Practices for Lesson 11

Practices Overview

In these practices, you will use the multithreaded features of the Java programming language.

Practice 11-1: Summary Level: Synchronizing Access to Shared Data

Overview

In this practice, you will add code to an existing application. You must determine whether the code is run in a multithreaded environment, and, if so, make it thread-safe.

Assumptions

You have reviewed the sections covering the use of the `Thread` class and the `synchronized` keyword of this lesson.

Summary

You will open a project that purchases shirts from a store. The file-reading code will be provided to you. Your task is to add the appropriate exception handling code.

Tasks

1. Open the project `Synchronized` as the main project.
   a. Select File > Open Project.
   b. Browse to `D:\labs\11\practices`. `(or your other folder)`
   c. Select `Synchronized` and select the "Open as Main Project" check box.
   d. Click the Open Project button.
2. Expand the project directories but avoid opening and review the provided classes at this point. You will attempt to discover whether this application is multithreaded by observing the behavior of code that you provide.
3. Create a `PurchasingAgent` class in the `com.example` package.
4. Complete the `PurchasingAgent` class.
   a. Add a `purchase` method.

```
public void purchase() {}
```

   b. Complete the `purchase` method. The purchase() method should:
      - Obtain a `Store` reference. Note that the `Store` class implements the Singleton design pattern.

```
Store store = Store.getInstance();
```

- Buy a `Shirt`.
  - Verify that the store has at least one shirt in stock.

```
store.getShirtCount()
```

  - Use the store to authorize a credit card purchase. Use a credit card account number of "1234" and a purchase amount of 15.00. A `boolean` result is returned.

```
store.authorizeCreditCard("1234", 15.00)
```

  - If there are shirts in stock and the credit card purchase was authorized, you should take a shirt from the store.

```
Shirt shirt = store.takeShirt();
```

  - Print out the shirt and a success message if a shirt was acquired or a failure message if one was not.

5. Run the project multiple times. Note that the store contains only a single shirt. You can see many possible variations of output. You might see:
   - Two success messages and two shirts (output may appear in varying order)
   - Two success messages, one shirt, and one null
   - Two success messages, one shirt, and one exception
   - One success message, one shirt, and one failure message (desired behavior, but least likely)
6. Discover how the `PurchasingAgent` class is being used.
   a. Use a constructor and a print statement to discover how many instances of the `PurchasingAgent` class are being created when running the application.

      Reminder: Sometimes objects are created per-request and sometimes an object may be shared by multiple requests. The variations in the model affect which code must be thread-safe.

   b. Within the `purchase` method use the `Thread.currentThread()` method to obtain a reference to the thread currently executing the `purchase()` method. Use a single print statement to print the name and ID of the executing thread.

   c. Run the project and observe the output.
7. Open the `Store` class and add a delay to the `authorizeCreditCard` method.
   - Obtain a random number in the range of 1–3, the number of seconds to delay. Print a message indicating how many seconds execution will be delayed.

```
int seconds = (int) (Math.random() * 3 + 1);
```

   - Use the appropriate static method in the `Thread` class to delay execution for 1 to 3 seconds.

   Optional Task: What if your delay is interrupted? How can you be sure that execution is delayed for the desired number of seconds? Or should a different action be taken?
8. Run the project multiple times. You should see a stack trace for a `java.util.NoSuchElementException`. Locate the line within the `com.example.PurchasingAgent.purchase` method that is displayed in the stack trace. Review the action occurring on that line.

---

9. Use a `synchronized` code block to create predictable behavior.
    - Modify the `purchase` method in the `PurchasingAgent` class to contain a synchronized code block.

      Note: Adding `synchronized` to the method signature or using a `synchronized` block that uses the `this` object's monitor will not work.
10. Run the project. You should now see the desired behavior. In the output window, you should see one success message, one shirt, and one failure message.

Practice 11-1: Detailed Level: Synchronizing Access to Shared Data

Overview

In this practice, you will add code to an existing application. You must determine whether the code is run in a multithreaded environment, and, if so, make it thread-safe.

Assumptions

You have reviewed the sections covering the use of the `Thread` class and the `synchronized` keyword of this lesson.

Summary

You will open a project that purchases shirts from a store. The file-reading code will be provided to you. Your task is to add the appropriate exception handling code.

Tasks

1. Open the project `Synchronized` as the main project.
   a. Select File > Open Project.
   b. Browse to `D:\labs\11\practices`. `(or your other folder)`
   c. Select `Synchronized` and select the "Open as Main Project" check box.
   d. Click the Open Project button.
2. Expand the project directories but avoid opening and review the provided classes at this point. You will attempt to discover whether this application is multithreaded by observing the behavior of code that you provide.
3. Create a `PurchasingAgent` class in the `com.example` package.
4. Complete the `PurchasingAgent` class.
   a. Add a `purchase` method. The purchase() method should:
      - Obtain a `Store` reference. Note that the `Store` class implements the Singleton design pattern.
      - Buy a `Shirt`.
        - Verify that the store has at least one shirt in stock.
        - Use the store to authorize a credit card purchase. Use a credit card account number of "1234" and a purchase amount of 15.00. A `boolean` result is returned.
        - If there are shirts in stock and the credit card purchase was authorized, you should take a shirt from the store.
        - Print out the shirt and a success message if a shirt was acquired or a failure message if one was not.

```
public class PurchasingAgent {

    public void purchase() {
        Store store = Store.getInstance();
        if (store.getShirtCount() > 0 &&
store.authorizeCreditCard("1234", 15.00)) {
            Shirt shirt = store.takeShirt();
            System.out.println("The shirt is ours!");
            System.out.println(shirt);
        } else {
            System.out.println("No shirt for you");
        }
    }
}
```

5. Run the project multiple times. Note that the store contains only a single shirt. You can see many possible variations of output. You might see:

   - Two success messages and two shirts (output may appear in varying order)

```
Adding a shirt to the store.
Total shirts in stock: 1
The shirt is ours!
The shirt is ours!
Shirt ID: 1
Description: Polo
Color: Rainbow
Size: Large

Shirt ID: 1
Description: Polo
Color: Rainbow
Size: Large
```

   - Two success messages, one shirt, and one null

```
Adding a shirt to the store.
Total shirts in stock: 1
The shirt is ours!
The shirt is ours!
null
Shirt ID: 1
Description: Polo
Color: Rainbow
Size: Large
```

   - Two success messages, one shirt, and one exception

```
Adding a shirt to the store.
Total shirts in stock: 1
The shirt is ours!
Shirt ID: 1
Description: Polo
Color: Rainbow
Size: Large
Exception in thread "Thread-0" java.util.NoSuchElementException
```

- One success message, one shirt, and one failure message (desired behavior but least likely)

```
Adding a shirt to the store.
Total shirts in stock: 1
The shirt is ours!
Shirt ID: 1
Description: Polo
Color: Rainbow
Size: Large

No shirt for you
```

6. Discover how the `PurchasingAgent` class is being used.

   a. In the `PurchasingAgent` class, use a constructor and a print statement to discover how many instance of the `PurchasingAgent` class are being created when running the application.

```
public PurchasingAgent() {
    System.out.println("Creating a purchasing agent");
}
```

   Reminder: Sometimes objects are created per-request and sometimes an object may be shared by multiple requests. The variations in the model affect which code must be thread-safe.

   b. Within the `purchase` method use the `Thread.currentThread()` method to obtain a reference to the thread currently executing the `purchase()` method. Use a single print statement to print the name and ID of the executing thread.

```
Thread t = Thread.currentThread();
System.out.println("Thread:" + t.getName() + "," + t.getId());
```

   c. Run the project and observe the output.

7. Open the `Store` class and add a delay to the `authorizeCreditCard` method.
   - `Math.random()` is used to obtain a random number in the range of 1–3, the number of seconds to delay.

```
int seconds = (int) (Math.random() * 3 + 1);
System.out.println("Sleeping for " + seconds + " seconds");
try {
    Thread.sleep(seconds * 1000);
} catch (InterruptedException e) {
    System.out.println("Interrupted");
}
```

8. Run the project multiple times. You should see a stack trace for a `java.util.NoSuchElementException`. Locate the line within the `com.example.PurchasingAgent.purchase` method that is displayed in the stack trace. The exception is being generated by the call to `store.takeShirt()`.

   Note: The delay introduced in the previous step makes it more likely that concurrent `PurchasingAgent.purchase` methods calls will both believe that a shirt can be taken but take the shirt at a different time. Taking the shirt at the near the same time typically results in some of the other errors shown in step 5.

9. Use a `synchronized` code block to create predictable behavior.
   - Modify the `purchase` method in the `PurchasingAgent` class to contain a synchronized code block.

```
synchronized (store) {
    if (store.getShirtCount() > 0 &&
store.authorizeCreditCard("1234", 15.00)) {
        Shirt shirt = store.takeShirt();
        System.out.println("The shirt is ours!");
        System.out.println(shirt);
    } else {
        System.out.println("No shirt for you");
    }
}
```

   Note: Adding `synchronized` to the method signature or using a `synchronized` block that uses the `this` object's monitor will not work.

10. Run the project. You should now see the desired behavior. In the output window, you should see one success message, one shirt, and one failure message.

```
Adding a shirt to the store.
Total shirts in stock: 1
The shirt is ours!
Shirt ID: 1
Description: Polo
Color: Rainbow
Size: Large

No shirt for you
```

Practice 11-2: Summary Level: Implementing a Multithreaded Program

Overview

In this practice, you will create a new project and start a new thread.

Assumptions

You have reviewed the sections covering the use of the `Thread` class.

Summary

You will create a project that slowly prints an incrementing number. A new thread will be used to increment and print the number. The application should wait for Enter to be pressed before interrupting any threads.

Tasks

1. Create a new project `ThreadInterrupted` as the main project.
   a. Select File > New Project.
   b. Select Java under Categories and Java Application under Projects. Click the Next button.
   c. Enter the following information in the "Name and Location" dialog box:
      - Project Name: `ThreadInterrupted`
      - Project Location: `D:\labs\11\practices`. `(or your other folder)`
      - (checked) Create Main Class: `com.example.ThreadInterruptedMain`
      - (checked) Set as Main Project
   d. Click the Finish button.
2. Create a `Counter` class in the `com.example` package.
3. Complete the `Counter` class. The `Counter` class should:
   - Implement the `Runnable` interface.
   - Within the `run` method:
     - Create an `int` variable called `x` and initialize it to zero.
     - Construct a loop that will repeat until the executing thread is interrupted.
     - Within the loop, print and increment the value of `x`.
     - Within the loop, delay for 1 second. Return from the `run` method or exit the loop if the thread is interrupted while delayed.
4. Add the following to the `main` method in the `ThreadInterruptedMain` class:
   - Create a `Counter` instance.
   - Create a thread and pass to its constructor the runnable Counter.
   - Start the thread.

5. Run the project. You should see an incrementing sequence of numbers with a one second delay between each number. Notice that while the `main` method has completed the application continues to run.
6. Stop the project.
   a. Open the Run menu.
   b. Click Stop Build/Run.

   Note: You can also stop a build/run by clicking the red square along the left side of the output window.
7. Modify the project properties to support the `try`-with-resources statement.
8. Modify the main method in the `ThreadInterruptedMain` class.
   - After starting the thread, wait for Enter to be pressed in the output window. You can use the following code:

```
try(BufferedReader br = new BufferedReader(new
InputStreamReader(System.in))) {
    br.readLine();
} catch (IOException e) {}
```

     Note: You may need to fix your imports and update the project properties to support JDK 7 features.
   - Print out a message indicating whether or not the thread is alive.
   - Interrupt the thread.
   - Delay for one second (to allow the thread time to complete) and then print out a message indicating whether or not the thread is alive.
9. Run the project. You should see an incrementing sequence of numbers with a one second delay between each number. Press Enter while the output window is selected to gracefully terminate the application.

Practice 11-2: Detailed Level: Implementing a Multithreaded Program

Overview

In this practice, you will create a new project and start a new thread.

Assumptions

You have reviewed the sections covering the use of the `Thread` class.

Summary

You will create a project that slowly prints an incrementing number. A new thread will be used to increment and print the number. The application should wait for Enter to be pressed before interrupting any threads.

Tasks

1. Create a new project `ThreadInterrupted` as the main project.
   a. Select File > New Project.
   b. Select Java under Categories and Java Application under Projects. Click the Next button.
   c. Enter the following information in the "Name and Location" dialog box:
      - Project Name: `ThreadInterrupted`
      - Project Location: `D:\labs\11\practices. (or your other folder)`
      - (checked) Create Main Class: `com.example.ThreadInterruptedMain`
      - (checked) Set as Main Project
   d. Click the Finish button.
2. Create a `Counter` class in the `com.example` package.
3. Complete the `Counter` class. The `Counter` class should:
   - Implement the `Runnable` interface.
   - Within the `run` method:
     - Create an `int` variable called `x` and initialize it to zero.
     - Construct a loop that will repeat until the executing thread is interrupted.
     - Within the loop, print and increment the value of `x`.
     - Within the loop, delay for 1 second. Return from the `run` method or exit the loop if the thread is interrupted while delayed.

```
int x = 0;
while(!Thread.currentThread().isInterrupted()) {
    System.out.println("The current value of x is: " + x++);
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        return;
    }
}
```

4. Add the following to the `main` method in the `ThreadInterruptedMain` class:
   - Create a `Counter` instance.
   - Create a thread and pass to its constructor the runnable Counter.
   - Start the thread.

```
Runnable r = new Counter();
Thread t = new Thread(r);
t.start();
```

5. Run the project. You should see an incrementing sequence of numbers with a one second delay between each number. Notice that while the `main` method has completed the application continues to run.
6. Stop the project.
   a. Open the Run menu.
   b. Click Stop Build/Run.

   Note: You can also stop a build/run by clicking the red square along the left side of the output window.
7. Modify the project properties to support the `try`-with-resources statement.
   a. Right-click the `ThreadInterrupted` project and click Properties.
   b. In the Project Properties dialog box select the Sources category.
   c. In the Source/Binary Format drop-down list select JDK 7.
   d. Click the OK button.
8. Modify the main method in the `ThreadInterruptedMain` class.
   - After starting the thread, wait for Enter to be pressed in the output window. You can use the following code:

```
try(BufferedReader br = new BufferedReader(new
InputStreamReader(System.in))) {
    br.readLine();
} catch (IOException e) {}
```

- Add the needed import statements.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

- Print out a message indicating whether or not the thread is alive.

```
System.out.println("Thread is alive:" + t.isAlive() );
```

- Interrupt the thread.

```
t.interrupt();
```

- Delay for one second (to allow the thread time to complete), and then print out a message indicating whether or not the thread is alive.

```
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
}
System.out.println("Thread is alive:" + t.isAlive());
```

9. Run the project. You should see an incrementing sequence of numbers with a one second delay between each number. Press Enter while the output window is selected to gracefully terminate the application.