CPSC 5011 Object-Oriented Concepts
P1 exercises your understanding of class design principles

For an acceptable P1 submission:

1. use **C#** and Visual Studio
2. upload all files individually (NOT a project, NOT a zip file) to Canvas
3. use Programming by Contract to specify contractual design, placing
    a. class and interface invariants at the top of *jumpPrime.cs* file
    b. implementation invariant at the end of *jumpPrime.cs* file
    c. pre & postconditions (if needed) before each method header
4. write readable code – see codingStd.pdf on Canvas
    a. use functional decomposition => no monolithic drivers
    b. do not hard code: replace arbitrary literals (such as "42") with constant values
5. employ the OOP tenets of abstraction, encapsulation and information hiding
    a. do not use a random number generator inside the class
6. document your driver:
    a. ProgrammingByContract **NOT** used for **drivers**
    b. DO NOT assume that the reader has access to this assignment specification
    c. *provide an overview of your program*
    d. *explicitly state <u>ALL assumptions</u>*
7. Neither the class nor the driver should request user input

<u>TWO perspectives demonstrated in this assignment:</u>
-- the class designer (designs and implements the *jumpPrime* class)
-- the client (the software that uses *jumpPrime* objects; simulated by the driver *P1.cs*)

Part I: Class Design (jumpPrime.cs)

Each *jumpPrime* object encapsulates a positive integer, which must be at least 4-digits long, and yields the prime numbers closest to that number. For example, an active *jumpPrime* `myObj` that encapsulates 2488:

```
myObj.up();          // returns prime number 2503
myObj.down();        // returns prime number 2477
```

Additionally, the encapsulated number will 'jump' after a limited number of queries, a number determined by the distance between the two closest prime numbers. Hence, the active *jumpPrime* myObj that encapsulates 2488 will 'jump' to a value > 2503 or to a value < 2477 after 26 queries. Every *jumpPrime* object is initially active but transitions to inactive once the number of jumps exceeds a bound. The client may **reset** as well as **revive** a *jumpPrime* object. An attempt to revive an active *jumpPrime* object causes that object to be permanently deactivated.

Many details are missing. You MUST make and DOCUMENT your design decisions!!

This assignment is an abstract realization of a data sink (store) that yields specific information upon query but can age and become invalid. With the interface described above, your design should encapsulate and control state as well as the release of information.

Part II: Driver (P1.cs) -- External Perspective of Client – tests your class design

Design a **FUNCTIONALLY DECOMPOSED** driver to demonstrate the program requirements.
Use many distinct objects, initialized appropriately, i.e. random distribution of *jumpPrimes*, etc.
Adequate testing requires varied (random) input sufficient enough to yield objects in different states and the seamless alteration of the state of some objects.
Make your output readable but not exceedingly lengthy.

Sample Grading: Class Design (75 points)

Form (25 points)

Contractual Design	10 points
Interface and Implementation invariants	
Pre & Post conditions	
Proper Accessibility (public, private)	5 points
Appropriate functionality present in interface	5 points
Readable code (no hardcoding)	5 points

Functionality (50 points)

Data Type implemented as required	15 points
clear, effective interface	
minimal getSize() or the like	
no dependencies on implementation exposed, etc.	
Data members defined & manipulated as required	10 points
Effective representation of state	
Constructor(s)	10 points
put object in proper initial state	
State transitions correct	10 points

Sample Grading: Driver (25 points)

Form (10 points)

Program overview	5 points
PROGRAMMER name, date, revision history, platform, etc.	
Description of process(es) performed by program	
Explanation of user interface (input, meaning of output)	
Comments on use and validity (error processing)	
Statement of assumptions	
Maintainable code (no hardcoding)	5 points

Functionality (15 points): Correlate to Documented Design Decisions

Requirements fulfilled – type tested appropriately	10 points
No unnecessary dependencies	
User Interface (clear, correct & effective)	5 points
Output (readable & consistent with design)	
Random number generator used appropriately	