**Programming Exercises**

**1. Pet Class**

Write a class named Pet, which should have the following data attributes:

• \_ \_name (for the name of a pet)

• \_ \_animal\_type (for the type of animal that a pet is. Example values are ‘Dog’, ‘Cat’,

and ‘Bird’)

• \_ \_age (for the pet’s age)

The Pet class should have an \_ \_init\_ \_ method that creates these attributes. It should also

have the following methods:

• set\_name

This method assigns a value to the \_ \_name field.

• set\_animal\_type

This method assigns a value to the \_ \_animal\_type field.

• set\_age

This method assigns a value to the \_ \_age field.

• get\_name

This method returns the value of the \_ \_ name field.

• get\_animal\_type

This method returns the value of the \_ \_animal\_type field.

• get\_age

This method returns the value of the \_ \_age field.

Once you have written the class, write a program that creates an object of the class and

prompts the user to enter the name, type, and age of his or her pet. This data should be stored as the object’s attributes. Use the object’s accessor methods to retrieve the pet’s name, type, and age and display this data on the screen.

**2. Car Class**

Write a class named Car that has the following data attributes:

• \_ \_year\_model (for the car’s year model)

• \_ \_make (for the make of the car)

• \_ \_speed (for the car’s current speed)

The Car class should have an \_ \_init\_ \_ method that accepts the car’s year model and make

as arguments. These values should be assigned to the object’s \_ \_year\_model and \_ \_make

data attributes. It should also assign 0 to the \_ \_speed data attribute.

The class should also have the following methods:

• accelerate

The accelerate method should add 5 to the speed data attribute each time it is called.

• brake

The brake method should subtract 5 from the speed data attribute each time it is called.

• get\_speed

The get\_speed method should return the current speed.

Next, design a program that creates a Car object and then calls the accelerate method

five times. After each call to the accelerate method, get the current speed of the car and

display it. Then call the brake method five times. After each call to the brake method, get

the current speed of the car and display it.

**3. Personal Information Class**

Design a class that holds the following personal data: name, address, age, and phone number.

Write appropriate accessor and mutator methods. Also, write a program that creates

three instances of the class. One instance should hold your information, and the other two

should hold your friends’ or family members’ information.

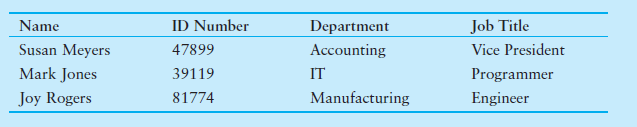
**4. Employee Class**

Write a class named Employee that holds the following data about an employee in attributes:

name, ID number, department, and job title.

Once you have written the class, write a program that creates three Employee objects to

hold the following data:



The program should store this data in the three objects and then display the data for each

employee on the screen.

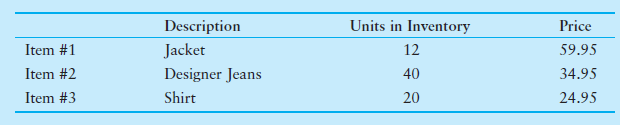
**5. RetailItem Class**

Write a class named RetailItem that holds data about an item in a retail store. The class

should store the following data in attributes: item description, units in inventory, and price.

Once you have written the class, write a program that creates three RetailItem objects

and stores the following data in them:



**6. E mployee Management System**

This exercise assumes that you have created the Employee class for Programming Exercise 4.

Create a program that stores Employee objects in a dictionary. Use the employee ID number

as the key. The program should present a menu that lets the user perform the following

actions:

• Look up an employee in the dictionary

• Add a new employee to the dictionary

• Change an existing employee’s name, department, and job title in the dictionary

• Delete an employee from the dictionary

• Quit the program

When the program ends, it should pickle the dictionary and save it to a file. Each time the

program starts, it should try to load the pickled dictionary from the file. If the file does not

exist, the program should start with an empty dictionary.

**7. Cash Register**

This exercise assumes that you have created the RetailItem class for Programming

Exercise 5. Create a CashRegister class that can be used with the RetailItem class. The

CashRegister class should be able to internally keep a list of RetailItem objects. The

class should have the following methods:

• A method named purchase\_item that accepts a RetailItem object as an argument.

Each time the purchase\_item method is called, the RetailItem object that is passed

as an argument should be added to the list.

• A method named get\_total that returns the total price of all the RetailItem objects

stored in the CashRegister object’s internal list.

• A method named show\_items that displays data about the RetailItem objects stored

in the CashRegister object’s internal list.

• A method named clear that should clear the CashRegister object’s internal list.

Demonstrate the CashRegister class in a program that allows the user to select several

items for purchase. When the user is ready to check out, the program should display a list

of all the items he or she has selected for purchase, as well as the total price.

**8. Circle Class**

Write a Circle class that has the following fields:

• **radius**: a double

• **PI**: a final double initialized with the value 3.14159

The class should have the following methods:

• **Constructor**. Accepts the radius of the circle as an argument.

• **Constructor**. A no-arg constructor that sets the radius field to 0.0.

• **setRadius**. A mutator method for the radius field.

• **getRadius**. An accessor method for the radius field.

• **getArea**. Returns the area of the circle, which is calculated as

area = PI \* radius \* radius

• **getDiameter**. Returns the diameter of the circle, which is calculated as

diameter = radius \* 2

• **getCircumference**. Returns the circumference of the circle, which is calculated as

circumference = 2 \* PI \* radius

Write a program that demonstrates the Circle class by asking the user for the circle’s radius,

creating a Circle object, and then reporting the circle’s area, diameter, and circumference.

**9. Temperature Class**

Write a Temperature class that will hold a temperature in Fahrenheit, and provide methods

to get the temperature in Fahrenheit, Celsius, and Kelvin. The class should have the

following field:

• ftemp – A double that holds a Fahrenheit temperature.

The class should have the following methods:

• Constructor – The constructor accepts a Fahrenheit temperature (as a double) and

stores it in the ftemp field.

• setFahrenheit – The setFahrenheit method accepts a Fahrenheit temperature (as a

double) and stores it in the ftemp field.

• getFahrenheit – Returns the value of the ftemp field, as a Fahrenheit temperature (no

conversion required).

• getCelsius – Returns the value of the ftemp field converted to Celsius.

• getKelvin – Returns the value of the ftemp field converted to Kelvin.

**10. Days in a Month**

Write a class named MonthDays. The class’s constructor should accept two arguments:

• An integer for the month (1 5 January, 2 February, etc.).

• An integer for the year

The class should have a method named getNumberOfDays that returns the number of days

in the specified month. The method should use the following criteria to identify leap years:

1. Determine whether the year is divisible by 100. If it is, then it is a leap year if and if

only it is divisible by 400. For example, 2000 is a leap year but 2100 is not.

2. If the year is not divisible by 100, then it is a leap year if and if only it is divisible by

4. For example, 2008 is a leap year but 2009 is not.

Demonstrate the class in a program that asks the user to enter the month (letting the user

enter an integer in the range of 1 through 12) and the year. The program should then display

the number of days in that month. Here is a sample run of the program:

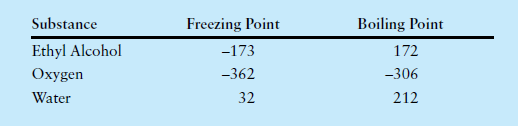
Enter a month (1-12): **2 [Enter]**

Enter a year: **2008 [Enter]**

29 days

**11. Freezing and Boiling Points**

The following table lists the freezing and boiling points of several substances.



Design a class that stores a temperature in a temperature field and has the appropriate

accessor and mutator methods for the field. In addition to appropriate constructors, the

class should have the following methods:

• **isEthylFreezing**. This method should return the boolean value true if the temperature

stored in the temperature field is at or below the freezing point of ethyl alcohol.

Otherwise, the method should return false.

• **isEthylBoiling**. This method should return the boolean value true if the temperature

stored in the temperature field is at or above the boiling point of ethyl alcohol.

Otherwise, the method should return false.

• **isOxygenFreezing**. This method should return the boolean value true if the temperature

stored in the temperature field is at or below the freezing point of oxygen.

Otherwise, the method should return false.

• **isOxygenBoiling**. This method should return the boolean value true if the temperature

stored in the temperature field is at or above the boiling point of oxygen. Otherwise,

the method should return false.

• **isWaterFreezing**. This method should return the boolean value true if the temperature

stored in the temperature field is at or below the freezing point of water. Otherwise,

the method should return false.

• **isWaterBoiling**. This method should return the boolean value true if the temperature

stored in the temperature field is at or above the boiling point of water. Otherwise,

the method should return false.

Write a program that demonstrates the class. The program should ask the user to enter a

temperature, and then display a list of the substances that will freeze at that temperature

and those that will boil at that temperature. For example, if the temperature is –20 the class

should report that water will freeze and oxygen will boil at that temperature.

**12. SavingsAccount Class**

Design a SavingsAccount class that stores a savings account’s annual interest rate and balance.

The class constructor should accept the amount of the savings account’s starting

balance. The class should also have methods for subtracting the amount of a withdrawal,

adding the amount of a deposit, and adding the amount of monthly interest to the balance.

The monthly interest rate is the annual interest rate divided by twelve. To add the monthly

interest to the balance, multiply the monthly interest rate by the balance, and add the result

to the balance.

Test the class in a program that calculates the balance of a savings account at the end of a

period of time. It should ask the user for the annual interest rate, the starting balance, and the number of months that have passed since the account was established. A loop should

then iterate once for every month, performing the following:

a. Ask the user for the amount deposited into the account during the month. Use the

class method to add this amount to the account balance.

b. Ask the user for the amount withdrawn from the account during the month. Use the

class method to subtract this amount from the account balance.

c. Use the class method to calculate the monthly interest.

After the last iteration, the program should display the ending balance, the total amount of

deposits, the total amount of withdrawals, and the total interest earned.

**13. Deposit and Withdrawal Files**

Use Notepad or another text editor to create a text file named Deposits.txt. The file

should contain the following numbers, one per line:

100.00

124.00

78.92

37.55

Next, create a text file named Withdrawals.txt. The file should contain the following numbers,

one per line:

29.88

110.00

27.52

50.00

12.90

The numbers in the Deposits.txt file are the amounts of deposits that were made to a savings

account during the month, and the numbers in the Withdrawals.txt file are the amounts

of withdrawals that were made during the month. Write a program that creates an instance

of the SavingsAccount class that you wrote in Programming Challenge 12. The starting balance

for the object is 500.00. The program should read the values from the Deposits.txt file

and use the object’s method to add them to the account balance. The program should read

the values from the Withdrawals.txt file and use the object’s method to subtract them from

the account balance. The program should call the class method to calculate the monthly

interest, and then display the ending balance and the total interest earned.

**Exercises**

1. Write a class called Investment with fields called principal and interest. The constructor

should set the values of those fields. There should be a method called value\_after that

returns the value of the investment after *n* years. The formula for this is *p*(1 + *i*)*n*, where *p* is

the principal, and *i* is the interest rate. It should also use the special method \_\_str\_\_ so that

printing the object will result in something like below:

Principal - $1000.00, Interest rate - 5.12%

2. Write a class called Product. The class should have fields called name, amount, and price,

holding the product’s name, the number of items of that product in stock, and the regular

price of the product. There should be a method get\_price that receives the number of

items to be bought and returns a the cost of buying that many items, where the regular price is charged for orders of less than 10 items, a 10% discount is applied for orders of between

10 and 99 items, and a 20% discount is applied for orders of 100 or more items. There should

also be a method called make\_purchase that receives the number of items to be bought and

decreases amount by that much.

3. Write a class called Password\_manager. The class should have a list called old\_passwords

that holds all of the user’s past passwords. The last item of the list is the user’s current password.

There should be a method called get\_password that returns the current password

and a method called set\_password that sets the user’s password. The set\_password

method should only change the password if the attempted password is different from all

the user’s past passwords. Finally, create a method called is\_correct that receives a string

and returns a boolean **True** or **False** depending on whether the string is equal to the current

password or not.

4. Write a class called Time whose only field is a time in seconds. It should have a method called

convert\_to\_minutes that returns a string of minutes and seconds formatted as in the following

example: if seconds is 230, the method should return '5:50'. It should also have

a method called convert\_to\_hours that returns a string of hours, minutes, and seconds

formatted analogously to the previous method.

5. Write a class called Wordplay. It should have a field that holds a list of words. The user

of the class should pass the list of words they want to use to the class. There should be the

following methods:

• words\_with\_length(length)— returns a list of all the words of length length

• starts\_with(s)— returns a list of all the words that start with s

• ends\_with(s)— returns a list of all the words that end with s

• palindromes()— returns a list of all the palindromes in the list

• only(L)— returns a list of the words that contain only those letters in L

• avoids(L)— returns a list of the words that contain none of the letters in L

6. Write a class called Converter. The user will pass a length and a unit when declaring an

object from the class—for example, c = Converter(9,'inches'). The possible units are

inches, feet, yards, miles, kilometers, meters, centimeters, and millimeters. For each of these

units there should be a method that returns the length converted into those units. For example,

using the Converter object created above, the user could call c.feet() and should get

0.75 as the result.

7. Use the Standard\_deck class of this section to create a simplified version of the game War.

In this game, there are two players. Each starts with half of a deck. The players each deal

the top card from their decks and whoever has the higher card wins the other player’s cards

and adds them to the bottom of his deck. If there is a tie, the two cards are eliminated from

play (this differs from the actual game, but is simpler to program). The game ends when one

player runs out of cards.