

13.14 The class hierarchy

In Java, all classes extend some other class. The most basic class is called `Object`. It contains no instance variables, but it does provide the methods `equals` and `toString`, among others.

Many classes extend `Object`, including almost all of the classes we have written and many of the built-in classes, like `Rectangle`. Any class that does not explicitly name a parent inherits from `Object` by default.

Some inheritance chains are longer, though. For example, in Appendix D.6, the `Slate` class extends `Frame`, which extends `Window`, which extends `Container`, which extends `Component`, which extends `Object`. No matter how long the chain, `Object` is the ultimate parent of all classes.

All the classes in Java can be organized into a “family tree” that is called the class hierarchy. `Object` usually appears at the top, with all the “child” classes below. If you look at the documentation of `Frame`, for example, you will see the part of the hierarchy that makes up `Frame`’s pedigree.

13.15 Object-oriented design

Inheritance is a powerful feature. Some programs that would be complicated without it can be written concisely and simply with it. Also, inheritance can facilitate code reuse, since you can customize the behavior of build-in classes without having to modify them.

On the other hand, inheritance can make programs difficult to read, since it is sometimes not clear, when a method is invoked, where to find the definition. For example, one of the methods you can invoke on a `Slate` is `getBounds`. Can you find the documentation for `getBounds`? It turns out that `getBounds` is defined in the parent of the parent of the parent of the parent of `Slate`.

Also, many of the things that can be done using inheritance can be done almost as elegantly (or more so) without it.

13.16 Glossary

object method: A method that is invoked on an object, and that operates on that object, which is referred to by the keyword `this` in Java or “the current object” in English. Object methods do not have the keyword `static`.

class method: A method with the keyword `static`. Class methods are not invoked on objects and they do not have a current object.

current object: The object on which an object method is invoked. Inside the method, the current object is referred to by `this`.

this: The keyword that refers to the current object.

implicit: Anything that is left unsaid or implied. Within an object method, you can refer to the instance variables implicitly (without naming the object).

explicit: Anything that is spelled out completely. Within a class method, all references to the instance variables have to be explicit.

13.17 Exercises

Exercise 13.2

Transform the following class method into an object method.

```
public static double abs (Complex c) {  
    return Math.sqrt (c.real * c.real + c.imag * c.imag);  
}
```

Exercise 13.3 Transform the following object method into a class method.

```
public boolean equals (Complex b) {  
    return (real == b.real && imag == b.imag);  
}
```

Exercise 13.4

This exercise is a continuation of Exercise 9.3. The purpose is to practice the syntax of object methods and get familiar with the relevant error messages.

- a. Transform the methods in the **Rational** class from class methods to object methods, and make the necessary changes in **main**.
- b. Make a few mistakes. Try invoking class methods as if they were object methods and vice-versa. Try to get a sense for what is legal and what is not, and for the error messages that you get when you mess things up.
- c. Think about the pros and cons of class and object methods. Which is more concise (usually)? Which is a more natural way to express computation (or, maybe more fairly, what kind of computations can be expressed most naturally using each style)?

Chapter 14

Linked lists

14.1 References in objects

In the last chapter we saw that the instance variables of an object can be arrays, and I mentioned that they can be objects, too.

One of the more interesting possibilities is that an object can contain a reference to another object of the same type. There is a common data structure, the **list**, that takes advantage of this feature.

Lists are made up of **nodes**, where each node contains a reference to the next node in the list. In addition, each node usually contains a unit of data called the **cargo**. In our first example, the cargo will be a single integer, but later we will write a **generic** list that can contain objects of any type.

14.2 The Node class

As usual when we write a new class, we'll start with the instance variables, one or two constructors and `toString` so that we can test the basic mechanism of creating and displaying the new type.

```
public class Node {
    int cargo;
    Node next;

    public Node () {
        cargo = 0;
        next = null;
    }

    public Node (int cargo, Node next) {
        this.cargo = cargo;
    }
}
```

```

        this.next = next;
    }

    public String toString () {
        return cargo + "";
    }
}

```

The declarations of the instance variables follow naturally from the specification, and the rest follows mechanically from the instance variables. The expression `cargo + ""` is an awkward but concise way to convert an integer to a `String`.

To test the implementation so far, we would put something like this in `main`:

```

Node node = new Node (1, null);
System.out.println (node);

```

The result is simply

1

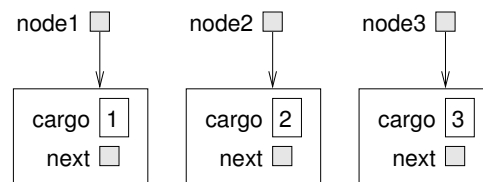
To make it interesting, we need a list with more than one node!

```

Node node1 = new Node (1, null);
Node node2 = new Node (2, null);
Node node3 = new Node (3, null);

```

This code creates three nodes, but we don't have a list yet because the nodes are not **linked**. The state diagram looks like this:



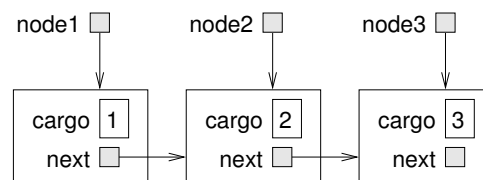
To link up the nodes, we have to make the first node refer to the second and the second node refer to the third.

```

node1.next = node2;
node2.next = node3;
node3.next = null;

```

The reference of the third node is `null`, which indicates that it is the end of the list. Now the state diagram looks like:



Now we know how to create nodes and link them into lists. What might be less clear at this point is why.

14.3 Lists as collections

The thing that makes lists useful is that they are a way of assembling multiple objects into a single entity, sometimes called a collection. In the example, the first node of the list serves as a reference to the entire list.

If we want to pass the list as a parameter, all we have to pass is a reference to the first node. For example, the method `printList` takes a single node as an argument. Starting with the head of the list, it prints each node until it gets to the end (indicated by the `null` reference).

```
public static void printList (Node list) {
    Node node = list;

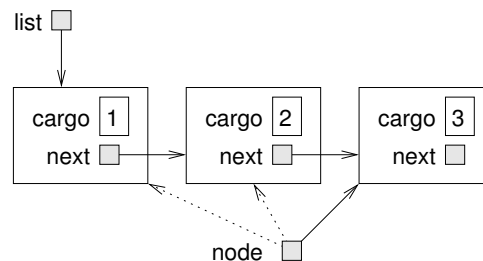
    while (node != null) {
        System.out.print (node);
        node = node.next;
    }
    System.out.println ();
}
```

To invoke this method we just have to pass a reference to the first node:

```
printList (node1);
```

Inside `printList` we have a reference to the first node of the list, but there is no variable that refers to the other nodes. We have to use the `next` value from each node to get to the next node.

This diagram shows the value of `list` and the values that `node` takes on:



This way of moving through a list is called a **traversal**, just like the similar pattern of moving through the elements of an array. It is common to use a loop variable like `node` to refer to each of the nodes in the list in succession.

The output of this method is

```
123
```

By convention, lists are printed in parentheses with commas between the elements, as in (1, 2, 3). As an exercise, modify `printList` so that it generates output in this format.

As another exercise, rewrite `printList` using a `for` loop instead of a `while` loop.

14.4 Lists and recursion

Recursion and lists go together like fava beans and a nice Chianti. For example, here is a recursive algorithm for printing a list backwards:

1. Separate the list into two pieces: the first node (called the head) and the rest (called the tail).
2. Print the tail backwards.
3. Print the head.

Of course, Step 2, the recursive call, assumes that we have a way of printing a list backwards. But *if* we assume that the recursive call works—the leap of faith—then we can convince ourselves that this algorithm works.

All we need is a base case, and a way of proving that for any list we will eventually get to the base case. A natural choice for the base case is a list with a single element, but an even better choice is the empty list, represented by null.

```
public static void printBackward (Node list) {  
    if (list == null) return;  
  
    Node head = list;  
    Node tail = list.next;  
  
    printBackward (tail);  
    System.out.print (head);  
}
```

The first line handles the base case by doing nothing. The next two lines split the list into `head` and `tail`. The last two lines print the list.

We invoke this method exactly as we invoked `printList`:

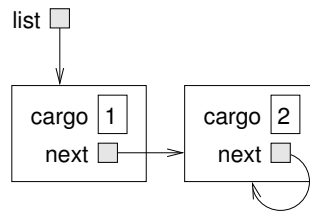
```
printBackward (node1);
```

The result is a backwards list.

Can we prove that this method will always terminate? In other words, will it always reach the base case? In fact, the answer is no. There are some lists that will make this method crash.

14.5 Infinite lists

There is nothing to prevent a node from referring back to an earlier node in the list, including itself. For example, this figure shows a list with two nodes, one of which refers to itself.



If we invoke `printList` on this list, it will loop forever. If we invoke `printBackward` it will recurse infinitely. This sort of behavior makes infinite lists difficult to work with.

Nevertheless, they are occasionally useful. For example, we might represent a number as a list of digits and use an infinite list to represent a repeating fraction.

Regardless, it is problematic that we cannot prove that `printList` and `printBackward` terminate. The best we can do is the hypothetical statement, “If the list contains no loops, then these methods will terminate.” This sort of claim is called a **precondition**. It imposes a constraint on one of the parameters and describes the behavior of the method if the constraint is satisfied. We will see more examples soon.

14.6 The fundamental ambiguity theorem

There is a part of `printBackward` that might have raised an eyebrow:

```

Node head = list;
Node tail = list.next;

```

After the first assignment, `head` and `list` have the same type and the same value. So why did I create a new variable?

The reason is that the two variables play different roles. We think of `head` as a reference to a single node, and we think of `list` as a reference to the first node of a list. These “roles” are not part of the program; they are in the mind of the programmer.

The second assignment creates a new reference to the second node in the list, but in this case we think of it as a list. So, even though `head` and `tail` have the same type, they play different roles.

This ambiguity is useful, but it can make programs with lists difficult to read. I often use variable names like `node` and `list` to document how I intend to use a variable, and sometimes I create additional variables to disambiguate.

I could have written `printBackward` without `head` and `tail`, but I think it makes it harder to understand:

```

public static void printBackward (Node list) {
    if (list == null) return;

    printBackward (list.next);
}

```

```

        System.out.print (list);
    }

```

Looking at the two function calls, we have to remember that `printBackward` treats its argument as a list and `print` treats its argument as a single object.

Always keep in mind the **fundamental ambiguity theorem**:

A variable that refers to a node might treat the node as a single object or as the first in a list of nodes.

14.7 Object methods for nodes

You might have wondered why `printList` and `printBackward` are class methods. I have made the claim that anything that can be done with class methods can also be done with object methods; it's just a question of which form is cleaner.

In this case there is a legitimate reason to choose class methods. It is legal to send `null` as an argument to a class method, but it is not legal to invoke an object method on a null object.

```

Node node = null;
printList (node);           // legal
node.printList ();          // NullPointerException

```

This limitation makes it awkward to write list-manipulating code in a clean, object-oriented style. A little later we will see a way to get around this, though.

14.8 Modifying lists

Obviously one way to modify a list is to change the cargo of one of the nodes, but the more interesting operations are the ones that add, remove, or reorder the nodes.

As an example, we'll write a method that removes the second node in the list and returns a reference to the removed node.

```

public static Node removeSecond (Node list) {
    Node first = list;
    Node second = list.next;

    // make the first node refer to the third
    first.next = second.next;

    // separate the second node from the rest of the list
    second.next = null;
    return second;
}

```

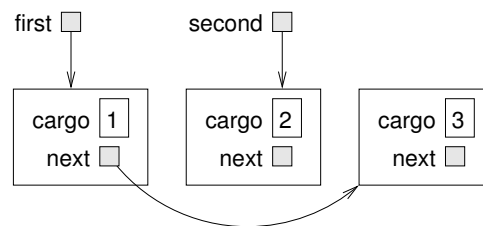

Again, I am using temporary variables to make the code more readable. Here is how to use this method.

```
printList (node1);
Node removed = removeSecond (node1);
printList (removed);
printList (node1);
```

The output is

```
(1, 2, 3)           the original list
(2)                 the removed node
(1, 3)              the modified list
```

Here is a state diagram showing the effect of this operation.



What happens if we invoke this method and pass a list with only one element (a **singleton**)? What happens if we pass the empty list as an argument? Is there a precondition for this method?

14.9 Wrappers and helpers

For some list operations it is useful to divide the labor into two methods. For example, to print a list backwards in the conventional list format, (3, 2, 1) we can use the `printBackwards` method to print 3, 2, but we need a separate method to print the parentheses and the first node. We'll call it `printBackwardNicely`.

```
public static void printBackwardNicely (Node list) {
    System.out.print ("");

    if (list != null) {
        Node head = list;
        Node tail = list.next;
        printBackward (tail);
        System.out.print (head);
    }
    System.out.println ("");
}
```

Again, it is a good idea to check methods like this to see if they work with special cases like an empty list or a singleton.

Elsewhere in the program, when we use this method, we will invoke `printBackwardNicely` directly and it will invoke `printBackward` on our behalf. In that sense, `printBackwardNicely` acts as a **wrapper**, and it uses `printBackward` as a helper.

14.10 The `IntList` class

There are a number of subtle problems with the way we have been implementing lists. In a reversal of cause and effect, I will propose an alternative implementation first and then explain what problems it solves.

First, we will create a new class called `IntList`. Its instance variables are an integer that contains the length of the list and a reference to the first node in the list. `IntList` objects serve as handles for manipulating lists of `Node` objects.

```
public class IntList {
    int length;
    Node head;

    public IntList () {
        length = 0;
        head = null;
    }
}
```

One nice thing about the `IntList` class is that it gives us a natural place to put wrapper functions like `printBackwardNicely`, which we can make an object method in the `IntList` class.

```
    public void printBackward () {
        System.out.print ("");

        if (head != null) {
            Node tail = head.next;
            Node.printBackward (tail);
            System.out.print (head);
        }
        System.out.println ("");
    }
```

Just to make things confusing, I renamed `printBackwardNicely`. Now there are two methods named `printBackward`: one in the `Node` class (the helper) and one in the `IntList` class (the wrapper). In order for the wrapper to invoke the helper, it has to identify the class explicitly (`Node.printBackward`).

So, one of the benefits of the `IntList` class is that it provides a nice place to put wrapper functions. Another is that it makes it easier to add or remove the first element of a list. For example, `addFirst` is an object method for `IntLists`; it takes an `int` as an argument and puts it at the beginning of the list.

```
public void addFirst (int i) {  
    Node node = new Node (i, head);  
    head = node;  
    length++;  
}
```

As always, to check code like this it is a good idea to think about the special cases. For example, what happens if the list is initially empty?

14.11 Invariants

Some lists are “well-formed;” others are not. For example, if a list contains a loop, it will cause many of our methods to crash, so we might want to require that lists contain no loops. Another requirement is that the `length` value in the `IntList` object should be equal to the actual number of nodes in the list.

Requirements like this are called **invariants** because, ideally, they should be true of every object all the time. Specifying invariants for objects is a useful programming practice because it makes it easier to prove the correctness of code, check the integrity of data structures, and detect errors.

One thing that is sometimes confusing about invariants is that there are some times when they are violated. For example, in the middle of `addFirst`, after we have added the node, but before we have incremented `length`, the invariant is violated. This kind of violation is acceptable; in fact, it is often impossible to modify an object without violating an invariant for at least a little while. Normally the requirement is that every method that violates an invariant must restore the invariant.

If there is any significant stretch of code in which the invariant is violated, it is important for the comments to make that clear, so that no operations are performed that depend on the invariant.

14.12 Glossary

list: A data structure that implements a collection using a sequence of linked nodes.

node: An element of a list, usually implemented as an object that contains a reference to another object of the same type.

cargo: An item of data contained in a node.

link: An object reference embedded in an object.

generic data structure: A kind of data structure that can contain data of any type.

precondition: An assertion that must be true in order for a method to work correctly.

invariant: An assertion that should be true of an object at all times (except maybe while the object is being modified).

wrapper method: A method that acts as a middle-man between a caller and a helper method, often offering an interface that is cleaner than the helper method's.

14.13 Exercises

Exercise 14.1

Start by downloading the file `IntList.java` from <http://thinkapjava.com/code/IntList>. It contains the definitions of `IntList` and `Node` from this chapter, along with code that demonstrates and tests some of the methods. Compile and run the program. The output should look like this:

```
(1, 2, 3)
(3, 2, 1)
```

The following exercises ask you to write additional object methods in the `IntList` class, but you might want to write some helper methods in the `Node` class as well.

After you write each method, add code to `main` and test it. Be sure to test special cases like empty lists and singletons.

For each method, identify any preconditions that are necessary for the method to work and add comments that document them. Your comments should also indicate whether each method is a constructor, function, or modifier.

- a. Write a method named `removeFirst` that removes the first node from a list and returns its cargo.
- b. Write a method named `set` that takes an index, `i`, and an item of cargo, and that replaces the cargo of the `i`th node with the given cargo.
- c. Write a method named `add` that takes an index, `i`, and an item of cargo, and that adds a new node containing the given cargo in the `i`th position.
- d. Write a method named `addLast` that takes an item of cargo and adds it to the end of the list.
- e. Write a method called `reverse` that modifies an `IntList`, reversing the order of the nodes.
- f. Write a method named `append` that takes an `IntList` as a parameter and appends a *copy* of the nodes from the parameter list onto the current list. You should be able to take advantage of code you have already written.
- g. Write a method named `checkLength` that returns true if the length field equals the number of nodes in the list, and false otherwise. The method should not cause an exception under any circumstances, and it should terminate even if the list contains a loop.

Exercise 14.2 One way to represent very large numbers is with a list of digits, usually stored in reverse order. For example, the number 123 might be represented with the list (3,2,1).

Write a method that compares two numbers represented as `IntLists` and returns 1 if the first is larger, -1 if the second is larger, and 0 if they are equal.

Chapter 15

Stacks

15.1 Abstract data types

The data types we have looked at so far are all concrete, in the sense that we have completely specified how they are implemented. For example, the `Card` class represents a card using two integers. As I discussed at the time, that is not the only way to represent a card; there are many alternative implementations.

An **abstract data type**, or ADT, specifies a set of operations (or methods) and the semantics of the operations (what they do) but it does not specify the implementation of the operations. That's what makes it abstract.

Why is that useful?

- It simplifies the task of specifying an algorithm if you can denote the operations you need without having to think at the same time about how the operations are performed.
- Since there are usually many ways to implement an ADT, it might be useful to write an algorithm that can be used with any of the possible implementations.
- Well-known ADTs, like the `Stack` ADT in this chapter, are often implemented in standard libraries so they can be written once and used by many programmers.
- The operations on ADTs provide a common high-level language for specifying and talking about algorithms.

When we talk about ADTs, we often distinguish the code that uses the ADT, called the **client** code, from the code that implements the ADT, called **provider** code because it provides a standard set of services.

15.2 The Stack ADT

In this chapter we will look at one common ADT, the stack. A stack is a collection, meaning that it is a data structure that contains multiple elements. Other collections we have seen include arrays and lists.

As I said, an ADT is defined by a set of operations. Stacks can perform the following set of operations:

constructor: Create a new, empty stack.

push: Add a new item to the stack.

pop: Remove and return an item from the stack. The item that is returned is always the last one that was added.

isEmpty: Check whether the stack is empty.

A stack is sometimes called a “last in, first out,” or LIFO data structure, because the last item added is the first to be removed.

15.3 The Java Stack Object

Java provides a built-in object type called `Stack` that implements the Stack ADT. You should make some effort to keep these two things—the ADT and the Java implementation—straight. Before using the `Stack` class, we have to import it from `java.util`.

Then the syntax for constructing a new `Stack` is

```
Stack stack = new Stack ();
```

Initially the stack is empty, as we can confirm with the `isEmpty` method, which returns a `boolean`:

```
System.out.println (stack.isEmpty ());
```

A stack is a generic data structure, which means that we can add any type of item to it. In the Java implementation, though, we can only add object types. For our first example, we'll use `Node` objects, as defined in the previous chapter. Let's start by creating and printing a short list.

```
IntList list = new IntList ();  
list.addFirst (3);  
list.addFirst (2);  
list.addFirst (1);  
list.print ();
```

The output is (1, 2, 3). To put a `Node` object onto the stack, use the `push` method:

```
stack.push (list.head);
```

The following loop traverses the list and pushes all the nodes onto the stack:


```
for (Node node = list.head; node != null; node = node.next) {
    stack.push (node);
}
```

We can remove an element from the stack with the `pop` method.

```
Object obj = stack.pop ();
```

The return type from `pop` is `Object`! That's because the stack implementation doesn't really know the type of the objects it contains. When we pushed the `Node` objects, they were automatically converted to `Objects`. When we get them back from the stack, we have to cast them back to `Nodes`.

```
Node node = (Node) obj;
System.out.println (node);
```

Unfortunately, the burden falls on the programmer to keep track of the objects in the stack and cast them back to the right type when they are removed. If you try to cast an object to the wrong type, you get a `ClassCastException`.

The following loop is a common idiom for popping all the elements from a stack, stopping when it is empty:

```
while (!stack.isEmpty ()) {
    Node node = (Node) stack.pop ();
    System.out.print (node + " ");
}
```

The output is 3 2 1. In other words, we just used a stack to print the elements of a list backwards! Granted, it's not the standard format for printing a list, but using a stack it was remarkably easy to do.

You should compare this code to the implementations of `printBackward` in the previous chapter. There is a natural parallel between the recursive version of `printBackward` and the stack algorithm here. The difference is that `printBackward` uses the run-time stack to keep track of the nodes while it traverses the list, and then prints them on the way back from the recursion. The stack algorithm does the same thing, just using a `Stack` object instead of the run-time stack.

15.4 Wrapper classes

For every primitive type in Java, there is a built-in object type called a **wrapper class**. For example, the wrapper class for `int` is called `Integer`; for `double` it is called `Double`.

Wrapper classes are useful for several reasons:

- You can instantiate wrapper classes and create objects that contain primitive values. In other words, you can wrap a primitive value up in an object, which is useful if you want to invoke a method that requires an object type.

- Each wrapper class contains special values (like the minimum and maximum values for the type), and methods that are useful for converting between types.

15.5 Creating wrapper objects

The most straightforward way to create a wrapper object is to use its constructor:

```
Integer i = new Integer (17);
Double d = new Double (3.14159);
Character c = new Character ('b');
```

Technically `String` is not a wrapper class, because there is no corresponding primitive type, but the syntax for creating a `String` object is the same:

```
String s = new String ("fred");
```

On the other hand, no one ever uses the constructor for `String` objects, because you can get the same effect with a simple `String` value:

```
String s = "fred";
```

15.6 Creating more wrapper objects

Some of the wrapper classes have a second constructor that takes a `String` as an argument and tries to convert to the appropriate type. For example:

```
Integer i = new Integer ("17");
Double d = new Double ("3.14159");
```

The type conversion process is not very robust. For example, if the `Strings` are not in the right format, they will cause a `NumberFormatException`. Any non-numeric character in the `String`, including a space, will cause the conversion to fail.

```
Integer i = new Integer ("17.1");           // WRONG!!
Double d = new Double ("3.1459 ");          // WRONG!!
```

It is usually a good idea to check the format of the `String` before you try to convert it.

15.7 Getting the values out

Java knows how to print wrapper objects, so the easiest way to extract a value is just to print the object:

```
Integer i = new Integer (17);
Double d = new Double (3.14159);
System.out.println (i);
System.out.println (d);
```

Alternatively, you can use the `toString` method to convert the contents of the wrapper object to a `String`

```
String istring = i.toString();
String dstring = d.toString();
```

Finally, if you just want to extract the primitive value from the object, there is an object method in each wrapper class that does the job:

```
int iprim = i.intValue ();
double dprim = d.doubleValue ();
```

There are also methods for converting wrapper objects into different primitive types. You should check out the documentation for each wrapper class to see what is available.

15.8 Useful methods in the wrapper classes

As I mentioned, the wrapper classes contain useful methods that pertain to each type. For example, the `Character` class contains lots of methods for converting characters to upper and lower case, and for checking whether a character is a number, letter, or symbol.

The `String` class also contains methods for converting to upper and lower case. Keep in mind, though, that they are functions, not modifiers (see Section 7.9).

As another example, the `Integer` class contains methods for interpreting and printing integers in different bases. If you have a `String` that contains a number in base 6, you can convert to base 10 using `parseInt`.

```
String base6 = "12345";
int base10 = Integer.parseInt (base6, 6);
System.out.println (base10);
```

Since `parseInt` is a class method, you invoke it by naming the class and the method in dot notation.

Base 6 might not be all that useful, but hexadecimal (base 16) and octal (base 8) are common for computer science related things.

15.9 Postfix expressions

In most programming languages, mathematical expressions are written with the operator between the two operands, as in `1+2`. This format is called **infix**. An alternative format used by some calculators is called **postfix**. In postfix, the operator follows the operands, as in `1 2+`.

The reason postfix is sometimes useful is that there is a natural way to evaluate a postfix expression using a stack.

- Starting at the beginning of the expression, get one term (operator or operand) at a time.

- If the term is an operand, push it on the stack.
- If the term is an operator, pop two operands off the stack, perform the operation on them, and push the result back on the stack.
- When we get to the end of the expression, there should be exactly one operand left on the stack. That operand is the result.

As an exercise, apply this algorithm to the expression `1 2 + 3 *`.

This example demonstrates one of the advantages of postfix: there is no need to use parentheses to control the order of operations. To get the same result in infix, we would have to write `(1 + 2) * 3`. As an exercise, write a postfix expression that is equivalent to `1 + 2 * 3`.

15.10 Parsing

In order to implement the algorithm from the previous section, we need to be able to traverse a string and break it into operands and operators. This process is an example of **parsing**, and the results—the individual chunks of the string—are called **tokens**.

Java provides a built-in class called a `StringTokenizer` that parses strings and breaks them into tokens. To use it, you have to import it from `java.util`.

In its simplest form, the `StringTokenizer` uses spaces to mark the boundaries between tokens. A character that marks a boundary is called a **delimiter**.

We can create a `StringTokenizer` in the usual way, passing as an argument the string we want to parse.

```
StringTokenizer st = new StringTokenizer ("Here are four tokens.");
```

The following loop is a standard idiom for extracting the tokens from a `StringTokenizer`.

```
while (st.hasMoreTokens ()) {  
    System.out.println (st.nextToken());  
}
```

The output is

```
Here  
are  
four  
tokens.
```

For parsing expressions, we have the option of specifying additional characters that will be used as delimiters:

```
StringTokenizer st = new StringTokenizer ("11 22+33*", " +-*/*");
```

The second argument is a `String` that contains all the characters that will be used as delimiters. Now the output is:

```
11
22
33
```

This succeeds at extracting all the operands but we have lost the operators. Fortunately, there is one more option for `StringTokenizers`.

```
StringTokenizer st = new StringTokenizer ("11 22+33*", " +-*/", true);
```

The third argument says, “Yes, we would like to treat the delimiters as tokens.” Now the output is

```
11
22
+
33
*
```

This is just the stream of tokens we would like for evaluating this expression.

15.11 Implementing ADTs

One of the fundamental goals of an ADT is to separate the interests of the provider, who writes the code that implements the ADT, and the client, who uses the ADT. The provider only has to worry about whether the implementation is correct—in accord with the specification of the ADT—and not how it will be used.

Conversely, the client *assumes* that the implementation of the ADT is correct and doesn’t worry about the details. When you are using one of Java’s built-in classes, you have the luxury of thinking exclusively as a client.

When you implement an ADT, on the other hand, you also have to write client code to test it. In that case, you sometimes have to think carefully about which role you are playing at a given instant.

In the next few sections we will switch gears and look at one way of implementing the Stack ADT, using an array. Start thinking like a provider.

15.12 Array implementation of the Stack ADT

The instance variables for this implementation are an array of `Objects`, which will contain the items on the stack, and an integer index which will keep track of the next available space in the array. Initially, the array is empty and the index is 0.

To add an element to the stack (**push**), we’ll copy a reference to it onto the stack and increment the index. To remove an element (**pop**) we have to decrement the index first and then copy the element out.

Here is the class definition:

```
public class Stack {
    Object[] array;
    int index;

    public Stack () {
        this.array = new Object[128];
        this.index = 0;
    }
}
```

As usual, once we have chosen the instance variables, it is a mechanical process to write a constructor. For now, the default size is 128 items. Later we will consider better ways of handling this.

Checking for an empty stack is trivial.

```
public boolean isEmpty () {
    return index == 0;
}
```

It is important to remember, though, that the number of elements in the stack is not the same as the size of the array. Initially the size is 128, but the number of elements is 0.

The implementations of `push` and `pop` follow naturally from the specification.

```
public void push (Object item) {
    array[index] = item;
    index++;
}

public Object pop () {
    index--;
    return array[index];
}
```

To test these methods, we can take advantage of the client code we used to exercise the built-in `Stack`. All we have to do is comment out the line `import java.util.Stack`. Then, instead of using the stack implementation from `java.util` the program will use the implementation we just wrote.

If everything goes according to plan, the program should work without any additional changes. Again, one of the strengths of using an ADT is that you can change implementations without changing client code.

15.13 Resizing arrays

A weakness of this implementation is that it chooses an arbitrary size for the array when the `Stack` is created. If the user pushes more than 128 items onto the stack, it will cause an `ArrayIndexOutOfBoundsException` exception.

An alternative is to let the client code specify the size of the array. This alleviates the problem, but it requires the client to know ahead of time how many items are needed, and that is not always possible.

A better solution is to check whether the array is full and make it bigger when necessary. Since we have no idea how big the array needs to be, it is a reasonable strategy to start with a small size and double it each time it overflows.

Here's the improved version of `push`:

```
public void push (Object item) {
    if (full ()) resize ();

    // at this point we can prove that index < array.length

    array[index] = item;
    index++;
}
```

Before putting the new item in the array, we check if the array is full. If so, we invoke `resize`. After the `if` statement, we know that either (1) there was room in the array, or (2) the array has been resized and there is room. If `full` and `resize` are correct, then we can prove that `index < array.length`, and therefore the next statement cannot cause an exception.

Now all we have to do is implement `full` and `resize`.

```
private boolean full () {
    return index == array.length;
}

private void resize () {
    Object[] newArray = new Object[array.length * 2];

    // we assume that the old array is full
    for (int i=0; i<array.length; i++) {
        newArray[i] = array[i];
    }
    array = newArray;
}
```

Both methods are declared `private`, which means that they cannot be invoked from another class, only from within this one. This is acceptable, since there is no reason for client code to use these functions, and desirable, since it enforces the boundary between the provider code and the client.

The implementation of `full` is trivial; it just checks whether the index has gone beyond the range of valid indices.

The implementation of `resize` is straightforward, with the caveat that it assumes that the old array is full. In other words, that assumption is a precondition of this method. It is easy to see that this precondition is satisfied, since

the only way `resize` is invoked is if `full` returns true, which can only happen if `index == array.length`.

At the end of `resize`, we replace the old array with the new (causing the old to be garbage collected). The new `array.length` is twice as big as the old, and `index` hasn't changed, so now it must be true that `index < array.length`. This assertion is a **postcondition** of `resize`: something that must be true when the method is complete (as long as its preconditions were satisfied).

Preconditions, postconditions, and invariants are useful tools for analyzing programs and demonstrating their correctness. In this example I have demonstrated a programming style that facilitates program analysis and a style of documentation that helps demonstrate correctness.

15.14 Glossary

abstract data type (ADT): A data type (usually a collection of objects) that is defined by a set of operations, but that can be implemented in a variety of ways.

client: A program that uses an ADT (or the person who wrote the program).

provider: The code that implements an ADT (or the person who wrote it).

wrapper class: One of the Java classes, like `Double` and `Integer` that provide objects to contain primitive types, and methods that operate on primitives.

private: A Java keyword that indicates that a method or instance variable cannot be accessed from outside the current class definition.

infix: A way of writing mathematical expressions with the operators between the operands.

postfix: A way of writing mathematical expressions with the operators after the operands.

parse: To read a string of characters or tokens and analyze their grammatical structure.

token: A set of characters that are treated as a unit for purposes of parsing, like the words in a natural language.

delimiter: A character that is used to separate tokens, like the punctuation in a natural language.

predicate: A mathematical statement that is either true or false.

postcondition: A predicate that must be true at the end of a method (provided that the preconditions were true at the beginning).

15.15 Exercises

Exercise 15.1 Write a method named `reverse` that takes an array of integers, traverses the array pushing each item onto a stack, and then pops the items off the stack, putting them back into the array in the reverse of their original order.

The point of this exercise is to practice the mechanisms for creating wrapper objects, pushing and popping objects, and typecasting generic Objects to a specific type.

Exercise 15.2 This exercise is based on the solution to Exercise 14.1. Start by making a copy of your implementation of `IntList` called `LinkedList`.

- a. Transform the linked list implementation into a generic list by making the cargo an `Object` instead of an integer. Modify the test code accordingly and run the program.
- b. Write a `LinkedList` method called `split` that takes a `String`, breaks it up into words (using spaces as delimiters), and returns a list of `Strings`, with one word per list node. Your implementation should be efficient, meaning that it takes time proportional to the number of words in the string.
- c. Write a `LinkedList` method named `join` that returns a `String` that contains the `String` representation of each of the objects in the list, in the order they appear, with spaces in between.
- d. Write a `toString` method for `LinkedList`.

Exercise 15.3 Write an implementation of the Stack ADT using your `LinkedList` implementation as the underlying data structure. There are two general approaches to this: the Stack might contain a `LinkedList` as an instance variable, or the Stack class might extend the `LinkedList` class. Choose whichever sounds better to you, or, if you are feeling ambitious, implement both and compare them.

Exercise 15.4 Write a program called `Balance.java` that reads a file and checks that the parentheses `()` and brackets `[]` and squiggly-braces `{}` are balanced and nested correctly.

HINT: See Section C for code that reads lines from a file.

Exercise 15.5 Write a method called `evalPostfix` that takes a `String` containing a postfix expression and returns a double that contains the result. You can use a `StringTokenizer` to parse the `String` and a Stack of Doubles to evaluate the expression.

Exercise 15.6 Write a program that prompts the user for a mathematical expression in postfix and that evaluates the expression and prints the result. The following steps are my suggestion for a program development plan.

- a. Write a program that prompts the user for input and prints the input string, over and over, until the user types “quit”. See Section C for information about getting input from the keyboard. You can use the following code as a starter:

```
public static void inputLoop () throws IOException {  
    BufferedReader stdin =  
        new BufferedReader (new InputStreamReader (System.in));
```

```
        while (true) {
            System.out.print ("=>");        // print a prompt
            String s = stdin.readLine();    // get input
            if (s == null) break;
            // check if s is "quit"
            // print s
        }
    }
```

- b. Identify helper methods you think will be useful, and write and debug them in isolation. Suggestions: `isOperator`, `isOperand`, `parseExpression`, `performOperation`.
- c. We know we want to push `int` values onto the stack and pop them off, which means we will have to use a wrapper class. Make sure you know how to do that, and test those operations in isolation. Maybe make them helper methods.
- d. Write a version of `evaluate` that only handles one kind of operator (like addition). Test it in isolation.
- e. Connect your evaluator to your input/output loop.
- f. Add the other operations.
- g. Once you have code that works, you might want to evaluate the structural design. How should you divide the code into classes? What instance variables should the classes have? What parameters should be passed around?
- h. In addition to making the design elegant, you should also make the code bullet-proof, meaning that it should not cause an exception under any circumstances, even if the user types something weird.

Chapter 16

Queues and Priority Queues

This chapter presents two ADTs: Queues and Priority Queues. In real life a **queue** is a line of customers waiting for service of some kind. In most cases, the first customer in line is the next customer to be served. There are exceptions, though. For example, at airports customers whose flight is leaving imminently are sometimes taken from the middle of the queue. Also, at supermarkets a polite customer might let someone with only a few items go first.

The rule that determines who goes next is called a **queueing discipline**. The simplest queueing discipline is called **FIFO**, for “first-in-first-out.” The most general queueing discipline is **priority queueing**, in which each customer is assigned a priority, and the customer with the highest priority goes first, regardless of the order of arrival. The reason I say this is the most general discipline is that the priority can be based on anything: what time a flight leaves, how many groceries the customer has, or how important the customer is. Of course, not all queueing disciplines are “fair,” but fairness is in the eye of the beholder.

The Queue ADT and the Priority Queue ADT have the same set of operations and their interfaces are the same. The difference is in the semantics of the operations: a Queue uses the FIFO policy, and a Priority Queue (as the name suggests) uses the priority queueing policy.

As with most ADTs, there are a number of ways to implement queues. Since a queue is a collection of items, we can use any of the basic mechanisms for storing collections, including arrays and lists. Our choice among them will be based in part on their performance— how long it takes to perform the operations we want to perform— and partly on ease of implementation.

16.1 The queue ADT

The queue ADT is defined by the following operations:

constructor: Create a new, empty queue.

add: Add a new item to the queue.

remove: Remove and return an item from the queue. The item that is returned is the first one that was added.

isEmpty: Check whether the queue is empty.

Here is an implementation of a generic Queue, based on the built-in class `java.util.LinkedList`:

```
public class Queue {
    private LinkedList list;

    public Queue () {
        list = new LinkedList ();
    }

    public boolean isEmpty () {
        return list.isEmpty ();
    }

    public void add (Object obj) {
        list.addLast (obj);
    }

    public Object remove () {
        return list.removeFirst ();
    }
}
```

A queue object contains a single instance variable, which is the list that implements it. For each of the other methods, all we have to do is invoke one of the methods from the `LinkedList` class.

We can write the same implementation a bit more concisely by taking advantage of inheritance:

```
public class Queue extends LinkedList {

    public Object remove () {
        return removeFirst ();
    }
}
```

Ok, it's a lot more concise! Because `Queue` extends `LinkedList`, we inherit the constructor, `isEmpty` and `add`. We also inherit `remove`, but the version of `remove` we get from the `LinkedList` class doesn't do what we want; it removes the *last* element in the list, not the first. We fix that by providing a new version of `remove`, which **overrides** the version we inherited.

The choice between these implementations depends on several factors. Inheritance can make an implementation more concise, as long as there are methods

in the parent class that are useful. But it can also make an implementation more difficult to read and debug, because the methods for the new class are not in one place. Also, it can lead to unexpected behavior, because the new class inherits *all* the methods from the parent class, not just the ones you need. That means that the second version of `Queue` provides methods like `removeLast` and `clear` that are not part of the `Queue` ADT. The first implementation is safer; by declaring a `private` instance variable, it prevents clients from invoking methods on the `LinkedList`.

16.2 Veneer

By using a `LinkedList` to implement a `Queue`, we were able to take advantage of existing code; the code we wrote just translates `LinkedList` methods into `Queue` methods. An implementation like this is called a **veneer**. In real life, veneer is a thin coating of good quality wood used in furniture-making to hide lower quality wood underneath. Computer scientists use this metaphor to describe a small piece of code that hides the details of an implementation and provides a simpler, or more standard, interface.

The `Queue` example demonstrates one of the nice things about a veneer, which is that it is easy to implement, and one of the dangers of using a veneer, which is the **performance hazard**!

Normally when we invoke a method we are not concerned with the details of its implementation. But there is one “detail” we might want to know—the performance characteristics of the method. How long does it take, as a function of the number of items in the list?

To answer that question, we have to know more about the implementation. If we assume that `LinkedList` is really implemented as a linked list, then the implementation of `removeFirst` probably looks something like this:

```
public Object removeFirst () {
    Object result = head;
    if (head != null) {
        head = head.next;
    }
    return result.cargo;
}
```

We assume that `head` refers to the first node in the list, and that each node contains `cargo` and a reference to the next node in the list.

There are no loops or function calls here, so the run time of this method is pretty much the same every time. Such a method is called a **constant time** operation. In reality, the method might be slightly faster when the list is empty, since it skips the body of the conditional, but that difference is not significant.

The performance of `addLast` is very different. Here is a hypothetical implementation:

```

public void addLast (Object obj) {
    // special case: empty list
    if (head == null) {
        head = new Node (obj, null);
        return;
    }
    Node last;
    for (last = head; last.next != null; last = last.next) {
        // traverse the list to find the last node
    }
    last.next = new Node (obj, null);
}

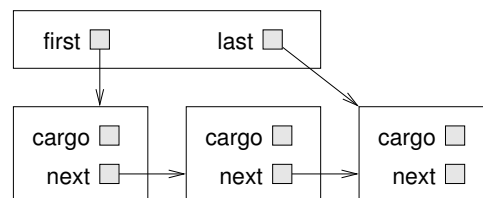
```

The first conditional handles the special case of adding a new node to an empty list. In this case, again, the run time does not depend on the length of the list. In the general case, though, we have to traverse the list to find the last element so we can make it refer to the new node.

This traversal takes time proportional to the length of the list. Since the run time is a linear function of the length, we would say that this method is **linear time**. Compared to constant time, that's very bad.

16.3 Linked Queue

We would like an implementation of the Queue ADT that can perform all operations in constant time. One way to accomplish that is to implement a **linked queue**, which is similar to a linked list in the sense that it is made up of zero or more linked `Node` objects. The difference is that the queue maintains a reference to both the first and the last node, as shown in the figure.



Here's what a linked `Queue` implementation looks like:

```

public class Queue {
    public Node first, last;

    public Queue () {
        first = null;
        last = null;
    }

    public boolean isEmpty () {

```