# Lecture Notes,

# CS345, Winter 2018

# Frameworks and Reuse

## *Reuse?*

Reuse is based on the idea that we can combine "pieces" of software into larger pieces. For example, the function call is a way to combine software. The function call allows us to effectively, insert a block of code, the code for the function, into our program at the point of the function call. The arguments in the call are "substituted" for the parameters in the function. (Of course, it's not really that simple.)

Object-oriented programming gives us additional ways to combine software:

- Inheritance: When you define a subclass you combine the software in the subclass with the software in the superclass.
- Composition: Composition allows you to combine your software with the bundle of data and operations in the composed class.

In this way we can think of inheritance and composition as mechanisms that allow us to reuse software.

There are different kinds of reuse:

- You write a function that is used in many places in your program
- Software libraries and frameworks
  - Bodies (libraries) of code that you incorporate into your program
  - These can be narrow—a set of utilities that are used just by you or your company—or wide —the java libraries
- External programs that provide services such as compilers, databases, etc.

The ideal is that any software you write is potentially reusable.

The reality is that reusable software must meet higher standards than non-reusable components:

- The functionality provided must work for a large community of users rather than a single application.
- The interface(s) must be cleaner and easier to use than those for a single application.
- The interface(s) must be documented for use by other developers.
- The implementation has to account for unanticipated usage patterns.
- The implementation should be carefully tested for conformance to its interface specifications.

For these reasons you shouldn't try to develop reusable components without a reasonable expectation that the benefit of the reuse will exceed the additional cost to develop the software.

A general rule of thumb for developing reusable components: You need three good examples or use-cases before you know enough to design and build a reusable component. Until you have three good examples you don't know what parts of the component change from use to use and which parts are common to all uses.

## *Object-Oriented Software and Frameworks*

A framework is a body of software (a library) that can be extended to create a specific application.

Most major frameworks are based on object-oriented programming languages.

Consider a library of functions:

- You write code that uses library functions.
- You have to write the overall controlling logic and invoke library functions as needed.
    - A library of mathematical functions works this way

Consider an object-oriented framework:

- You can extend functionality by inheritance and composition.
    - You can inherit from a class in the framework. This allows you to extend the framework by supplying methods in a subclass that are invoked by methods in the superclass.
    - You can have a class in the framework that composes (HAS-A) an instance of a class that you provide. Typically, your class implements an interface from the framework or extends an abstract class provided by the framework.
    - You can write a class that composes (HAS-A) functionality from one or more instances of framework classes.
- The framework can use *inversion-of-control*. The framework can provide the controlling logic and invoke application specific functionality you provide. This is frequently done by inheriting from a framework class that includes this logic.
- You can still (and will need to) invoke library functions that are part of the framework.

When you use a framework you have to obey the rules of the framework.

Consider a framework for a Graphical User Interface program:

- The framework imposes a structure for the overall application. The framework specifies how you plug your application specific functionality into the framework.
    - In JavaFX you provide a subclass of the Application class that has a start() method that builds the GUI and starts the application.
- The framework specifies the structure of the interface, for example, how the menus work.

## *APIs*

An Application Programming Interface (API). An API is an interface that one or more software components provide that allows other software components—the application(s)—to access a set of functionality. An API is an <u>interface</u>. The API does not specify its implementation. Some examples of potential APIs:

- Networking
- Database access

- Email manipulation

The API can also be thought of as a contract between the developer of the API and the user of the API:

- The developer provides the software with the needed behavior.
- The user follows the rules for using the software and gets the benefit of the behavior provided by the framework without having to implement that behavior themselves.

The contract behavior includes both the software interface and any additional relationships and constraints specified by the developer and expected of the user. This additional behavior needs to be documentated as part of the API since it cannot be captured by the formal specifications.

Some examples of additional behavior (not captured by the methods and interfaces of the API):

- In Java, all classes inherit from the class `Object`. One of the behaviors expected of all Objects is: if `obj1.equals(obj2)` is true, then `obj1.hashCode() == obj2.hashCode()`
- In a Collection (see Collections Framework, below) the `isEmpty` method returns true if the collection is empty. There is a `size` method that returns the number of elements in the collection. Part of the additional behavior is that `c.isEmpty()` is true, if and only if `c.size() > 0`. (Note: Why an `isEmpty` method? Because it may be significantly faster to decide whether a collection is empty that to decide the exact number of objects in the collection. Think about linked lists.)
- Sorts can be stable or unstable. (A stable sort is one that preserves the order of equal elements. Merge sort is stable. Quick Sort is unstable.) The interface to a sort function does not distinguish between stable and unstable sorts. However, this can be an important part of the behavior from the perspective of the client of the sort.

## *Frameworks, Libraries, and APIs*

An ***API*** is an Application Programming Interface. An API is the interface to a software component that provides some specific functionality.

A software *library* is a collection of software that can be used by an application to assist in providing the functionality implemented by that application.

A software *framework* is a collection of software that can be extended in order to create an application.

A little thought will convince the reader that there a no bright line differences between these three concepts. An API can be the interface to a library or framework. A framework is typically a library of components. A library can be thought of as a framework and as providing an API.

## *A Reference on API design in Java*

Joshua Bloch[1] has a talk entitled “How To Design A Good API And Why It Matters”. This talk gives a lot of practical advice about API design. Much of the talk is about general concepts and approaches to API design. Some of the talk is Java specific, in particular, describing some dos and don'ts in Java API design. Here is a video of the author giving a Google Tech Talk of this presentation.

http://www.youtube.com/watch?v=aAb7hSCtvGw

The presentation can also be found in a number of different forms online by searching for the title.

1 Worked for both Sun and Google. Author of *Effective Java*.

## *Object-Oriented Framework Mechanisms*

From an object-oriented perspective, there are a number of mechanisms available for providing a framework:

- Interfaces
  - Interfaces are a way that the framework can specify both (1) expectations of client code and (2) the behavior provided by the framework.
  - Code in the framework can work with an object that implements the given interface. Using interfaces means that user code does not have to inherit from framework code.
  - Provided classes can implement interfaces. This allows both (1) the framework to hide implementation details and (2) user classes and framework classes to be intermingled in a single application.
- Abstract classes
  - An abstract class provides a skeleton for an implementation. Both framework and user classes can inherit from an abstract class and extend the functionality of that class.
- Concrete classes
  - Concrete classes can provide the functionality of the API.
  - Concrete classes can provide implementation of interfaces and concrete versions of abstract classes for standard use-cases.
- Functions
  - APIs can provide standard functions that work in conjunction with the interfaces and classes that are provided by the API. In Java, a function typically appears as a static method.

## *An Example: Java Collections Framework*

References:

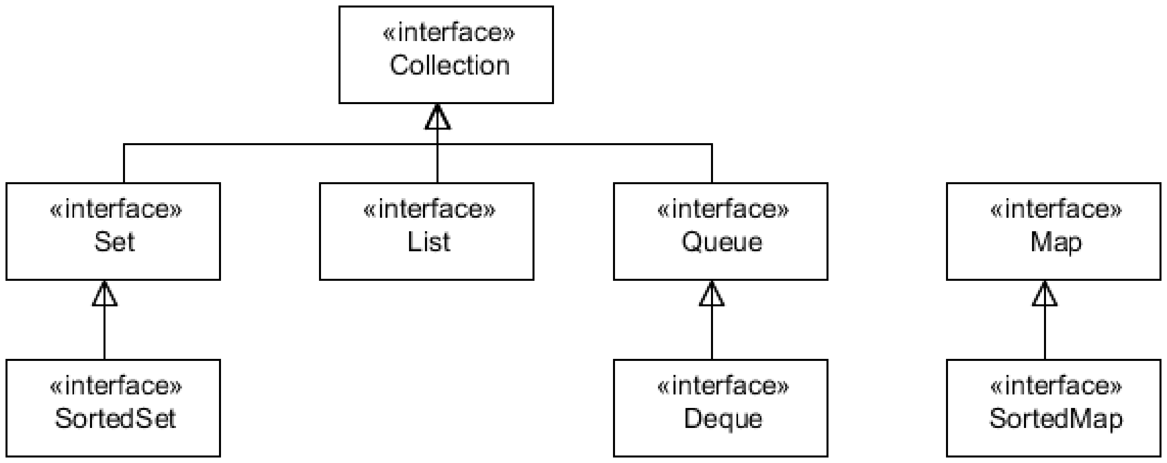
*Java SE Documentation: The Collections Framework*, at http://docs.oracle.com/javase/8/docs/technotes/guides/collections/index.html .

*The Java Tutorials: Trail: Collections*, at http://docs.oracle.com/javase/tutorial/collections/index.html .

Some important lessons in the tutorial trail:

- Introduction to Collections
  - Describes how the concepts of interfaces, implementations, and algorithms (interfaces, classes, and functions) are used in the framework.
  - Why collections?
    - Reuse – you don't need to do your own implementation every time you need a collection, which is all the time.
    - Quality – you don't have to fix bugs in your own implementation
    - Commonality – Have a common collections framework makes it easier for different libraries to exchange information. You don't have to transform data from one libraries' custom collection to the other libraries' custom collection.

- Interfaces
    - Discusses the key interfaces that are part of the framework. Here's a UML diagram showing the key interfaces:



    - One of the key decisions in the collections framework was to not have separate interfaces for synchronized, mutable/immutable, fixed-size/variable-size, and append-only collections. If an operation is not supported for a given collection, that operation is still part of the interface. However, invoking that operation will result in an `UnsupportedOperationException`. Operations that don't have to be supported as part of an interface are documented as "optional" operations.

      An interesting question is whether this constitutes a violation of the Liskov Substitution Principle. In one sense it does not: The expected behavior of mutation operations on an immutable collection is to throw the specified exception. On the other hand: Mutable and immutable collections (appear to) have different behavior in certain cases. Passing an immutable collection to a method expecting a mutable collection is not checked by the compiler and will cause a runtime exception if the method attempts to modify the collection.
- The Collection Interface
    - This lesson describes the `Iterable` and `Iterator` interfaces. Iterators are objects that control traversal (iteration) through a collection. The iterator interface is part of the Java language and is used by the compiler to implement the for-each construct:

      ```
      for (E e : c) { … }
      ```

      In fact this construct is equivalent to:

      ```
      Iterator<E> iter = c.iterator(); // c implements Iterable<E>
      while (iter.hasNext()) {
          E e = iter.next();
          … }
      ```

      Here's an alternative way to code exactly the same thing:

      ```
      for (Iterator<E> iter = c.iterator(); iter.hasNext(); ) {
          E e = iter.next();
      ```

```
        … }
```

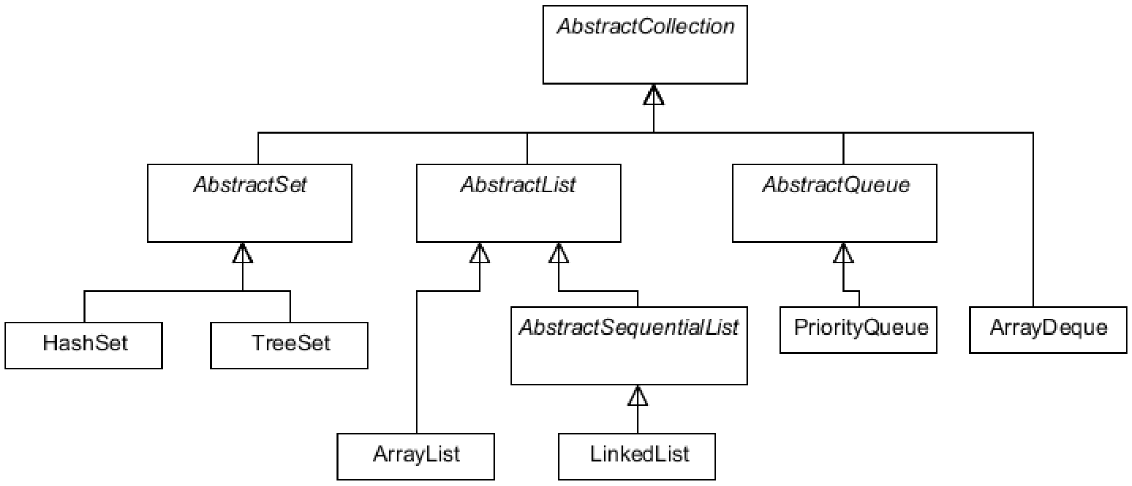
- Note that `Iterator` is an interface. That means that you can create your own iterators with additional operations. For example, Java Lists have a `listIterator` method that returns a `ListIterator` type. Instances of this class allow both forward and reverse iteration and adding, replacing, and removing elements of the list relative to the current location of the iteration in the list.

- Implementations (concrete classes)
  - Discusses the concrete classes that are provided as implementation of the various interfaces that are provided in the framework.
  - Here's a brief synopsis of the major implementation for the major interfaces:

| **Interface** | **Implementations** |
|---|---|
| `Set` | `HashSet, TreeSet, LinkedHashSet` |
| `Map` | `HashMap, TreeMap, LinkedHashMap` |
| `List` | `ArrayList, LinkedList` |
| `Deque` | `ArrayDeque, LinkedList` |
| `Queue` | `ArrayDeque, LinkedList, PriorityQueue` |

- Wrappers
  - The framework provides "wrappers" for a number of different uses. These wrappers "wrap" a collection to change it's behavior in some way. Two useful types of wrappers:
    - synchronization – makes the collection safe to use in a multi-threaded environment. The static functions `Collections.synchornized`*`interface`*, where *interface* is List, Map, etc., takes an ordinary object of type *interface* and returns one which is correctly synchronized.
    - mutability – makes a collection that cannot be changed. All the update methods are disabled. The static functions `Collections.unmodifiable`*`interface`* do this.
- Algorithms
  - Discusses algorithms that have been provided as part of the framework. (To quote: "With luck you will never have to write your own sort routine again!")
  - Some standard algoriths:
    - sort a list
    - shuffle a list
    - binarySearch a sorted list
    - compute min and max of a collection
- Custom Implementations (Abstract classes)
  - The custom implementation include abstract classes that are provided to make it easier to implement your own collections.

- Here is a UML diagram of some of the standard abstract and concrete classes that implement the standard interfaces:



  - `ArrayList` extends `AbstractList`. Extending `AbstractList` class requires implementation of
    - `get(int)` and `size()` methods for any list
    - `set(int)` for modifiable lists
    - `add(int, E)` and `remove(int)` for lists that can change length
  - `LinkedList` extends `AbstractSequentialList`. Extending `AbstractSequentialList` requires implementation of the `listIterator()` and `size()` methods. The `listIterator()` method returns a `ListIterator` that can be used to traverse and modify the list.