

Lab 1 - TOCTOU

Part 0

See `lab1_solution.zip`.

Part 1: Exploit your program

- **What is the shared resource? Who is sharing it?**

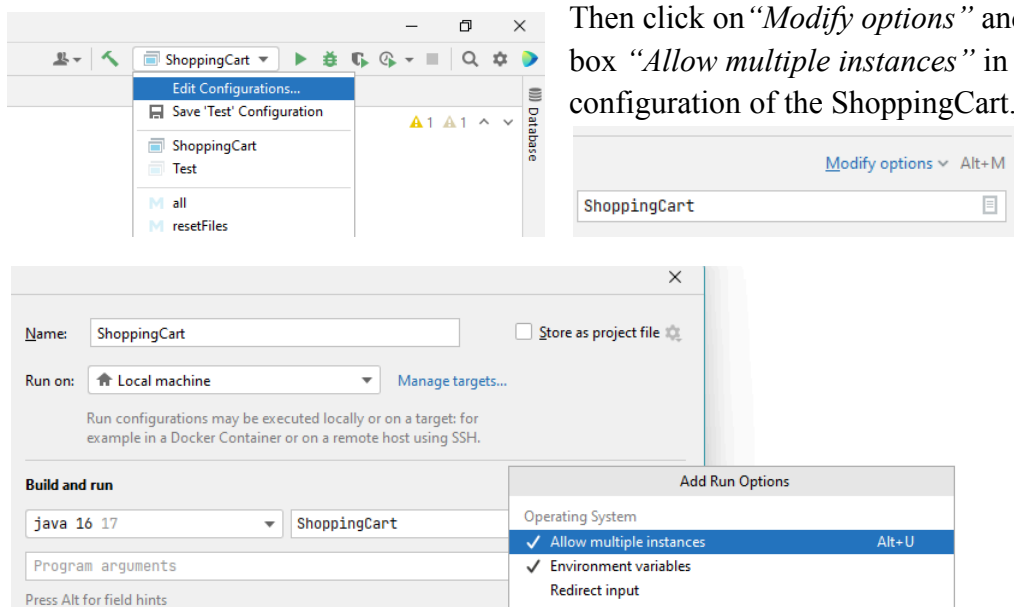
The shared resource is the **wallet.txt** file. It is shared by all users who are executing `ShoppingCart.java`. Another shared resource is the **pocket.txt** file. It is also shared by all users who are executing `ShoppingCart.java`.

- **What is the root of the problem?**

The root of the problem is a data race condition that happens when 1 process reads the balance in the wallet and sees that it is enough to purchase an item. It then proceeds in the code to purchase the item. Before it updates the balance in the wallet by subtracting the balance with the value of the purchased item, another process reads the balance and sees that it is enough to purchase an item. After process 1 updates the balance the money in the updated wallet is not enough for process 2 to purchase the item. However since process 2 checked the balance before process 1 updated it, process 1 also manages to purchase an item even though it should not be allowed.

- **Explain in detail how you can attack this system.**

Allow the `ShoppingCart.java` program to run in multiple instances. This is achieved in IntelliJ by clicking on the “*Edit configurations*” option next to the run button.

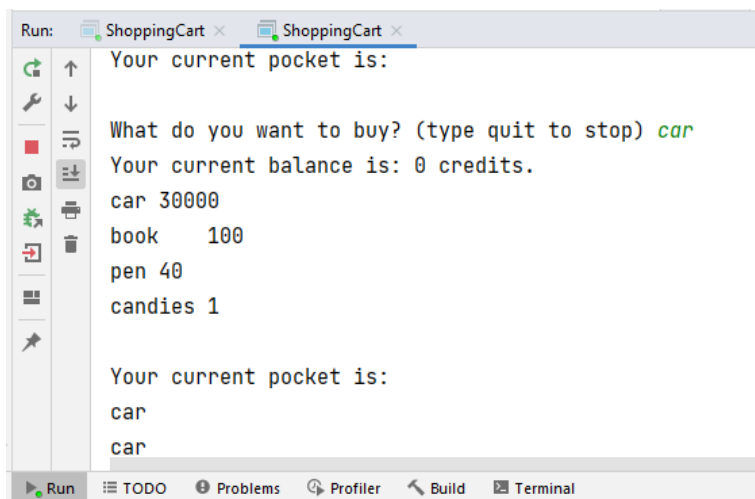


After that, add a call to `Thread.sleep(5000)` in the code of `ShoppingCart.java` before the line of code that updates the balance of the wallet.

```
//Step 6
int newBalance = balance - price;
Thread.sleep( millis: 5000);
wallet.setBalance(newBalance);
```

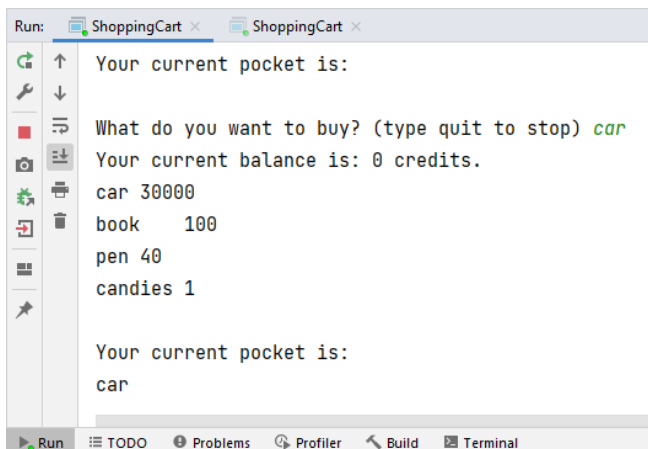
Then, run 2 instances of `ShoppingCart.java`.

In the first instance, purchase a car. Open the second running instance of the program, and purchase a car there as well(before 5 seconds has passed in the first program). After a few seconds, you will see that the pocket will contain 2 cars.



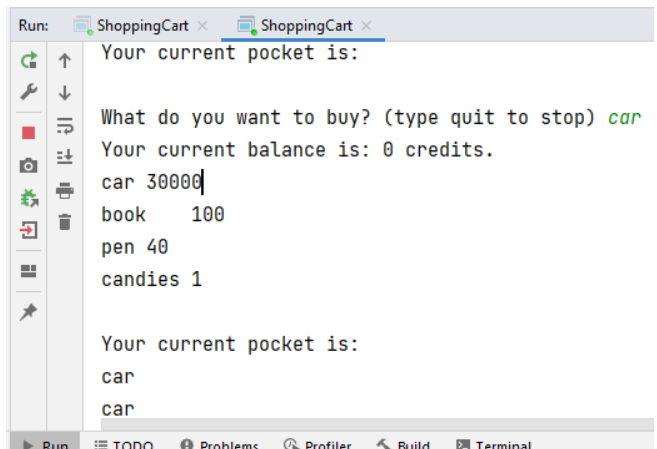
```
Run: ShoppingCart x ShoppingCart x
Your current pocket is:
What do you want to buy? (type quit to stop) car
Your current balance is: 0 credits.
car 30000
book 100
pen 40
candies 1
Your current pocket is:
car
car
```

- Provide the program output and result, explaining the interleaving to achieve them.



```
Run: ShoppingCart x ShoppingCart x
Your current pocket is:
What do you want to buy? (type quit to stop) car
Your current balance is: 0 credits.
car 30000
book 100
pen 40
candies 1
Your current pocket is:
car
```

Program instance 1



```
Run: ShoppingCart x ShoppingCart x
Your current pocket is:
What do you want to buy? (type quit to stop) car
Your current balance is: 0 credits.
car 30000
book 100
pen 40
candies 1
Your current pocket is:
car
car
```

Program instance 2

The resulting pocket contains 2 cars. The resulting credit is 0 credits. The reason this works is because process 2 checks the balance before process 1 has updated the balance after purchasing the car. This means that process 2 sees the balance as 30000 and thus purchases a car as well, even though process 1 has purchased the car and the balance should be 0 for process 2.

Part 2: Fix the API

Create this method in `Wallet.java`:

```
public boolean safeWithdraw(int valueToWithdraw) throws Exception {
    try {
        int balance = getBalance();
        // Acquire the lock
        fileLock = fileChannel.lock();
        // Check if sufficient balance
        if (valueToWithdraw <= balance) {
            // Simulate processing time
            Thread.sleep(5000);
            // Update balance
            setBalance(balance - valueToWithdraw);
            return true;
        } else {
            return false;
        }
    } finally {
        // Release the lock in a finally block to ensure cleanup
        if (fileLock != null && fileLock.isValid()) {
            fileLock.release();
        }
    }
}
```

Rewrite ShoppingCart.java to use the safeWithdraw() method from Wallet.java. Inside ShoppingCart.java, replace step 5,6 and 7 with:

```
if (wallet.safeWithdraw(price)) {  
    pocket.addProduct(product);  
} else  
{  
    break;  
}
```

For the full code that secures the buggy API, see *lab2_solution.zip*.

- Were there other APIs or resources suffering from possible races? If so, please explain them and update the APIs to eliminate any race problems.

Although the Wallet.java class was the biggest issue when it came to data race, since it allowed a user to purchase items whose value exceeded their balance, there were other possible data races in the program also. This was the case with the shared pocket.txt file. The addProduct() method of the Pocket class could cause a race condition where 1 process called the function and read the from the pocket.txt file. Before it updated it with an item, another process would also call the addProduct() function and read from the pocket.txt file. The result was that the 2 processes would write to the same line in the pocket.txt file, effectively making 1 process overwrite the line that another process had written. To prevent this bug, we implemented a file lock on the addProduct function similar to the file lock on the safeWithdraw() function in the wallet class.

Lastly, the getter methods in both the Wallet.java and Pocket.java classes needed to have file locks, this included the getBalance() method in the Wallet class and the getPocket() method in the Pocket class. The reason is not that they cause a data race in a similar way as the setter methods which could lead to an exploit, but instead that they can cause an exception to occur if 2 processes purchase an item at the same time. When one of the processes tries to call the getter methods, it will be locked by the process having the file lock, and so the process will get an exception and the program will crash.

- When eliminating all race problems, it is important to not implement the protections too aggressively, as it could lead to performance hits in the real world. This is something you should consider in the lab.

We only put the part of the code that needs to be protected from concurrent access inside the file lock in all methods that use file lock. For example, the `getBalance()` method our `Wallet` class looked like this:

```
public int getBalance() throws IOException {  
    this.file.seek(0);  
  
    try {  
        fileLock = fileChannel.lock();  
        return Integer.parseInt(this.file.readLine());  
    } finally {  
        if (fileLock != null && fileLock.isValid()) {  
            fileLock.release();  
        }  
    }  
}
```

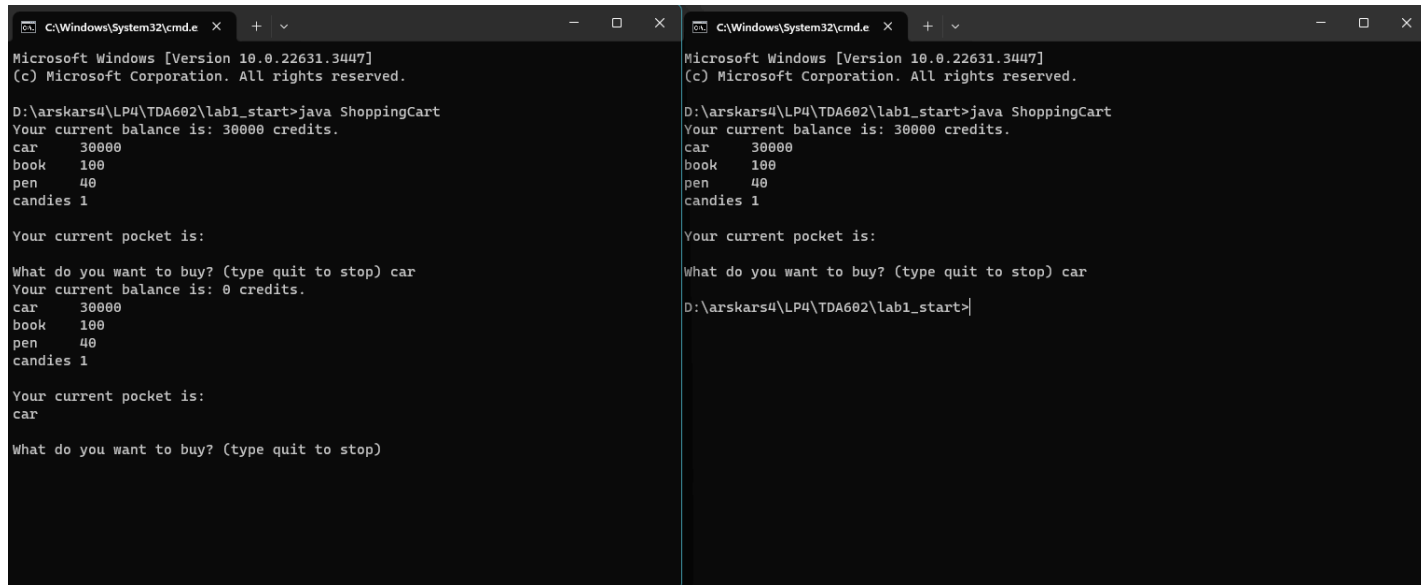
We can see that we put the line of code `this.file.seek(0);` outside of the file lock since it did not need to be protected by a file lock. All other code needed to be protected by file lock.

- Why are these protections enough and at the same time not too excessive? Make sure to test that the attack from Part 1 now fails.

The protection is enough and not too excessive since it locks only the part of code that needs to be protected from concurrent access inside the methods that are protected by file locks.

- Make sure that your solution works outside any IDE you might use.

We tried on windows cmd and it did not work at first. We realized that it was because we needed to build the project to get the updated class files. When running using intelliJ, this was not necessary. So we used the command “*make all*” in git bash which rebuilt all files including ShoppingCart.java. After that, the output was as expected and data race was prevented.



```
C:\Windows\System32\cmd.exe x + v
Microsoft Windows [Version 10.0.22631.3447]
(c) Microsoft Corporation. All rights reserved.

D:\arskars4\LP4\TDA602\lab1_start>java ShoppingCart
Your current balance is: 30000 credits.
car      30000
book     100
pen       40
candies  1

Your current pocket is:

What do you want to buy? (type quit to stop) car
Your current balance is: 0 credits.
car      30000
book     100
pen       40
candies  1

Your current pocket is:
car

What do you want to buy? (type quit to stop)

C:\Windows\System32\cmd.exe x + v
Microsoft Windows [Version 10.0.22631.3447]
(c) Microsoft Corporation. All rights reserved.

D:\arskars4\LP4\TDA602\lab1_start>java ShoppingCart
Your current balance is: 30000 credits.
car      30000
book     100
pen       40
candies  1

Your current pocket is:

What do you want to buy? (type quit to stop) car
D:\arskars4\LP4\TDA602\lab1_start>
```