Chapter 1

Object-Oriented Programming Concepts

In order to support object-oriented programming, a language must include: abstract data types, inheritance, and dynamic binding. We typically discuss the object-oriented features present in a language, and, based upon those characteristics, then determine if the language qualifies as object-oriented. The blurry distinction can lead to aggressive, pointless debate. Languages evolve, and at some point they possess enough features for developers to find them useful when developing in the object-oriented style.

The major features we consider in this course include:

- Abstraction: Creating functions and classes
- Encapsulation and Information Hiding: Providing the correct interface
- Inheritance: Extending code developed in other objects
- Dynamic Binding: Polymorphism

1.1 Abstraction

Per Sebesta, abstraction is, "a view or representation of an entity that includes only the most significant attributes."

What are the minimum features or characteristics a thing must possess to be named that thing? What are the minimum features in a car? A plane? A writing instrument? With abstraction, we try to distill a concept to its most essential components. For example, in order to be considered an omelette, what characteristics must the thing have?
Per Dictionary.com, it is "a dish of beaten eggs cooked in a frying pan until firm, often with a filling added while cooking, and usually served folded over."

This definition makes no mention of the specifics of the filling, its texture, or the number of eggs used. Were one constructing a menu of all the restaurant’s available omelette options, however, one would not need these data. Simply knowing the item is an omelette suffices. Ham and Cheese, Italian, Egg-white only – the specifics of the variations play no role in determining their rightful place on the menu in the omelette section.

By reducing an object to its most essential components, we simplify the resultant software. If every omelette definition included a place-holder for the number of olives present, this field would waste space in the majority of omelette recipes. What about the variation of green or black olives? Instead of including these fields in the definition of every omelette, these features only belong in the specific variety of omelette that includes them. Omelettes also possess a temperature, for one would not likely enjoy the dish after it sits on a plate for three hours. Should the omelette abstraction include its temperature?

The fewer hooks and connections an object offers, the less complex it is to use the object. Should the class include the instance’s current temperature? If these data fields are used for a menu generator, these data possess no significance, so including them in the abstraction serves little purpose and requires additional space.

Note: Make certain the abstraction represents a class and not simply the role it plays in the code. That is, does the software warrant Mother and Father classes, or are these roles of a Person class with two instances of mother and father?

1.1.1 Process Abstraction

In Java, when one wishes to send output to the terminal screen, one typically uses some variant of System.out.println("Driving on nine"). Alternatively, one could adjust the individual pixel values for the appropriate display positions on a pixel-by-pixel basis to achieve this result. Doing so, however, proves unnecessarily tedious. The process of writing output to the screen is effectively the same when printing "you could be a shadow" or "beneath a streetlight." Only the output values – the specifics of what one is printing at that time – that changes.
```cpp
#include <iostream>
using namespace std;

void _init();

int main(){
    _init();
}

void _init(){
    // ... insert initialization sequence here
    cout << "Initialized." << endl;
}
```

Figure 1.1: One example of process abstraction in the C++ language. The _init function abstracts away the concept of initialization.

Here, we see the need for process abstraction. Rather than setting the values for each pixel by adjusting the output value of a CPU register, one can group the essential characteristics of this concept into a function and then call this function by name. The System.out.println method does one thing, and it does so cleanly.

The C language supports the creation of functions, like Java, Python, C++, and Elixir, so all these languages support process abstraction. This does not make these all object-oriented languages, however, for process abstraction is only one feature in an object-oriented language.

1.1.2 Data Abstraction

In the C language, a struct represents a record of data. One may view a struct as the first, sputtering start toward data abstraction. Recall that with these data types one may begin grouping together like items into a single, monolithic entity. Rather than storing an array of student identification numbers and an array of grades, one may create a student abstraction through a structure, and inside this abstraction it includes both the grade and identification number, for these fields are closely related.

An album possesses a list of the names of contributing artists, and it is likely that when one inspects the bassist of a band that one would also want to parse the lead guitarist or drummer. Thus, keeping these fields collected together makes sense. Due to the principle of locality, which captures the
```cpp
#include <iostream>
#include <vector>

int main()
{
    // init the object holding the specified values
    std::vector<int> v = {3, 20, 4, 40, 4};

    // Manipulate the object by adding to its end
    v.push_back(96);

    // Iterate and print values of vector
    for(int i : v) {
        std::cout << i << std::endl;
    }

    v.clear();
}
```

Figure 1.2: An example of an abstract data type in C++ using a `vector`. This object holds the data and includes methods to manipulate the items inside.

The idea that computer processors tend to access the same set, or near adjacent set, of memory locations. This concept includes both the physical location in memory, but it also extends into the time domain. That is, after using a variable, the computer will likely use it again in a short period of time.

Although a step in toward object-oriented, these concepts fail to fully meet the object-oriented concept of data abstraction. Object-oriented languages support abstract data types (ADT). An ADT includes the data fields, like those in a record, but it also includes the functions, sub-programs, or methods necessary to manipulate the data. These objects group together both the items one wishes to hold, but also the functions necessary for adding or removing from the structure.

Abstractions within an object-oriented programming model communicate by passing messages. Rather than directly modifying or accessing a variable, one uses methods to reach into the class and get these data. It is rude for a waiter to reach into a customer’s bag for payment – one must request money instead of directly taking it from someone.
Heuristic: A class should capture one, and only one, key abstraction.

### 1.2 Encapsulation and Information Hiding

The concepts of *encapsulation* and *information hiding* remain closely linked, but they serve different purposes.

Encapsulation is a strategy for achieving information hiding.

Heuristic: All data should be hidden within its class, for if all data is public, how do we separate the essential? Where are the dependencies?

Heuristic: Beware of classes with many getters and setters, for they reflect poor design encapsulation.

### 1.2.1 Access Modifiers and Specifiers

Supporting the object oriented concepts of encapsulation and information hiding, programming languages like Java and C++ include special keywords to indicate the visibility of a given variable within a scope. The goal remains simplifying the interface to other components so that only the essential is exposed.

The C programming language provides no mechanism for access modification, for it does not include classes and the structures remain data focused. That is, structures in C hold data, but they do not include methods or the notion of operator overloading. Python fully supports classes, but it does not provide a method for access modification.

To indicate a variable or method is internal in Python, one simply prefixes its name with an underscore character. This naming convention helps developers achieve much of the same encapsulation without forcing it in software.

**In Java** Access modifiers are the keywords which are used with classes, variables, methods and constructors to control their level of access. Java has four access modifiers: *public*, *private*, *protected*, and *package* (or *default*).
When no access modifier is specified, its visibility is limited within the package.

- **public**: It has visibility everywhere.
- **private**: It has visibility only within its own class.
- **protected**: It has scope within all inheriting classes and objects in the same package.

**Figure 1.3: Access specifiers in Java.**

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Visibility within current scope</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>public</strong></td>
<td>It has visibility everywhere.</td>
</tr>
<tr>
<td><strong>private</strong></td>
<td>It has visibility only within its own class.</td>
</tr>
<tr>
<td><strong>protected</strong></td>
<td>It has scope within all inheriting classes.</td>
</tr>
</tbody>
</table>

**Figure 1.4: Access specifiers in C++.**

In C++, in classes and structures, three keywords impact a variable’s visibility within the current scope: **public, private, and protected.**

**Friendly Modifiers**

A friend function is a function outside the class with special permission to access the internal state of the class. Because it may access the internal workings of a class, the target class must specify that the friend function has access with the **friend** keyword before use. That is, a class must explicitly identify all external functions it considers friendly.

**1.2.2 Naming Encapsulations**

As software systems pass beyond the toy, educational level application and into real-world applications, they grow increasingly complex. Moreover, they likely include code from a number of sources. Some of these sources might be other, in-house developed applications, but they may also come from third-party sources. One might wish to include computer vision software modules developed by industry experts rather than developing one from scratch.

As these other tools become part of a software package, the global functions and variables they define also become part of the current software design. To help prevent these clashes, or **name collisions**, many languages include the concept of a **namespace.**

---

1 hint, hint
```cpp
// classes_as_friends1.cpp
// compile with: /c

class B;

class A {
public:
    int Func1( B& b );

private:
    int Func2( B& b );
};

class B {
private:
    int _b;

    // A::Func1 is a friend function to class B
    // so A::Func1 has access to all members of B
    friend int A::Func1( B& );
};

int A::Func1( B& b ) { return b._b; } // OK
```

Figure 1.5: Friendly example from Microsoft.
namespace Fleet
{
    // a class object in the namespace
    class SuperDimensionalFortress
    {
    public:
        void Transform();
        bool isSelfAware();
    }

    // This is C++, so a function without a class in the namespace
    void CheckStatus(SuperDimensionalFortress){}
};

Figure 1.6: An example of namespace in C++

Fleet::SuperDimensionalFortress sdf;
sdf.Transform();
Fleet::CheckStatus(sdf);

Figure 1.7: Working with the fully-qualified name.

The use of a namespace helps developers organize their code into meaningful units. All the items within the same namespace may "see" one another without a fully-qualified name. A namespace is a way to identify which variables or functions should take preference over others when they have the same name, and it allows developers to write code using shorthand – for the namespace means one need not include the entire qualification.

Namespaces in C++

If one does not include a namespace identifier in a source unit, then the compiler automatically places it in the global namespace. Keeping items in this namespace serves useful purposes in an educational environment, but in any meaningful program good form dictates that developers place their classes in an appropriate namespace. C++ places some of the most universally useful tools in the std namespace. These subroutines and objects create the C++ Standard Library.

In addition to defining a namespace at the global level, one might wish to break sub-sections of code into their own, distinct namespaces. That is,
using Fleet::SuperDimensionalFortress;
SuperDimensionalFortress sdf;

Figure 1.8: Bringing one identifier into scope with the using directive.

using namespace Fleet;
SuperDimensionalFortress sdf;
sdf.Transform();
CheckStatus(sdf);

Figure 1.9: Grabbing everything in the namespace and bringing it into scope.

one may wish to nest namespaces. Entities within a nested namespace have access to all other entities within the same, nested-namespace, but they may also access entities within their parent namespace.

On the other hand, entities in the outer namespace may not access the members of the inner namespace without qualification.

Namespaces in Java

In Java, one may group together related classes in a common package. Were one developing custom data structures, for example, one might elect to place them all in the package datastructures. This way, when you reference your custom Vector object, it does not conflict with a similar structure in the Java Collections Framework.

All entities within the same package in Java have access to the public, protected, and unspecified access permission level (package-scope) of all other types within the package. That is, everything in the same package may peer into all but the inner-most, private workings of the objects.

1.3 Inheritance

Sometimes, one needs to build a program up from the ground by its bootstraps, but other times it’s nice to suddenly gain powers due to your parent’s status. The inheritance or an estate goes to the next-of-kin in a simple model. Object-oriented languages support this abstraction in some way.

---

2 Or edu.sdsu.cs.datastructures if you have an awesome 310 instructor
3 Genetic testing may complicate this in the future.
Using inheritance, the new types defined by the software become a specialized type of the parent class. They possess an IS-A relationship. Using inheritance, we may establish a model for a simple Duck class. In this model, we want to simulate multiple duck types: Wood Ducks, Mallards, and Canvasback. All three of these animals possess the abilities to quack and fly.

For ducks, apparently, each type produces a slightly different call. Consequently, in this application, we want to specify that all ducks possess this ability, but we want to require every duck type to implement their own duck sound. For our purposes, the birds in this model all fly the same way, so they do not need unique behavior for that method.

Our goal is to:

- Create a simple, universal fly method all ducks can share.
- Require each implementing type to define their own quack noises.

We could implement this functionality directly in each specific class. Instead, we will add a level of abstraction to the problem. Although we want to actually model three specific duck types, we will begin by introducing a higher-level abstraction: the Duck. We do not intend for implementations to actually instantiate bland duck objects, but every mallard and wood duck share functions, and one might combine these functions into a parent class object so one only needs to write the code in one place.

One major advantage of Inheritance emerges in its ability to introduce new functionality into the inheriting classes without changing their source code. So far, we have discussed inheriting functions in advance and by design, but we may also use inheritance during the maintenance cycle. If one were to modify the Duck parent class by adding new functions and their associated code, then every existing class that inherits from Duck also gains that ability.
```cpp
class name : access_mode parent_class_name
{
    // Define the subclass here . . .
};
```

Figure 1.11: Basic structure for inheritance in C++.

- Pro: Model relationships between objects
- Pro: Code-reuse
- Pro: Add new features
- Con: Introduces dependencies which may be difficult to unravel
- Con: Tight coupling

1.3.1 Inheritance in C++

We may model this problem in C++ directly using the built-in classes and the language’s inheritance procedure. We simply need to define the base behavior in the Duck class, and then all children gain the ability.

To help support the concepts of information hiding, C++ permits access specifiers which modify the visibility of the inheritance. That is, one might wish to inherit the public members of a class in a way that hides those features from the outside world, for if you derive from a class with public functions, then those functions become a part of the child as well. This might clutter up the interface.

Consequently, C++ permits the developer to specify the way to treat the public methods and fields it receives from its base class. The modifiers follow standard C++ convention with: public, protected, and private. What use could this be – Implement a stack.

Virtual Functions

The member function area has been declared as virtual in the base class because it is later redefined in each of the derived classes. Non-virtual members can also be redefined in derived classes, but non-virtual members of derived classes cannot be accessed through a reference of the base class: i.e., if virtual is removed from the declaration of area in the example above, all
```cpp
#include <iostream>

namespace Targets
{
    class Duck
    {
    public:
        virtual void quack(void) {}
        void fly() { std::cout << "Flap Flap" << std::endl; }
    };

    class WoodDuck : public Duck
    {
    public:
        void quack(void);
    };

    class Mallard : public Duck
    {
    public:
        void quack(void);
    };

    class Canvasback : public Duck
    {
    public:
        void quack(void);
        void fly(void) { std::cout << "Canvas Flap" << std::endl; }
    };
    
};
```

Figure 1.12: An example of Inheritance in C++ using ducks.
using namespace Targets;

void WoodDuck::quack(void){
    std::cout << "Woody Quack!" << std::endl;
};

void Mallard::quack(void){
    std::cout << "Malignant Quack!" << std::endl;
};

void Canvasback::quack(void){
    std::cout << "Burlap Quack!" << std::endl;
}

int main(){
    Targets::WoodDuck fred;
    fred.quack();
    fred.fly();

    Mallard mary;
    mary.quack();
    mary.fly();

    Canvasback carl;
    carl.quack();
    carl.fly();

    std::cout << "Complete!" << std::endl;
}

Figure 1.13: Using the earlier duck example defining class methods outside their definition.
Woody Quack!
Flap Flap
Malignant Quack!
Flap Flap
Burlap Quack!
Canvas Flap
Complete!

Figure 1.14: Output produced after running the duck example.

three calls to area would return zero, because in all cases, the version of the base class would have been called instead.

Declare the methods in the base class that software can redefine in derived classes.

Virtual functions act as a type of interface to methods in derived classes. All derived classes which provide implementations for these methods are said to override them. Although listed under inheritance, virtual functions help abstraction, encapsulation, and dynamic binding.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>virtual</td>
<td>The method <em>may</em> be overridden</td>
</tr>
<tr>
<td>=0</td>
<td>The method <em>must</em> be virtual and it <em>must</em> be overridden</td>
</tr>
<tr>
<td>override</td>
<td>The method is meant to override a virtual method in the base class</td>
</tr>
<tr>
<td>final</td>
<td>The method is not meant to be overridden</td>
</tr>
</tbody>
</table>

Pure Virtual Base Classes

One may specify individual methods are virtual and meant to be overridden, but what if the class definition does not include any member fields, so it does not need state? This class might not need a Constructor. What if every method defined were virtual and the file does not provide implementations. That is, what if the class definition file only required implementing classes to perform work? The Class itself does nothing except specify the interface one plans to use when interacting with objects of this type.

Abstract Base Classes

Sometimes, one wishes to provide a class implementation with *most* of the software provided so that developers only need to spend time working on


```cpp
class Polygon {
    protected:
        int num_sides;

    public:
        virtual void area() = 0;
};
```

Figure 1.15: An example of an abstract class. No meaningful default exists for the `area` method. By declaring it `virtual` we require all children classes to provide the code for this message.

Creating only the most relevant code. This simplifies work by allowing significant code-reuse. One might wish to provide a default implementation and require modified and new implementations to then override this behavior.

What if no meaningful default exists?

Multiple Inheritance

The C++ language includes code inheritance from multiple sources, but it did not do so in the original C++ language definition, and the language committee only included it in response to an apparent user demand. The demand, in part, was due to software developers at the time not understanding what they requested. By introducing the feature, the language created a maintenance apocalypse. Liberated from the constraints of good software design, developers were finally able to create garbled, intermingled classes without a clear inheritance chain!

Heuristic: If you have Multiple Inheritance in your design, assume you made a mistake and prove otherwise.

**Done Wrong** Imagine trying to model an automobile through multiple inheritance. We can say that our Car object inherits from the Cockpit, Drive-train, and Engine. The Cockpit includes a Radio and Steering Wheel, and the Radio includes Electronic Components and Screws.

You can turn a Screw, and you can turn a Steering Wheel, and you can turn the knob on the radio. Because with multiple inheritance the inheriting class gains all the capabilities of its parent, in this example, everything you can do with a screw, you can do with a Car, and this does not make sense.
Certainly, although a car operator CAN turn screws in the vehicle, this is not something typically performed while driving.

1.3.2 Inheritance in Java

In response to the maintenance nightmare multiple-inheritance unleashes on a code-base when used inappropriately, Java took the position that a class may only directly inherit code from one class. Thus, two classes cannot provide a child class with instance variables.

Objects rarely fall into a single abstraction, however, so this restriction places severe limits on the utility of Dynamic Binding and impairs software development. Consequently, Java allows classes to inherit requirements from any number of classes. These special parent classes serve as interfaces. They provide absolutely no code to inherit – instead, they possess method definitions.

Java uses an interface file to define an abstraction without providing any implementation details. Pure virtual classes in C++ serve a similar purpose. Although these other classes provide no code, any implementing class may then, through type inheritance, act like an instance of the parent class.

You may only extend the code of one class in Java, but one may implement any number of additional requirements.

Note: Java 8 introduced the default keyword in interface files, and this now allows developers to provide interface files with code that classes can inherit from. Consequently, Java 8 now supports multiple inheritance.

1.3.3 Inheritance in Python

Python includes support for inheritance in its class design. In fact, every class in Python inherits from the super-class Object. This common class establishes the framework for the class itself.
abstract class IteratorHelper<E> implements Iterator<E> {
    protected int index;
    protected Node<Entry<K,V>> [] nodeArray;

    public IteratorHelper(){
        // setup the same for everyone
    }
    public boolean hasNext(){
        return index < size();
    }

    // this is the one abstract method every child must implement
    public abstract E next();
    public void remove(){
        throw new UnsupportedOperationException();
    }
}

class KeyIteratorHelper<K> extends IteratorHelper<K>{
    public K next(){
        if(!hasNext()) throw new NoSuchElementException();
        return (K) nodeArray[index++].getKey();
    }
}

class ValueIteratorHelper<V> extends IteratorHelper<V>{
    public V next(){
        if(!hasNext()) throw new NoSuchElementException();
        return (V) nodeArray[index++].getValue();
    }
}

Figure 1.16: Useful inheritance in Java for a Key and Value iterator
// A simple Java interface file illustrating default methods
interface ExampleInterface
{
    public int countThings();

    // default method
default void status()
    {
        System.out.println("Having this makes you feel good.");
    }
}

Figure 1.17: Java 8 introduces default methods in interface files.

class DerivedClassName(BaseClassName):
    pass

Figure 1.18: The basic template for declaring a derived class in Python.

<table>
<thead>
<tr>
<th>Method or Attribute</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>self</td>
<td>Similar to this in Java or a pointer to the base class in C++</td>
</tr>
<tr>
<td><strong>init</strong></td>
<td>Constructor</td>
</tr>
<tr>
<td><strong>str</strong></td>
<td>The toString method or &lt;&lt; C++ operator</td>
</tr>
<tr>
<td><strong>dir</strong></td>
<td>Returns a list of attributes supported in the class</td>
</tr>
<tr>
<td><strong>hash</strong></td>
<td>A unique integer. Two equivalent objects must produce the same code.</td>
</tr>
<tr>
<td><strong>doc</strong></td>
<td>Provides the class’ docstring.</td>
</tr>
</tbody>
</table>

Multiple Inheritance in Python

Unlike Java, Python supports Multiple Inheritance. Because all classes in Python inherit from Object, every class with multiple inheritance must include an inheritance diamond pointing to the common Object class. To resolve conflicts in naming, Python attempts to bind methods by searching almost linearly, depth-first through the inheriting classes. Python cannot reliably guarantee it always fulfills the call from left to right due to limits
```
class Person:
    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last

    def Name(self):
        return self.firstname + " " + self.lastname

class Wizard(Person):
    def __init__(self, first, last, mana):
        Person.__init__(self, first, last)
        self.mana = mana

    def GetMana(self):
        return self.Name() + ", " + self.mana

settler = Person("Preston", "Garvey")
quest_giver = Wizard("Morgan", "la Fay", "1000")
```

Figure 1.19: A basic example of class inheritance using Python.

```
class DerivedClassName(BaseClassName, OtherClassName, YetAnotherClassName):
    pass
```

Figure 1.20: An example of multiple-inheritance in Python.

imposed by the software’s design (e.g., calls to super in the methods might interfere with this).

### 1.4 Dynamic Binding or Dynamic Dispatch

Frequently when coding software tasks, one must repetitively perform the same operation on collections of dramatically different objects. The single operation they share may be their only commonality. That is, the only thing the two objects have in common may be the desire to print their description or ability to periodically update themselves in response to a changing clock. Let us create an example of a Non-Player-Character and it’s current mood.

We want to model an entity with a dynamically adjusting description
based upon the character’s current mood. When one looks at the character, one should receive some type of useful reflection of their mood. We will use the general format, "The character looks <MOOD HERE >." Implementing this by storing an integer we evaluate to compute the mood, where negative scores reflect poor moods and positive scores represent good moods could quickly establish a working system, but what if we want more? A software update might want to establish a more advanced mood evaluation strategy. How do quests impact the character’s mood? Its interactions with other NPCs?

Some characters, like simple vendors, might use something simple, but more interesting characters – characters central to the plot – warrant more complicated mood calculations. Should we create two classes, one for primitive characters and one for advanced characters each descending from some common character? Does anything else change between these characters, or is it just the intelligence? What if we created a single NonPlayerCharacter class which contains, (i.e. a HAS-A relationship), an internal Mood object.

Moreover, if we declare Mood as an interface or pure-virtual class, then we could create child classes, which type-inherit Mood but calculate their current mood in dramatically different manners. The simple vendor or NPC extra in a crowd could use a trivial mood calculator – perhaps one even hard-coded inside – and advanced characters could use computationally intensive algorithms. In the code, the developers do not need to know the specifics of the individual Mood child classes. They only need to know that every child of Mood contains a method that returns a string and describes the current mood.

The class may then communicate its message to the mood object through the getMood method. One could even change a NPC’s mood calculation object dynamically on the fly at run-time. Perhaps a spell briefly changes how a NPC feels about the player, one might wish to swap in a new Mood object. Dynamic dispatch makes these type of run-time changes possible. Using a language with only static binding, one could only find similarities at compile time.

One may only determine the type of a polymorphic variable or method at run-time.
Chapter 2

Design Patterns

Using the features provided by an object-oriented language, one may begin creating greater abstractions. For example, rather than saying, ”Use polymorphism to create a method in every object called next, and then use this to retrieve the next item in a sequence,” we say something like, ”make that object Iterable.” Design patterns combine encapsulation, abstraction, and dynamic binding to help better structure code.

Although not a course in object-oriented programming, no discussion of the topic feels complete without a few meaningful examples. Design patterns break down into three major categories.

- Behavioral: Identify common message patterns between objects and create abstractions around those.
- Creational: Instantiating an object in a specific way.
- Structural: Organizing different objects to build new functionality or frameworks

2.1 Behavioral: Iterator

Modern programming languages support the for-each conditional statement. The idea being that instead of iterating from a zero index up to the index of the stopping point, rephrase this concept and write the software to work with the items. For each item in this collection, do the following. The Iterator object provides this mechanism. It manages sequencing through a collection of objects. Typically, one pairs this with the Iterable pattern.
```cpp
int main()
{
    int arr[] = {1, 2, 3, 4};

    for (int curItem : arr)
        cout << curItem << endl;
}
```

Figure 2.1: A C++ program demonstrating iteration using a for-each statement.

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

using namespace std;

int main()
{
    // Iterating over whole array
    vector<int> v = {0, 1, 2, 3, 4, 5};
    for (auto thing : v)
        cout << i << ' ';
}
```

Figure 2.2: A C++ program demonstrating its range-based iteration.

An iterator provides the interface between algorithms and the data structures, or containers, that contain them.

<table>
<thead>
<tr>
<th>Iterable</th>
<th>This object is able to produce an Iterator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator</td>
<td>This object handles an iteration</td>
</tr>
</tbody>
</table>

### 2.2 Behavioral: Comparator and Comparable

How does one need to sort a collection of items from first to last? Given numbers, or letters, this seems trivial. For example, if you were given: 19,
22, 14, 4, and 11 and then told to place them in ascending order, you might return: 4, 11, 14, 19, and 22. If one were placing items into their natural order, this would be valid. What if we wanted them in inverse order? What if they weren’t numbers at all?

The original sequence (19, 22, 14, 4, and 11) – are already in increasing order, for they represent the driver finishing order for the Saturday, April 13th Richmond NASCAR race. Unfortunately, Joey Logano finished second.

2.3 Structural: Adapter

What happens when the vendor who supplied the GPS module your company used for the past five years stops developing your module or changes the command protocol? If your software makes direct calls to the third-party tool directly, then every class that touches it must change. Consequently, swapping out the component becomes expensive. Instead, what if the developers coded their main application using a custom, internal component?

Rather than making calls directly to the soon-to-be-obsolete part, create software that sends messages to an object with an interface under your control. Then, if the vendor changes, one only needs to update the Adapter class. This limits the changes to a single class. One may also use an adapter to convert an existing data structure to a new type.

2.4 Creational: Object Pool

In situations where building an object requires extensive time, what if one simply instantiates a block of them once and then manages their distribution? Rather than directly getting instances of a class, request these from the Object Pool. When one finishes using the object, one simply returns it to the pool. Obviously, objects in the pool need some method to clear or reset themselves, but otherwise the allocation completes only once – when they are loaded into the pool.

Particle systems and objects in game worlds frequently use object pools, for the sparkles radiating from a laser always look the same; Vehicles simply need their damage reset.
Chapter 3

Memory Management and Error Handling

3.1 Garbage Collection

Recall when one allocates memory on the heap in C++, one must manually return these or they remain in place for the life of the program. Should developers forget to release these before program exit, the operating system will almost certainly free up the heap memory allocated for that process. For persistent, real-time systems, however, the process may never terminate, so memory the application fails to return is wasted.

Some programming languages include a mechanism for handling these loose-ends. Rather than require the developer to explicitly instruct the machine to return the memory, implement a sub-system that runs behind the scenes and automatically detects when memory is unused.

Garbage collection seeks to efficiently manage dynamically allocated memory independent of the source unit. Both Java and C# supply garbage collectors. Python automatically allocates and deallocates memory as needed, so it includes garbage collection by default, but it does so slightly differently.

Why would one chose a language without garbage collection? In some real-time applications, the OS simply lacks the resources to support the action. Garbage collected languages certainly help with several targeted errors, but this does not eliminate the error, and it requires additional overhead, for one cannot trust the Java Virtual Machine to eliminate unused references on systems without JVM running.

Garbage collection requires computation, so it must receive CPU time. When it receives this time remains up to the implementation. It might run
{ int *dangling;
{ // declare something on the stack
int onstack = 320;
// point the outside variable at it
dangling = &onstack;
}
// stack frame pops, but dangling points to an address inside!
}

Figure 3.1: A dangling-pointer bug in C++ code blocks.

{ Robot *target = new Robot("IG-88");
  // some code
  target = new Robot("HK-47");
}

Figure 3.2: A memory-leak after pointing target to a new object. IG-88 is still floating around on the heap.

concurrently as a scheduled task, or it might trigger in response to certain actions.

3.1.1 Targeted Errors

Recall that a dangling-pointer develops when an aliased variable falls out of scope. That is, a pointer stores a reference to an invalid memory address. Memory leaks also manifest in C and C++ when one loses a reference to dynamically-allocated heap memory. This might happen when switching pointers. This error tends to manifest subtly as programs grow more complicated and objects begin interacting with one another.

Another error that garbage collection seeks to remedy is the double-free bug\[1\]. It emerges when a program attempts to return to the heap memory already freed. This type of error may appear due to conditional logic and

\[1\] Possibly discovered at a Doubletree
GiantFightingRobot *target = new GiantFightingRobot("Mazinger");
Thing *one;
one = target;

// some code
if( one->isDestroyed() ) free(one);

// cleanup before exit
free(target);

Figure 3.3: A double-free bug in C++. The GiantFightingRobot descends from Thing.

good-intentions. That is, this error appears when one overly-aggressively frees memory. One should make sure cleanup happens only once for any variable in languages without garbage collection.

Garbage collection represents one solution to these errors. Alternatively, consistent programming practices and software engineering will help mitigate them. Readily-available tools in languages without a garbage collection scan either source code or profile a running program and effectively identify their presence in most of these languages, so by no means are they a plague in current programming.

3.1.2 Tracing

The garbage collectors provided by both the Java Virtual Machine and the .NET framework in C# use a strategy of reachability. The collector identifies all heap memory allocations it may reach by tracing a path from all current reachable objects. For this strategy to work, the garbage collector needs some base set of root objects it never collects. These could include the stack, program registers, or other system-wide objects. These would be the essential components of any program.

Then, stepping through each one, it identifies every location accessible through a root or something contained recursively therein. The collector considers anything referenced from a reachable object as reachable.

Java Garbage Collection and Security

As part of efficient memory management, the JVM periodically moves objects in memory. A system memory space might become peppered with
small allocations which make allocating a contiguous block of memory impossible. Recall that the `malloc` and `calloc` operations in C guarantee the data will be together sequentially without gaps. Thus, the largest block of memory the system may provide is defined by the biggest gap between objects in memory.

When Java moves these objects, it leaves behind the dirty memory location. Thus, even if an object takes care to keep private data in non-volatile memory and clear them before destruction, a full read of the system memory would reveal the original data potentially lingering in an unused cell. Several early attacks targeted the garbage collector for this purpose.

In much the same way snooping through a real trash-can can reveal credit-card information, so can scanning all the objects in the garbage collector as well. Moreover, because these copies occur behind the scenes, the developer cannot control when they happen. [https://www.cs.princeton.edu/~appel/papers/memerr.pdf](https://www.cs.princeton.edu/~appel/papers/memerr.pdf)

Thankfully, the JVM garbage collector remains the focus of intense study, and these types of errors appear less frequently.

**Weak References**

Java includes several data structures which support weak references. Unlike the references used in tracing, a weak-reference does not prolong the life of an object like a typical pointer reference would. The collector considers weak-references as useful, but possibly invalid, data. Consequently, the main application can free the variable by removing its reference without having to notify or update the weak reference.

In the Observer design pattern, the publisher must maintain a link to every observer object. If implemented as strong references and stored in a typical aggregate data structure, their presence in the publisher will prevent them from being garbage collected.

A `WeakHashMap` prevents this behavior, for the garbage collector will collect objects if they only appear in the `WeakHashMap`.

### 3.1.3 Reference Counting

Python’s garbage collector uses a reference count for each allocated block of memory. When the reference count reaches zero, the collector knows the area is unused, so it may return the area to the system for use.

An object with a reference cycle will not automatically disappear when garbage collection runs. This memory area, however, may very well be
import gc
num_collected = gc.collect()

print("GC {} items.").format(num_collected))

Figure 3.4: Calling the garbage collector in Python for long-running server programs. This code should collect zero objects.

import gc
def cycle():
    tmp = {}
    tmp[0] = tmp;

    for count in range(320):
        cycle()

num_collected = gc.collect()
print("GC {} items.").format(num_collected))

Figure 3.5: The cycle function creates a tmp dictionary with a self-reference, so it will always have a reference to itself.
otherwise unreachable, for if the only reference to an object is through itself, the client program cannot access the data. Python’s garbage collector will not automatically detect this, so the user must initiate garbage collection manually to recover the memory.

The automatic garbage collector uses reference counting, but manually triggering garbage collection will collect unreferenced objects by the main program. Thus, Python mixes reference counting which it performs automatically with manually-activated tracing.

General heuristics about when to trigger a manual garbage collection:

- Python stops automatic garbage collection when low on memory
- Consider running it after the program finishes loading, booting, and initializing.
- Prior to initiating time critical sections of code where background collection could impact performance.
- After blocks of code that free large blocks of variables (perhaps when reading in data)

### 3.1.4 Destructors and Constructors

In Java, the JVM calls the `finalize()` method attached to every object. For almost all applications, the default method provided in the Object class remains sufficient, but for objects with complicated resource allocations, users may wish to override this behavior. One might wish to disconnect from an external server, shutdown resources, or notify an Observer. To do so, one need only override the method in the class definition.

The C++ language also provides a default destructor, and for many applications the default suffices. If an object allocates heap memory, however, one must explicitly remove the reference and free the memory. Failing to do so might lead to a memory leak. Destructors are noted by a tilde appearing in front of the class name.

### 3.2 Exception Handling

*Exception:* a person or thing that is excluded from a general statement or does not follow a rule.

- Dictionary.com
```cpp
#include <cstdlib>
using namespace std;

class Label {
  private:
    int serno;
  public:
    Label(int number) { serno = number; }
};

class Orange {
  private:
    Label *farmer;
  public:
    Orange() { farmer = new Label(1); }
    ~Orange() { free(farmer); }
};
```

Figure 3.6: Example of a class with a destructor and constructor in C++. The allocation of Heap memory requires a `free` to avoid memory leaks.

Typically, failing to catch an exception results in a catastrophic failure. At a minimum, it clunks the program flow and slows down performance. Exceptions handle situations outside standard program flow. They might cover errors like the inability to get more memory, open a file, or reflect invalid input. They also cover invalid parameters provided to objects.

Should a constructor throw an exception? What are the implications?

### 3.2.1 Java: try-catch

https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/

Java requires the try-catch mechanism throw a Object of type `Throwable`. One cannot simply throw numbers or characters as a result. Java allows one to catch the general `Exception` object which emulates the C++ `catch(...)` behavior. Java also allows one to include the `throws` keyword to indicate an object produces a particular error under knowable state.

**Checked Exceptions** are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception
using the throws keyword. In general, these are errors one might easily recover from with some other valid input or another try.

The price of checked exceptions is an Open/Closed Principle violation. If you throw a checked exception from a method in your code and the catch is three levels above, you must declare that exception in the signature of each method between you and the catch. This means that a change at a low level of the software can force signature changes on many higher levels. –Robert C. Martin, Clean Code, page 107

The open/closed principle states that software modules should be open to extension and closed to modification.

When a single change to a program results in a cascade of changes to dependent modules, that program exhibits the undesirable attributes that we have come to associate with "bad" design. [...] The open-closed principle attacks this in a very straightforward way. It says that you should design modules that never change. –Robert C. Martin, The Open-Closed Principle, 1996

Use checked exceptions for conditions from which the caller can reasonably be expected to recover. [...] By confronting the API user with a checked exception, the API designer presents a mandate to recover from the condition. –Joshua Bloch, Effective Java, page 244

**Unchecked Exceptions**  In Java exceptions under Error and RuntimeException classes are unchecked exceptions, everything else under Throwable is checked.

Java includes the `finally` block which always executes after the try-catch block completes. One might use this to close external resources. No matter what happens in the try-catch behavior, it will always call finally last.

A well-documented list of the unchecked exceptions that a method can throw effectively describes the preconditions for its successful execution. –Joshua Bloch, Effective Java, page 252
class Main {
    public static void main(String[] args) {
        FileReader file = new FileReader("./out.txt");
        BufferedReader fileInput = new BufferedReader(file);

        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        fileInput.close();
    }
}

Figure 3.7: This Java code will not compile because it does not catch FileNotFoundException.

3.2.2 C++: catchall

All C++ compilers support the catchall format, but some consider it dated and bad form like using raw pointers. Instead, the C++11 standard library includes a new exception type to work with.

```cpp
try {
    // code here
} catch (int param) { cout << "int exception"; }
    catch (char param) { cout << "char exception"; }
    catch (...) { cout << "default exception"; }
```

Instead, C++11 introduces the concept of an `std::current_exception` object. This object captures the current exception object and creates an `std::exception_ptr` strongly holding a copy to the exception object. Creating this object does not, itself, cause a new exception, so one must typically:

1. Catch the current exception in a catchall
2. Build a `current_exception` object from the try-catch context
3. Rethrow the exception as a `current_exception`
4. Catch the std::exception_ptr thrown and continue error analysis

33
```cpp
int main()
{
    int x = 320;
    char *ptr;

    try {
        if (x < 0)
        {
            throw x;
        }
        if (ptr == NULL)
        {
            throw "Dead stars still burn."
        }
    }
    catch (int x)
    {
        std::cout << "Suboptimal: " << x << std::endl;
    }
    catch (...) {
        std::cout << "Exception occurred: exiting " << std::endl;
        exit(0);
    }
}
```

Figure 3.8: Example of the try-catch mechanic in C++. Note the use of a catch-all block.
```cpp
#include <iostream>
#include <string>
#include <exception>
#include <stdexcept>

void handle_eptr(std::exception_ptr eptr)
{
    try {
        if (eptr) {
            // Rethrow the exception using an eptr object so we process it here
            std::rethrow_exception(eptr);
        }
    } catch (const std::exception& e) {
        std::cout << "Caught " << e.what() << "\n";
    }
}

int main()
{
    std::exception_ptr eptr;
    try {
        // ...
    } catch(...) {
        eptr = std::current_exception();
    }
    handle_eptr(eptr);
}
```

Figure 3.9: The updated exception handling procedure in C++11 using the standard library. This method provides an exception object to work with.
Chapter 4

Generic Programming

Generic programming facilitates designing algorithms and data structures in terms of their operation and use and not the specifics of what they hold. It attempts to abstract out the key concepts by deferring many of the implementation specifics until the point of instantiation. A queue for coffee, for example, provides the same operations as a queue for cattle on a dairy. The items in line, be they coffee or people, sequentially exit in the order they arrived. Thus, one can program all the operations on the queue independent of what goes inside.

Languages provide differing levels of support for Generic programming, and they tend to refer to it in slightly different terms. Be it generics in Java and Python or templates in C++, generic programming ultimately seeks to reduce the amount of code developers need to write and maintain.

The Python programming does not require any formal syntax with generics. Instead, it uses a strategy called duck typing. The idea follows the saying, "if it looks like a duck and quacks like a duck, it’s a duck." Instead of trying to verify the operation is permitted at compile time, Python assumes it is and acts accordingly.

4.1 Java Generics

Initial versions of the Java language lacked type-safe, generic support. Rather than include extra syntax, like C++ does when it identifies a generic function or class, it simply relied upon the inheritance tree to achieve this functionality. Recall that technical authors frequently employ the use of 'recall' to make their sentences seem more formal. Also, all objects in java descend from the Object super-class.
Rather than creating a linked-list for Droid objects, the early Java programmer envisioned it as a linked-list of type Object which the developer understood to be Droid. That is, the compiler possessed no way to verify or enforce that only Droid objects entered the list. Because everything descends from Object, a linked list in this manner might contain a mixed bag of Droid, Duck, and Exception.

With the Java 1.5 update, the language began including additional syntax for generic support. This syntax permits compile-time type checking to verify the software uses an object in the intended fashion. The extra notation helped catch errors as well as supplying additional documentation to the reader about a class’ intended use.

Java’s updated generic implementation, however, serves only as syntactic sugar over the original method. Behind the syntax, the Java compiler converts the generic references indicated in the brackets into casts it then applies to the Object. That is, verifies the objects inside the class match the appropriate type at compile-time, but it then converts the code into a new object which uses casts inside this single object.

The process of type-erasure removes the generic types identified in the generic object and converts it into a single, non-generic object capable of containing any style Object with casts to each of the generic types used. Consequently, an ArrayList<Integer> has no way to tell itself apart from a ArrayList<Float>, for they both become the ArrayList with the appropriate casts inside.

This feature serves as the root of data structure developers familiar warning: Unchecked cast from Object to Integer. Java warns the developer that the item referenced may, very well, not be the expected type.

Because there is only one copy of a static class at runtime, all type-specifications of a generic object use the same static variables. A static counter in MyCounter<Integer> references the exact same variable as one in MyCounter<Long>.

One may not call a generic constructor because, due to type-erasure, at run-time the program possesses no way to know what to instantiate, for it has broken free from the chains of typing. It follows that one may not perform a new T() operation if the program cannot distinguish T between a Droid or a ArrayList.
4.2 Templates in C++

Templates support generic programming and design abstraction. They represent a general method for creating source code to accomplish a task. In C++, the process of generating the source code from the template is called template instantiation. The resulting template code is a template specialization. Multiple specializations may result from a single template.

One cannot explicitly state a template’s requirements in code, and the user must glean this from the design. It behaves somewhat like a macro substitution. One can easily create a syntax error by calling a macro with invalid input. Likewise, one might introduce an error by instantiating a template in C++ with an object that doesn’t support the required types or includes nonsensical values.

Templates in C++ hide the intended abstraction from the compiler whereas those in Java help enforce it.

**Class Templates** provide a way to specify that some of the class’ members come from templates.

**Function Templates** identify that a function, independent of a class, accepts generic parameters.

```cpp
List v = new ArrayList();
v.add("test");
Integer i = (Integer)v.get(0);
```

Figure 4.1: An example of generic syntax motivation in Java. Generating a compile time error here catches an error before run-time, but this framework lacks the capability. Provided by Wikipedia.

```cpp
List<String> v = new ArrayList<String>();
v.add("test");
Integer i = v.get(0);
```

Figure 4.2: The updated generic syntax catches these types of type error at compile-time. Provided by Wikipedia.
template <class Receiver>
class SimpleCommand : public Command {
    Action _act;
    Receiver *_rec;

public:
    typedef void (Receiver::* Action)();

    SimpleCommand(Receiver* r, Action a) :
        _rec(r), _act(a) {}

    virtual void Execute(){_act->Execute();}
}

m1a1 = Tank();
SimpleCommand<Tank> tnkOff = SimpleCommand(m1a1, &Tank::powerOff())
tnkOff.Execute();

Figure 4.3: An example of a C++ template for the Command design pattern in Design Patterns.

Non-type Parameters extend beyond simply specifying that a type in
the code is generic, one may even supply generic values. This functionality
extends C++ templates into a new level of potential complexity.

template <class T, int N>
void DynamicArray<T,N>::set(int x, T value) {
    storage[x]=value;
}

4.2.1 Metaprogramming with Templates

http://people.cs.uchicago.edu/~jacobm/pubs/templates.html

template <int n>
class Fact {
    public:
        static const int val = Fact<n-1>::val * n;
};

class Fact<0> { public: static const int val = 1; };
Chapter 5

Functional Programming

Functional programming gravitates toward several central themes:

- Immutability of Data: Errors emerge when the data an arbitrary algorithm uses changes in another thread.
- No Side Effects: The order of operations should not impact the results.
- Recursion: The use of function calls to perform iteration instead of loops.

Each of these address problems that frequently manifest in concurrent programming situations like those on the Internet or in multi-processor systems. Most modern programming languages support the idea of a first-class function – if not directly, then through a library addition. Java, for example, provides very little built-in support for functional programming, for one cannot use functions the same way as we do in C++ and Python. That is, one cannot set the value of a variable to a function.

In computer science, a programming language is said to have first-class functions if it treats functions as first-class citizens. This means the language supports passing functions as arguments to other functions, returning them as the values from other functions, and assigning them to variables or storing them in data structures.

–Wikipedia

A higher-order function accepts a function as one of its input parameters or returns a function as a result. Functions in Python’s map, filter, reduce algorithms frequently accept a function as the input parameter.
```python
def recsum(x):
    if (x==1):
        return x
    else:
        return x + recsum(x-1)
```

Figure 5.1: A general recursive sum function

```python
def tailrecsum(x, running_total=0):
    if (x==0):
        return running_total
    else:
        return tailrecsum(x-1, running_total+x)
```

Figure 5.2: A tail recursive sum function

*Pure functions* produce output exclusively based upon its input parameters. That is, these functions use no stored state or external data to complete their operation. The mathematical operations provide an excellent illustration of a pure-function ideal. Addition remains independent of the temperature, barometric pressure, or previous winner.

*Anonymous functions* are not associated with a function label. These frequently take the form of *lambda* expressions. Alanzo Church proposed a Turing-complete, formal system for expressing computation based on functions.

**Tail Call** Recursion, as previously discussed, requires the creation of an activation record, and many recursive algorithms possess the potential for unbounded stack growth. That said, when the recursive call is the last item the function performs before exiting, the compiler may optimize the entire recursive function into an iterative while loop. This style of optimization remains essential in a functional compiler.

## 5.1 Python

As demonstrated previously with the *decorator* function, Python supports the functional style of programming. One may obviously create closures with the language, but it also supports *lambdas* and anonymous functions.
def read_lines(filename):
    with open(filename, 'r') as the_file:
        return [line for line in the_file]

ex = read_lines("sample.txt")

Figure 5.3: A functional method for opening files in Python

add_it = lambda a, b: a + b
add_it(1,1)
add_it(10,-20)

Figure 5.4: A basic lambda definition in Python

5.1.1 Map, Filter, and Reduce

Although Python supports functional programming and includes the map, filter, and reduce algorithms, list comprehensions better fit the coding style.

5.2 Functional in Java

Until Java version 1.8, the language provided very little functional programming support. Designed as an object-oriented language, many of the functional concepts appear antithetical to Java’s syntax and design, for one cannot display text on the screen in Java without a class. That said, some functional concepts fit well with any programming style. Lambda expressions and anonymous functions are convenient syntax for complicated operations, so they found a place in the language itself.

Functional Interfaces in Java are interfaces which include a single method definition. Then, lambda expressions may subsequently define the method to use as needed.

\[ \text{new_read = list(map(lambda a : a.upper(), read_lines("sample.txt")))} \]

Figure 5.5: A map lambda expression in Python
def my_filter(the_test, data):
    return Map(
        datum for datum
        in data if the_test(datum) is True)

Figure 5.6: The filter algorithm in Python

import functools
import operator

the_data = [x ** 2 for x in range(10)]
print(functools.reduce(operator.add, the_data))
print(functools.reduce(operator.mul, the_data))

Figure 5.7: Reducing in Python.

def test_func(datum):
    return (datum * 2) + 1

def my_comp(num):
    return [test_func(x) for x in range(num)]

def my_mapping(num):
    return list(map(test_func, range(num)))

Figure 5.8: List comprehension and a map applied to the same function.
interface FuncInterface
{
    void singleFunction(int x);
}

class Test
{
    public static void main(String args[])
    {
        FuncInterface fobj = (int x)->System.out.println(2*x);

        fobj.singleFunction(160);
    }
}

Figure 5.9: A basic Java Functional Interface.

5.3 Functional in C++11

Although not a functional programming language, C++11 and later include support for some popular functional features. With each subsequent revision, the language evolves to expand its functional capabilities. The functional header file includes several useful programming abstractions so one need not develop these basic ideas from scratch.

5.3.1 Lambdas, Anonymous Functions, and Closures

A closure is an instance of a function, a value, whose non-local variables have been bound either to values or to storage locations.

–Wikipedia

Sometimes, when working with higher-order functions, one needs a small, special-purpose function. When infrequently used, creating a named function might introduce more coding and complexity than necessary. In these situations, one might use an anonymous function. Anonymous functions in C++ take the general form: [capture](parameters) -> return_type

```
[](int x, int y) -> int { return x + y; }
```
Figure 5.10: This computes the total of all elements in the list. The variable total is stored as a part of the lambda function’s closure. Since it is a reference to the stack variable total, it can change its value.

<table>
<thead>
<tr>
<th>Capture Specifier</th>
<th>Types of Capture</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>Nothing used in outer scope.</td>
</tr>
<tr>
<td>[identifier]</td>
<td>Copy identifier by value.</td>
</tr>
<tr>
<td>[=identifier]</td>
<td>Copy identifier by value.</td>
</tr>
<tr>
<td>&amp;[identifier]</td>
<td>Copy identifier by reference.</td>
</tr>
<tr>
<td>&amp;[&amp;first,=second]</td>
<td>Copy first by reference, second by value.</td>
</tr>
</tbody>
</table>

Figure 5.11: Lambda capture specifiers if the object should be copied by value or reference.

One may capture variables in the greater program scope by including them in the square brackets.
Bibliography

