Exercise 2.44 Give the class two constructors. One should take a single parameter that specifies the price, and the other should take no parameter and set the price to be a default value of your choosing. Test your implementation by creating machines via the two different constructors.

Exercise 2.45 Implement a method, **`empty`**, that simulates the effect of removing all money from the machine. This method should have a **`void`** return type, and its body should simply set the **`total`** field to zero. Does this method need to take any parameters? Test your method by creating a machine, inserting some money, printing some tickets, checking the total, and then emptying the machine. Is the **`empty`** method a mutator or an accessor?

2.12 Reflecting on the design of the ticket machine

From our study of the internals of the **`TicketMachine`** class, you should have come to appreciate how inadequate it would be in the real world. It is deficient in several ways:

- It contains no check that the customer has entered enough money to pay for a ticket.
- It does not refund any money if the customer pays too much for a ticket.
- It does not check to ensure that the customer inserts sensible amounts of money. Experiment with what happens if a negative amount is entered, for instance.
- It does not check that the ticket price passed to its constructor is sensible.

If we could remedy these problems, then we would have a much more functional piece of software that might serve as the basis for operating a real-world ticket machine.

In the next few sections, we shall examine the implementation of an improved ticket machine class that attempts to deal with some of the inadequacies of the naïve implementation. Open the *better-ticket-machine* project. As before, this project contains a single class: **`TicketMachine`**. Before looking at the internal details of the class, experiment with it by creating some instances and see whether you notice any differences in behavior between this version and the previous naíve version.

One specific difference is that the new version has an additional method, **`refundBalance`**. Take a look at what happens when you call it.

Code 2.8 A more sophisticated **`TicketMachine`**

```
/**
 * TicketMachine models a ticket machine that issues
 * flat-fare tickets.
 * The price of a ticket is specified via the constructor.
 * Instances will check to ensure that a user only enters
 * sensible amounts of money, and will only print a ticket
 * if enough money has been input.
 *
```

Code 2.8 continued
A more sophisticated `TicketMachine`

```
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29
 */
public class TicketMachine
{
    // The price of a ticket from this machine.
    private int price;
    // The amount of money entered by a customer so far.
    private int balance;
    // The total amount of money collected by this machine.
    private int total;

    /**
     * Create a machine that issues tickets of the given price.
     */
    public TicketMachine(int cost)
    {
        price = cost;
        balance = 0;
        total = 0;
    }

    /**
     * @Return The price of a ticket.
     */
    public int getPrice()
    {
        return price;
    }

    /**
     * Return The amount of money already inserted for the
     * next ticket.
     */
    public int getBalance()
    {
        return balance;
    }

    /**
     * Receive an amount of money from a customer.
     * Check that the amount is sensible.
     */
    public void insertMoney(int amount)
    {
        if(amount > 0) {
            balance = balance + amount;
        }
        else {
            System.out.println("Use a positive amount rather than: " +
                               amount);
        }
    }
```