Section 1:

Consider Queue.java[1].

```
public class Queue <E> {

    private List<E> elements;
    private int size;

    public Queue() {
        this.elements = new ArrayList<E>();
        this.size = 0;
    }

    public void enQueue (E e) {
        elements.add(e);
        size++;
    }

    public E deQueue () {
        if (size == 0) throw new IllegalStateException("Queue.deQueue");
        E result = elements.get(0);
        elements.remove(0);
        size--;
        return result;
    }

    public boolean isEmpty() {
        return size == 0;
    }
}
```

Question 1.1: For enQueue, write (i) a partial contract and (ii) a total contract. For each part, if you need to change the code for the contract, do so and explain what you did

Here I am denoting some parts of the contract as public and others as private. This is to indicate which (public) parts of the contract should be visible to the caller versus which (private) parts are only needed internally to ensure a proper implementation. For example, the caller does not need to know that the elements field is non-null, since the elements field should be hidden from the caller entirely via encapsulation.

Partial Contract:

- Preconditions:
    - (private) Field elements is not null.
    - (private) Field size equals elements.size().
    - (private) Field size >= 0.
    - (private) Field size < 2147483647, because otherwise size would overflow.
- Modifies: this
- Postconditions:
    - (public) The reference “e” will be pushed onto the top of the stack.

[1] Obtained from “2.4 In class 2A”: https://nguyenthanhvuh.github.io/class-oo/schedule.html

    - (public) The size of the stack will be incremented by one.
    - (private) Field size will still equal elements.size().
    - (private) elements.size() == (prior elements.size() + 1).
    - (private) elements.get(size - 1) == reference “e”.

Total Contract:

- Preconditions: None; otherwise, it would not be a total contract!
- Modifies: this
- Postconditions (hierarchical list):
    - (private) Field elements is not null..
        - I would add an assertion or guava Verify-ication to verify this.
    - (private) Field size equals elements.size().
        - I would add an assertion or guava Verify-ication to verify this.
    - (public) If the queue is ready to overflow, throw an IllegalStateException .
        - In older days, this would be an “impossibility”; however, with heap sizes growing to truly unimaginable levels (TBs, see here), checking the upper bound could be seen as future proofing.
        - The queue is ready to overflow, if size == 2147483647.
    - (public) The reference “e” will be pushed onto the top of the stack.
    - (public) The size of the stack will be incremented by one.
    - (private) Field size will still equal elements.size().
        - I would add an assertion or guava Verify-ication to verify this.
    - (private) elements.size() == (prior elements.size() + 1).
        - This is more documentation, since adding the item to the list implies this is true.
    - (private) elements.get(size - 1) == reference “e”.
        - This is more documentation to ensure the implementation adds the element to the correct end of the list.

Question 1.2: Write the rep invs for this class. Explain what they are.

Note: Upon creating the rep-invariant, some of the preconditions and postconditions could be removed, as they do not need to be stated in both places. However, I am under the impression that the questions are to be answered in order.

- Field *elements* is not null.
- Field *size* equals elements.size().
- Field *size* >= 0 and size <= 2147483647.
- The elements in the queue are ordered from head to tail, where tail is the most recently added element.
- If the elements list is non empty, then the head of the queue is at elements[0].

A representation invariant constrains the number of valid internal states of the Queue object, by placing constraints on the values of the fields therein, and by placing constraints on the relationships of the fields therein.

Question 1.3: Write a reasonable toString() implementation. Explain what you did

In this particular case, one could simply reuse the ArrayList's toString() in order to be reasonably consistent with the behavior of classes in the Java Collection Framework. In that case, a Queue containing the elements 100, 200, and 300, would have a string representation of "[100, 200, 300]".

```
@Override
public String toString ()
{
    return elements.toString();
}
```

The downside to the aforesaid approach is that it is not an "Abstraction Function" in the strictest sense, as defined by Liskov, because it does not map all of the object's abstract state onto the string. More specifically, the string representation does not contain the "size" field explicitly. Thus, we should modify the aforesaid implementation to be a little more in line with Liskov, as follows.

```
@Override
public String toString ()
{
    return String.format("size = %d; elements = %s", size, elements.toString());
}
```

Question 1.4: Consider a new method, deQueueAll(), which does exactly what the name suggests. Write a reasonable contract for this method and then implement it. Be sure to follow Bloch's advice with respect to generics. Explain what you did

First, we need to decide on what the signature of the method shall be:

- public void deQueueAll()
    - In this case, the method would behave like Queue.clear(). The elements would be dequeued, but not returned.
- public int deQueueAll()
    - In this case, the method would still behave like Queue.clear(), but the number of dequeued elements would be returned.
- public E[] deQueueAll(Class<? super E> type)
    - In this case, the method would deque the elements, but return them as an array. In order to allow the type of the array to match type E, the caller would have to pass in a Class object. If the caller passed in a Class representing a supertype of E, then the method would still work, since Java arrays are covariant.
- public <T super E> T[] deQueueAll(T[] array)

    - This option would be similar to the previous one, but is more consistent with the Collection.toArray(T[]) method.
- public Iterable<E> deQueueAll()
    - Bloch says to prefer Lists over arrays. However, I take this to mean prefer collection types over arrays, not necessarily Lists per se. To that end, one may consider returning an Iterable, which is basically the most abstract option. This will allow the implementation maximum flexibility as to what data-structure to use for the return type. The Guava library has really influenced me to use Iterable in more places, as it frees one from only thinking in terms of the Java Collections Framework (JCF).
- public Collection<E> deQueueAll()
    - However, Iterable<E> can be hard to pass into JCF constructors. For example, using Iterable<E> would make things like “new HashSet<>(queue.deQueueAll())” impossible. Thus, we could consider the use of Collection<E> as the return type.
- public List<E> deQueueAll()
    - In general, one wants the return type of a method to be more specific, and the parameter type(s) to be less specific. In this case, one can make an argument that the order of the dequeued elements is an important piece of information to retain. Therefore, we want a list type in order to preserve the ordering.
    - According to Bloch, “*If appropriate interface types exist, then parameters, return values, variables, and fields should all be declared using interface types.*”; therefore, the return type should be List<E>, rather than a specified implementation type such as ArrayList<E>.

Contract:

- Preconditions: None
- Modifies: this
- Postconditions (hierarchical list):
    - (public) If the queue is empty, throw IllegalStateException.
    - (public) After the method returns, the queue will be empty.
    - (public) An immutable list of the elements, which were removed by the call, will be returned.
    - (public) The returned elements will be in the same order as they previously appeared in the queue.
    - (private) size = 0
    - (private) elements.size() == 0

There are two ways to implement this.

I would probably use this approach, since I would also make the *elements* field final. I prefer to make as many fields as final as possible, since it makes the class easier to understand. In addition, returning a standardized immutable list (i.e. one created by List.copyOf()) can have some performance advantages later on, if the calling method also performs a copyOf(), for example, because the library can detect that the list is immutable.

```
public List<E> deQueueAll ()
{
    if (isEmpty()) { throw new IllegalStateException("empty queue"); }

    // Note: List.copyOf() is a new method in JRE 10,
    // which creates an *immutable* copy of elements.
    final List<E> retval = List.copyOf(elements);
    elements.clear();
    size = 0;
    return retval;
}
```

That said, the following implementation is arguably a little more performant, as it avoids the need to allocate an entire copy of the list, but I believe is less readable.

```
public List<E> deQueueAll ()
{
    if (isEmpty()) { throw new IllegalStateException("empty queue"); }

    // In this case, retval will be effectively immutable,
    // because the next line replaces the mutable one.
    final List<E> retval = Collections.unmodifiableList(elements);
    elements = new ArrayList<>();
    size = 0;
    return retval;
}
```

As you can see, in both cases, I would return an immutable list, because:

1. Returning defensive copies is a good practice, in general, according to Bloch.
2. Returning an immutable list prevents the caller from becoming dependent on the performance characteristics of subsequent insertions into the returned list.

The size field should just be removed from the class in general. The size field can be replaced by elements.size(). The contract would be simpler then.

Question 1.5: Rewrite the deQueue() method for an immutable version of this class. Explain what you did

In order to make the method apply to an immutable variant of Queue, we need to:

- Do not modify the given internal state of “this” during the operation.
- Return a “modified copy” of the Queue, which represents the change in state. In this particular case, we do not actually need to “copy” anything, because deQueueAll() resets the queue back to its initial empty state.

```
public Queue<E> deQueueAll ()
{
    return new Queue<>();
}
```

Question 1.6: Write a reasonable implementation of clone(). Explain what you did.

```
@SuppressWarnings("unchecked")
public Queue<E> clone ()
{
    try
    {
        final Queue<E> clone = (Queue<E>) super.clone();
        clone.elements = new ArrayList(this.elements);
        return clone;
    }
    catch (CloneNotSupportedException ex)
    {
        throw new AssertionError("never happens");
    }
}
```

Comments:

- I added `@SuppressWarnings("unchecked")`, because `(Queue<E>)` is an “unchecked cast”, which we know will always succeed in this case.
- super.clone() creates an object that is a “shallow copy” of “this”. In other words, both the original “this” and the clone both reference the exact same “elements” list object. That is a problem, since any subsequent modifications of the clone would also modify the original! Thus, I manually copied the elements list after calling super.clone().

Personally, I have never found a use for clone() in the real world. My experience is that clone() has been relegated to merely being the whipping boy of a bygone era. Thus, I foresee continuing to follow Bloch’s advice, as follows, and implement copy constructors whenever the need arises.

Bloch (Chapter 3):

> *“The copy constructor approach and its static factory variant have many advantages over Cloneable/clone: they don’t rely on a risk-prone extralinguistic object creation mechanism; they don’t demand unenforceable adherence to thinly documented conventions; they don’t conflict with the proper use of final fields; they don’t throw unnecessary checked exceptions; and they don’t require casts.”*

Section 2:

Consider Bloch's final version of his Chooser example, namely GenericChooser.java.

```
// List-based Chooser - typesafe

public class Chooser<T> {

    private final List<T> choiceList;

    public Chooser(Collection<T> choices) {
        choiceList = new ArrayList<>(choices);
    }

    public T choose() {
        Random rnd = ThreadLocalRandom.current();
        return choiceList.get(rnd.nextInt(choiceList.size()));
    }
}
```

Question 2.1: What would be good rep invariants for this class? Explain each.

- choiceList is not null.
    - You actually need a list in order to store the choices.
- choiceList.size() equals the number of choices.
    - Similarly to what we saw w.r.t. the elements field in the Stack example of Question #3 of this exam, the size of choiceList could theoretically be larger than the number of choices actually stored therein, if the class was implemented differently.
- choiceList.size() > 0
    - In order to make a choice, there actually must be a choice to choose.
    - I did not go with "choiceList.size() > 1", because sometimes you could want to cause the probability of a choice being selected to be 100%.

I also considered adding "choiceList does not contain null" as a rep-invariant. However, I believe that this class could serve as a utility class in situations where a null choice may be useful. An example would be randomly selecting inputs for test cases.

I also considered adding "choiceList does not contain duplicates". One could make an argument either way for the existence of this rep-invariant. In favor of adding the constraint, one could argue that duplicates affect the probability that a specified choice will be selected. If one wants all of the choices to have equal probability, then there should be no duplicates. On the other hand, one can argue that allowing the probability to be affected eases use of the class in the common case, and is generally understandable to the reader.

Question 2.2: Supply suitable contracts for the constructor and the choose() method and recode if necessary. The contracts should be consistent with your answer to the previous question. Explain exactly what you are doing and why.

Contract of the Constructor:

- Preconditions: None
- Postconditions:
    - If “choices” is null, then throw a NullPointerException.
    - If “choices” is empty, then throw an IllegalArgumentException.
    - “this” will contain the available choices for later random selection therefrom.

Contract of choose():

- Preconditions: None
- Postconditions:
    - A randomly selected element from the collection of choices will be returned, such that each element in the collection of choices is given equal probability. If the same choice is stored (N) times herein, then that choice will be returned with (N) times the probability of a choice that is only stored once herein.

Question 2.3. Argue that the choose() method, as documented and possibly updated in your previous answers, is correct. You don’t have to be especially formal, but you do have to ask (and answer) the right questions.

First, let me present my modified version of the class.

```
public final class Chooser<T> {

    private final List<T> choiceList;

    public Chooser(Collection<T> choices)
    {
        if (choices == null) { throw new NullPointerException(“choices is null”); }
        if (choices.isEmpty()) { throw new IllegalArgumentException(“empty choices”); }
        choiceList = new ArrayList<>(choices);
    }

    public T choose()
    {
        Random rnd = ThreadLocalRandom.current();
        return choiceList.get(rnd.nextInt(choiceList.size()));
    }
}
```

Let us look at each constraint imposed by the contract and evaluate whether it is obeyed. If so, then the program is correct, since “*correctness”* means that the code obeys its specification.

Contract of the Constructor:

- Preconditions: None
    - Since there are no preconditions, the code cannot possibly violate them.
- Postconditions:
    - If “choices” is null, then throw a NullPointerException.
        - There is an if-statement that enforces this; therefore, this constraint is obeyed.
    - If “choices” is empty, then throw an IllegalArgumentException.
        - There is an if-statement that enforces this; therefore, this constraint is obeyed.
    - “this” will contain the available choices for later random selection therefrom.
        - The constructor makes a defensive copy of the list, and stores it for later use; therefore, this constraint is obeyed.

Contract of choose():

- Preconditions: None
    - Since there are no preconditions, the code cannot possibly violate them.
- Postconditions:
    - A randomly selected element from the collection of choices will be returned, such that each element in the collection of choices is given equal probability. If the same choice is stored (N) times herein, then that choice will be returned with (N) times the probability of a choice that is only stored once herein.
        - The code uses a ThreadLocalRandom to retrieve and return one of the elements; therefore, this constraint is obeyed.

Rep Invariant:

- choiceList is not null.
    - The constructor always creates a new list and assigns it to the field. Null is never subsequently assigned to the field. Therefore, this constraint is obeyed.
- choiceList.size() equals the number of choices.
    - The choices themselves are provided as a collection given to the constructor. The constructor then copies them into an equally sized list. The constructor does not somehow “preallocate” a list that is larger than the number of choices. Further, the list is never modified after construction. Therefore, this constraint is obeyed.
- choiceList.size() > 0
    - The constructor throws an IllegalArgumentException, if the collection of choices is of size zero. A negative size is not possible. Since size of the list will always equal the number of choices given to the constructor, this constraint is also obeyed.

In addition, notice that I made the class *final*. Per Bloch, one should document for inheritance; otherwise, prohibit it. If one does not prohibit inheritance, then someone could do something

such as the following, which my team and I discussed a variant of on Discord while studying for the final exam.

```
public MutableChooser<T> extends Chooser<T>

    private List<T> myElements;

    public MutableChooser(Collection<T> elements)
    {
        super(elements);
        this.myElements = new ArrayList<>(elements);
    }

    public T gaussianChoose()
    {
        Random rnd = ThreadLocalRandom.current();
        // More or less correct, I hope. :-)
        int random = Math.abs(Math.floor(ThreadLocalRandom.current().nextGaussian());
        return myElements.get(index % myElements.size());
    }

    public void addElement (T element)
    {
        myElements.add(element);
    }
}
```

As you can see, there are now two lists storing the choices, namely the choiceList in the superclass, and myElements in the subclass. Consequently, the MutableChooser will appear to work fine initially, because both lists will contain exactly the same elements. However, over time, the behavior will become unexpected, as elements are added via addElement(). Specifically, choose() (inherited) will never be able to return the newly added elements, because it is using the choiceList (superclass), whereas gaussianChoose() is using myElements (subclass).

My team and I discussed a slightly different variant, yesterday, where the superclass implemented equals() and hashCode(). In that case, the need for making the class final was even clearer.

(Intentionally Blank)

Section 3:

Consider StackInClass.java. Note that the push() method is a variation on Bloch's code.

```
public class Stack {
    private Object[] elements; private int size = 0;

    public Stack() { this.elements = new Object[0]; }

    public void push (Object e) {
        if (e == null) throw new NullPointerException("Stack.push");
        ensureCapacity(); elements[size++] = e;
    }

    public void pushAll (Object[] collection) { for (Object obj: collection) {
push(obj); } }

    public Object pop () {
        if (size == 0) throw new IllegalStateException("Stack.pop");
        Object result = elements[--size];
        elements[size] = null;
        return result;
    }

    @Override public String toString() {
        String result = "size = " + size;
        result += "; elements = [";
        for (int i = 0; i < elements.length; i++) {
            if (i < elements.length-1)
                result = result + elements[i] + ", ";
            else
                result = result + elements[i];
        }
        return result + "]";
    }
}
```

Question 3.1. What is wrong with toString()? Fix it.

The loop relies on the (elements.length), rather than the (size) field. The length may grow when elements are added to the stack, but will not shrink when elements are removed. Thus, the toString() will end up printing out (elements.length - size) number of more elements (which happen to be nulls) than actually are present.

Fixed toString():

```
@Override
public String toString() {
    final StringBuilder string = new StringBuilder();
    string.append("size = ").append(size).append("; elements = [");
    for (int i = 0; i < size; i++)
    {
        string.append(elements[i])
              .append((i < size - 1) ? ",” : ””);
    }
    string.append("]");
    return string.toString();
}
```

Question 3.2. As written, pushAll() requires documentation that violates encapsulation. Explain why and then write a contract for pushAll().

The Stack class is not final; therefore, it may be subclassed. The pushAll() method is defined in terms of the push() virtual method. This must be documented, because overriding the push() method can change the behavior of pushAll() unexpectedly.

Contract:

- Preconditions:
- Modifies: this
- Postconditions:
    - If the collection is null, throw a NullPointerException.
    - After the invocation, all of the elements in the collection will be on the stack, and consequently the size of the stack will be incremented by the number of elements in the collection. Of note, the elements of the collection will be pushed onto the stack in the order defined by the collection's iterator().

Question 3.3. Rewrite the pop() method for an immutable version of the Stack class. Keep the same instance variables. Rewrite what you did.

```
private Stack (final int size,
               final Object[] elements)
{
    this.size = size;
    this.elements = elements;
}

public Object peek ()
{
    if (size == 0)
    {
        throw new IllegalStateException("empty stack");
    }
    else
    {
        return elements[size - 1];
    }
}

public Stack pop ()
{
    if (0 == size)
    {
        throw new IllegalStateException("empty stack");
    }
    else
    {
        final Object[] members = Arrays.copyOfRange(elements, 0, size - 2);
        return new Stack(size - 1, members);
    }
}
```

First, I added a private constructor, which allows me to create a modified copy of a Stack.

Second, I created a peek() method, which allows callers to obtain the topmost element from the Stack. This is needed, because the rewritten pop() method will now return a Stack, rather than the element that is on top of the Stack.

Third, I rewrote the pop() method to return a “modified copy” of “this”.

Question 3.4. Implementing the equals() method for this class is a messy exercise, but would be much easier if the array was replaced by a list. Explain why. Note: You are not required to provide an implementation in your answer, but if you find it helpful to do so, that’s fine.

If the array was replaced with a list, then we could reuse the list’s equals method, as follows.

Side Note: This solution assumes that replacing the array with a list also does away with the size field. In theory, one could replace the array with an ArrayList, but leave the size field in place, which would not benefit the equals() method. In short, here, I am assuming that the size of elements changes when elements are both added and removed.

```
@Override
public boolean equals (Object other)
{
    return (other == this)
        || (other instanceof Stack && elements.equals(((Stack) other.elements)));
}
```

Otherwise, we would have to implement the array comparison manually ourselves, as follows:

```
@Override
public boolean equals (Object other)
{
    if (other == this)
    {
        return true;
    }
    else if (other instanceof Stack == false)
    {
        return false;
    }
    else if (size != ((Stack) other).size)
    {
        return false;
    }

    final Stack peer = (Stack) other;
```

```
    for (int i = 0; i < size; i++)
    {
        if (Objects.equals(elements[i], peer.elements[i]) == false)
        {
            Return false;
        }
    }

    return true;
}
```

Section 4:

Consider the program below (y is the input).

```
{y ≥ 1} // precondition

x := 0;
while(x < y)
    x += 2;

{x ≥ y} // post condition
```

Question 4.1: Informally argue that this program satisfies the given specification (pre/post conditions).

In order to show that the program is correct w.r.t. the precondition, we simply need to show that when the precondition is true, then the postcondition is also true. The precondition states that { y >= 1 }. We see that x starts at zero, and then increments, until it is greater than y.

By case analysis, we can informally analyze this loop itself.

- Case #1 - The loop never iterates. We can ignore this case, since { x = 0 & x < y} means that the loop will iterate at least once.
- By substitution, we know that the loop only iterates when { 0 < y } (i.e. 1, 2, 3, …). We can divide this series into two parts, namely the even numbers (2, 4, 6, …), and the odd numbers (1, 3, 5, …).
- Case #2 - The loop iterates, and y is even. In this case, the loop will terminate when { x = y }.
- Case #3 - The loop iterates, and y is odd. In this case, the loop will terminate when { x = y + 1}.

To summarize, the loop will terminate when either { x = y | x = y + 1}, which fulfills the postcondition { x >= y }. Since the postcondition logically follows from the precondition, the program must be correct w.r.t. its specification.

Question 4.2: Give 3 loop invariants for the while loop in this program. For each loop invariant, informally argue why it is a loop invariant.

- x >= 0
  - This is a loop invariant, because x is set to zero before the loop and only incremented by the loop. Thus, after the loop, x must either be zero (loop never entered) or greater than zero.
- y >= 1
  - This is a loop invariant, because it is a precondition, and the loop does not modify the y variable.
- TRUE
  - This is always a loop invariant. :-)

Question 4.3: Sufficiently strong loop invariants: Use a sufficiently strong loop invariant to formally prove that the program is correct with respect to given specification. This loop invariant can be one of those you computed in the previous question or something new.

- **Note: show all works for this step (e.g., obtain weakest preconditions, verification condition, and analyze the verification condition).**
- **Recall that if the loop invariant is strong enough, then you will be able to do the proof. In contrast, if it is not strong enough, then you cannot do the proof.**

In order to compute the WP, we first must compute the WP of the loop.

```
WP(while [I] b do S, {Q})
  = I
  & ((I &  b) => WP(S, I))
  & ((I & !b) => Q)

b: x < y
Q: x >= y

Conjunctive Case 1 = I:

y >= 1

Conjunctive Case 2 = ((I & b) => WP(S, I)):

  I & b => WP(x := x + 2, y >= 1))
= I & b => (y >= 1)
= (y >= 1) & (x < y) => (y >= 1)
= !((y >= 1) & (x < y)) | (y >= 1)
= (x >= y) | (y < 1) | (y >= 1)
```

```
Conjunctive Case 3 = ((I & !b) => Q):

  I & !b => (x >= y)
= (y >= 1) & !(x < y) => (x >= y)
= (y >= 1) & (x >= y) => (x >= y)
= !((y >= 1) & (x >= y)) | (x >= y)
= (x < y) | (y < 1) | (x >= y)

Thus, the WP of the loop itself is
= (Conjunctive Case 1) & (Conjunctive Case 2) & (Conjunctive Case 3)
= (y >= 1) & ((x >= y) | (y < 1) | (y >= 1)) & ((x < y) | (y < 1) | (x >= y))

Compute the WP of the statement list given the WP of the loop:

WP(x := 0; while [y >= 0] x < y do x := x + 2 { x >= y })
= WP(x := 0; WP(while [y >= 0] x < y do x := x + 2, { x >= y }))
= WP(x := 0; (y >= 1) & ((x >= y) | (y < 1) | (y >= 1)) & ((x < y) | (y < 1) | (x >= y)))
= (y >= 1) & ((0 >= y) | (y < 1) | (y >= 1)) & ((0 < y) | (y < 1) | (0 >= y))
   (Due to the first conjunctive clause, we know y >= 1)
= (y >= 1) & (~~(0 >= y) | (y < 1)~~ | (y >= 1)) & ((0 < y) | ~~(y < 1)~~ | ~~(0 >= y)~~)
= (y >= 1) & (y >= 1) & (0 < y)
= (y >= 1) & (y >= 1) & (y >= 0)
= (y >= 1) & (y >= 0)
= (y >= 1)

Compute the verification condition vc (P => WP(...)):

  P => WP(x := 0; WP(while [y >= 0] x < y do x := x + 2, { x >= y }))
= P => (y >= 1)
= (y >= 1) => (y >= 1)
= TRUE

Analyze the vc to determine whether the program is proved or not:
```

As you can see, if we assume that the precondition (y >= 1) holds true, then the verification condition will also be true, because the precondition is one of the disjunctive clauses within the verification condition. Therefore, the program is proven correct.

Question 4.4: Insufficiently strong loop invariants: Use another loop invariant (could be one of those you computed previously) and show that you cannot use it to prove the program. Note: show all work as the previous question.

In order to compute the WP, we first must compute the WP of the loop.

```
WP(while [I] b do S, {Q})
  = I
  & ((I &  b) => WP(S, I))
  & ((I & !b) => Q)

b: x < y
Q: x >= y

Conjunctive Case 1 = I:

x >= 0

Conjunctive Case 2 = ((I & b) => WP(S, I)):

  I &  b => WP(x := x + 2, x >= 0))
= I & b => ((x + 2) >= 0)
= (x >= 0) & (x < y) => ((x + 2) >= 0)
= (x >= 0) & (x < y) => (x >= -2)
= !((x >= 0) & (x < y)) | (x >= -2)
= ((x < 0) | (x >= y)) | (x >= -2)
= (x < 0) | (x >= y) | (x >= -2)

Conjunctive Case 3 = ((I & !b) => Q):

  I & !b => (x >= y)
= (x >= 0) & !(x < y) => (x >= y)
= (x >= 0) & (x >= y) => (x >= y)
= !((x >= 0) & (x >= y)) | (x >= y)
= !((x >= 0) & (x >= y)) | (x >= y)
= ((x < 0) | (x < y)) | (x >= y)
= (x < 0) | (x < y) | (x >= y)


Thus, the WP of the loop itself is
= (Conjunctive Case 1) & (Conjunctive Case 2) & (Conjunctive Case 3)
= (x >= 0) & ((x < 0) | (x >= y) | (x >= -2)) & ((x < 0) | (x < y) | (x >= y))
= (x >= 0) & (~~(x < 0) |~~ (x >= y) ~~| (x >= -2)~~) & (~~(x < 0) |~~ (x < y) | (x >= y))
= (x >= 0) & (x >= y) & ((x < y) | (x >= y))
= (x >= 0) & (x >= y) & (~~(x < y) |~~ (x >= y))
= (x >= 0) & (x >= y) & (x >= y)
= (x >= 0) & (y < 0)
```

```
Compute the WP of the statement list given the WP of the loop:

WP(x := 0; while [x >= 0] x < y do x := x + 2 { x >= y })
= WP(x := 0; WP(while [x >= 0] x < y do x := 2, { x >= y }))
= WP(x := 0; (x >= 0) & (y < 0))
= (0 >= 0) & (y < 0)
= TRUE & (y < 0)
= (y < 0)

Compute the verification condition vc (P => WP(...)):

  P => WP(x := 0; WP(while [x >= 0] x < y do x := x + 2, { x >= y }))
= P => (y < 0)
= (y >= 1) => (y < 0)
= FALSE

Analyze the vc to determine whether the program is proved or not:
```

Clearly, the program is not proven, since the vs is false.

(Intentionally Blank)

Section 5:

Note: you can reuse your answers/examples in previous questions to help you answer the following questions.

Question 5.1: What does it mean that a program (or a method) is correct? Give (i) an example showing a program (or method) is correct, an (ii) an example showing a program (or method) is incorrect.

A correct program or method is one that obeys its contract (preconditions, invariants, postconditions).

Example Correct Method:

```
// Preconditions:
// + student is not null.
// + correct_answers >= 0
// + correct_answers <= total_answers
//
// Postconditions:
// + Returns the student's score [0..100] for the semester.
// + Returns a perfect score (100), if the student is Mackenzie High.
public double computeFinalGrade (String student,
                                 int correct_answers,
                                 int total_answers)
{
    if ("Mackenzie High".equalsIgnoreCase(student))
    {
        return 100;
    }
    else
    {
        return 100 * (((double) correct_answers) / total_answers);
    }
}
```

Blatantly Incorrect Method:
This method does not obey its most important postcondition (assigning my score). Since I have answered fewer questions correctly than the total number of questions, this method will return a value that is less than one hundred. However, per the contract, the method should return (100).

```
// Preconditions:
// + student is not null.
// + correct_answers >= 0
// + correct_answers <= total_answers
//
// Postconditions:
// + Returns the student's score [0..100] for the semester.
// + Returns a perfect score (100), if the student is Mackenzie High.
public double computeFinalGrade (String student,
                                 int correct_answers,
                                 int total_answers)
{
    return 100 * (((double) correct_answers) / total_answers);
}
```

Question 5.2: Explain the difference between rep invariants, loop invariants, and contract/specifications (i.e., pre/post conds). Use concrete examples to demonstrate the difference.

Rather than giving a rote textbook answer, let me explain how I think of them.

When I think of preconditions, invariants, postconditions, of any form, I think of them as being *constraints* on the *state space* of a program. For example, imagine we have the following two integer variables.

```
int X;
int Y;
```

Since an int can have 2**32 possible values (i.e. states), we know that a program consisting of two variables can have (2**32)**2 states.

Wow, that is a lot! However, a program consisting of two variables and no operations is pointless. However, every program, like this toy example, has a state space whether we realize it or not. I view our job as programmers as trying to (1) limit that state space to only the states that are useful, (2) defining orderly transitions between the states.

For example, we could say, constrain Y to be equal to 2**2**X, and greater than zero. This would limit Y from being in any one of 2**32 states to only be in one of three possible states, namely 3, 5, 17, and 257, 65537 (i.e. the Fermat numbers that will fit in an int).

That said, computer programs change state as they are executed. Thus, it is not possible to simply instruct the computer to constrain the variables, and then ask it for the answer. Instead, we must write computer programs that take input states and produce output states. That is where preconditions and postconditions come into play.

Let us take for example the following function in some C-like language:

```
long fib (long N) { return (N == 1 || N == 2) ? 1 : fib(N-1) + fib(N-2); }
```

Without getting enamored with details, we can say that the function has two variables of type long, namely (N) which is clearly visible to the reader, and the return value which is implicit.

We can think of the function as being a *temporal constraint* on the return value. In other words, after fib(X) executes, the return value must be in one of a specific set of states (a fibonacci number). We call this state the *postcondition* of function fib.

Thus, *postconditions* allow us to reason about what happens *over time*. For example, in the following code, we know that Z will be equal to (21). How do we know this?

- According to the postcondition of fib(6), Y must be equal to (8).
    - Recall that the postcondition says the return value is the Nth fibonacci number.
- According to the postcondition of fib(8), Z must be equal to (21).

```
int X = 6;
int Y = fib(X);
int Z = fib(Y)
```

The *postconditions* are not restricted to merely stating what the return value will be. Rather, they indicate what that state of part of the system (function, method, object) will be after some fragment of code is executed.

Sometimes an operation is not well defined for all inputs. In that case, the precondition may be used to map an input state onto an “error state”, such as raising an exception or returning an indicator (-1 in C, for example).

However, sometimes programmers deem it safe to “assume” that the input state is acceptable, and will not preclude the postcondition from becoming true. Such an assumption is known as a *precondition*. For example, in the fib() function above, we assumed that (N >= 1). Thus, one can say that the fib() function has the precondition (N >= 1). If you study the fib() function, you will see that providing it with a negative input will result in wildly inappropriate behavior performance wise.

To summarize what has been said thus far:

- A *precondition* is an assumption about the input state before a code fragment executes.
- A *postcondition* describes the output state that will result from the execution.

Sometimes there are constraints that should hold true both before and after a code fragment executes. These types of constraints are known as *invariants*.

One particular type of invariant is a so-called *loop invariant*. A loop invariant is a constraint that must hold true before and after the execution of a given loop. For example, in the following code (X != 0) is a loop invariant, because it is true before the loop is entered, and will be true after the loop regardless of whether the loop iterates or not.

```
int X = 1;
for (int i = 0; i < 10; i++)
{
    X = X + 1;
}
```

As we have seen, preconditions, postconditions, and loop invariants are temporal constraints that are used to describe how the state of an executing program changes over time. However, this concept can be expanded upon even farther.

Once we introduce user-defined data-structures into a language, it becomes important to be able to describe the states in which the data-structure can exist. For example, let us assume that we are creating a Plain Old Java Object describing a Person. A person has a name, an age, a weight, tand a height. A rep-invariant allows us to place restrictions on the person data-type, such as “0 <= weight < 1000” and “0 <= age < 150”.

Again, to summarize what has been said thus far:

- A *precondition* is a constraint that is assumed to be true before some code executes.
- A *postcondition* is a constraint that must be true after the code executes.
- A *loop invariant* is a special type of constraint that holds true before and after a loop.
- A *rep-invariant* is a special type of constraint that restricts the states of a data-structure.

Question 5.3: What are the benefits of using JUnit Theories compared to standard JUnit tests. Use examples to demonstrate your understanding.

In a standard JUnit test, one generally supplies a set of example values to a method in order to determine whether the method behaves as expected. For example, given fib() defined above, one may write a test that verifies whether fib(1) = 1, fib(2) = 1, fib(3) = 2, fib(4) = 3, fib(5) = 5, and fib(6) = 8. If all of these tests pass, then the programmer may conclude that the method works as intended according to its specification.

On the other hand, JUnit Theories, and other property-based testing frameworks, are designed to auto generate the input instances to test the method upon. In other words, the programmer specifies a property that must be true, such as the output of fib(N) is the Nth fibonacci number. The framework then generates test inputs, and uses them to search for violations of the property.

The frameworks are more adept at generating the input test samples than often is. For example, the framework may generate all possible inputs, or randomly generate a large set of wide ranging inputs when an exhaustive analysis would be impossible computationally.

Consequently, such frameworks are often able to find missed edge cases, which the programmer either did not know about or did not bother to test. In the fib() example, the framework would likely find that fib(some large number) overflows, fib(0) is undefined, and fib(negative number) combinatorially explodes.

Question 5.4: Explain the differences between proving and testing.

Testing involves running a program, supplying it with a set of inputs, and then determining whether the program behaves correctly as a result. This can miss important edge cases, since the program may not be supplied with all possible problematic inputs.

On the other hand, static analysis analyzes the source code of a program in order to determine whether it obeys its specification given all possible inputs.

In summary, testing builds confidence that a program is correct, whereas static analysis guarantees that the program is correct.

In addition, if you cannot prove (e.g., using Hoare logic), then what does that mean about the program (e.g., is it wrong)?

The program is not necessarily wrong.

For example, the Halting Problem is famous for showing that proving a program will halt is computationally undecidable in the general case. In other words, no algorithm can possibly exist that will always be able to prove the Total Correctness of all programs. This is because there are some pathological programs that would be incorrect when they *continue a little bit farther in execution*; however, no generalized algorithm can figure out whether the program in question would actually *continue a little bit farther in execution*.

Question 5.5: Explain the Liskov Substitution Principle (LSP). Use a concrete example to demonstrate LSP. Note: use a different example than the one given in Liskov.

In short, the LSP states that wherever a type T is used, it should be possible to replace T with another type S, where S is a supertype of T, without changing the correctness of a program. In order to ensure that this property holds, the LSP places restrictions on subtypes. Specifically, subtypes must have signatures and specifications that are compatible with those of their supertypes.

For example, let use consider the following contrived supertype:

```
public interface Rocket
{
    // Returns null, if the launch failed;
    // otherwise, returns the orbital trajectory of a rock launch.
    public OrbitalPath launch (LaunchParams params);
}
```

We may define an implementation, such as the following:

```
public final class DeltaIV implements Rocket
{
    public OrbitalPath launch (LaunchParams params)
    {
        if (params.success())
        {
            return fancyMath();
        }
        else
        {
            return null;
        }
    }
}
```

The aforesaid implementation can safely be substituted wherever a Rocket is required, because it obeys the contract, and the signature of the launch() method is compatible.

On the other hand, we may define another implementation, such as the following.

```
public final class SpaceShuttle implements Rocket
{
    Vector launch (ShuttleLaunchParams params) throws LaunchAbortException
    {
        if (params.success())
        {
            return fancyMath();
        }
        else
        {
            throws new LanchAbortException();
        }
    }
}
```

The aforesaid implementation violates the LSP, because it is signature incompatible with Rocket for four reasons.

1. The launch() method in the subtype has stricter access (package-private) than the launch method() in the supertype.
2. The launch() method returns a Vector, which is not a subclass of class OrbitalPath. In other words, the return type of launch() in the subtype is not covariant w.r.t. the return type of launch() in the supertype.
3. The launch() method throws a checked exception in the subtype, but not in the supertype.
4. The launch() method in the subtype takes ShuttleLaunchParams as a parameter, which is a more specific type (i.e. ShuttleLaunchParams subtypes LaunchParams) then is done in the supertype.

Again, we may try to implement the Rocket interface, as follows:

```
public final class Falcon9 implements Rocket
{
    public OrbitalPath launch (LaunchParams params)
    {
        if (params.success())
        {
            return fancyMath();
        }
        else
        {
            throw new DragonAbortException();
        }
    }
}
```

In this implementation, the launch() method in the subtype is signature compatible with the overridden one from the supertype. However, the postcondition defined in the supertype has been violated. Specifically, the postcondition says to return null on a launch failure. On the other hand, the implementation throws an unchecked exception on launch failure. Therefore, the subtype would behave differently/unexpectedly from what is expected by reading the supertype's specification.

(Intentionally Blank)

Section 6:

This question helps me determine the grade for group functioning. It does not affect the grade of this final.

Question 6.1: Who are your group members?

Question 6.2: For each group member, rate their participation in the group on the following scale: (a) Completely absent, (b) Occasionally attended, but didn't contribute reliably, (c) Regular participant; contributed reliably

Group: #6 Members:

- Lucas Leitz (lleitz@gmu.edu)
- Luke Young (lyoung29@gmu.edu)
- Mozhgansadat Momtaz Dargahi (mmomtazd@gmu.edu)
- M. Mackenzie High (mhigh@gmu.edu)

Each member of my group always attended and always contributed. We quickly developed a flow and took on specialized tasks within the group early in the semester.

For the reading reflections, we would each read the book independently. We then created a Google Doc in which we collaboratively wrote and reviewed. As you can probably tell, I like to write; therefore, I tried to add a real-world experience story to each reflection to show depth of understanding. Finally, on each Sunday night, we held a phone call and a Discord chat to discuss the document in more detail and prepare for submitting it. Late in the semester, we did away with the phone calls entirely, as Discord proved to be more efficient (maybe that is a generational thing).

For the assignments, we basically split into two subteams. Mozhgan and I served as the development subteam, while Luke and Lucas handled the testing/review/submission. Basically that meant that Mozhgan and I would write up the code late Saturday night or more usually Sunday afternoon. Then, Luke and Lucas would test/review/submit it before class on Monday. This worked out quite well. In particular, I am working full-time and taking another class simultaneously; therefore, having Luke and Lucas be able to review and submit the assignment on Monday helped to free up my Saturday to focus on the other class.

In short, I was quite happy with all of my team members. Honestly, given that we are leaving with a *perfect team score*[2], I think that it is self-evident that we worked quite well together.

[2] Thanks for that!

Section 7:

There is no right or wrong answer for the below questions, but they can help me improve the class. I might present your text verbatim (but anonymously) to next year's students when they are considering taking the course (e.g., in the first week of class) and also add your advice to the project description pages.

Question 7.1:

What were your favorite and least aspects of this class? Favorite topics?

- Favorite: Reading Reflections, because I like to write.
- Least Favorite: Manual Hoare Logic (extremely tedious). This may be something an animation would help to teach better, and reduce the amount of time spent on it.

Question 7.2: Favorite things the professor did or didn't do?

- No rush on quizzes, etc. No matter how well I do on assignments, or anything non-timed, I always freeze up under time pressure. This has cost me so many points over the years that I would not want to count. The longer time limits here reduces stress, which enables one to more clearly think about and articulate answers.
- The TA was quite helpful. This was the first class where someone asked why I did not submit a quiz (hit save, instead of submit).

Question 7.3: What would you change for next time?

- I think that more topics could be covered, if less of the semester was spent fixated on a few topics, such as equals() / hashCode(). In particular, I think clone() is a relic of the past, which could be summarized as "Don't Use This".
- Other aspects of object oriented design could be covered, such as design patterns.
- A more explicit and direct introduction to SOLID.
- The MapPoly example should be replaced. I believe the concepts would be clearer, if they were introduced using more simplistic examples. For example, the professor's class Dog example for covariance/contravariance was a *good example*.
- From life experience, more emphasis should be given to teaching interfaces, and why they are useful. equals() and hashCode() are going to get auto-generated most of the time. On the other hand, it is surprising how many developers seem to avoid interfaces, and detest composition over inheritance.
- I would add an assignment on using Coverity or some other static analysis tool.
- I am not sure what would be the best approach exactly, but I would put far more emphasis on the fact that these principles can be applied to other languages as well. This class is not really a "learn Java" class. Rather, this is a "learn good OOP principles" class, but comes off as being a little too Java centric.

All that said, this was my favorite class this semester. Way back in 2010, I took CS 332 with Paul Ammann, which I would rank as one of the two most influential in the undergrad CS curriculum. This class seems very similar to my recollection of that class, but this class was more approachable and fun, which I would argue can make it more educational.