Collections of Objects

Mikael Svahnberg*

2023-08-30

Contents

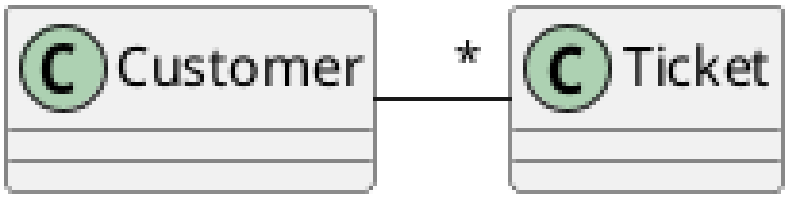
*Mikael.Svahnberg@bth.se

1 Introduction

- Barnes & Kölling Chapter 4, Grouping Objects
- Barnes & Kölling Chapter 7, Fixed-Size Collections – Arrays
- Collections of Objects
  - ArrayList
  - Array
- Iteration
  - Iterators
  - for-each
  - `while`
  - `for`
- Java Standard Library
- C++ Standard Libraries

2 Collections

- Examples
  - A *collection of Customers*
  - A *collection of Movie Shows*
  - A Cinema has *many Seats*
  - A Customer has *one or several Tickets*
- Shown in Class Diagrams:



3 Fixed or Flexible Size

- *Sometimes* the collection has a fixed size
  - Or at least a fixed upper bound
- *Most of the time*, the size of a collection is not known at design time.

Prefer Flexible Data Structures for collections
Examples of collection data structures:

Vector A flexble-size Array. Prefer `java.util.ArrayList`

Set Unordered collection where duplicates are not allowed.

Bag Unordered collection where duplicates *are* allowed.

Queue first in, first out

Stack Last in, first out

Dictionary Tuples of the form `<key, value>`

Linked List Mostly an internal implementation detail these days

Tree Can be quite useful for some problem types

HashMap A hash-value for each element decides where it is stored, makes searching fast

4 ArrayList

- Stores a collection of *objects*
- Is called a *generic class*
  - it can be instantiated with any type of objecs
  - uses the *diamond notation* `<classname>`
  - once instantiated, it can only store elements of the type `<classname>`

```
import java.util.ArrayList;

ArrayList<String> myList = new ArrayList<>(); // Element Type is inferred from the variabl
myList.add("Hello");
```

```
myList.add(new String("World"));
System.out.println(myList);

// Built-in Datatypes do not work
// ArrayList<int> myIntList = new ArrayList<int>();
ArrayList<Integer> myIntList = new ArrayList<>();
myIntList.add(Integer.valueOf(42));
myIntList.add(12); // 12 can be "upgraded" to an instance of the Integer class
System.out.println(myIntList);
System.out.println(myIntList.get(0) instanceof Integer);
```

5 Java Standard Library

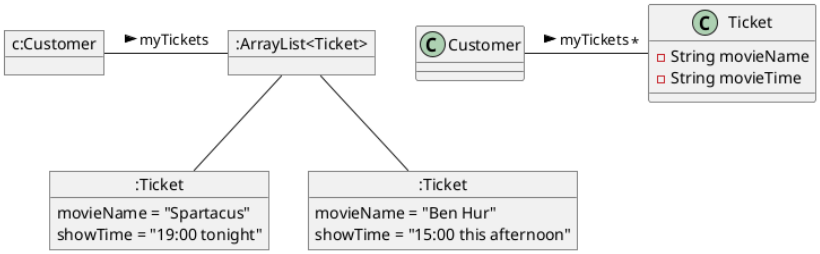
- `https://docs.oracle.com/en/java/javase/20/docs/api/index.html`
- Organised into a set of *Modules* and *Packages*
  - You will use the `java.base` most of the time.
  - `https://docs.oracle.com/en/java/javase/20/docs/api/java.base/module-summary.html`
- Example; the page for the `ArrayList<>` class:
  - `https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/ArrayList.html`
  - Class Overview
  - "See Also" (Similar classes, maybe you prefer to use one of these instead?)
  - Summary and details for
    - Fields (attributes)
    - Constructors
    - Methods

6 ArrayList in Action

```
import java.util.ArrayList;

public class Customer {
  private ArrayList<Ticket> myTickets;

  public Customer() {
    myTickets = new ArrayList<>();
  }

  public void addTicket(Ticket theTicket) {
    myTickets.add(theTicket); // Note that we no longer need a dedicated variable for theT
  }
```

```
  public int countTickets() {
    return myTickets.size();
  }

  public ArrayList<Ticket> getTickets() { // CAREFUL HERE
    return myTickets;                     // This is *not* a good idea to return
  }                                       // your internal and private attributes

  public void describeTicket(int ticketNumber) {
    if (0 <= ticketNumber   // We start at index 0
        && myTickets.size() > ticketNumber) { // size() is just outside of the collection.
      String details = myTickets.get(ticketNumber).toString();
      System.out.println(details);
    }
  }
}

public class Ticket {
  private String movieName;
  private String showTime;

  public Ticket() {
    this("-- not specified -- ", "-- not specified --");
  }

  public Ticket(String theName, String theTime) {
    movieName = theName;
    showTime = theTime;
  }

  public String toString() {
    return "Ticket for " + movieName + " at " + showTime;
  }
}


public class Start {
  public static void main(String [] args) {
    Customer c = new Customer();
    Ticket t = new Ticket("Spartacus", "19:00 tonight");

    c.addTicket(t);
    c.addTicket(new Ticket("Ben Hur", "15:00 this afternoon"));

    System.out.println(c.countTickets());
    c.describeTicket(0);
    c.describeTicket(1);
    c.describeTicket(3); // Will not print anything, since 3 is currently out of bounds.
  }
}
```

7 A Look at the Objects and Classes



- Design Principle: *Separation of Concerns*
- Customer does not know how many Tickets it has
- Customer does not know the details of any Tickets

8 Index number in Collection

- Elements in an ArrayList range from `[0 \dots size()-1]`
- When accessing an element, *check* that the index is within this range.
- Adding or removing an element in the middle reorders every element after.
- Accessing elements by index *may* be useful
  - Personally, I prefer not to if I can avoid it.

```
if (0 <= ticketNumber   // We start at index 0
    && myTickets.size() > ticketNumber) { // size() is just outside of the collection.
  String details = myTickets.get(ticketNumber).toString();
  System.out.println(details);
}
```

9 Traversing a Collection: for-each

```
Customer c = new Customer();
c.addTicket(new Ticket("Spartacus", "19:00 tonight"));
c.addTicket(new Ticket("Ben Hur", "15:00 this afternoon"));

for (Ticket t : c.getTickets()) { // For each element t of the type Ticket in collection c
  System.out.println(t.toString());
}
```

- This was the reason for the method `Customer.getTickets()`, and it is a bad reason
- The rest of the world does not need to know *how* Customer stores its tickets.
- Only the Customer class should operate on a customer's Tickets.

- Design Principle: *Low Coupling*
  - The lesser I know, the more loosely connected we are
- Design Principle: *High Cohesion*
  - Each class has sole responsibility for its own data.

10 Filtering a Collection

```
Customer c = new Customer();
c.addTicket(new Ticket("Spartacus", "19:00 tonight"));
c.addTicket(new Ticket("Ben Hur", "15:00 this afternoon"));

for (Ticket t : c.getTickets()) {
  if (t.getName().contains("tonight")) {
    System.out.println(t.toString());
  }
}
```

- Filtering collections is *extremely* useful
- In functional programming, it is usually the starting point of nearly *everything*:
  1. Given a collection
  2. Filter to remove everything not relevant
  3. Do something with the remaining elements
  4. (maybe) repeat steps 2 and 3
  5. Translate whatever remains to the format you want
  6. ...
  7. Success!
- Many languages have built-in support for this.
- Later versions of Java support it with the `Streams` API (which we will not cover in this course).

11 Other forms of Iteration: while

```
int x = 5;

while (0 <= x) {
  System.out.print(" " + x);
  x--; // If you forget this line, x will never update and the while loop will continue fo
}

System.out.println();
System.out.println("x = "+x);
```

- Repeat while some condition tests to `true`
- Can go on forever, if you are not careful
- Often used if you do not know when to end, e.g.
  - *while (user has not exited the menu)*
  - *while (there are more elements in the database)*
  - *while (there are more lines in this file)*
  - *while (I still have not found a movie that shows tonight)*
- Boolean expression can be arbitrarily complex: `while (index < myTickets.size() && !found && !userAborted)`

12 While without index

```
int f0=0;
int f1=1;
int fn=f1 + f0; // Fibonacci Sequence

while (fn < 100) {
  System.out.print(" " + fn);
  f0 = f1;
  f1 = fn;
  fn = f1 + f0;
}
```

13 Iterators

```
import java.util.ArrayList;
import java.util.Iterator;

Iterator<Ticket> it = myTickets.iterator(); // Get an iterator for the myTickets collectio
while (it.hasNext()) {
  Ticket t = it.next();
  t.getDetails();
}
```

- Goes hand in hand with a collection
- We *could* linger at a particularly troublesome element before calling `it.next()`
  - ... this rarely happens in practice, though.
- I suspect that `for-each` uses Iterators internally
- We are not allowed to remove elements in a `for-each`, but we can if we use the Iterator.
  - `it.remove()` ← can only remove the *current* element.

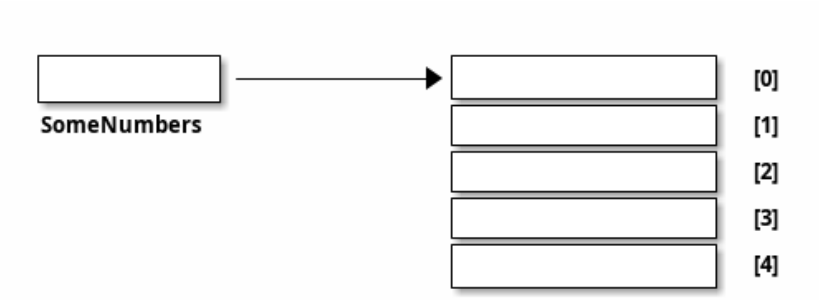
14 Arrays

```
int[] someNumbers = new int[5];
someNumbers[0] = 42;
someNumbers[3] = 12;

for(int num : someNumbers) {
  System.out.print(" " + num);
}


System.out.println();
int [] otherNumbers = {23, 31, 57};
System.out.println(otherNumbers.length); // Note -- length is an attribute and not a metho
```

- Arrays are built-in and can operate on any type.
- Arrays are *fixed size*, extending or shrinking has to be implemented by yourself
- Inserting and removing elements has to be implemented by yourself
- *May* have better performance than e.g. ArrayList

15 Deeper into Array Creation

```
int [] someNumbers;
someNumbers = new int[5];
```



- `int[] someNumbers` creates a variable that holds a reference to an array
- `new int[5]` allocates *consecutive* space for 5 integers.

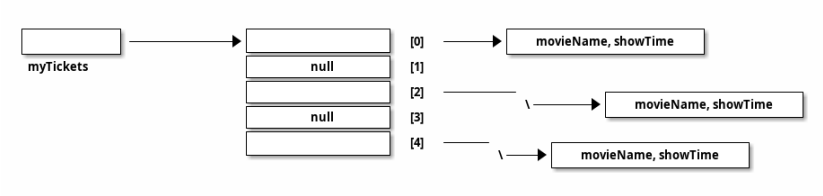
16 Array of Objects

- What happens if we allocate an array of e.g. Tickets:
  - `Ticket[] myTickets`
  - `myTickets = new Ticket[5]`

- What is the output of:

```
Ticket [] myTickets = new Ticket[5];

for(Ticket t : myTickets) {
  System.out.println(t.toString());
}
```

16.1 Objects and Object References



17 Yet another iteration: for

```
for (int i = 0; i < 10; i++) {
  System.out.print(" " + i);
}
```

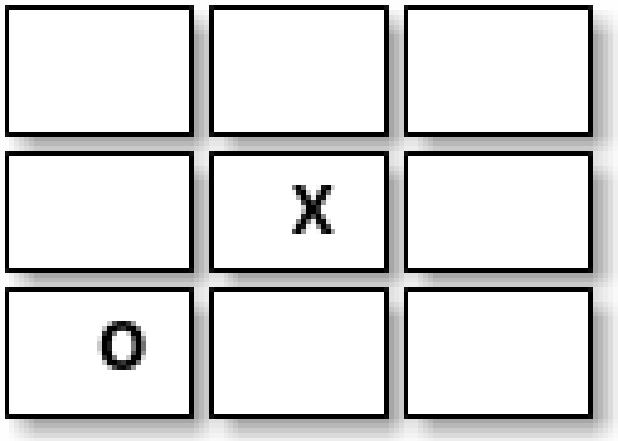
- `for (/<initialisation>/ ; /<condition>/ ; /<increment>/) { /<statements>/ }`
- Difference to for-each is that we can use the iterator (e.g. `i` above) inside the loop
- Does not have to look like above, initialisation, condition, and increment can be quite different:

```
for (Query q=new Query("Select * from Users") ; q.hasMoreElements(); q.nextElement() ) {
  System.out.println(q.currentElement());
}

for (Iterator<Ticket> it = myTickets.iterator() ; it.hasNext() ; /* empty increment */ ) {
  Ticket t = it.next();
  // ...
}
```

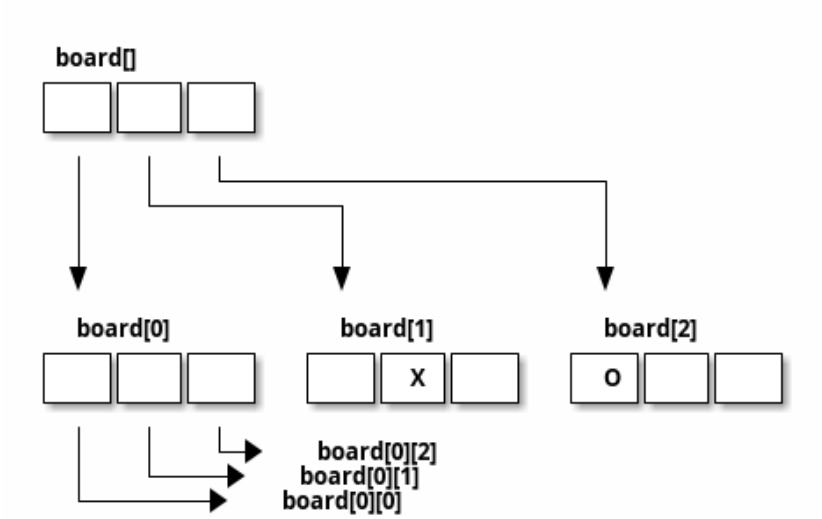
-Compare to the while loop:

<initialisation>;
while (*<condition>*) {
    <statements>
    <increment>
}

18 Two-dimensional arrays



- Example *Tic-Tac-Toe*
- Can be seen as an *array of arrays*: `board[row][column]` , or `(board[row])[column]`
- Whether to put *rows* or *columns* first depends:
  - Number of rows vs number of columns
  - Read and write behaviour
    - mostly along a column, or
    - mostly along a row




```
char board[][] = new char[3][3];
board[1][1] = 'X';
board[2][0] = 'O';

for (int row = 0; row < board.length; row++) {
  for (int column = 0; column < board[row].length; column++) {
    System.out.print(" " + board[row][column]);
  }
  System.out.println();
}

System.out.println("----------");

for (char[] row : board) {
  for (char pos : row) {
      System.out.print(" " + pos);
  }
  System.out.println();
}
```

19 Again in C++

19.1 Standard Library, Containers

- C++ Standard Library: `https://en.cppreference.com/w/cpp`
- Part of the *Containers* library, split into
  - Sequence Containers
    - `array` fixed-size wrapper around built-in arrays
    - `vector` flexible sized array (corresponds to Java ArrayList)

        * `deque` Double-ended queue
        * `forward_list` singly-linked list
        * `list` doubly-linked list
    - Associative Containers
        * `set`
        * `map`
        * `multiset`
        * `multimap`
    - Container Adaptors
        * `stack`
        * `queue`
        * ...

19.2 std::vector

```
#include <iostream>
#include <vector>

int main(void) {
  std::vector<int> myIntList; // C++ vector works with built-in datatypes as well as user-
  myIntList.push_back(12);
  myIntList.push_back(42);

  std::cout << myIntList.size() << std::endl
            << myIntList[0] << " " << myIntList.at(0) << std::endl
            << myIntList.front() << " " << myIntList.back() << std::endl;
}
```

19.3 Vector in action

1. Class Customer

```
#ifndef CUSTOMER_HH
#define CUSTOMER_HH
#include <vector>
#include "ticket.hh"

class Customer {
public:
  Customer(void) {};
  void addTicket(Ticket* theTicket);
  int countTickets();
  std::vector<Ticket*> getTickets(); // Same caution as before, this exposes your inte
  void describeTicket(int theTicketNumber);
private:
  std::vector<Ticket*> myTickets;
};
#endif
```

```
#include <vector>
#include <iostream>
using namespace std;
#include "customer.hh"
#include "ticket.hh"

void Customer::addTicket(Ticket* theTicket) {
  myTickets.push_back(theTicket);
}

int Customer::countTickets(void) {
  return myTickets.size();
}

vector<Ticket*> Customer::getTickets() {
  return myTickets;
}

void Customer::describeTicket(int theTicketNumber) {
  if (0 <= theTicketNumber && myTickets.size() > theTicketNumber) {
    cout << myTickets[theTicketNumber]->toString() << endl;
  }
}
```

2. Class Ticket

```
#ifndef TICKET_HH
#define TICKET_HH
#include <string>


class Ticket {
public:
  Ticket(void) : Ticket("-- not specified -- ", "-- not specified --") {}
  Ticket(std::string theName, std::string theTime) : movieName(theName),showTime(theTi
  std::string toString(void) {
    return "Ticket for " + movieName + " at " + showTime;
  }
private:
  std::string movieName;
  std::string showTime;
};

#endif
```

3. main function

```
#include <iostream>
#include "customer.hh"
#include "ticket.hh"
```

```
int main(void) {
    Customer* c = new Customer();
    Ticket* t = new Ticket("Spartacus", "19:00 tonight");

    c->addTicket(t);
    c->addTicket(new Ticket("Ben Hur", "15:00 this afternoon"));

    std::cout << c->countTickets() << std::endl;
    c->describeTicket(0);
    c->describeTicket(1);
    c->describeTicket(3); // Will not print anything, since 3 is currently out of boun

}
```

19.4 Vector as a pointer

```
#include <vector>
#include <iostream>
using namespace std;

int main(void) {
  vector<int>* myList = new vector<int>();
  myList->push_back(12);
  myList->push_back(42);
  cout << myList->at(1) << endl;
  // cout << myList->at(3) << endl; // Will throw a runtime error
  // cout << myList[1] << endl; // CAUTION this will not work!
  cout << (*myList)[0] << endl;  // First we need to dereference the pointer (*myList)
                                 // Then, it can be used as a "normal" variable

  // Why? It has to do with C++ arrays...
}
```

19.5 Traversing Collections

- `for(:)` Similar to for-each in Java
  - There is also a `for_each` , but it requires a pointer to a function.
- `while` as in Java
- `for` as in Java
- C++ Also have `iterators`, but I'm not so sure about their necessity

```
#include <vector>
#include <iostream>

using namespace std;
```

```
  int main(void) {
    vector<int> myList{3,12,42,44,8,25};

    for(int element : myList) {
      cout << element << " ";
    }
    cout << endl;

    // Can also use the "auto" type, to infer the type for the elements:
    for(auto element : myList) { cout << element << " "; }
    cout << endl;

    int i = 0;
    while(myList.size() > i) {
      cout << myList[i] << " ";
      i++;
    }
    cout << endl;

    // for-loop
    for(int i = 0; i < myList.size(); i++) { cout << myList[i] << " "; }
    cout << endl;

    // for-loop with an iterator
    vector<int>::iterator itr;
    for(itr=myList.begin(); itr!=myList.end(); ++itr) { cout << *itr << " "; }
    cout << endl;
}
```

19.6 Arrays

- Arrays and pointers are often interchangeable
- Most of the time we don't know the size of the array during startup
  - instead, we declare a pointer and allocate an array later on.
- Weaving back and forth between arrays and pointers is powerful and very useful
  - it is also dangerous: c++ does not stop you from doing stupid things.

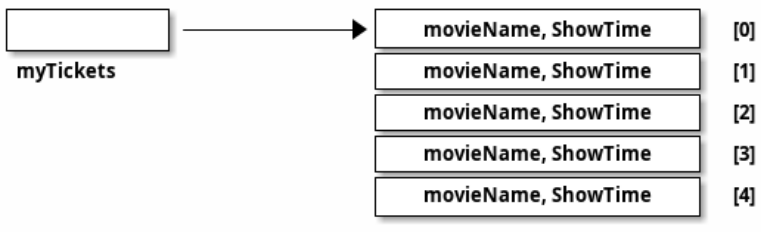
```
#include <iostream>
using namespace std;

int main(void) {
  int someSize = 10;

  int anArray[5];                          // Size is given either as a literal...
  int anotherArray[someSize];              // ...or using another variable
  long long bit64[] = {10,9,8,7};          // An array given an initial set of values
```

```
  int* initByPointer = new int[someSize]; // declared as a pointer, memory allocated with

  int* start = anotherArray; // The array is just a pointer with space allocated upon crea
  *start = 5;                // Write '5' to the position that start is pointing at.
  cout << anotherArray[0] << *anotherArray
       << *start << start[0] << *(&start[0]) << endl;

  start = &anotherArray[0];  // '&' = the address of the 0th position of the anotherArray

  anotherArray[1] = 12;
  cout << *(start+1) << *(bit64+1) << endl;  // C++ keeps track of the size of the entitie
  cout << 1[bit64] << endl; // Just to mess with your heads, this is also valid!

  int valueOfPositon = anotherArray[120];  // No bounds-check, we are reading memory which
                                           // Worse, we can also write here!

  // A special case, char* arrays:
  char aText[] = "abc";
  char* pos = &aText[2];
  cout << aText[1] << *pos << aText[3] << endl; // Strings are terminated with \0

  for (char* pos = aText; '\0' != *pos; pos++) { cout << *pos; }
  cout << endl;
}
```

19.7 Deeper into C++ Arrays I

```
Ticket* myTickets = new Ticket[5];
```

- `myTickets` is a pointer
  - `*myTickets` == The value of the memory position stored in variable `myTickets`
  - `&myTickets` == The memory position for the variable `myTickets`
  - `myTickets[1]` == The value of the first position after the memory position stored in variable `myTickets`



19.8 Deeper into C++ Arrays II

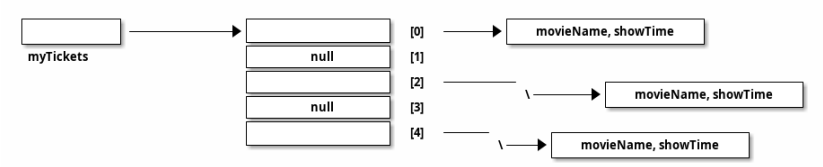
- More often we want to have an array of pointers to objects

- More similar to Java

```
#include "ticket.hh"

int main(void) {
  Ticket** myTickets = new Ticket*[5];
  myTickets[0] = new Ticket("Spartacus", "tonight");
  myTickets[2] = new Ticket("Ben Hur", "tomorrow");
  myTickets[4] = new Ticket("Also Spartacus", "tomorrow");

  myTickets[0]->toString();
}
```

As before:

- `myTickets` is a pointer
  - `myTickets[0]` == The value of the memory position stored in variable `myTickets`
    - Which is *also* a pointer
    - `*myTickets[0]` == The value of the memory position in the memory position stored in the variable `myTickets`



19.9 Two-dimensional Arrays

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

int main(void) {
  // char board[3][3]; // This would also work, but less fun for me

  char** board = new char*[3]; // An array of pointers

  // Must initialise with a meaningful value, otherwise it will be full of
  // whatever junk was lying in those memory addresses before.

  for (int row=0; row<3; row++) {
    board[row] = new char[3];  // The first thing we must do is to create a new array to r
    for (int col=0; col<3; col++) {
      board[row][col] = '-';   // Then we can fill this second array with good values.
    }
  }
```

```
  // Now we have initialised all the memory, and can reference it just like before:
  board[1][1] = 'X';
  board[2][0] = 'O';

  for (int row=0; row<3; row++) {
    for (int col=0; col<3; col++) {
      cout << board[row][col];
    }
    cout << endl;
  }
}
```

20 Summary

- Design Principle: *Separation of Concerns*
- Design Principle: *Low Coupling*
- Design Principle: *High Cohesion*
- Standard Library
  - `https://docs.oracle.com/en/java/javase/20/docs/api/index.html`
  - `https://en.cppreference.com/w/cpp`
- `java.util.ArrayList`
- `std::vector`
- Array[]
- Conditionals and Loops
  - if
  - while
  - for
  - for-each `for(:)`
- C++ Pointers again
- Collections are *extremely important* in order to hold and manage a set of objects of the same type

21 Next Lecture: Inheritance and Polymorphism

- Barnes & Kölling Chapter 8, Designing Classes
- Design Principle: *Low Coupling*
- Design Principle: *Encapsulation*

- Inheritance
- Polymorphism
- Multiple Inheritance
  - `extends` vs `implements`
  - abstract classes