Program a Game Engine from Scratch

Mark Claypool

Development Checkpoint #3

Vector & Object

This document is part of the book "Dragonfly – Program a Game Engine from Scratch", (Version 5.0). Information online at: `http://dragonfly.wpi.edu/book/`

Copyright ©2012–2017 Mark Claypool and WPI. All rights reserved.

4.5 The Game World

The game world itself is full of objects: bad guys running around; walls that enclose buildings and spaces; trees, rocks and other obstacles; and the hero, rushing to save the day. The exact type of objects depends upon the type of game, of course, but in nearly all games, the game engine has many objects to manage (game objects are introduced in Section 4.5.1 on page 72).

The game world needs to store and access groups of objects. In addition, the game programmer needs to access game objects in order to make them react to input or perform game-specific functions. So, the game world needs to manage them efficiently and present them in a convenient fashion. The game programmer might want a list of all the solid objects, a list of all the objects within the radius of an explosion, or a list of all the Saucer objects. It is the job of the world manager to store the game objects and provide these lists in response to game programmer queries. Section 4.5.2 on page 76 introduces lists of game objects.

The game world not only stores and provides access to the game objects, it also needs to update them, moving them around, see if they collide, and more. Section 4.5.4 (page 85) introduces methods to update game objects, with Section 4.5.5 (page 86) providing details on events, of vital importance for understanding how a game engine connects to game programmer code.

4.5.1 Game Objects

Game objects are a fundamental game programmer abstraction for items in the game. For example, consider Saucer Shoot from Section 3.3. As in many games, there are opponents to defeat (i.e., Saucers), the player character (i.e., the Hero), projectiles that can be launched (i.e., Bullets), and other game objects (e.g., Explosions, Points, etc.). Other games may have obstacles or boundaries to bypass (e.g., walls, doors), items that can be picked up (e.g., gold coins, health packs), and even other characters to interact with (e.g., non-player characters). The game engine needs to access all these objects, for example, to get an object's position. The game engine also needs to update these objects, for example, to change the location as the object moves. Thus, a core attribute for a game object, and the first one used in Dragonfly, is the object's position, stored as a 2d vector.

4.5.1.1 The Vector Class

The Vector class represents a 2d vector. When used for a position, the Vector is sufficient to hold a location in the game world. Some future version of Dragonfly could provide a third dimension, z, and/or provide coordinates as floating point numbers. The header file for the Vector class is described in Listing 4.25. Vector mostly holds the attributes `x` and `y`, with methods to get and set them. In addition to the default constructor (which set `x` and `y` to 0), on line 9, Vector has a constructor that sets `x` and `y` to initial values.

Listing 4.25: Vector.h

```
0 class Vector {
1
```

```
2  private:
3   float m_x;  // Horizontal component.
4   float m_y;  // Vertical component.
5
6  public:
7
8   // Create Vector with (x,y).
9   Vector(float init_x, float init_y);
10
11  // Default 2d (x,y) is (0,0).
12  Vector();
13
14  // Get/set horizontal component.
15  void setX(float new_x);
16  float getX() const;
17
18  // Get/set vertical component.
19  void setY(float new_y);
20  float getY() const;
21
22  // Set horizontal & vertical components.
23  void setXY(float new_x, float new_y);
24
25  // Return magnitude of vector.
26  float getMagnitude() const;
27
28  // Normalize vector.
29  void normalize();
30
31  // Scale vector.
32  void scale(float s);
33
34  // Add two Vectors, return new Vector.
35  Vector operator+(const Vector &other) const;
36 };
```

To make a Vector more generally useful, methods and operators on lines 26 to 35 are provided.

Vector `getMagnitude()`, shown in Listing 4.26, returns the magnitude (size) of the vector.

Listing 4.26: Vector getMagnitude()

```
0 // Return magnitude of vector.
1 void Vector::getMagnitude()
2   float mag = sqrt(x*x + y*y)
3   return mag
```

Vector `scale()`, shown in Listing 4.26, resizes (changes the magnitude) of the vector by the scale factor, leaving the direction the vector is pointing in the same.

Listing 4.27: Vector scale()

```
0 // Scale vector.
1 void Vector::scale(float s)
2   x = x * s
```

```
  y = y * s
```

Vector `normalize()`, shown in Listing 4.28, takes a vector of any length and, keeping it pointing in the same direction, changes its length to 1 (also called a unit vector). The `if` check is to avoid a possible division by zero.

Listing 4.28: Vector normalize()

```
// Normalize vector.
void Vector::normalize()
  length = getMagnitude()
  if length > 0 then
    x = x / length
    y = y / length
  end if
```

Overloading the addition operator (`+` in `v1 + v2`) for an Vector is shown in Listing 4.29. The method is called on the first vector (the Vector on the left-hand side of the '`+`'), with the second vector provided as an argument (the Vector on the right-hand side of the '`+`'). The variable `v` holds the new Vector, with `x` and `y` values added from the components of the other two vectors and then returned.

Listing 4.29: Vector operator+

```
// Add two Vectors, return new Vector.
Vector Vector::operator+(const Vector &other) const {
  Vector v                // Create new vector.
  v.x = x + other.x       // Add x components.
  v.y = y + other.y       // Add y components.
  return v                // Return new vector.
```

Other Vector operators (optional) The addition operator (`+`) is core since adding two vectors is used for many operations. However, there are other operators that may useful for a general Vector class, including: subtraction (`-`), multiplication (`*`), division (`/`), comparison (`==` and `!=`) and not (`!`). The aspiring programmer may want to implement them, using Listing 4.29 as a reference.

4.5.1.2 The Object Class

With the Vector class in place, the Object class can now be specified. The Object class definition is provided in Listing 4.30.

Listing 4.30: Object.h

```
// System includes.
#include <string>

// Engine includes.
#include "Vector.h"

class Object {

 private:
```

```
9   int m_id;             // Unique game engine defined identifier.
10  std::string m_type;   // Game-programmer defined identification.
11  Vector m_position;    // Position in game world.
12
13 public:
14  // Construct Object. Set default parameters and
15  // add to game world (WorldManager).
16  Object();
17
18  // Destroy Object.
19  // Remove from game world (WorldManager).
20  virtual ~Object();
21
22  // Set Object id.
23  void setId(int new_id);
24
25  // Get Object id.
26  int getId() const;
27
28  // Set type identifier of Object.
29  void setType(std::string new_type);
30
31  // Get type identifier of Object.
32  std::string getType() const;
33
34  // Set position of Object.
35  void setPosition(Vector new_pos);
36
37  // Get position of Object.
38  Vector getPosition() const;
39 };
```

Each Object has a unique id, initially set in the constructor (`Object()`), that may be of use in some games to uniquely identify an game object.[6] The id is obtained from a `static` integer that starts at 0 and is incremented each time `Object()` is called. The method `getId()` can be used to obtain an Object's id. The method `setId()` can be used to set an id manually, but in most cases this is never used.[7]

The type is a string primarily used to identify the Object in game code. For example, a Bullet object can `setType("Bullet")`, allowing a Saucer object to query a collision event to see whether or not the type was a "bullet", destroying oneself as an action. For the base Object constructor, the type can just be set to `"Object"`.

The Object `setPosition()` and `getPosition()` methods should allow changing the attribute `position` (via set) and retrieving it (via get).

Objects will have many attributes eventually (such as altitude, solidness, bounding boxes for collisions, sprites for animating, ...) but for now merely keeping the position of the game world is sufficient.

Note that the destructor for Object is `virtual` on line 20. This is necessary since

[6] In most cases, the game programmer can uniquely identify an object by its memory address, but an integer may still be more convenient. Moreover, a network game cannot count on a game object that is replicated on another computer to have the same memory address.

[7] A common exception is for synchronizing game worlds in a networked computer game.

Objects are deleted by the engine, not the game programmer, and the `virtual` keyword makes sure the right destructor is called. If the destructor was not `virtual`, when an Object was deleted by the engine, only `~Object()` would be invoked and not the destructor for a game programmer object that inherited from it.

4.5.2 Lists of Objects

In order to handle the management and presentation of objects to the game programmer, the world manager needs a data structure that supports lists of objects, lists that can be created and passed around (inside the engine and to the game programmer) in a convenient to use and efficient to handle manner. In passing the lists to the game programmer, if the Objects themselves are changed (e.g., say, by decreasing the hit points of Objects damaged in an explosion), updates need to happen to the "real" objects as seen by the world manager. While there are numerous libraries that could be used for building efficient lists (e.g., the *Standard Template Library* or the *Boost C++ Library*), list performance is fundamental to game engine performance, both for Dragonfly as well as for other game engines, so implementing game object lists provides an in-depth understanding of game engine performance. Plus, there are additional programming skills to be gained by implementing lists from scratch.

There are different implementation choices possible for lists of game objects, including linked lists, arrays, hash tables, trees and more. For ease of implementation *and* performance efficiency, an array is used for lists of game objects in Dragonfly. In addition, the *iterator* design pattern is used to provide a way to access the list without exposing the underlying data representation. In general, not having iteration as part of the container class separates the functionality for the collection from the functionality for iteration. This simplifies the collection (not having it cluttered with iteration methods), allows several iterations to be active at the same time, and decouples collection algorithms from collection data structures, while still leaving the details of the container implementation encapsulated. This last point means the internals of the list data structure can be changed (e.g., replace the array with a linked list) without changing the rest of the game engine or any dependent game programmer code.

For reference, consider a basic list of integers (`int`s), shown in Listing 4.31. The list starts out empty, by setting the count of items to 0 in the constructor. Basic operations allow the programmer to `insert()` items, `remove()` items, and `clear()` the entire list. Clearing the list and adding items to the list are quite efficient. Removing items from the list is rather inefficient, requiring the entire list to be traversed each time. However, perhaps most importantly, copying the list, which in a game engine happens often as many lists are created and destroyed by both the engine and the game programmer, is efficient as compilers (and programmers) handle fixed sized chunks of memory efficiently.

Listing 4.31: List of integers implemented as an array

```
0 const int MAX = 100;
1
2 class IntList {
3
4 private:
5   int list[MAX];
```

```
  int count;

public:
  IntList() {
    count = 0;
  }

  // Clear list.
  void clear() {
    count = 0;
  }

  // Add item to list.
  bool insert(int x) {
    if (count == MAX)  // Check if room.
      return false;
    list[count] = x;
    count++;
    return true;
  }

  // Remove item from list.
  bool remove(int x) {
    for (int i=0; i<count; i++) {
      if (list[i] == x) {            // Found...
        for (int j=i; j<count-1; j++) // ...so scoot over
          list[j] = list[j+1];
        count--;
        return true; // Found.
      }
    }
    return false; // Not found.
  }
};
```

While arrays are efficient, it is still not a good design for the game engine to have entire objects inside the list. In other words, the `int` in Listing 4.31 should *not* be replaced by an `Object` representing a game object. Instead, lists of game objects are handled by having *pointers* to the game objects. So, `int` is replaced with `Object *`. Using pointers allows the game engine to reference the game object's attributes and methods, but is much more efficient for creating the many needed lists of objects needed. Basically, copying a list of pointers is much faster (and uses less memory) than copying a list of game objects. In addition, lists passed from the engine to, say, the game code refer to the original Objects (via the pointers) and not copies. Last but not least, having Object pointers allows for polymorphism, as in Listing 1.1 on page 7, when Object methods are resolved.

Listing 4.32: ObjectList.h

```
const int MAX_OBJECTS = 5000;

#include "Object.h"
#include "ObjectListIterator.h"

class ObjectListIterator;
```

```
6
7  class ObjectList {
8
9  private:
10   int m_count;                      // Count of objects in list.
11   Object *m_p_obj[MAX_OBJECTS];     // Array of pointers to objects.
12
13 public:
14   friend class ObjectListIterator;
15
16   // Default constructor.
17   ObjectList();
18
19   // Insert object pointer in list.
20   // Return 0 if ok, else -1.
21   int insert(Object *p_o);
22
23   // Remove object pointer from list.
24   // Return 0 if found, else -1.
25   int remove(Object *p_o);
26
27   // Clear list (setting count to 0).
28   void clear();
29
30   // Return count of number of objects in list.
31   int getCount() const;
32
33   // Return true if list is empty, else false.
34   bool isEmpty() const;
35
36   // Return true if list is full, else false.
37   bool isFull() const;
38 };
```

Notice that line 14 of Listing 4.32 refers to an "ObjectListIterator" class that has not been defined. This class is defined in Section 4.5.2.1 and is used for efficient traversal of the list without exposing the attributes and internal structure of the list publicly. Line 5 is needed to act as a forward reference for the compiler, allowing compilation to proceed, as long as the ObjectListIterator class is defined before linking. Make sure this is also put inside the `df::` namespace (see page 51).

4.5.2.1 ObjectList Iterators

The ObjectList class, as described in Section 4.5.2, is a fine container class, but does not allow traversal of the items (Objects) in the list. Fortunately, this can be rectified by defining an *iterator* for the ObjectList class. In general, iterators "know" how to traverse through a container class without exposing the internal data methods and structure publicly. Iterators do this by being a `friend` of the container class, giving them access to the private and protected attributes of the class. Defining an iterator decouples traversing the container from the container iteration. This allows, for instance, the structure of the container to be changed (e.g., from a static array to a linked list) without redefining all the code that uses the container.

There can be more than one iterator for a given list instance, each keeping its own position. Note, however, that adding or deleting items to a list during iteration may cause the iterator to skip or repeat iteration of an item (not necessarily the one added) – the program should not crash, but the iteration may not touch each item once and only once.

There are 3 primary steps in coding an iterator for a container:

1. Understand container class (e.g., List)
2. Design iterator class for container class
3. Add iterator materials:
    - Add iterator as friend class of container class

To illustrate these steps, consider creating a IntListIterator for the List defined in Listing 4.31 on page 76. Step 1 is to understand the implementation of List, in terms of the attributes `p_obj[]` array and the `count` used to store and keep track of list members. Step 2 is to define an iterator for IntList, provided by IntListIterator in Listing 4.33. The constructor for IntListIterator (line 7) needs a pointer to the IntList object it will iterate over, which it stores in attribute `p_list`. Since the iterator does not change the contents of the list, this pointer is declared as `const`. The `index` attribute is used to keep track of where the iterator resides in the list during iteration. The `first()` method resets the iterator to the beginning of the list. Subsequently, `next()` and `isDone()` allow iteration until the end of the list. The method `currentItem()` returns the current item that the iterator is on. Note, although not shown for brevity, `index` should be error checked for bounds in `currentItem()` and `next()`.

Listing 4.33: Iterator for IntList class

```
0  class IntListIterator {
1
2  private:
3    const IntList *p_list;  // Pointer to IntList iterating over.
4    int index;              // Index of current item.
5
6  public:
7    IntListIterator(const IntList *p_l) {
8      p_list = p_l;
9      first();
10   }
11
12   // Set iterator to first item.
13   void first() {
14     index = 0;
15   }
16
17   // Iterate to next item.
18   void next() {
19     if (index < p_list -> count)
20       index++;
21   }
22
```

```
23   // Return true if done iterating, else false.
24   bool isDone() {
25     return (index == p_list -> count);
26   }
27
28   // Return current item.
29   int currentItem() {
30     return p_list -> item[index];
31   }
32 };
```

For step 3, in order for the IntListIterator to access the private member of the IntList class, namely the `item[]` array and the list `count`, IntListIterator must be declared as a `friend` class inside `IntList.h`.

```
0 friend class IntListIterator;
1 ...
```

Both `IntList.h` and `IntListIterator.h` need forward references to class IntListIterator and class IntList, respectively.

Once the IntListIterator is defined, a programmer that wants to iterate over an instance of List, say `my_list`, first creates an iterator:

```
0 IntListIterator li(&my_list);
```

Then, the programmer calls `first()`, `currentItem()`, and `next()` until `isDone()` returns true.

Listing 4.34: Iterator with while() loop

```
0 li.first();
1 while (!li.isDone()) {
2   int item = li.currentItem();
3   li.next();
4 }
```

A `for` loop provides for a bit shorter syntax:

Listing 4.35: Iterator with for() loop

```
0 for (li.first(); !li.isDone(); li.next())
1   int item = li.currentItem();
```

For Dragonfly, the complete header file for an ObjectListIterator is defined in Listing 4.36. Having the default constructor `private` on line 8 makes it explicit that a List must be provided to the iterator when created.

Listing 4.36: ObjectListIterator.h

```
0 #include "Object.h"
1 #include "ObjectList.h"
2
3 class ObjectList;
4
5 class ObjectListIterator {
6
```

```
7  private:
8   ObjectListIterator();      // Must be given list when created.
9   int m_index;               // Index into list.
10  const ObjectList *m_p_list; // List iterating over.
11
12 public:
13  // Create iterator, over indicated list.
14  ObjectListIterator(const ObjectList *p_l);
15
16  void first();          // Set iterator to first item in list.
17  void next();           // Set iterator to next item in list.
18  bool isDone() const;   // Return true if at end of list.
19
20  // Return pointer to current Object, NULL if done/empty.
21  Object *currentObject() const;
22 };
```

4.5.2.2 Overloading + for ObjectList (optional)

A useful abstraction for game programmers is to combine two ObjectLists, the result being a third, combined list holding all the elements of the first list and all the elements of the second list. A method named `add()` could combine two lists, written as part of the ObjectList class (e.g., `ObjectList::add()`) or as a stand alone function. However, a natural abstraction is to use the addition ('`+`') operator, overloading it to combine ObjectLists in the expected way.

Operators are just functions, albeit special functions that perform operations on objects without directly calling the objects' methods each time. Unary operators act on a single piece of data (e.g., `myInt++`), while binary operators operate on two pieces of data (e.g., `newInt = myInt1 + myInt2`). For ObjectLists, overloading the binary addition '(`+`)' operator is helpful. The syntax for overloading an operator is the same as for declaring a method, except the keyword `operator` is used before the operator itself.

Overloading the addition operator for an ObjectList is shown in Listing 4.37. The method is called on the first list (the ObjectList on the left-hand side of the '`+`'), with the second list provided as an argument (the ObjectList on the right-hand side of the '`+`'). The variable `big_list` holds the combined list, starting out by copying the contents of the first list ((`*this`) on line 4). The method then proceeds to iterate through the second list, inserting each element from the second list into the first list on line 10. Once finished iterating over all elements in the second list, the method returns the combined list `big_list` on line 14.

Listing 4.37: ObjectList operator+

```
0 // Add two lists, second appended to first.
1 ObjectList ObjectList::operator+(ObjectList list)
2
3   // Start with first list.
4   ObjectList big_list = *this
5
6   // Iterate through second list, adding each element.
7   ObjectListIterator li(&list)
```

```
  for (li.first(); not li.isDone(); li.next())
    Object *p_o = li.currentObject()
    big_list.insert(p_o) // Add element from second, to first list
  end for

  // Return combined list.
  return big_list
```

Since ObjectLists are implemented as arrays, a more efficient '+' operation could allocate one array of memory large enough for both lists, then, using `memcpy()` or something similar, copy the first list then the second list into the allocated memory. Care must be taken to get the pointers and memory block length correct. That is left as option for the reader to explore outside this text.

However, as an advantage, the implementation in Listing 4.37 is agnostic of the actual implementation of ObjectList. The lists could be implemented as either arrays or linked lists with pointers or some other internal structure and the code would still work.

Once defined, the ObjectList '+' operator can be called explicitly, such as:

```
ObjectList list_1, list_2;
ObjectList list_both = ObjectList+(list_1, list_2);
```

but a more natural representation is to call it as intended:

```
ObjectList list_1, list_2;
ObjectList list_both = list_1 + list_2;
```

4.5.2.3 Dynamically-sized Lists of Objects (optional)

A significant potential downside of the code shown in Listing 4.31 and Listing 4.32 is that the maximum size of the list needs to be specified at compile time. If the list grows larger than this maximum, items cannot be added to the list – the container class data structure cannot do anything besides return an error code. This is true even when there is memory available on the computer to store more list items. A full list is potentially problematic – for example, the world manager can no longer manage any more Ogres or the player cannot put more Oranges into a backpack. Specifying a larger maximum size and then recompiling the game engine and the game is hardly an option for most players!

What can be done instead is to make arrays that dynamically resize themselves to be larger as more items are required to be stored in the list. This has two tremendous advantages: 1) the maximum size of the list does not need to be known by the engine ahead of time, and 2) game object lists do not have to all be as large as the potential maximum size, but can instead be small when a small list is required and only become large when a large list is required, thus saving runtime memory and runtime processing time when lists are copied and returned. The downside of this approach is that more runtime overhead can be incurred when a list grows. If done right, however, this runtime overhead can be infrequent and fairly small.

The basic idea of dynamically sized lists is to allocate a relatively small array to start. Then, if the array gets full (via `insert()`), the memory is re-allocated to make the array larger. In order to avoid having the re-allocation happen every time a new item is inserted,

the re-allocation is for a large chunk of memory. A good guideline for the size of the larger chunk is twice the size of the list that is currently allocated.

In order to make this change, first, the ObjectList attribute for the list needs to be changed from an array to a list of pointers, such as `Object **p_list`.[8]

Next, the ObjectList constructor needs to allocate memory for the list dynamically. This can certainly be done via `new`, but memory can be efficiently resized using C's `realloc()`.[9] The initial allocation uses `malloc()` to create a list of size `MAX_COUNT_INIT`. `MAX_COUNT_INIT` is defined to be 1, but other sizes can certainly be chosen. The ObjectList destructor should `free()` up memory, if it is allocated.

Listing 4.38: Re-declaring list to be dynamic array label

```
0 max_count = MAX_COUNT_INIT;  // initial list size (e.g., 1)
1 p_item = (Object **) malloc(sizeof(Object *));
```

In the `insert()` method, if the list is full (`isFull()` returns `true`) then the item array is re-allocated to be twice as large, shown in Listing 4.39.

Listing 4.39: Re-allocating list size to twice as large

```
0 Object **p_temp_item;
1 p_temp_item = (Object **)
2               realloc(p_item, 2*sizeof(Object *) * max_count);
3 p_item = p_temp_item;
4 max_count *= 2;
```

The default copy constructor and assignment operator provided by C++ only do a "shallow" copy, meaning any dynamically allocated data items are not copied. Since the revised List class has dynamically allocated memory for the items, a copy constructor and assignment operator need to be created, doing a "deep" copy. The copy constructor and assignment operator prototypes look like:

Listing 4.40: Copy and assignment operator prototypes

```
0 ObjectList::ObjectList(const ObjectList &other);
1 ObjectList &operator=(const ObjectList &rhs);
```

In the assignment operator, memory for the copy needs to be dynamically allocated and copied over, along with the static attributes:

Listing 4.41: Deep copy of list memory

```
0 p_item = (Object **) malloc(sizeof(Object *) * other.max_count);
1 memcpy(p_item, other.p_item, sizeof(Object *) * other.max_count);
2 max_count = other.max_count;
3 count = other.count;
```

[8]The variable name is changed from `list` to explicitly depict that this is a pointer with dynamically allocated memory.

[9]Preliminary investigation running the Bounce benchmark and Saucer Shoot on both Linux Mint and Windows 7 suggests about 20% of the time when a list needs to expand, the memory block can be extended via `realloc()`, while 80% of the time the new block must be allocated elsewhere.

The assignment operator is similar, but with two additions before doing the "deep" copy: 1) the item being copied, `rhs`, must be checked to see if it is the same object (`*this`) to avoid copying the list over itself. Doing this check makes sense for efficiency and can also prevent some crashes in copying the memory over itself; 2) if the current object (`*this`) has memory allocated (`p_item` is not `NULL`), then that memory should be `free()`'d. Not doing this results in a memory leak.

Note, the above code needs error checking since calls to `malloc()` and `realloc()` can fail (returning `NULL`).

Lastly, the ObjectList destructor needs to check if (`p_item` is not `NULL`), and, if so, then that memory needs to be `free()`'d.

4.5.3 Development Checkpoint #3!

If you have not continued to do so, resume development now.

1. Create the Vector and Object classes, using the headers from Listing 4.25 and Listing 4.30, respectively. Add `Vector.cpp` and `Object.cpp` to the project and stub out each method so it compiles.

2. Implement both Vector and Object. Then, test even though the logic is fairly simple in both of these classes since they are primarily holders of attributes.

3. Create the ObjectList class, using the header from Listing 4.32. Refer to Listing 4.31 for method implementation details, remembering that ObjectList uses a static array of Object pointers. Add `.cpp` code to the project and stub out each method so it compiles.

4. Implement ObjectList and test. At this point, write a test program that inserts and removes elements from an ObjectList. Be sure to test boundary conditions – check if the `isFull()` method works, too, when the list reaches maximum size.

5. Create the ObjectListIterator class, using the header from Listing 4.36. Refer to Listing 4.33 for method implementation details. Add `.cpp` code to the project and stub out each method so it compiles.

6. Compile the iterator and write test code, referencing Listing 4.34 or Listing 4.35 for example code that uses the iterator. While iterating, adjust Object positions during run-time and print out values to verify code is working. Test cases where the list is empty, too.

Make sure to test all the above code thoroughly to be sure code is trustworthy (robust). Make sure the code is easy to read and commented sufficiently so it can be re-factored later as needed - the Object class is definitely re-visited as game objects grow in attributes and functionality.