CSCI 200: Foundational Programming Concepts & Design

Final Exam Review

1. Struct

Create a **`struct`** called **`Point`** that has two data members that are **`double`**s called **`x`** and **`y`**.

Additionally, write a function called **`distance`** that accepts two **`Point`** variables as input and returns a **`double`**. The function needs to return the Euclidean distance between the two points.

Write a full program that will read in four floating point values from the user separated by spaces. The first pair of values will correspond to the **`x`** and **`y`** values for **`Point a`**. The second pair of values will correspond to the **`x`** and **`y`** values for **`Point b`**.

Finally, print the distance between **`a`** and **`b`** with four decimal places.

Include all necessary headers.

Note: While it is possible to solve this problem without writing a **`struct`** or a function, on the exam to receive full credit you must write a **`struct`** and function.

1. Struct

```
#include <cmath>
#include <iomanip>
#include <iostream>
using namespace std;

struct Point {
  double x, y;
};

double distance( const Point& LEFT, const Point& RIGHT ) {
  return sqrt( pow(LEFT.x - RIGHT.x, 2.0) + pow(LEFT.y - RIGHT.y, 2.0) );
}

int main() {
  Point a, b;
  cin >> a.x >> a.y >> b.x >> b.y;
  cout << fixed << setprecision(4) << distance(a, b) << endl;
  return 0;
}
```

2. Lists

- What are the pros/cons of an Array?
- What are the pros/cons of a LinkedList?
- What is the same about an Array & a LinkedList?
- What is different about an Array & a LinkedList?
- When should one be used over the other?
- *Consider memory, run time complexity, and other considerations*

2. Lists

	Array	LinkedList
Pro	Constant element access Less bookkeeping	Constant add or remove to the front or back Can potentially store more elements
Con	Linear add/remove One big contiguous block of memory	Linear element access More bookkeeping to alter list Uses total more memory
Same	Store an ordered list of values that can change size	
Different	One big contiguous block of memory	Many small fragmented blocks of memory
Use cases	Small(er) amount of data Size does not change frequently Many random accesses	Large(r) amount of data Size continually changing Many sequential accesses

3. Sorting

- Given a list (Array or LinkedList), implement one of Selection/Insertion/Bubble Sort to sort the list in ascending order.

3. Sorting

- Selection

```
IList<T>* pList = new ???;
// populate list
for(int i = 0; i < pList->size(); i++) {
  int minPos = i;
  for(int j = i+1; j < pList->size(); j++) {
    if(pList->get(j) < pList->get(minPos)) {
      minPos = j;
    }
  }
  T temp = pList->get(minPos);
  pList->set(minPos, pList->get(i));
  pList->set(i, temp);
}
```

3. Sorting

- Insertion

```
IList<T>* pList = new ???;
// populate list
for(int i = 1; i < pList->size(); i++) {
  int temp = pList->get(i);
  int j = i-1
  while(j >= 0 && pList->get(j) > temp) {
    pList->set(j+1, pList->get(j));
    j--;
  }
  pList->set(j+1, temp);
}
```

3. Sorting

- Bubble

```
IList<T>* pList = new ???;
// populate list
for(int i = 0; i < pList->size(); i++) {
  int numSwaps = 0;
  for(int j = 0; j < pList->size() - i - 1; j++) {
    if(pList->get(j) > pList->get(j + 1)) {
      T temp = pList->get(j + 1);
      pList->set(j + 1, pList->get(j));
      pList->set(j, temp);
      numSwaps++;
    }
  }
  if(numSwaps == 0) {
    break;
  }
}
```

4. Exceptions

- What is the structure sequence necessary when implementing exception handling?
  - What is the role of each component?

- Why should we perform exception handling in our code?

4. Exceptions

- try – throw - catch
  - What is the role of each component?

- Why should we perform exception handling in our code?

4. Exceptions

- try – throw - catch
  - try: test dangerous code
  - throw: raise an exception up to the calling code block
  - catch: handle exception

- Why should we perform exception handling in our code?

4. Exceptions

- try – throw - catch
  - try: test dangerous code
  - throw: raise an exception up to the calling code block
  - catch: handle exception

- Prevent run time errors (logic or run time)
  - Cleanly handle errors that occur at runtime
  - Provide better user experience

5. Fill in the missing pieces of code

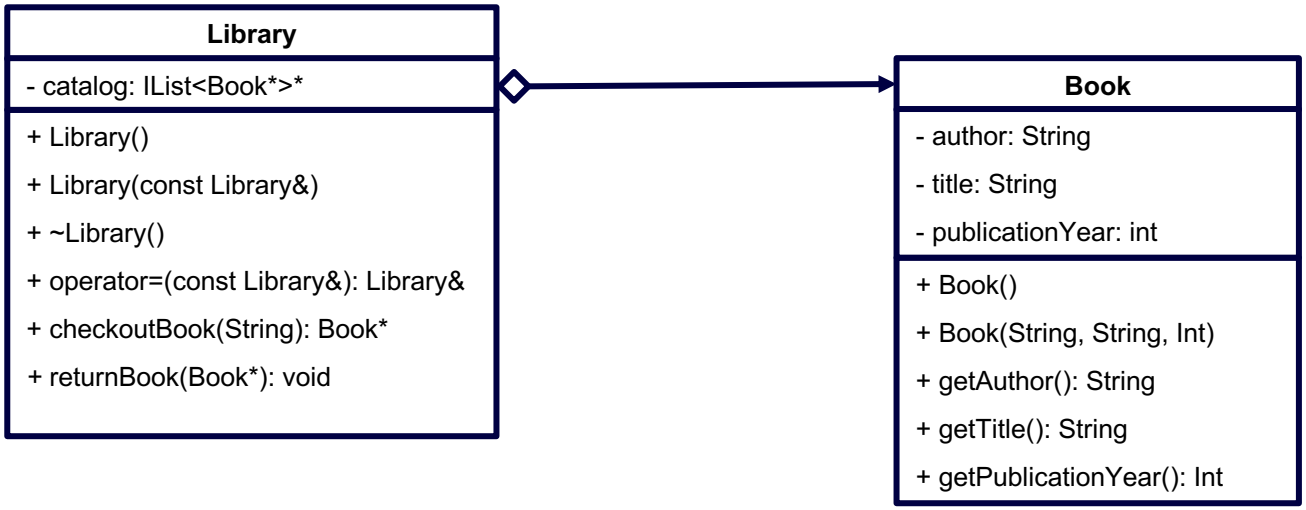
```
// TODO 2: complete the following function named populate_array
/**
 * @brief populates an array by setting each element equal to its index i % MOD_VALUE
 * @param arr integer array
 * @param SIZE size of array
 * @param MOD_VALUE modulus value to apply to each element
 * @return int sum of all element values
 */

// TODO 1: create a static integer array with 100 elements

// TODO 3: call populate_array with a modulus value of 21 and store result

// TODO 4: print result of TODO 3
```

5. Fill in the missing pieces of code

```
// TODO 2: complete the following function named populate_array
/**
 * @brief populates an array by setting each element equal to its index i % MOD_VALUE
 * @param arr integer array
 * @param SIZE size of array
 * @param MOD_VALUE modulus value to apply to each element
 * @return int sum of all element values
 */
int populate_array(int arr[], const int SIZE, const int MOD_VALUE) {
  int sum = 0;
  for(int i = 0; i < SIZE; i++) {
    arr[i] = i % MOD_VALUE;
    sum += arr[i];
  }
  return sum;
}

// TODO 1: create a static integer array with 100 elements
const int SIZE = 100;
int array[SIZE] = {0};

// TODO 3: call populate_array with a modulus value of 21 and store result
const int MOD_VALUE = 21;
int result = populate_array(array, SIZE, MOD_VALUE);

// TODO 4: print result of TODO 3
cout << result;
```

6. From UML to Class

- Write the C++ Classes corresponding to the following UML diagrams.



6. From UML to Class

```
class Book {
public:
  Book();
  Book(const string,
       const string,
       const int);
  string getAuthor() const;
  string getTitle() const;
  int getPublicationYear() const;
private:
  string _author;
  string _title;
  int _publicationYear;
};
```

```
class Library {
public:
  Library();
  Library(const Library&);
  ~Library();
  Library& operator=(const Library&);
  Book* checkoutBook(const string);
  void returnBook(Book*);
private:
  IList<Book*> *_pCatalog;
};
```

7. Create the Book

- Create the following Book functions:
  - Default constructor → Creates book with following members:
    - The C++ Programming Language
    - Bjarne Stroustrup
    - 1986
  - Parameterized constructor → sets each data member
  - Getters → returns each data member

7. Create the Book

```
Book::Book() {
  _author = "Bjarne Stroustrup";
  _title = "The C++ Programming Language";
  _publicationYear = 1986;
}

Book::Book(const string AUTHOR,
           const string TITLE,
           const int PUBLICATION_YEAR) {
  _author = AUTHOR;
  _title = TITLE;
  _publicationYear = PUBLICATION_YEAR;
}

string Book::getAuthor() const { return _author; }

string Book::getTitle() const { return _title; }

int Book::getPublicationYear() const { return _publicationYear; }
```

8. Create the Library

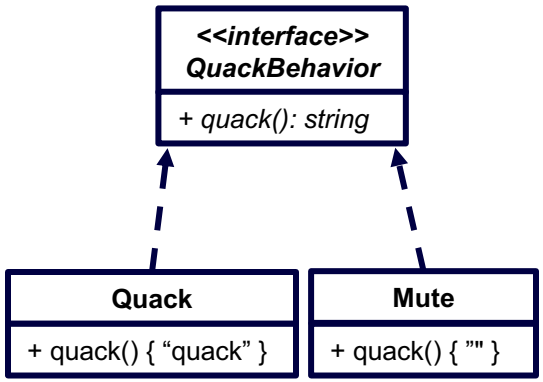
- Create the following Library functions:
  - Default constructor → Initializes an empty list
  - Copy constructor → Performs a Deep Copy Making New Books
  - checkoutBook → If book with title is in list, removes from list and returns. If title is not in list, returns null pointer
  - returnBook → puts book in list

8. Create the Library

```
Library::Library() {
  _pCatalog = new LinkedList<Book*>();
}

Library::Library(const Library& OTHER) {
  _pCatalog = new LinkedList<Book*>();
  for(int i = 0; i < OTHER._pCatalog->size(); i++) {
    Book* pBook = OTHER._pCatalog->get(i);
    _pCatalog->insert( i, new Book(*pBook) );
  }
}

Book* Library::checkoutBook(const string TITLE) {
  for(int i = 0; i < _pCatalog->size(); i++) {
    if(_pCatalog->at(i)->getTitle() == TITLE) {
      Book* pBook = _pCatalog->at(i);
      _pCatalog->remove(i);
      return pBook;
    }
  }
  return nullptr;
}

void Library::returnBook(Book* pBook) {
  _pCatalog->insert( 0, pBook );
}
```

9. From UML to Class II

- Write the C++ Classes corresponding to the following UML diagrams.



9. From UML to Class II

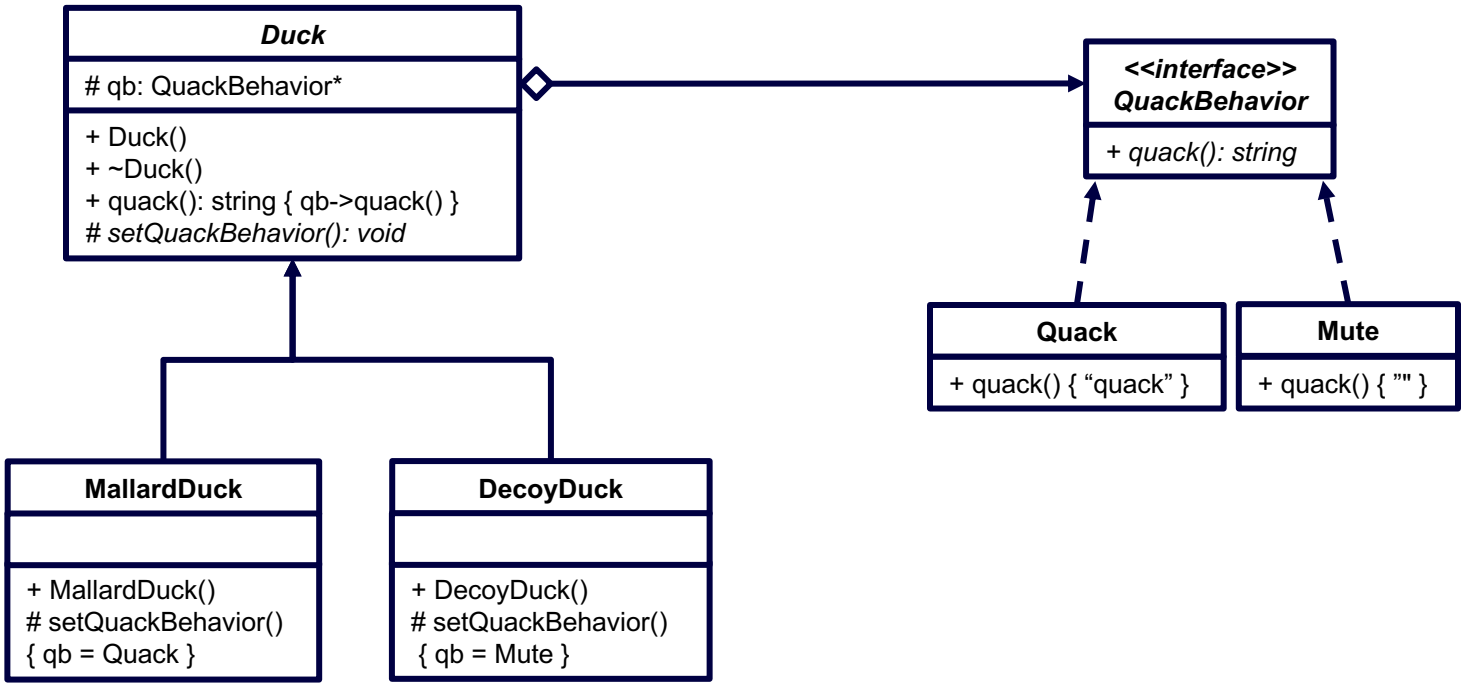
```
class IQuackBehavior {
public:
  virtual ~IQuackBehavior() {}
  virtual string quack() const = 0;
};

class Quack final : public IQuackBehavior {
public:
  string quack() const override { return "quack"; }
};

class Mute final : public IQuackBehavior {
public:
  string quack() const override { return ""; }
};
```

9. From UML to Class II

```
class IQuackBehavior {
public:
  virtual ~IQuackBehavior() = default;
  virtual string quack() const = 0;
};

class Quack final : public IQuackBehavior {
public:
  string quack() const override { return "quack"; }
};

class Mute final : public IQuackBehavior {
public:
  string quack() const override { return ""; }
};
```

10. From UML to Class III

- Write the C++ Classes corresponding to the following Duck UML diagrams. Have the Duck constructor call the setQuackBehavior method.



10. From UML to Class III

```
class Duck {
public:
  Duck() { mpQB = nullptr; }
  virtual ~Duck() { delete mpQB; }
  string quack() const { return mpQB->quack(); }
protected:
  virtual void mSetQuackBehavior() = 0;
  IQuackBehavior *mpQB;
};

class MallardDuck final : public Duck {
public:
  MallardDuck() { mSetQuackBehavior(); }
protected:
  void mSetQuackBehavior() override { mpQB = new Quack(); }
};

class DecoyDuck final : public Duck {
public:
  DecoyDuck() { mSetQuackBehavior(); }
protected:
  void mSetQuackBehavior() override { mpQB = new Mute(); }
};
```

Aside: mSetQuackBehavior()

- Can't call abstract function in abstract constructor
  - Constructor called before concrete implementation exists
  - Every concrete constructor must call method...or...

```
class Duck {
public:
  Duck() { mSetQuackBehavior(); }
  virtual ~Duck() { delete mpQB; }
  string quack() const { return mpQB->quack(); }
protected:
  virtual void mSetQuackBehavior() = 0;
  IQuackBehavior *mpQB;
};

class MallardDuck final : public Duck {
public:
  MallardDuck() { /* calls Duck() before MallardDuck() */ }
protected:
  void mSetQuackBehavior() override { mpQB = new Quack(); }
};
```

Aside: Proper Way To Make Ducks

```
class Duck {
public:
  Duck() { mpQB = nullptr; }
  virtual ~Duck() { delete mpQB; }
  string quack() const { return mpQB->quack(); }
  template<typename> friend class DuckFactory;
protected:
  virtual void mSetQuackBehavior() = 0;
  IQuackBehavior *mpQB;
};

class MallardDuck final : public Duck {
public:
  template<typename> friend class DuckFactory;
protected:
  void mSetQuackBehavior() override { mpQB = new Quack(); }
private:
  MallardDuck() {}
};

template<typename DuckT>
class DuckFactory {
  static_assert(std::is_base_of<Duck, DuckT>::value, "DuckT must derive from Duck");
public:
  static Duck* createDuck() {
    Duck *pDuck = new DuckT();
    pDuck->mSetQuackBehavior();
    return pDuck;
  }
};

Duck *pMyDuck = DuckFactory<MallardDuck>::createDuck();
```

11. The Duck Pond

- Create a list of Ducks with 10 ducks. The odd numbered ducks are decoys, the even numbered ducks are mallards.
- Then iterate through the list and make each quack.
- Cleanup all memory.

- Use runtime polymorphism!

11. The Duck Pond

```
IList<Duck*>* pDuckpond = new LinkedList<Duck*>();

for(int i = 0; i < 10; i++) {
  if(i % 2) {
    pDuckpond->insert( i, new DecoyDuck() );
  } else {
    pDuckpond->insert( i, new MallardDuck() );
  }
}

for(int i = 0; i < pDuckpond->size(); i++) {
  cout << "Duck #" << i << " says " << pDuckpond->get(i)->quack() << endl;
}

for(int i = 0; i < pDuckpond->size(); i++) {
  delete pDuckpond->get(i);
}

delete pDuckpond;
```

Aside: Using the Factory

```
IList<Duck*>* pDuckpond = new LinkedList<Duck*>();

for(int i = 0; i < 10; i++) {
  if(i % 2) {
    pDuckpond->insert( i, DuckFactory<DecoyDuck>::createDuck() );
  } else {
    pDuckpond->insert( i, DuckFactory<MallardDuck>::createDuck() );
  }
}

for(int i = 0; i < pDuckpond->size(); i++) {
  cout << "Duck #" << i << " says " << pDuckpond->get(i)->quack() << endl;
}

for(int i = 0; i < pDuckpond->size(); i++) {
  delete pDuckpond->get(i);
}

delete pDuckpond;
```

12. 2D Arrays

- Create a static 2D Character Array with 5 rows and 6 columns. Initialize each element to a space.

12. 2D Arrays

```
const int NUM_ROWS = 5;

const int NUM_COLS = 6;

char pWordle[NUM_ROWS][NUM_COLS];

for(int i = 0; i < NUM_ROWS; i++) {
  for(int j = 0; j < NUM_COLS; j++) {
    pWordle[i][j] = ' ';
  }
}
```

13. 2D Arrays II

- Prompt the user to enter the number of rows *r* and columns *c*.
- Create a dynamic 2D Array of Books (see #6) called bookCase that has *r* shelves and *c* books per shelf.
- Make every book B. Stroustrup's book.

13. 2D Arrays II

```
int numShelves, booksPerShelf;
cin >> numShelves >> booksPerShelf;
Book** pBookCase = new Book*[numShelves];
for(int shelf = 0; shelf < numShelves; shelf++) {
  pBookCase[shelf] = new Book[booksPerShelf];
}
```

14. 2D Arrays III

- Prompt the user to enter the number of shelves *s* and books *b*.
- Create a dynamic 2D Array of Books (see #6) called bookCase that has *s* shelves and *b* books per shelf.
- When creating each book, prompt the user for the book information.
- Then sort the books on each shelf alphabetically.

14. 2D Arrays III

```
int numShelves, booksPerShelf;
cin >> numShelves >> booksPerShelf;
Book** pBookCase = new Book*[numShelves];
string title, author;
cin.ignore();
for(int shelf = 0; shelf < numShelves; shelf++) {
  pBookCase[shelf] = new Book[booksPerShelf];
  for(int book = 0; book < booksPerShelf; book++) {
    getline(cin, title);  pBookCase[shelf][book].setTitle(title);
    getline(cin, author); pBookCase[shelf][book].setAuthor(author);
  }
  /* perform sort
   * if( pBookCase[shelf][i].title > pBookCase[shelf][j].title ) swap
   */
}
```

15. 2D Arrays IV

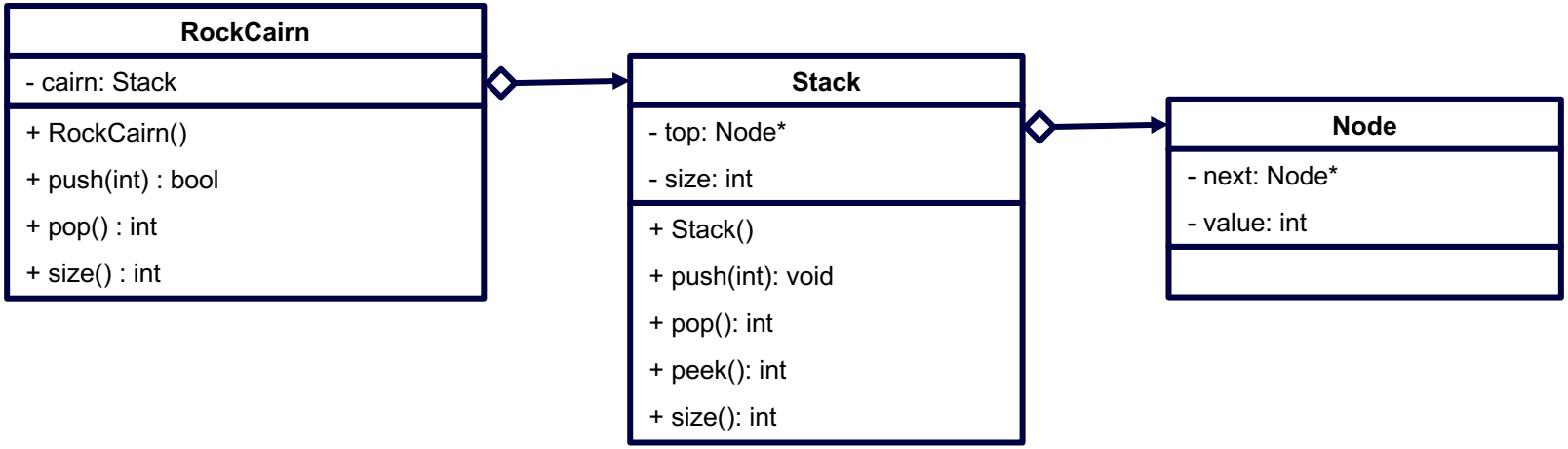
- After the books are sorted, ask the user for a book name.
  - If it exists, tell the user which shelf the book is on and what position on the shelf it is in.
  - Otherwise, tell the user the book cannot be found.
- What is a good search strategy?
- What is the run time to search the bookcase?
  - Express it in terms of *s* and *b*

15. 2D Arrays IV

```
int numShelves, booksPerShelf;
cin >> numShelves >> booksPerShelf;
Book** pBookCase = new Book*[numShelves];
// insert & sort each shelf
string targetTitle;
getline(cin, targetTitle);
int targetShelf = -1, targetBook = -1;
for(int shelf = 0; shelf < numShelves; shelf++) {
  // binary search amongst pBookCase[shelf] using bookPos
  // set targetShelf = shelf and targetBook = bookPos
  // if targetShelf != -1, break
  // full runtime is O(s log b)
}
if(targetShelf != -1) {
  cout << "Book is on shelf #" << targetShelf << " and is book #" << targetBook << endl;
} else {
  cout << "Book cannot be found" << endl;
}
```

16. Stack Class

- Create a class called RockCairn that contains a stack of integers. RockCairn::push() will only push to its stack if the value to be pushed is smaller than the current top value of the stack. It returns true if the value was pushed, false if not. You can assume Stack & Node are implemented.



16. Stack Class

```
class RockCairn {
public:
  RockCairn() { }
  bool push(const int VALUE) {
    if(VALUE < _stack.peek()) {
      _stack.push(VALUE);
      return true;
    }
    return false;
  }
  int pop() {
    if(_stack.size() > 0) return _stack.pop();
    else throw out_of_range("stack is empty");
  }
  int size() const { return _stack.size(); }
private:
  Stack _stack;
};
```

17. SOLID

- What are the SOLID Principles?
- What is a scenario that each would be applied to?

17. SOLID

- Single Responsibility – a Book stores the information for a Book. A Printer prints the book, the book doesn't print itself.
- Open/Closed – A stack would contain a LinkedList, not be a LinkedList with hidden functionality
- Liskov Substitution – a square is a quadrilateral, but not a rectangle
- Interface Separation – a microphone can record audio, a speaker can play audio
- Dependency Inversion – the pizza shop sells pizzas, not pepperoni pizzas & cheese pizzas