.length;index++) {
        if (items[index] > items[maxIndex])
            maxIndex = index;
    }

    return maxIndex;
}
```

The trouble with this code is that we cannot quite reuse it with other types. For example, suppose we have an array of `Cat[]`, where the `Cat` class has the code below:

```
public class Cat {
    private String name;
    private int weight;
    private int age;

    public Cat(String name, int weight, int age) {
        this.name = name;
        this.weight = weight;
        this.age = age;
    }

    public String getName() { return name; }
    public int getWeight() { return weight; }
    public int getAge() { return age; }
}
```

There are a few issues at hand: First, it is unclear how to compare two instances of `Cats`—do we mean heavier (i.e., a higher weight value)? Or do we mean the name appearing later in the lexicographical sense? Second, even if the intent is clear—for example, we wish to compare them by weight—we still have to work to say that.

To lay out design choices, we put on pause the `maxIndex` discussion and toy with a much simpler function `max`, which returns the larger of the two objects. Below are two compelling ideas for expressing how two objects are to be compared:

- **Idea #1: Explicit higher-order comparison function.** In this proposal, we pass in a comparison function to the function directly. Hence, the shape of `max` will be as follows (in pseudocode):

  ```
  def max(x, y, compare):
      if compare(x, y) == LARGER:
          return x
      else:
          return y
  ```

- **Idea #2: Subtyping polymorphism comparison.** Differently, in this version, we insist that the objects we are working with have certain methods that allow for comparing them. Hence, the shape of `max` in this case will be as follows (in pseudocode):

  ```
  def max(x, y):
      if x.largerThan(y):
          return x
      else:
          return y
  ```

Quick Discussion. Before diving into the technical maneuvers required to implement these ideas, we discuss their merits and drawbacks and when each proposal should be used. Using the explicit higher-order function approach, we can choose the comparison function, leading to more control and explicitness. In contrast, in the subtyping approach, the object itself makes the choice, leading to more concise code provided that the default choice is

the right choice for our code. In general, the subtyping approach is often used to provide a default comparator, whereas the higher-order function approach supplements it when users want to specify their own comparator.

Higher-Order Functions: The User Provides A Comparator

A *higher-order function* (HoF) is a function that treats another function as its (input) data. How can we implement and use functions that take as input another function? There are two stories here: the story prior to Java 8 and the story in Java 8 and on. We will talk about both mechanisms—sadly, the ancient syntax, as annoying it may be, is still prevalent.

Prior to Java 8. Prior to Java 8, there are no reference types pointing to functions. What this means is that we could not write a function that has a "function" type, as there was simply no data type for functions.

We can, however, work around this limitation by defining our own interface. We then represent a concrete function by writing a method that conforms to this interface. For a simple example, we will begin by writing an interface that defines any function that takes in an integer and returns an integer—an `IntUnaryFunction`:

```
public interface IntUnaryFunction {
    int apply(int x);
}
```

Remember that an interface defines, in essence, a type—and a contract specifying what methods must exist in the class implementing it. In the present case, this interface says a class satisfying the interface must have a public method `apply` that takes as input an `int` and returns an `int`. When we want to pass a function (`int` to `int`) into another function, we will implement it as this `apply` method. Below is an example that implements a concrete function in a class which implements this interface:

```
class DoubleFunction implements IntUnaryFunction {
    public int apply(int x) { return 2*x; }
}
```

We now give an example of a function that uses it:

```
int twice(IntUnaryFunction f, int x) {
    return f.apply(f.apply(x));
}
// Example Usage
System.out.println(twice(new DoubleFunction(), 5));
```

The intention of the `twice` method is to take in a function `f` and an input integer `x`, and return the result of apply `f` repeatedly twice to `x` (i.e., $f(f(x))$). Notice that by design, `f.apply` is the act of calling `f`. In the example usage, we will see 20 printed out.

The New Way: Java 8 and On. The situation in Java 8 and later is much more pleasant. There are now reference types to refer to functions. The code below shows how we can reference functions and pass them to another function:

```
import java.util.function.*;
class HoFDemoJava8 {
    static int doubleFunc(int x) { return 2*x; }
    static int twice(Function<Integer, Integer> f, int x) {
        return f.apply(f.apply(x));
    }

    public static void main(String[] args) {
        int result = twice(HoFDemoJava8::doubleFunc, 5);
        System.out.println(result);
    }
}
```

Several features should be noted: First, one can implement a function that will be passed into another function as an ordinary method, in this case a static method. Java 8 and on have a concept of functional references and the expression `HoFDemoJava8::doubleFunc` yields the reference to this function.

Second, the functional types are part of `java.util.function.*`. In this case, the function type we are after is a function from integers to integers. However, generic types have to be reference types, so we use the `Integer` type for integers, resulting in `Function<Integer, Integer>`.

Writing `maxIndex` using HoF. If we remember how `maxIndex` was written for the `int[]` array, the main difference for a generic array is that we need a different way to compare two given items: Is object `a` larger than object `b`? By asking the user to pass in a comparison function, our `maxIndex` can use it to make comparisons appropriately. We design the comparison function to take two objects `a` and `b`, and return whether the former object `a` is larger than the latter object `b`, leading to the following method declaration:

```
static<T> int maxIndex(
    T[] items,
    BiFunction<T, T, Boolean> isLarger) { ... }
```

where the `BiFunction<...>` type declaration represents a function that takes two arguments (both of type `T`) and returns a boolean value. The body of the function stays mostly the same; the only change is to the line that performs the comparison:

```
static<T>
int maxIndex(T[] items, BiFunction<T, T, Boolean> isLarger) {
    if (items.length == 0)
        return -1;
    int maxIndex = 0;
    for (int index=0;index<items.length;index++) {
        if (isLarger.apply(items[index], items[maxIndex]))
            maxIndex = index;
    }

    return maxIndex;
}
```

This means that `maxIndex` will be replaced whenever the item at `index` is larger than the item at `maxIndex` according to `isLarger`.

To use this `maxIndex` implementation, we can invoke it like the following example shows:

```
class MaxDemo {
    static boolean isLargerByWeight(Cat x, Cat y) {
        return x.getWeight() > y.getWeight();
    }
    public static void main(String[] args) {
        Cat[] items = ... // list of cats omitted
        int sampleRet;
        sampleRet = maxIndex(items, Maximum::isLargerByWeight);
        sampleRet = maxIndex(
            items,
            // OR: declare a function in place
            (Cat x, Cat y) -> x.getWeight() > y.getWeight()
        );
    }
}
```

Subtyping: The User Provides A Default Comparator

Often, there is a default way to compare objects. For the purpose of this discussion, it is obvious that for cats, we should be comparing weight. Therefore, we wish to be able to write the following line when we make a comparison:

```
if (items[index].isLargerThan(items[maxIndex]))
```

The trouble is, how can we know whether the item object has a `.isLargerThan`? Because our implementation is supposed to work for every object type, some objects—but not all—will have this method. For this code to work, we need a way to ensure that the item objects we are working with have the method. The trick is to tell Java that the kind of objects that we support is exactly those that have `.isLargerThan`. We will do this in two steps through the help of Java interfaces:

- define an interface that promises such a method; and
- tell Java that you only want items of that kind.

Similar to what we have seen before, the interface below promises a method `isLargerThan` that takes in an argument of type `T` and returns a boolean:

```
public interface HasIsLarger<T> {
    boolean isLargerThan(T that);
}
```

The next piece of the puzzle is to write `maxIndex` to only accept items that conform to this interface. We use the following familiar declaration:

```
static<T extends HasIsLarger<T>> int maxIndex(T[] items) {...}
```

This declaration says the type for items isn't any `T` but rather any `T` that satisfies the `HasIsLarger<T>` interface. Below is the rest of the implementation:

```
static<T extends HasIsLarger<T>> int maxIndex(T items[]) {
    if (items.length == 0)
        return -1;

    int maxIndex = 0;
    for (int index=0;index<items.length;index++) {
        if (items[index].isLargerThan(items[maxIndex]))
            maxIndex = index;
    }

    return maxIndex;
}
```

To complete this example, we show how to update the `Cat` class to comform to the `HasIsLarger` interface. There are only two simple changes: (i) declare that the class will implement the interface, and (ii) implement the promised method—i.e., `isLargerThan`. The code below shows the class declaration line and the added method; the remaining lines are omitted.

```
public class Cat implements HasIsLarger<Cat> {
    // existing code omitted
    public boolean isLargerThan(Cat that) {
        return this.weight > that.weight;
    }
}
```