Pre-Class Work 2: Pre/Post-Conditions and Exceptions

Pre-Conditions [Background Reading]

□ Motivation

Imagine it's 1928 and you're working as a human computer to help calculate whether the Afsluitdijk (a new dam in the Netherlands) can be successfully built. Your contract simply describes your job responsibilities as:

> Will calculate the square root of a given number.

Being so good at your job, when your boss asks what the square root of 6561 is, you can promptly inform them that the answer is 81. Work is great until one day your boss asks what the square root of -9 is. Your boss doesn't understand imaginary numbers, so you inform them that there isn't a square root of -9, or any negative number. However, your boss says that this isn't an answer and that you've violated your contract.

As a result, you're fired and forced to look for work elsewhere. You find a new job at NASA doing the same work as before but you've learned from the past and make sure your contract has an extra clause:

> Will calculate the square root of a given number.
>
> The given number must be greater than or equal to 0.

This one condition you've added has prevented an angry and confused boss from ever expecting impossible results. The same principle this imaginary contract uses can be applied to our code comments to ensure clients don't expect the impossible from our code. This extra "clause" in our comments is called a **pre-condition**.

□ What is a Pre-Condition?

A **pre-condition** has two key elements:

1. A statement used to inform the client about the **accepted range of input**. Only input that meets these criteria is **guaranteed** to produce a correct output. All other input (input outside the specified criteria) is not guaranteed to compute correctly.
2. Code that verifies the pre-condition is met before running any computations.

✏ Relevant Examples

Mathematical Ranges

Let's go back to the square root example. Imagine we wanted to write a trivial method for calculating square roots. It might look like this:

```
// Behavior: Calculates the square root of any given double input
// Parameter: double value to be square rooted
// Returns: the square root of the given parameter as a double
public static double squareRoot(double input) {
    double answer = Math.sqrt(input);
    return answer;
}
```

`Math.sqrt()` is a built in Java function for calculating the square root of a given double. Documentation for those interested can be found here. You'll notice this documentation makes use of **pre-conditions** and **post-conditions**!

While this method works well for positive numbers, remember that square roots can only be calculated for numbers greater than or equal to 0.

Therefore, if the client passes in a negative `double` like -9.0 then our code will produce an **unexpected behavior** even though the client did everything we asked by passing in a `double`. To avoid this problem we need to tell the client about the range of input data we can correctly produce output on (numbers >= 0). We should also write a check into our code to make sure we don't needlessly calculate the square root on any invalid input.

Doing so, our code might look like this:

```
// Behavior: Calculates the square root of a positive double input
// Parameter (input): double value to be square rooted.
// Pre-Condition: input must be >= 0
// Returns: the square root of the given parameter as a double
public static double sqaureRoot(double input) {
    double answer = -1.0;
    if (input >= 0.0) {
        answer = Math.sqrt(input);
    }
    return answer;
}
```

Notice now that we only calculate the square root when `input` is >= 0. All other times (when input is < 0) we return -1. Even though the square root of -9 definitely isn't -1 we only promised to calculate the square root if the input was in the correct range so our code does exactly what we say it will! It would be really nice if the client had some idea of what to expect if they didn't meet the **pre-condition** we set for them, but we'll discuss that in the following slide on **post-conditions**.

States and Types

Pre-conditions can also be used to ensure that the input to a method is of the correct type or meets certain requirements. For example, a method that adds the number 9 to every index in a given `int` array might run into problems if the array is of size 0 or the reference is `null`. Here is an example of what what such a method might look like after successfully implementing **pre-conditions** to avoid these potential problems.

```
// Behavior: Adds 9 to every index in the given array
// Parameter (nums): int array that will have 9 added to every index
//    Pre-Condition 1: nums is not null
//    Pre-Condition 2: size of nums is > 0
public static void squareRoot(int[] nums) {
    if (nums != null && nums.length > 0) {
        for (int i = 0; i < nums.length; i++) {
            nums[i] += 9;
        }
    }
}
```

Again, it would be great to let the client know what to expect after invalid input is entered but more on that on the next slide!

Main Points

- A **pre-condition** is a "clause" or statement in code comments we create that specifies a range of inputs or conditions in which our method is guaranteed to produce correct output.
- A **pre-condition** is like a contract between the method and client that defines valid input. With **pre-conditions**, clients can avoid generating unexpected behavior from our code.
- We implement **pre-conditions** by checking input in our methods before doing any computations. If the input does not meet the **pre-conditions**, then the method can choose to handle it in different ways, such as by throwing an error or returning a pre-defined value.
- Some sample areas in which we can use **pre-conditions** include mathematical ranges (e.g. non-negative input for square root calculation), input checking (e.g. making sure an array is not null and has at least 1 element), etc.

Post-Conditions [Background Reading]

Motivation

Using the same imaginary scenario where you work as a human computer lets explore another possible problem you may run into. Your current contract reads:

> Will calculate the square root of a given number.
>
> The given number must be greater than or equal to 0.

Your boss may ask for the square root of 84.2724 and you'll tell them 9.18 accordingly. But what happens when your boss asks you to find the square root of 2? Remember that the square root of 2 is irrational, meaning it goes on forever. Based on the current wording of your contract it's unclear if you need to spend your whole life calculating this problem or if you can simply round to 1.414. It's this uncertainty in your contract that we want to avoid. To fix this problem, in case you ever get a new boss who wants overly precise answers, you ask your boss to rewrite your contract and include an extra condition. This condition will only require you to produce answers rounded to the fourth decimal place.

Your new and final contract now reads:

> Will calculate the square root of a given number.
>
> The given number must be greater than or equal to 0.
>
> Answers will be rounded to the fourth decimal place.

This extra clause describing the output of your work is analogous to a **post-condition**. These conditions ensure that both you and any client (or in this scenario your computing boss) fully agree on what the **expected output** looks like.

What is a Post-Condition?

Like a **pre-condition**, a **post-condition** has two key elements:

1. A statement or expression informing the client on the **range of output** to expect from your code.
2. Code that produces the correct output.

Sometimes the behavior of our programs will change under different states. Think about a `indexOf` method that searches for a value in an `ArrayList` and returns the index. If the `ArrayList` contains that value then we can return the index, but if it doesn't contain that value then we might return a

default value like -1. Therefore we would have two post-conditions: one if the value is contained, and another for if the value is not contained. Thinking about the output of our programs in terms of post-conditions allows us as programmers to more easily **identify edge cases** and ensure our program in **robust** and **comprehensive**.

✏ Relevant Example

Multiple Outputs and Post-Condition Comments

Let's say we're making a method that searches for the number 9 in a given `int` array and returns the index. Lets assume our **pre-conditions** are the same as before such that the reference can't be null and the size has to be greater than 0.

Before adding post-conditions, our code looks like this:

```
// Behavior: Searches given array for the first index that equals 9 and returns it
// Parameter nums: int array that will be searched
// Pre-Condition 1: nums is not null
// Pre-Condition 2: size of nums is > 0
// Returns: first index that equals 9
public static int findNine(int[] nums) {
    // check for invalid input
    if (nums == null || nums.length <= 0) {
        return -1;
    }
    // valid input
    for (int i = 0; i < nums.length; i++) {
        if (nums[i] == 9) {
            return i;
        }
    }
    return -2;
}
```

Notice here how our code and comments are completely accurate but the client has to read the code to understand what the output might be if they don't meet the pre-conditions or there isn't a 9. We can use **post-conditions** to reduce this uncertainty around the varying input.

There are three cases we should consider:

1. The pre-conditions aren't met.
2. There is a 9.
3. There is not a 9.

Since we have **three possible outputs** we should have **three post-conditions** too! Based on our current code it looks like if the pre-conditions aren't met then we return -1. If the pre-conditions are met and there is a 9 then we return its index, but if there isn't a 9 then we return -2. Knowing all this, let's translate this into a **post-condition comment!**

```java
// Parameter nums: int array that will be searched
// Pre-Conditions:
//    nums is not null
//    size of nums is > 0
// Post-Conditions:
//    If the pre-conditions are not met: -1
//    If nine is found: the first index containing 9
//    If nine is not found: -2
public static int findNine(int[] nums) {
    // check for invalid input
    if (nums == null || nums.length <= 0) {
        return -1;
    }
    // valid input
    for (int i = 0; i < nums.length; i++) {
        if (nums[i] == 9) {
            return i;
        }
    }
    return -2;
}
```

Writing our comments with both **pre-conditions** and **post-conditions** ensures that the client fully knows what to expect without having to understand our potentially complicated code. **It also makes your job as a programmer easier!** This may seem counterintuitive at first but thinking about what to expect for different inputs is a key part of writing a **comprehensive** and **accurate** program.

□ Main Points

- A **post-condition** is a "clause" or statement in code comments we create that specifies the expected range of outputs or conditions after we run our method.
- **Post-conditions** ensure that clients know what to expect from our code and know of possible ending scenarios of our code, so we can avoid ambiguity and confusion.
- We should include multiple **post-conditions** in our code comments based on all sorts of conditions and scenarios (e.g. if a pre-condition is not met, if specific values are used, etc).
- **Post-conditions** can help programmers find edge cases and handle situations differently based on different input, allowing for our programs to be more clear.

Using Exceptions [Background Reading]

🤔 Motivation

Pre/post-conditions enable us to precisely describe and verify both the beginning and end states of our programs. This gives the client extra clarity and helps us as programmers build reliable and comprehensive programs. However, sometimes there are inputs that are so dangerous that the only output that makes sense is stopping the entire program and informing the client of their mistake.

🤔 What are Exceptions?

You've probably seen `Exception`s while debugging and maybe even thrown a few yourself. `Exception`s are most commonly used to alert a client that something has gone wrong. We use different types of `Exception`s to denote different types of problems. Unless a client catches an `Exception` (not relevant to this class), throwing an `Exception` will stop their program. This is incredibly useful for preventing potentially dangerous code from executing or wasting time on calculations that wouldn't be accurate with the given input.

✏ Using Exceptions

Important CSE 123 Exceptions

Here are some key examples of `Exception`s you'll probably encounter and throw yourself in CSE 123.

- `NullPointerException`: Trying to access a null object or value.
- `IllegalArgumentException`: A parameter doesn't meet the pre-conditions.
- `ArrayIndexOutOfBoundsException`: Trying to access an index greater than or equal to the length of the array, or less than 0.
- `ClassCastException`: Casting an object to another class but the original object is not an instance of that the class. This could occur when attempting to cast a `Scanner` to something that is definitely not a `Scanner` like a `HashMap`.

Invalid Pre-Conditions

Let's go back to our method `findNine()` from the previous slide. Currently the code produces three different types of output and looks like this:

```
// Parameter nums: int array that will be searched
// Pre-Conditions:
```

```
//    nums is not null
//    size of nums is > 0
// Post-Conditions:
//    If the pre-conditions are not met: -1
//    If nine is found: the first index containing 9
//    If nine is not found: -2
public static int findNine(int[] nums) {
    // check for invalid input
    if (nums == null || nums.length <= 0) {
        return -1;
    }
    // valid input
    for (int i = 0; i < nums.length; i++) {
        if (nums[i] == 9) {
            return i;
        }
    }
    return -2;
}
```

A common practice to help clients understand the **pre-conditions** for a method and quickly identify bugs in their code is to throw `Exception`s when the **pre-conditions** are not met. In this case we have two pre-conditions which both involve our parameter `nums`. As such, it would make sense to throw an `IllegalArgumentException` if either of these **pre-conditions** are not met.

Doing so, we would also need to adjust our **post-condition** comment to specify which `Exception` we throw and when. These changes could look like this:

```
// Parameter nums: int array that will be searched
// Pre-Conditions:
//    nums is not null
//    size of nums is > 0
// Post-Conditions:
//    If the pre-conditions are not met: throws IllegalArgumentException
//    If nine is found: returns the first index containing 9
//    If nine is not found: returns -2
public static int findNine(int[] nums) {
    // check for invalid input
    if (nums == null || nums.length <= 0) {
        throw new IllegalArgumentException();
    }
    // valid input
    for (int i = 0; i < nums.length; i++) {
        if (nums[i] == 9) {
            return i;
        }
    }
    return -2;
}
```

The exact styling of the comment is left up to the programmer to best determine but it's highly recommended to be consistent throughout your program.

Main Points

- While **pre-conditions** and **post-conditions** can help us make reliable programs, some input that users supply us is so dangerous that it's better to stop the program all together and tell the client their mistake. In this case, we would rather throw **exceptions**.
- Some important **exceptions** we will see and use throughout CSE 123 include `NullPointerException`, `IllegalArgumentException`, `ArrayIndexOutOfBoundsException`, and `ClassCastException`.
- To help clients understand **pre-conditions** of a method, we can throw an **exception** when a **pre-condition** is not met.
- When we throw an **exception** if a **pre-condition** is not met, we want to fix our **post-condition** comment so that is specifies what **exception** we will throw and when we will throw it.

Implementing Exceptions [Programming Question]

Implement the method in the `ArrayIntList` class called `set`. This method is designed to set the value of an index to a specific value. The client is responsible for passing in parameters for both the index and value.

Your task is to implement the `set` method so it's behavior is in line with the pre/post-conditions described below.

The pre-conditions for `set` are:

1. The given index must be greater than or equal to 0.
2. The given index must be less than the size of the `List`.

The post-conditions for `set` are:

1. If the given index is less than 0: throws an `IllegalArgumentException`.
2. If the given index is greater than or equal to size: throws an `IndexOutOfBoundsException`.
3. Updates the existing value at the given index to the given value.

An implementation with only working code will pass the tests but it's highly encouraged that you also practice writing comments describing the pre/post-conditions.

The entire `ArrayIntList` class is given for reference, but you only need to implement the `set` method at the top of the class.