Cpt S 321 – Exam 1
Spring 2021
25 points total
Total pages: 5

Name (First Last):	Ryan Rochleau
WSU ID:	11577056

Read the instructions carefully:

- This is an individual exam: you are **not allowed to communicate** with anyone regarding the exam questions.

- If something is unclear, ask for clarifications via the BB Discussion. If it is still unclear after my answer, then write down any assumptions that you are making.

- There are 3 questions. (The last question accounts for 19 points)

- What to submit:

- Commit a .PDF version of this document with your answers and your code in your repository for the in-class exercises in a branch "MidTerm1". Tag what you want to be graded with "MT1_DONE" tag.

1. (2 points) For each of the programs below, explain whether or not it will compile, if it will

crash, and justify your answer.

Programs	Explanation (compile/crash/result)

void F1(int b)
{
long a = b;
}
	Does it compile? (Y/N): Yes
	Does it crash? (Y/N/NA): No
	Justification: This is implicit conversion. Longs hold 64 bits worth of data while ints hold 32. So there is enough space for long a to hold all the data int b could possibly ever have.

void F2(int b)
{
short a = b;
}
	Does it compile? (Y/N): No
	Does it crash? (Y/N/NA): N/A
	Justification: Shorts are 16 bit numbers while ints are 32. There is no implicit conversion allowed for integers to shorts becuase shorts couldn't hold the value of an integer the majority of the time.

2. (4 points) Consider the following definitions for classes Animal, Dog, and Cat. For each of the programs below, explain whether or not it will compile, if it will crash, and justify your answer. **Justification is more important than the rest.**

```
public class Animal
{
   public String Eat()
   {
      return "Yummy!";
   }
}
```

```
public class Dog : Animal
{
   public String Eat()
   {
      return "Yummy woof!";
   }

   public String Bark()
   {
      return "Woof!!";
   }
}
```

```
public class Cat : Animal
{
   public String Purr()
   {
      return "Purr!";
   }
}
```

Programs	Explanation (compile/crash/result)

void F1()
{
Dog aDog = new Dog();
Animal anotherAnimal = aDog;
Console.WriteLine(anotherAnimal.Eat());
}
	Does it compile? (Y/N): Yes
	Does it crash? (Y/N/NA): No
	Result: Prints "Yummy!"
	Justification: Base classes don't inherit properties or methods from derived classes. Since anotherAnimal is an Animal which is the base class, we end up with the base classes string for Eat()

`void F2()`
`{`
`  Animal anAnimal = new Animal();`
`  Cat aCat = anAnimal;`
`}`
	Does it compile? (Y/N): No
	Does it crash? (Y/N/NA): N/A
	Justification: A cat is an animal, but an animal is not a cat. We are trying to downcast here which isn't a good idea. Since an animal is not a cat, trying to set aCat to something that isn't a cat will not work.

`void F3()`
`{`
`  Cat aCat = new Cat();`
`  Animal anotherAnimal = aCat;`
`  Console.WriteLine(anotherAnimal.Purr());`
`}`
	Does it compile? (Y/N): No
	Does it crash? (Y/N/NA): N/A
	Result: Nothing
	Justification: Once again, base classes don't inheritit properties or methods from derived classes. Since anotherAnimal is the base class Animal, it won't inherit the method Purr() from Cat.

`void F4()`
`{`
`  Animal anAnimal = new Animal();`
`  Dog aDog = (Dog)anAnimal;`
`  Console.WriteLine(aDog.Bark());`
`}`
	Does it compile? (Y/N): Yes
	Does it crash? (Y/N/NA): Yes
	Result: Crashes on the line Dog aDog = (Dog) anAnimal;
	Justification: We are downcasting here line in problem F2(). We are allowed to compile because we are casting anAnimal to the correct type Dog. However, even though we are type casting, anAnimal is not a Dog so ultimately, we crash becuase there's no way to convert an Animal to a Dog becuase an Animal is not a Dog.

3. (19 points) **Please read the entire question carefully before you start working.**

You are contacted by a local store to build a desktop application in C# that will allow employees to look up the availability of a certain products and to restock. All products have a unique ID and a description. Products can be physical (ex. pen) or electronic (ex. e-book). For physical products, the store will have 0 or more items of a certain product available at a specific moment. Electronic products are unlimited.

The set of features that the software must support are the following:

- **Search**: An employee can search for a product in the store database. Employees will enter a sequence of characters (which can enter either be a (partial) code and/or keywords) and the search should be performed on all possible fields (i.e., the employee should not be asked whether she/he wants to search by code or by keywords, this must be handled implicitly). If the sequence of characters contains one or more spaces, it would need to be split into tokens and the employee should

be asked whether this is an AND search (all tokens must be present in each product) or and OR search (at least one of the tokens must be present in each product). The result will be the list of products that match the search. For each product that matches the search all available information about the product and the current items associated with that product at the moment in the store must be displayed. If the user enters an empty sequence of characters, i.e., she/he simply hits enter or space(s), then the result should be all products in the store.

- **Save search**: If an employee selects this functionality, the last search that was performed will be saved to a file in a subdirectory of the project called "searches". The file name should indicate the date and time that the search was performed in the following format: <yyyy>-<mm>-<dd>-<hh>h<mm>m<ss>s.txt. For example, if the search was performed at 8:34:30pm on February 4, 2021 the filename would be "2021-02-04-20h34m30s.txt"
  The first line of the file should contain the sequences of characters that the employee used for the search and whether it was an AND or OR search if applicable. The rest of the file, i.e., starting from line 2, must contain the result of the search as it was shown to the employee.

- **Restock:** If an employee selects this functionality, all physical products with less than N items will be restocked, where N is provided by the employee. First, the list of products that match the search will be shown, i.e., all products where the number of items is less than N. Then, the employee will be asked whether she/he would like to restock for all of them: If the response is positive, then for each product that matches the search all available information about the product and the number of current items associated with that product at the moment in the store must be displayed and the employee would be asked how many additional items need to be purchased. After that, a confirmation must be shown that the purchase is successful, and the updated product information must be shown. If the response is negative, the user should be given the option to change N or to return to the main menu.

Using TDD, implement a prototype that would fit the needs of the client as described above. Note: You do not have to worry about a user interface for now. You can build a prototype in a console application (Choose "Console App (.NET Core)" or "Console App (.NET Core)" when you are creating the project).

Grading schema and point breakdown (19 points total):

- 9 points: Fulfill all the requirements above with no inaccuracies in the output and no crashes.
- 2 points: For a "healthy" version control history, i.e., 1) the prototype should be built iteratively, 2) every commit should be a cohesive functionality, 3) the commit message should concisely describe what is being committed, 4) you should follow TDD – i.e., write and commit tests first and then implement and commit the functionality.
- 2 points: Code is clean, efficient and well organized.
- 2 points: Quality of identifiers.
- 2 points: Existence and quality of comments.
- 2 points: Existence and quality of test cases. Normal cases and edge cases are both important to test.

General Homework Requirements	
Quality of Version Control	• Should be built iteratively (i.e., one feature at a time, not in one huge commit). • Each commit should have cohesive functionality. • Commit messages should concisely describe what is being committed. • TDD should be used (i.e, write and commit tests first and then implement and commit functionality). • Use of a .gitignore. • Commenting is done alongside with the code (i.e, there is commenting added in each commit, not done all at once at the end).
Quality of Code	• Each file should only contain one public class. • Correct use of access modifiers. • Classes are cohesive. • Namespaces make sense. • Code is easy to follow. • StyleCop is installed and configured correctly for all projects in the solution and all warnings are resolved. If any warnings are suppressed, a good reason must be provided. • Use of appropriate design patterns and software principles seen in class.

Quality of Identifiers	• No underscores in names of classes, attributes, and properties. • No numbers in names of classes or tests. • Identifiers should be descriptive. • Project names should make sense. • Class names and method names use PascalCasing. • Method arguments and local variables use camelCasing. • No Linguistic Antipatterns or Lexicon Bad Smells.
Existence and Quality of Comments	• Every method, attribute, type, and test case has a comment block with a minimum of <summary>, <returns>, <param>, and <exception> filled in as applicable. • All comment blocks use the format that is generated when typing "///" on the line above each entity. • There is useful inline commenting in addition to comment blocks that explains how the algorithm is implemented.
Existence and Quality of Tests	• Normal, boundary, and overflow/error cases should be tested for each feature. • Test cases should be modularized (i.e, you should have a separate test case for each feature or scenario that you test - do not combine them into one large test case).