# Exercises

## Exercise 1 — MySavings

Create a MySavings application that displays a menu of choices for entering pennies, nickels, dimes, and quarters into a piggy bank and then prompts the user to make a selection. The MySavings application should include a PiggyBank object that can add coins to the piggy bank, remove coins, and return the total amount in the bank. Application output should look similar to:

```
1. Show total in bank.
2. Add a penny.
3. Add a nickel.
4. Add a dime.
5. Add a quarter.
6. Take money out of bank.
Enter 0 to quit
Enter you choice: 5
```

## Exercise 2 — DigitExtractor

Create a DigitExtractor application that prompts the user for an integer and then displays the ones, tens, and hundreds digit of the number. The DigitExtractor application should include a Num object that can return the ones digit, tens digit, hundreds digit, and the whole number. Application output should look similar to:

```
Enter an integer: 123
show (W)hole number.
show (O)nes place number.
show (T)ens place number.
show (H)undreds place number.
(Q)uit
Enter you choice: h
The hundreds place digit is: 1

show (W)hole number.
show (O)nes place number.
show (T)ens place number.
show (H)undreds place number.
(Q)uit
Enter you choice: q
```

## Exercise 3 — LunchOrder

Create a LunchOrder application that prompts the user for the number of hamburgers, salads, french fries, and sodas and then displays the total for the order. The LunchOrder application should include a Food object with a constructor that accepts the price, fat, carbs, and fiber for an item. Food methods should return the price of the item and return the fat, carbohydrates, and fiber. Use the chart below for food prices and nutrition information:

| Item | Price | Fat(g) | Carbohydrates(g) | Fiber(g) |
|---|---|---|---|---|
| hamburger | $1.85 | 9 | 33 | 1 |
| salad | $2.00 | 1 | 11 | 5 |
| french fries | $1.30 | 11 | 36 | 4 |
| soda | $0.95 | 0 | 38 | 0 |

Application output should look similar to:

```
Enter number of hamburgers: 3
Each hamburger has 9.0g of fat, 33.0g of carbs, and 1.0g of fiber.

Enter number of salads: 4
Each salad has 1.0g of fat, 11.0g of carbs, and 5.0g of fiber.

Enter number of fries: 2
French fries have 11.0g of fat, 36.0g of carbs, and 4.0g of fiber.

Enter number of sodas: 5
Each soda has 0.0g of fat, 38.0g of carbs, and 0.0g of fiber.

Your order comes to: $20.90
```

## Exercise 4 — DiceRollGame

In the Dice Roll game, the player begins with a score of 1000. The player is prompted for the number of points to risk and a second prompt asks the player to choose either high or low. The player rolls two dice and the outcome is compared to the player's choice of high or low. If the dice total is between 2 and 6 inclusive, then it is considered "low". A total between 8 and 12 inclusive is "high". A total of 7 is neither high nor low, and the player loses the points at risk. If the player had called correctly, the points at risk are doubled and added to the total points. For a wrong call, the player loses the points at risk. Create a DiceRollGame application that uses a DRPlayer object based on this specification. The DRPlayer object should have two Die member variables that represent the dice. The Die class should use a random number generator to determine the outcome in a roll() method. Application output should look similar to:

```
You have 1000 points.
How many points do you want to risk? (-1 to quit) 10
Make a call (0 for low, 1 for high): 1
You rolled: 10
You now have 1020 points.
How many points do you want to risk? (-1 to quit) 20
Make a call (0 for low, 1 for high): 1
You rolled: 11
You now have 1060 points.
How many points do you want to risk? (-1 to quit) 20
Make a call (0 for low, 1 for high): 1
You rolled: 7
You now have 1040 points.
How many points do you want to risk? (-1 to quit) -1
```

## Exercise 5 — Nim2

The game of Nim starts with a random number of stones between 15 and 30. Two players alternate turns and on each turn may take either 1, 2, or 3 stones from the pile. The player forced to take the last stone loses. Use object-oriented development to create a Nim2 application that allows the user to play Nim against the computer. The Nim2 application and its objects should:

- Generate the number of stones to begin with.
- Allow the player to go first.
- Use a random number generator to determine the number of stones the computer takes.
- Prevent the player and the computer from taking an illegal number of stones. For example, neither should be allowed to take three stones when there are only 1 or 2 left.

## Exercise 6 — GameOf21

In the Game of 21, a player is dealt two cards from a deck of playing cards and then optionally given a third card. The player closest to 21 points without going over is the winner. Use object-oriented development to create a Game of 21 application that allows the user to play the Game of 21 against the computer. The Game of 21 application and its objects should:

- Deal a card from a deck of playing cards by generating a random number between 1 and 13. A 1 corresponds to an Ace, numbers 2 through 10 correspond to those cards, and 11 through 13 correspond to Jack, Queen, and King. The Jack, Queen, and King have a value of 10 in the Game of 21. An Ace can have a value of either 1 or 11.
- Allow the player to stay with two cards or be given a third card.
- Announce the winner.
- Play rounds until the player says to stop.

## Exercise 7 — Bowling

In bowling, a ball is rolled down a lane, also called an alley, at a set of ten pins. A game consists of a bowler bowling for ten frames, where each frame consists of two chances (throws) to knock over all ten pins. Bowling centers often use computers to electronically keep scores for bowlers. Use object-oriented development to create a Bowling application that simulates a simplified game of bowling. The Bowling application and its objects should:

- Allow a bowler to bowl ten frames. Each frame consists of two throws, unless a strike is thrown.
- Award 20 points to the bowler when all ten pins are knocked over on the first throw of a frame.
- Award 15 points to the bowler when all ten pins are knocked over within the two throws of a frame.
- Award one point for each pin knocked over in the two throws of a frame when all ten pins are not knocked over.
- If there is more than one bowler in a game, then the bowlers take turns until each has bowled ten frames.
- Use a random number generator to determine how many pins a bowler has knocked over with each throw.
- Display an updated score after each frame.

## Exercise 8 — Adder

The Adder game prompts a player for the answer to an addition problem. The Adder game creates a problem from two randomly selected integers between 0 and 20. Adder allows the player three tries to enter a correct answer. If the correct answer is entered on the first try, the player is awarded 5 points. If the correct answer is entered on the second try, 3 points are awarded. The correct answer on the third try earns 1 point. If after three tries, the correct answer is still not entered, the player receives no points and the correct answer is displayed. The game continues until `999` is entered as an answer. At the end of the game, Adder displays the player's score. Application output should look similar to:

```
20 + 5 = 25
20 + 2 = 20
Wrong answer. Enter another answer: 22
3 + 6 = 5
Wrong answer. Enter another answer: 4
Wrong answer. Enter another answer: 7
5 + 12 = 17
14 + 18 = 999
Your score is: 13
```

## Programming Challenges


VideoNote
The Employee and Production-worker Classes Problem

### 1. `Employee` and `ProductionWorker` Classes

Design a class named `Employee`. The class should keep the following information in fields:

- Employee name
- Employee number in the format XXX–L, where each X is a digit within the range 0–9 and the L is a letter within the range A–M.
- Hire date

Write one or more constructors and the appropriate accessor and mutator methods for the class.

Next, write a class named `ProductionWorker` that inherits from the `Employee` class. The `ProductionWorker` class should have fields to hold the following information:

- Shift (an integer)
- Hourly pay rate (a `double`)

The workday is divided into two shifts: day and night. The shift field will be an integer value representing the shift that the employee works. The day shift is shift 1 and the night shift is shift 2. Write one or more constructors and the appropriate accessor and mutator methods for the class. Demonstrate the classes by writing a program that uses a `ProductionWorker` object.

### 2. `ShiftSupervisor` Class

In a particular factory a shift supervisor is a salaried employee who supervises a shift. In addition to a salary, the shift supervisor earns a yearly bonus when his or her shift meets production goals. Design a `ShiftSupervisor` class that inherits from the `Employee` class you created in Programming Challenge 1. The `ShiftSupervisor` class should have a field that holds the annual salary and a field that holds the annual production bonus that a shift supervisor has earned. Write one or more constructors and the appropriate accessor and mutator methods for the class. Demonstrate the class by writing a program that uses a `ShiftSupervisor` object.

### 3. `TeamLeader` Class

In a particular factory, a team leader is an hourly paid production worker who leads a small team. In addition to hourly pay, team leaders earn a fixed monthly bonus. Team leaders are required to attend a minimum number of hours of training per year. Design a `TeamLeader` class that inherits from the `ProductionWorker` class you designed in Programming Challenge 1. The `TeamLeader` class should have fields for the monthly bonus amount, the required number of training hours, and the number of training hours that the team leader has attended. Write one or more constructors and the appropriate accessor and mutator methods for the class. Demonstrate the class by writing a program that uses a `TeamLeader` object.

### 4. `Essay` Class

Design an `Essay` class that inherits from the `GradedActivity` class presented in this chapter. The `Essay` class should determine the grade a student receives on an essay. The student's essay score can be up to 100 and is determined in the following manner:

Grammar: 30 points
Spelling: 20 points
Correct length: 20 points
Content: 30 points

Demonstrate the class in a simple program.

### 5. Course Grades

In a course, a teacher gives the following tests and assignments:

- A **lab activity** that is observed by the teacher and assigned a numeric score.
- A **pass/fail exam** that has 10 questions. The minimum passing score is 70.
- An **essay** that is assigned a numeric score.
- A **final exam** that has 50 questions.

Write a class named `CourseGrades`. The class should have a `GradedActivity` array named `grades` as a field. The array should have four elements, one for each of the assignments previously described. The class should have the following methods:

| | |
|---|---|
| `setLab`: | This method should accept a `GradedActivity` object as its argument. This object should already hold the student's score for the lab activity. Element 0 of the `grades` field should reference this object. |
| `setPassFailExam`: | This method should accept a `PassFailExam` object as its argument. This object should already hold the student's score for the pass/fail exam. Element 1 of the `grades` field should reference this object. |
| `setEssay`: | This method should accept an `Essay` object as its argument. (See Programming Challenge 4 for the `Essay` class. If you have not completed Programming Challenge 4, use a `GradedActivity` object instead.) This object should already hold the student's score for the essay. Element 2 of the `grades` field should reference this object. |
| `setFinalExam`: | This method should accept a `FinalExam` object as its argument. This object should already hold the student's score for the final exam. Element 3 of the `grades` field should reference this object. |
| `toString`: | This method should return a string that contains the numeric scores and grades for each element in the `grades` array. |

Demonstrate the class in a program.

### 6. `Analyzable` Interface

Modify the `CourseGrades` class you created in Programming Challenge 5 so it implements the following interface:

```
public interface Analyzable
{
   double getAverage();
   GradedActivity getHighest();
   GradedActivity getLowest();
}
```

The `getAverage` method should return the average of the numeric scores stored in the `grades` array. The `getHighest` method should return a reference to the element of the grades array that has the highest numeric score. The `getLowest` method should return a reference to the element of the grades array that has the lowest numeric score. Demonstrate the new methods in a complete program.

### 7. `Person` and `Customer` Classes

Design a class named `Person` with fields for holding a person's name, address, and telephone number. Write one or more constructors and the appropriate mutator and accessor methods for the class's fields.

Next, design a class named `Customer`, which inherits from the `Person` class. The `Customer` class should have a field for a customer number and a `boolean` field indicating whether the customer wishes to be on a mailing list. Write one or more constructors and the appropriate mutator and accessor methods for the class's fields. Demonstrate an object of the `Customer` class in a simple program.

### 8. `PreferredCustomer` Class

A retail store has a preferred customer plan where customers can earn discounts on all their purchases. The amount of a customer's discount is determined by the amount of the customer's cumulative purchases in the store, as follows:

- When a preferred customer spends $500, he or she gets a 5% discount on all future purchases.
- When a preferred customer spends $1,000, he or she gets a 6% discount on all future purchases.
- When a preferred customer spends $1,500, he or she gets a 7% discount on all future purchases.
- When a preferred customer spends $2,000 or more, he or she gets a 10% discount on all future purchases.

Design a class named `PreferredCustomer`, which inherits from the `Customer` class you created in Programming Challenge 7. The `PreferredCustomer` class should have fields for the amount of the customer's purchases and the customer's discount level. Write one or more constructors and the appropriate mutator and accessor methods for the class's fields. Demonstrate the class in a simple program.

### 9. `BankAccount` and `SavingsAccount` Classes

Design an abstract class named `BankAccount` to hold the following data for a bank account:

- Balance
- Number of deposits this month
- Number of withdrawals
- Annual interest rate
- Monthly service charges

The class should have the following methods:

| | |
|---|---|
| Constructor: | The constructor should accept arguments for the balance and annual interest rate. |
| `deposit`: | A method that accepts an argument for the amount of the deposit. The method should add the argument to the account balance. It should also increment the variable holding the number of deposits. |
| `withdraw`: | A method that accepts an argument for the amount of the withdrawal. The method should subtract the argument from the balance. It should also increment the variable holding the number of withdrawals. |

`calcInterest`: A method that updates the balance by calculating the monthly interest earned by the account, and adding this interest to the balance. This is performed by the following formulas:

*Monthly Interest Rate = (Annual Interest Rate/12)*
*Monthly Interest = Balance * Monthly Interest Rate*
*Balance = Balance + Monthly Interest*

`monthlyProcess`: A method that subtracts the monthly service charges from the balance, calls the `calcInterest` method, and then sets the variables that hold the number of withdrawals, number of deposits, and monthly service charges to zero.

Next, design a `SavingsAccount` class that extends the `BankAccount` class. The `SavingsAccount` class should have a status field to represent an active or inactive account. If the balance of a savings account falls below $25, it becomes inactive. (The `status` field could be a `boolean` variable.) No more withdrawals can be made until the balance is raised above $25, at which time the account becomes active again. The savings account class should have the following methods:

`withdraw`: A method that determines whether the account is inactive before a withdrawal is made. (No withdrawal will be allowed if the account is not active.) A withdrawal is then made by calling the superclass version of the method.

`deposit`: A method that determines whether the account is inactive before a deposit is made. If the account is inactive and the deposit brings the balance above $25, the account becomes active again. The deposit is then made by calling the superclass version of the method.

`monthlyProcess`: Before the superclass method is called, this method checks the number of withdrawals. If the number of withdrawals for the month is more than 4, a service charge of $1 for each withdrawal above 4 is added to the superclass field that holds the monthly service charges. (Don't forget to check the account balance after the service charge is taken. If the balance falls below $25, the account becomes inactive.)

### 10. `Ship`, `CruiseShip`, and `CargoShip` Classes

Design a `Ship` class that the following members:

- A field for the name of the ship (a string)
- A field for the year that the ship was built (a string)
- A constructor and appropriate accessors and mutators
- A `toString` method that displays the ship's name and the year it was built

Design a `CruiseShip` class that extends the `Ship` class. The `CruiseShip` class should have the following members:

- A field for the maximum number of passengers (an `int`)
- A constructor and appropriate accessors and mutators

- A `toString` method that overrides the `toString` method in the base class. The `CruiseShip` class's `toString` method should display only the ship's name and the maximum number of passengers.

Design a `CargoShip` class that extends the `Ship` class. The `CargoShip` class should have the following members:

- A field for the cargo capacity in tonnage (an `int`)
- A constructor and appropriate accessors and mutators
- A `toString` method that overrides the `toString` method in the base class. The `CargoShip` class's `toString` method should display only the ship's name and the ship's cargo capacity.

Demonstrate the classes in a program that has a `Ship` array. Assign various `Ship`, `CruiseShip`, and `CargoShip` objects to the array elements. The program should then step through the array, calling each object's `toString` method. (See Code Listing 9-25 as an example.)