Design-by-Contract Activity
Explore Defensive Programming

Name: ____
Name: ____
One CM: ___________

1) Study the design and implementation of *dequeueTwo* (below) and note:

- It is a standalone operation
- It has a contract and a non-trivial *requires* clause
- The implementation assumes the calling operation has met the *requires* clause
- By visual inspection, it is arguably correct
- *dequeueTwo* handles all possible legal configurations of an incoming queue except queues whose length is ≤ 1

```
//! updates q
//! replaces y, z
//! requires |q| > 1
//! ensures q = #q[2,|#q|) and <y> = #q[0,1) and <z> = #q[1,2)
void dequeueTwo(QueueOfInteger& q, Integer& y, Integer& z)
{
   q.dequeue(y);
   q.dequeue(z);
} // dequeueTwo
```

2) Examine various defensive versions of *dequeueTwo*

Definition: Defensive Programming – an operation that is implemented so that it checks its own precondition

2.1 Defensive Version #1

To do:

1. Utilizing the definition above, is *dequeueTwoDefensiveV1* defensive? Yes, because it checks its precondition with an if statement
2. Update *dequeueTwoDefensiveV1*'s contract based on its implementation
   Hint: You might need to utilize additional logical operators, e.g., implication, or, not, etc.

```
void dequeueTwoDefensiveV1(QueueOfInteger& q, Integer& y, Integer& z)
{
   if (q.length() > 1) {
      q.dequeue(y);
      q.dequeue(z);
   } // end if
} // dequeueTwoDefensiveV1
```

Put *dequeueTwoDefensiveV1*'s contract here:

```
//! updates q, y, z
~~//! replaces~~
//! requires    |q| > 1 // Note: true would work but recommend sticking with the non-trivial requires clause
//! ensures     ((|q| > 1) → (q = #q[2,|#q|) and <y> = #q[0,1) and <z> = #q[1,2)))
//!             and
//!             (~(|q| > 1) → (q = #q and y = #y and z = #z))
```

To do:
List all the ways you can think of that the calling operation can determine if *dequeueTwoDefensiveV1* did anything or not?
In this case the calling operation of *dequeueTwoDefensiveV1* might obtain the length of the queue prior to the call and then compare that to the length after the call.

In general, there is not a good way for a calling operation to detect a *do nothing* situation.

Activity continued on the back

2.2 Defensive Version #2

To do: Update *dequeueTwoDefensiveV2*'s contract based on its implementation

```
void dequeueTwoDefensiveV2(QueueOfInteger& q, Integer& y, Integer& z, Boolean& successful)
{
   successful = (q.length() > 1);
   if (successful) {
      q.dequeue(y);
      q.dequeue(z);
   } // end if
} // dequeueTwoDefensiveV2
```

Put *dequeueTwoDefensiveV2*'s contract here:

```
//! updates q, y, z
~~//! replaces~~
//! requires   |q| > 1  // Note: true would work but recommend sticking with the non-trivial requires clause
//! ensures    ((|q| > 1) → (q = #q[2,|#q|) and <y> = #q[0,1) and <z> = #q[1,2) and successful))
//!         and
//!            (~(|q| > 1) → (q = #q and y = #y and z = #z and ~successful))
```

To do:
List all the ways you can think of that the calling operation can determine if *dequeueTwoDefensiveV2* did anything or not?

Check the Boolean parameter for true or false

2.3 Defensive Version #3

To do: Update *dequeueTwoDefensiveV3*'s contract based on its implementation

```
void dequeueTwoDefensiveV3(QueueOfInteger& q, Integer& y, Integer& z)
{
   if (q.length() > 1) {
      q.dequeue(y);
      q.dequeue(z);
   }
   else {
      throw EmptyQueueException();
   } // end if
} // dequeueTwoDefensiveV3
```

Put *dequeueTwoDefensiveV3*'s contract here:

```
//! updates q, y, z
~~//! replaces~~
//! requires   |q| > 1  // Note: true would work but recommend sticking with the non-trivial requires clause
//! ensures    ((|q| > 1) → (q = #q[2,|#q|) and <y> = #q[0,1) and <z> = #q[1,2)))
//!            and
//!            (~(|q| > 1) → (q = #q and y = #y and z = #z and throw an exception))
```

Note: "*and throw an exception*" adds an English description of what the operation will do in this non-steady-state situation, so now the ensures clause is now not a completely mathematical statement

To do:
List all the ways you can think of that the calling operation can determine if *dequeueTwoDefensiveV3* did anything or not?
Wrap up the call to dequeueTwoDefensiveV3 in a try-catch block