

Stacks and Queues

- 21.1 Prologue 2
- 21.2 Implementations of Stack 2
- 21.3 *Lab*: Browsing 5
- 21.4 The Hardware Stack 7
- 21.5 Implementations of Queues 9
- 21.6 *Case Study and Lab*: Actors World 10
- 21.7 Summary 15
 - Exercises 16

21.1 Prologue

A stack is useful for untangling nested structures and branching processes: when you jump to the next branch or level, you have to remember where you left off to be able to return there later. A stack is controlled by two operations referred to as *push* and *pop*. Push adds a value to the top of the stack; pop removes the value from the top of the stack.

One example of effective stack use is in a web browser. When you click on a link, the browser saves the current URL and your place in the document so that you can retrace your steps if necessary. When you follow another link, the current location is saved again. And so on. When you press the “Back” button, the browser needs to restore the location visited just before the current one. That is why a stack is appropriate for storing the sequence of visited locations.

In a *queue*, by contrast, data elements are retrieved in the same order as they were stored. The queue is controlled by two operations: *add* and *remove* (or *enqueue* and *dequeue*). *Add* inserts a value at the rear of the queue, and *remove* removes a value from the front of the queue.

In this chapter we will discuss different implementations of the stack and queue data structures. The *LineCruiser* lab in Section 21.3 illustrates the use of stacks in implementing “back” and “forward” buttons in a toy browser. The *Actors* lab in Section 21.6 uses a queue of messages to simulate a world of active “actors.”

21.2 Implementations of Stacks

As we saw in Chapter 19, the Java collections framework has a class `java.util.Stack<E>` class, which implements a stack of objects of the type *E*. `Stack<E>` extends `Vector<E>`. (`Vector` is similar to `ArrayList`; it is a relic from the first release of Java.) In general it is not hard to implement a stack with the help of a list. Both the `ArrayList` and `LinkedList` classes provide methods that make programming a stack of objects very easy. One approach is to derive your stack class from one of them, similar to what `Stack<E>` does. However, in that case your stack will also inherit all of `List`’s methods. If you prefer to keep your stack’s functionality pure, with only `push`, `pop`, `peek`, and `isEmpty` but no `List` methods or iterators, you can define your own `Stack` interface (Figure 21-1) and implement it using an `ArrayList<Object>` (Figure 21-2) or a `LinkedList<Object>` (Figure 21-3).

```
public interface Stack
{
    boolean isEmpty();
    Object peek();
    void push(Object obj);
    Object pop();
}
```

Figure 21-1. “Our own” Stack interface (JM\Ch21\Stack.java)

```
import java.util.ArrayList;

public class ArrayStack implements Stack
{
    private ArrayList<Object> items;

    public ArrayStack() { items = new ArrayList<Object>(); }
    public boolean isEmpty() { return items.isEmpty(); }
    public Object peek() { return items.get(items.size() - 1); }
    public void push(Object obj) { items.add(obj); }
    public Object pop() { return items.remove(items.size() - 1); }
}
```

**Figure 21-2. An implementation of stack based on ArrayList
(JM\Ch21\ArrayStack.java)**

```
import java.util.LinkedList;

public class ListStack implements Stack
{
    private LinkedList<Object> items;

    public ListStack() { items = new LinkedList<Object>(); }
    public boolean isEmpty() { return items.isEmpty(); }
    public Object peek() { return items.getFirst(); }
    public void push(Object obj) { items.addFirst(obj); }
    public Object pop() { return items.removeFirst(); }
}
```

**Figure 21-3. An implementation of stack based on LinkedList
(JM\Ch21>ListStack.java)**

In both implementations the `peek`, `push`, and `pop` methods run in $O(1)$ time.

When you implement a stack, the `pop` and `peek` methods are expected to “throw” an exception if the stack is empty.

For example:

```
if (isEmpty())
    throw new NoSuchElementException();
```

In our `ArrayStack` and `ListStack` classes we don't throw exceptions, because `ArrayList` and `LinkedList` do it for us.



It is also easy to program a specialized stack from scratch, using an array to hold the values. This may be useful for a stack of `ints` or `doubles`, so that you don't have to convert them into objects. This is one of the exercises at the end of this chapter. The array implementation maintains an integer index, called the stack pointer, which marks the current top of the stack. The stack usually grows toward the end of the array; the stack pointer is incremented when a new value is pushed onto the stack and decremented when a value is popped from the stack. In some implementations the stack pointer points to the top element of the stack; other programmers find it more convenient to point to the next available vacant slot on the stack. Figure 21-4 illustrates the latter implementation.

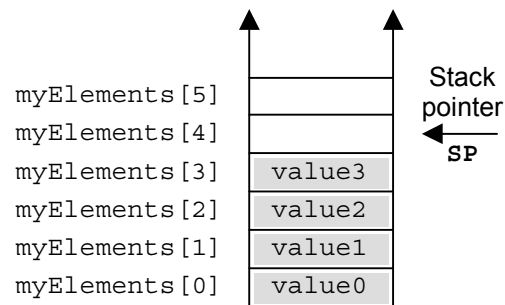


Figure 21-4. An array implementation of stack

In Java, a stack of objects holds references to objects. The objects potentially may change after they have been pushed on the stack (unless they are immutable). If you expect an object to change and you need to save its original state on a stack, create a copy of the object and push a reference to that copy onto the stack.

21.3 Lab: Browsing



In this lab we will implement a toy browser called *LineCruiser*. Rather than browsing web pages, our browser will “browse” several lines of text in the same file. Figure 21-5 shows the classes that we have to write for this project. As usual, this is a team effort: I’ll provide the `LineCruiser`, `BrowserMouseListener`, `BrowserControlPanel`, and `BrowserView` classes, and you work on the `BrowserModel` class.

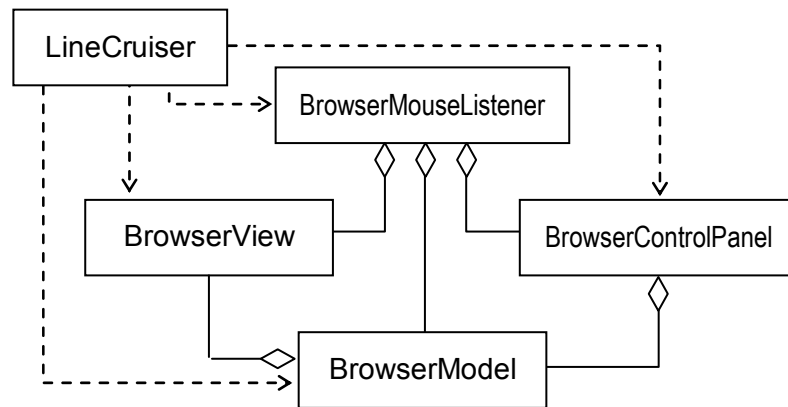


Figure 21-5. *LineCruiser* classes

Our browser only understands two very basic HTML tags: `` and `` (see Appendix <...> for an HTML tutorial). To avoid complicated parsing of strings, we’ll assume that these tags are written exactly as shown, where *name* is any non-empty alphanumeric string. If a line contains an `` tag, the whole line becomes a hyperlink. As usual, when such a hyperlink is clicked, our browser finds a line with a matching `` tag

and brings that line to the top of the screen. Our browser ignores all other tags and preserves the actual line breaks in the file rather than interpreting `<p>`, `
`, or other tags.

All these details are handled in my `BrowserView` class; you only need to know them if you want to create your own test data file. Or just use the one provided by us: `JM\Ch21\Browser\lines.html`.

Our browser has “Home,” “Back,” and “Forward” buttons, just like a real browser. An important part of your task is to figure out how exactly these buttons work in a real browser (such as *Firefox* or *Internet Explorer*). “Home,” “Back,” and “Forward” in our browser should work the same way.

Now let’s agree on more formal specifications.

My `BrowserView` class provides one method of interest to you:

```
public void update(int n)
```

It displays several lines from the file, as many as fit in the display, with the n -th line at the top of the display. Your class, `BrowserModel`, should call `update` as necessary.

Your `BrowserModel` class should provide one constructor and six methods.

`BrowserModel`’s constructor takes one parameter:

```
BrowserModel(BrowserView view)
```

Your constructor should save `view` to be able to call `view`’s `update` method later. Don’t forget to call `update` from `BrowserModel`’s constructor to initialize `view`.

Four of `BrowserModel`’s methods are used for navigation:

```
void back();  
void forward();  
void home();  
void followLink(int n);
```

I call `home`, `back`, and `forward` methods when a corresponding button is clicked. I call `followLink(n)` when a hyperlink pointing to the n -th line is clicked.

`BrowserModel`’s two remaining methods let me know whether the “Back” and/or “Forward” buttons should be enabled or disabled:

```
boolean hasBack();  
boolean hasForward();
```

The return `true` means enable; `false`, disable.

You'll find the *LineCruiser* files in `JM\Ch21\Browser`. The `lines.html` data file for testing our browser is in the same folder. Use `java.util.Stack<Integer>` for the stacks. Write the `BrowserModel` class and test the program thoroughly.

21.4 The Hardware Stack

What happens when a method is called? When method `caller` calls method `task`, how does `task` know where to return control after it has finished? Obviously `caller` has to pass along some return address so that `task` can send the CPU to that address when it is through. Let us consider several possible locations where that return address can be stored.

The first guess is that it could be saved in some specially reserved memory location. This could work if `task` did not call any other methods. If, however, `task` called another method, `subTask`, then its return address would go into the same memory location and overwrite the first return address. In a more elaborate scheme, the return address could go into some special memory area attached to the method code, for instance just before the beginning of the method code. This would solve the problem of methods calling other methods, because every method has its own storage for the return address. This is, in fact, how some early models of computers worked. A problem arises, however, if `task` is allowed to call itself, or when there are circular calls: `task` calls `subTask`, `subTask` calls `anotherSubTask`, `anotherSubTask` calls `task`. Then `task` will get confused about whether to return control to `anotherSubTask` or to `caller`.

The notion of a method calling itself may at first seem absurd. But, as we know, such recursive calls are extremely useful for dealing with nested structures or branching processes where substructures or branches are similar to the whole. As we saw in Chapter 4, recursive methods can greatly simplify algorithms.

Practically the only solution remaining is a stack. When `caller` calls `task`, it pushes the return address on the stack. When `task` has finished, it pops the return address from the stack and passes control back to it. `task` can use the stack for its own purposes and for calling other methods: the only requirement is that it restore the stack pointer to its initial value before returning.

This way, methods can call each other without any conflict. In particular, a method can call itself or methods can call each other in a circular manner. In addition to the return address, though, we have to be careful with the parameters and local variables. If a method in the middle of its course calls itself, what becomes of its local variables? Again, the stack offers the solution. The method's parameters and local variables can all reside on the stack. The stack pointer is adjusted to reserve some space for them when the method is called, and the stack pointer is restored when the method has finished its processing. That way we can use only one copy of the method code but multiple copies of the method's return address, parameters, and local variables, one for every call currently in progress. The area of the stack that holds all the information for a particular method call is called a *frame*. Figure 21-6 illustrates the frames created on the stack after several method calls.

In modern computers the stack method is supported in hardware. The hardware stack does not require any special memory. It is implemented simply as a stack pointer *register* that can point to a desired memory location and can be modified either directly or by the `push` and `pop` CPU instructions. The CPU instruction `call` automatically pushes the address of the next instruction onto the stack before passing control to a subroutine. The CPU instruction `ret` (return) automatically pops the return address from the stack and passes control back to that address.

When method `caller` calls method `task`, `caller` pushes the parameters that it wants to pass to `task` onto the stack, then passes control to `task`. `task` allocates some space on the stack for its own local variables. When `task` has finished its job, it wipes out its local variables from the stack. Either the caller or the called method, depending on the convention in the compiler, performs the final clean-up by removing the parameters from the stack.

The hardware stack is also used for saving the system state when it is interrupted by an external event. Pressing any key on the keyboard, for example, generates a hardware interrupt, a situation that needs the CPU's immediate attention. When this happens, the address of the current CPU instruction is pushed on stack and control is passed to the special interrupt handling routine. This routine pushes all CPU registers on stack to save the current state. Then it receives and processes the pressed key and places its code into the keyboard buffer for later use by the running program. After that the keyboard routine pops all the registers from the stack (in reverse order) and returns control to the interrupted program. The stack helps to handle nested interrupts (when one interrupt comes in the middle of processing another interrupt) properly. People often use a similar method when their tasks or conversations are interrupted.

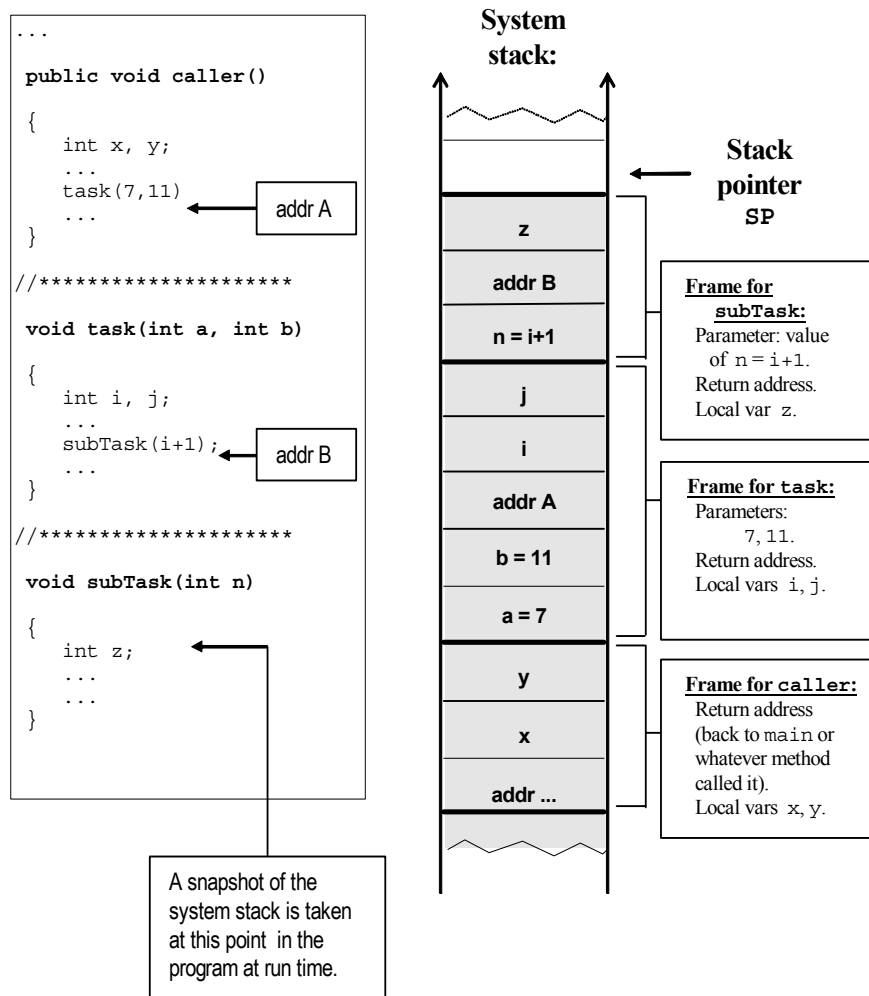


Figure 21-6. Frames on the system stack after a few method calls

21.5 Implementations of Queues

As we saw in Chapter 19, the `java.util` package defines an interface `Queue<E>`; `LinkedList<E>` implements that interface. We will only use four methods from `Queue<E>`: `isEmpty`, `peek`, `add`, and `remove`. In this implementation, `peek`, `add`, and `remove` run in $O(1)$ time. As we know, `LinkedList` is implemented as a doubly-linked list. In Section <...> we saw that it is easy to implement a queue using a singly-linked list with tail, and its methods will have the same $O(1)$ efficiency.

Another possible implementation uses a *ring buffer*, which is simply an array used in a circular manner. If we used an array in a regular linear manner, we would have to shift the whole array toward the front of the queue whenever we removed the first value. In a ring buffer we simply adjust a pointer that defines the “logical” first element. The state of the queue is maintained with the help of two indices, `front` and `rear`. `front` points to the first element in the queue, which will be returned by the next call to the `remove` method; `remove` also increments the `front` index. `rear` points to the empty slot following the last stored element. The `add` method stores the next value in the slot pointed to by `rear` and increments the `rear` index. Both `front` and `rear` wrap around the end of the array to the beginning (Figure 21-7). This mechanism helps to maintain a queue without shifting the whole array.

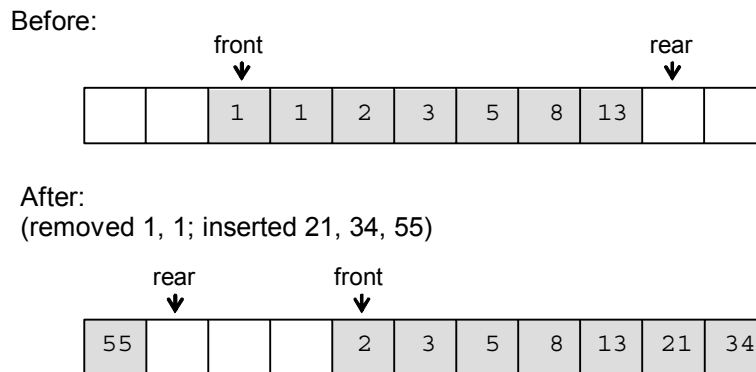


Figure 21-7. Ring-buffer implementation of a queue

21.6 Case Study and Lab: Actors World

After a skiing trip, Kitty, Tommy, and Lizzy discovered that they had misplaced some of their gloves. Some of their hats were missing, too. It turned out nothing was lost: they had simply been in a hurry and grabbed some random items that didn't belong to them. Kitty ended up with three right gloves and one hat; Tommy got only one left glove; Lizzy got two left gloves and two hats. (They all wear the same type and size of gloves and hats.) But, thanks to e-mail, they were quickly able to figure out who got what and swap, so that each ended up with a complete set.

In this case study we examine a program that simulates an *event-driven* world of “actors.” The concept of *event-driven computations* is one of the two key OOP concepts (the other one is inheritance). The von Neumann computer architecture,

with one CPU that controls all operations and has exclusive access to memory, imposes a particular view of programming: a program is thought of as a set of instructions to be performed in sequence. Research in artificial intelligence (or AI, a branch of computer science that strives to model intelligent systems and rational behaviors) and in cognitive science has drawn attention to models where many autonomous agents perform meaningful tasks by working together in parallel without rigid centralized control. Such models allow for the kind of parallelism and autonomy that scientists consider more typical of the cognitive activity in the human brain.

In event-driven models, independent agents can communicate with each other by sending signals or messages. Many things can be taking place at the same time without central synchronization of activities that do not have to be synchronized. In theory, this new model requires a new kind of computer. However, although research continues on various types of parallel computers and distributed systems, models based on active objects can run as simulations on conventional computers. One such model was implemented in the Simula programming language, developed in the mid-1960s (before Smalltalk) by Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Center (NCC) in Oslo. This model perhaps pioneered the OOP approach. Other well-known and influential models for concurrent OOP computations are known as *Actors* models. They originate from the work of Carl Hewitt at the Massachusetts Institute of Technology's AI Laboratory in the early 1970s.

A more recent metaphor for a distributed system of active agents is the Internet itself. It has millions of independent users exchanging messages and data asynchronously and without any centralized control. Yet things get done. The GIMPS (Great Internet Mersenne Prime Search)^{★mersenne} project, mentioned in Chapter 8, is one example.

In an “actor” model, each actor has a set of methods that allow it to recognize and process certain types of messages. An actor responds to a message by sending messages to other actors. Different mechanisms for distributing messages can be included in the model. For example, messages can be directed to one specific actor (somewhat like an e-mail message) or they can be broadcast to many actors at once (like newsgroups or e-mail lists). In the latter case, only those actors that can recognize a message and find it “relevant” respond to it.

When simulated on a conventional computer, the actors need a central mechanism for handling messages, a kind of a “message server.” The server places all incoming messages into a queue. The whole program becomes one large loop that checks whether there are any messages waiting in the queue and dispatches them to the appropriate actors. This is quite similar to what happens with events in GUI

operating systems, such as *Windows*. In Java, the keyboard, the mouse, and GUI components play the role of “actors,” and the `EventQueue` class in the `java.awt` package handles the management of the event queue. It dispatches the events to objects designated as “listeners” (`ActionListeners`, `KeyListener`s, `MouseListener`s, etc.). All this normally happens behind the scenes. In this case study we made the management of the message queue explicit to show you how it might work.



Figure 21-8 shows the classes and objects involved in our *Actors World* program and their public methods. An actor is implemented as an object of the class `Actor`. Each actor has certain “possessions”: a number of items out of a small list of items. In this example we have three actors and three types of items. The `main` method in the `ActorsWorld` class initializes the actors:

```
public static void main(String[] args)
{
    < ... window setup >

    String[] kittysPossessions = {"right glove", "right glove",
                                "right glove", "hat"};
    String[] tommysPossessions = {"left glove"};
    String[] lizzysPossessions = {"left glove", "left glove",
                                "hat", "hat"};

    List<Actor> actors = new ArrayList<Actor>();
    actors.add(new Actor("Kitty", Arrays.asList(kittysPossessions)));
    actors.add(new Actor("Tommy", Arrays.asList(tommysPossessions)));
    actors.add(new Actor("Lizzy", Arrays.asList(lizzysPossessions)));
    ...
}
```

Each actor can send or receive messages, either privately to another actor, or to the whole subscriber list. Messages do not have to be synchronized in any way. It is possible to envision a distributed computing environment in which our three actors would be implemented as three programs (or rather, three copies of the same program) running on computers in different cities; the programs would be connected to e-mail so that they could send and receive messages automatically. Each actor has its own “mailbox,” which is a queue that holds the messages awaiting processing.

A message is an object of the class `Message`. It is a data-carrier kind of object with a constructor `Message(Actor sender, Actor recipient, String text)` and three methods:

```
public Actor getSender()
public Actor getRecipient()
public String getText()
```

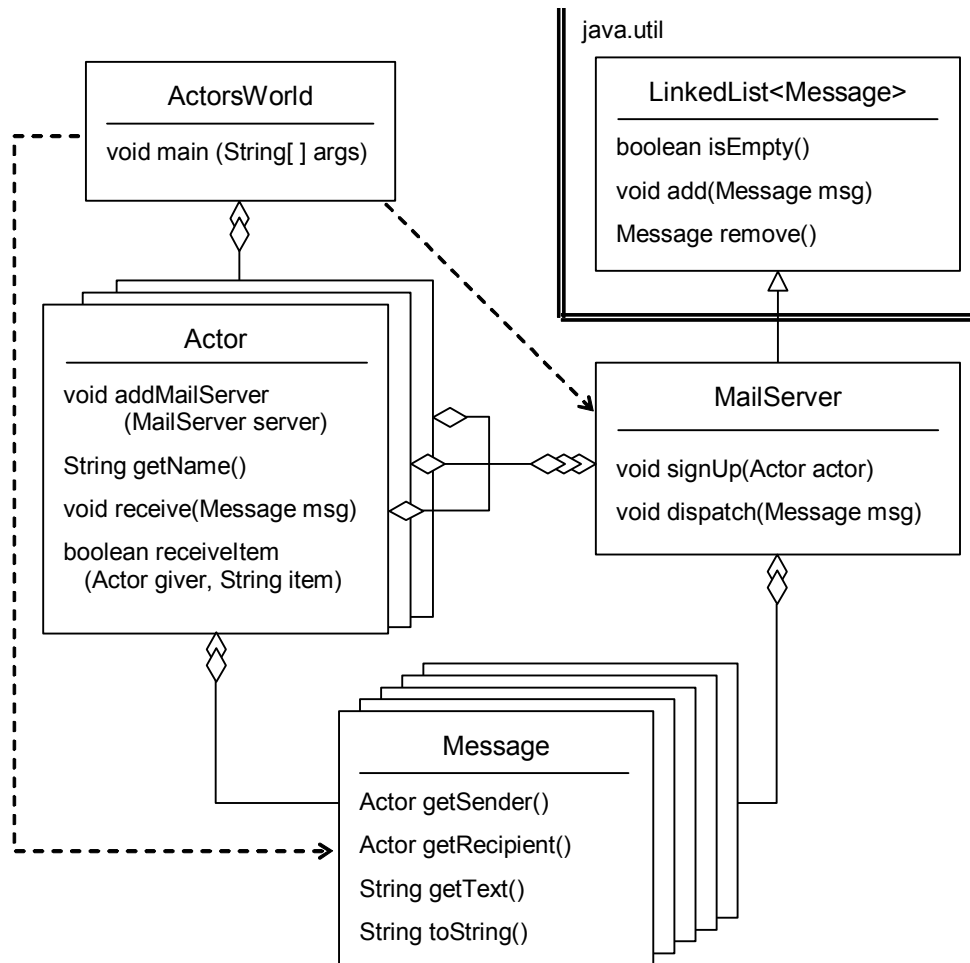


Figure 21-8. Classes, objects, and public methods in the *Actors World* program

If the recipient is null, the message is intended for the whole set of subscribers. A message also has a `toString` method that converts it into a readable message, such as

```

From: Kitty
To: Tommy
ship left glove
  
```

The `MailServer` class represents a mechanism for queuing and dispatching messages. A `MailServer` is a queue of messages (`MailServer` implements `Queue<Message>`), and it has a method to dispatch a message to the appropriate recipient. A `MailServer` also keeps a set of all “registered” subscribers.

After `main` has created the actors and a `MailServer`, it notifies each actor who its `MailServer` is:

```
MailServer email = new MailServer();
for (Actor actor : actors)
    actor.addMailServer(email);
```

This is analogous to adding a listener to an object in Java.

After that, the program goes into an “infinite” loop. On each iteration the program checks whether the `MailServer` has any messages in its queue, retrieves them one by one, displays each message, and dispatches it to its recipient(s) by calling `MailServer`’s `dispatch` method. Then it gives each actor time to “read e-mail”:

```
while (true)
{
    while (!email.isEmpty())
    {
        Message msg = email.remove();
        window.display.append(msg.toString() + "\n");
        email.dispatch(msg);
    }
    for (Actor actor : actors)
        actor.readMail();
}
```

Luckily Java does the same: it allocates CPU time to GUI components for processing their events, so when the “close window” button is clicked, our “infinite” `while` loop is interrupted and the program quits.



Your task in this project is to write the `MailServer` class and to supply two methods for the `Actor` class.

1. The `MailServer` class

This class should extend `LinkedList<Message>` (and, therefore, implement `Queue<Message>`). Only the `isEmpty`, `add`, and `remove` methods of `LinkedList` will be used.

A `MailServer` should maintain a set of “registered” subscribers (implemented as either a `TreeSet<Actor>`). A `MailServer` should provide a method `signUp(Actor actor)`, which adds `actor` to the set of the registered subscribers.

Also provide a method `dispatch(Message msg)`, which either sends `msg` to the recipient indicated in `msg` (by calling the recipient’s `receive(msg)`), or, if the recipient is `null`, to all registered subscribers (except the sender).

2. The Actor class

For this class, you have to fill in the blanks in the `receive` method and the `readMail` method (in `JM\Ch21\Actors\Actors.java`.)

`receive` simply adds `msg` to the actor’s mailbox.

The `readMail` method first checks this actor’s possessions and sends requests to everyone for each missing item. After that, `readMail` “reads” all messages in this actor’s mailbox and sends replies or takes appropriate actions. The proper responses to various messages are documented in `Actors.java` in the comment to the `readMail` method.



When you are done, set up a project with `ActorWorld.java` and `Message.java` (from `JM\Ch21\Actors`) and your `MailServer` and `Actors` classes, and test your program to see if every actor has found all its possessions at the end. All three should post a “thanks, all set” message.

21.7 Summary

A stack is a data structure used for storing and retrieving data elements in the LIFO (Last-In-First-Out) manner.

One possible implementation of a stack uses an array and an integer index, called the stack pointer, which marks the current top of the stack. The stack usually grows toward the end of the array; the stack pointer is incremented when a new value is pushed onto the stack and decremented when a value is popped from the stack.

A stack can also be easily implemented as a singly- or doubly-linked list with $O(1)$ running time for the `peek`, `push`, and `pop` methods. In the Java collections

framework, a stack is implemented in the class `java.util.Stack<E>`, where `E` is the type of objects on the stack.

Stacks are used for temporary storage; they are especially useful when you are working with nested structures or branching processes.

The queue data structure is used for storing and retrieving values in a First-In-First-Out (FIFO) manner. A queue can be implemented as a ring buffer, which is simply an array used in a circular way. The `front` index marks the beginning of the queue, where the next value will be removed; the `rear` index marks the end of the queue (the first available slot), where the next value will be inserted. Both pointers wrap around the end of the array. Another queue implementation may use a linked list with a tail or a doubly-linked list. In all these implementations, the `peek`, `add`, and `remove` methods run in $O(1)$ time. In the Java collections framework, a queue is implemented as `LinkedList<E>`.

Queues are used for processing events that have to be handled in the order of their arrival but may have to wait for available resources or an appropriate time. Queues are widely used for system tasks such as scheduling jobs, processing events, passing data between processes, and input/output buffering for peripheral devices.

Exercises

1. Mark true or false and explain:
 - (a) A stack is especially useful as temporary storage for events that have to be handled in the order of their arrival. _____ ✓
 - (b) The *Queue* data structure is a special case of the *Stack* data structure. _____
 - (c) A stack can be implemented efficiently (with constant time for `push` and `pop`) using either an array or a singly-linked list. _____
2. ■ Write and test a class `IntStack` that implements a stack of integers with methods similar to the ones in the `Stack` class, but handling values of the type `int` rather than `Object`. Provide a no-args constructor that allocates an array of some default capacity to hold the values and another constructor that allocates an array of a specified capacity. Use a “stack pointer” to hold the index of the first empty slot. Reallocate the array, doubling its size, if the stack runs out of space.

3. Suppose a stack is implemented as a linked list. The nodes are represented by `ListNode` objects, and the field `top` refers to the top node:

```
public class ListNodeStack implements Stack
{
    private ListNode top;
    ...
}
```

Given the code for the `push` method —

```
public void push(Object value)
{
    top = new ListNode(value, top);
}
```

— supply the code for the `isEmpty`, `peek`, and `pop` methods. (The `peek` and `pop` methods should throw a `NoSuchElementException` if the stack is empty.)

4. ■ The following method takes a number represented as a string of binary digits and returns the number as an `int`:

```
public static int binToInt(String binNum)
{
    Stack<Character> stack = new Stack<Character>();

    for (int i = 0; i < binNum.length(); i++)
        stack.push(binNum.charAt(i));

    int result = 0, power2 = 1;

    while (!stack.isEmpty())
    {
        char ch = stack.pop();
        int dig = Character.digit(ch, 2);
        result += dig * power2;
        power2 *= 2;
    }

    return result;
}
```

Explain why the use of a stack here is overkill. Rewrite without the stack. ✓

5. A stack of characters can be implemented using a `String`.

(a) Does the following code correctly implement the `push` and `pop` methods?

```
private String stack = "";
           // Declare an empty stack of chars;
...
public void push(char ch)
{
    stack += ch;
}

public char pop()
{
    int n = stack.length() - 1;
    if (n >= 0)
    {
        char ch = stack.charAt(n);
        stack = stack.substring(0, n);
        return ch;
    }
    else
        throw new NoSuchElementException();
}
```

(b) Discuss the merits of the above implementation of a stack. ✓

(c) Rewrite the `push` and `pop` methods so that `pop` returns the first character in the string.

(d) ■ Rewrite the above code using a `StringBuffer` object and its `append` and `deleteCharAt` methods.

6. A deck of cards is represented in a program as a stack of `Card` objects. Write a method

```
public boolean moveToTop(Stack deck, int n)
```

that takes the n -th card in the deck (counting the top card as the first one) and moves it to the top of the deck. The method returns `true` if the deck has n or more cards; otherwise it leaves the deck unchanged and returns `false`. Your method may use a temporary stack, but no other arrays or lists. ✓

7. A class `RingBuffer` implements a queue of chars. It has a constructor that allocates a character array of a given size and initializes it to an empty state:

```
public RingBuffer(int capacity)
{
    items = new char[capacity + 1];
    front = 0;
    rear = 0;
    last = capacity;
}
```

- (a) Write a void method `flush()` that empties the queue (without deallocating items). ✓
- (b) Write a boolean method `isEmpty()`.
- (c) Write a boolean method `add(char ch)`. The method should return `true` if the operation is successful and `false` if the queue is full.
8. Suppose a class `ListNodeQueue` implements `Queue<Object>` using a linked list of `ListNode` objects with references `front` and `rear` to the first and last nodes, respectively. Given the code for the `remove` method —

```
public Object remove()
{
    if (front != null)
    {
        Object obj = front.getValue();
        front = front.getNext();
        if (front == null)
            rear = null;
        return obj;
    }
    else
        throw new NoSuchElementException();
}
```

— supply the code for the `isEmpty`, `peek`, and `add` methods.

9. ■ The IBM PC BIOS (Basic IO System) uses a keyboard ring buffer of 16 bytes, starting at the address 40:1E. The two-byte locations 40:1A and 40:1C represent the front and the rear of the keyboard queue, respectively. These locations hold offsets from 40:00, stored with the least significant byte first. Each pressed keyboard key adds two bytes to the keyboard buffer: the ASCII code of the character followed by the so-called “scan code” that represents the location of the key on the keyboard. Examine the following hex memory dump and determine the current contents of the keyboard queue and the last eight characters typed. ✓

```

                _0 _1 _2 _3 _4 _5 _6 _7 _8 _9 _A _B _C _D _E _F
0040:0010                28 00 28 00 30 0B
0040:0020 3A 27 31 02 61 1E 0D 1C-64 20 20 39 34 05

```

10. How are “actor” models usually implemented?
- On parallel data flow computers
 - Using neural networks hardware
 - Through simulation on conventional von Neumann computers
 - On the Internet
11. If we add a call

```
Collections.shuffle(actors); // shuffle the actors list
```

before

```
for (Actor actor : actors)
    actor.readMail();
```

in `ActorsWorld's main` method, will Kitty, Tommy, and Lizzy still be able to recover their possessions? Test this hypothesis.