INTERFACES, GENERICS, AND THE JAVA COLLECTIONS FRAMEWORK

CSC 207 SOFTWARE DESIGN

Computer Science
UNIVERSITY OF TORONTO

Course Overview

Tools (Weeks 1-4)

- Java
- Version Control
- Software Tools

Design (Weeks 5-8)

- Clean Architecture
- SOLID
- Design Patterns

Professional Topics (Weeks 9-12)

- Ethics
- Internships
- GenAI

- Last week we talked about OOP in Java, including:
  - constructors, types, casting, inheritance and other features of Java
- This week, we will talk about
  - Interfaces
  - Abstract Classes
  - the Java Collections Framework (JCF)

Questions to be answered this week...

What is an Interface in Java?

How can we decide whether we should use an interface or an abstract class?

What is a generic type?

What is the Java Collections Framework?

LEARNING OUTCOMES

- Understand what an Interface is in Java
- Understand what Generics are in Java
- Explore the Java Collections Framework (JCF) to see Generics, Interfaces, and Abstract classes in action.
- Understand how ADTs are incorporated into Java by the JCF.

INTERFACES

RETHINKING INHERITANCE

Consider the following:

- Classes in Java have one parent.
- What if an abstract class only contains abstract methods but no instance variables?
  -
- What if we want our new class to inherit from both a concrete class and this abstract class?
  -

RETHINKING INHERITANCE

Consider the following:

- Classes in Java have one parent.
- What if an abstract class only contains abstract methods but no instance variables?
  - Seems wasteful (not making use of the instance variables)
- What if we want our new class to inherit from both a concrete class and this abstract class?
  - We are forced to pick which parent we care about more inheriting from!

ENTER INTERFACES!

- An Interface is like an abstract class, but:

  It is separate from the inheritance hierarchy, so a class can implement any number of interfaces.

  It can't have instance variables (so no state information).

  Its methods must be either abstract or provide a default implementation. Syntactically, we don't have to use the abstract keyword since methods are assumed to be abstract — the default keyword is used if the interface does provide a default implementation.

ENTER INTERFACES!

- Importantly, **we can create variables whose reference type is that of an Interface**!

  This allows for us to still benefit from polymorphism — we can create variables which can refer to any object which implements the provided interface!

  Instanceof can be used to check if an object implements an interface, just like it checks if an object is part of an inheritance hierarchy.

- Example:

  ```
  MyInterface mi = new MyClass();  // where MyClass implements MyInterface

  mi instanceof MyInterface;  // evaluates to True!
  ```

INTERFACES IN PRACTICE

We'll see lots of examples soon in the Java Collections Framework, but first it will be useful to learn about generics.

In IntelliJ, if you have a class and want to create an interface corresponding to some of its methods, the IDE can automatically help you do this:

- https://www.jetbrains.com/help/idea/extract-interface.html

GENERICS

GENERICS (FANCY TYPE PARAMETERS)

- "`class Foo<T>`" introduces a class with a type parameter `T`.
- "`<T extends Bar>`" introduces a type parameter that is required to be a descendant of the class `Bar` — with `Bar` itself a possibility.
    - In a type parameter, "`extends`" is also used to mean "`implements`".
- "`<? extends Bar>`" is a type parameter that can be any class that extends `Bar`. We'll never refer to this type later, so we don't give it a name.
- "`<? super Bar>`" is a parameter that can be any ancestor of `Bar`.

https://docs.oracle.com/javase/tutorial/java/generics/types.html

AN INTERFACE WITH GENERICS

```
public interface Comparable<T> {
    /**
     * Compare this object with o for order.
     * Return a negative integer, zero, or a
     * positive integer as this object is less
     * than, equal to, or greater than o.
     */
    int compareTo(T o);  // No body at all.
}

public class Student implements Comparable<Student> {
    . . .
    public int compareTo(Student other) {
        // Here we need to provide a body for the method.
    }
}
```

INTERFACE LEFT, CLASS RIGHT

- `List<String> ls = new List<>();` `// Fails`
- `List<String> ls = new ArrayList<>();`
- We can choose a different implementation of `List` at any point by editing a single line of code.

GENERICS: NAMING CONVENTIONS

- The Java Language Specification recommends these conventions for the names of type variables:
  - very short, preferably a single character
  - but evocative
  - all uppercase to distinguish them from class and interface names
- Specific suggestions:

Maps:	K, V
Exceptions:	X
Nothing particular:	T (or S, T, U or T1, T2, T3 for several)

THE JAVA COLLECTIONS FRAMEWORK

Now that we know about the basics of Interfaces and Generics, we are ready to look at examples of them in practice!

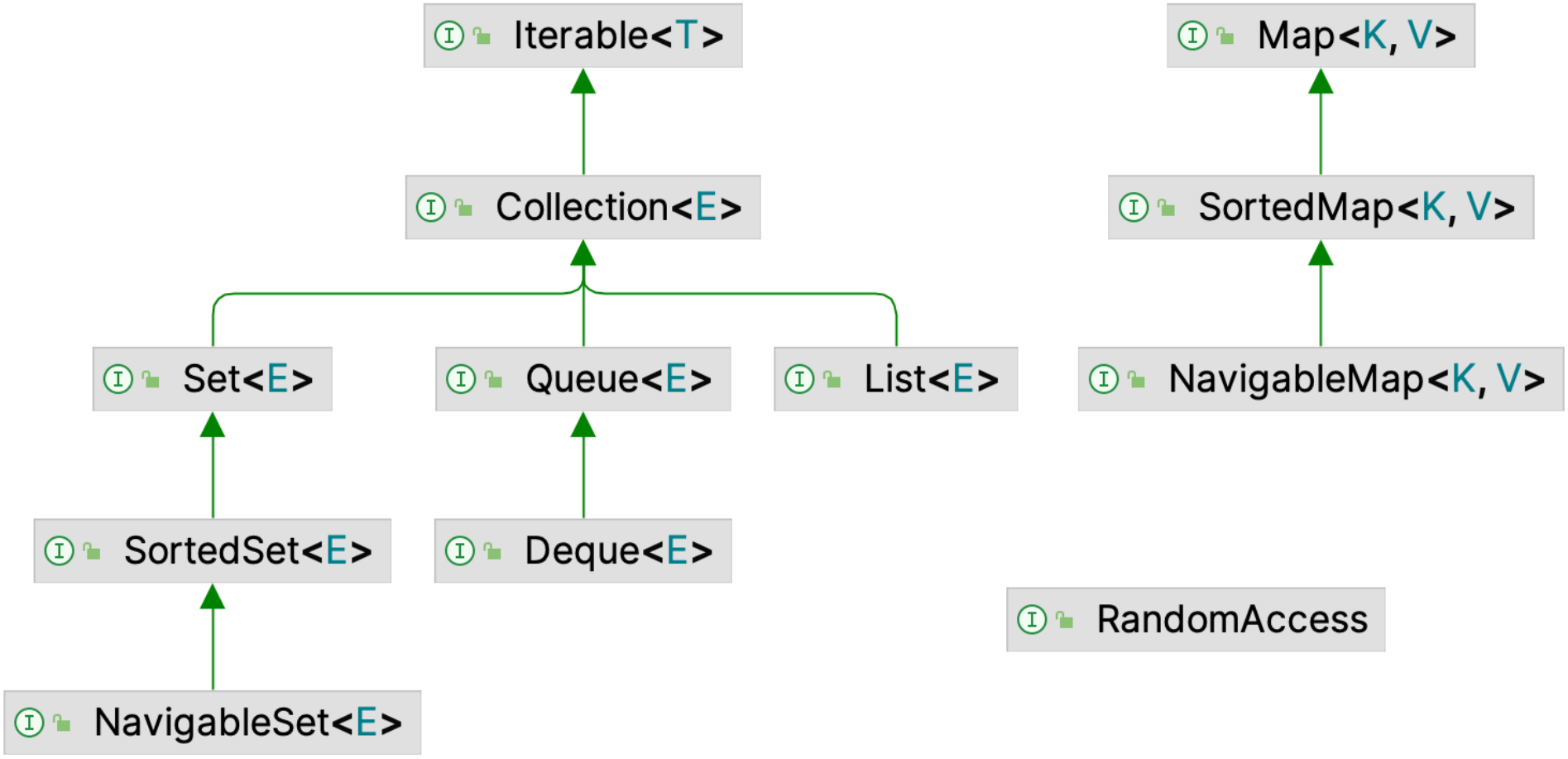
THE JAVA COLLECTIONS FRAMEWORK

Oracle's Collections Framework Overview:

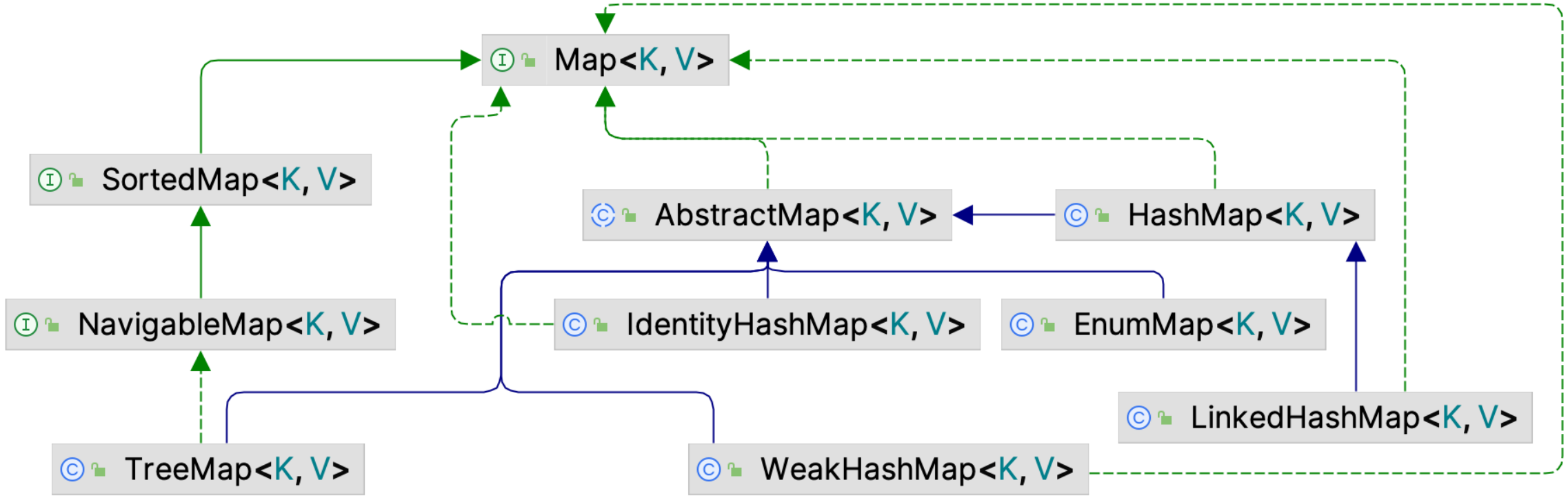
- "A *collection* is an object that represents a group of objects"
- "A *collections framework* is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details."

Examples of classes and interfaces in the framework include List, ArrayList, LinkedList, Set, Map, and HashMap.

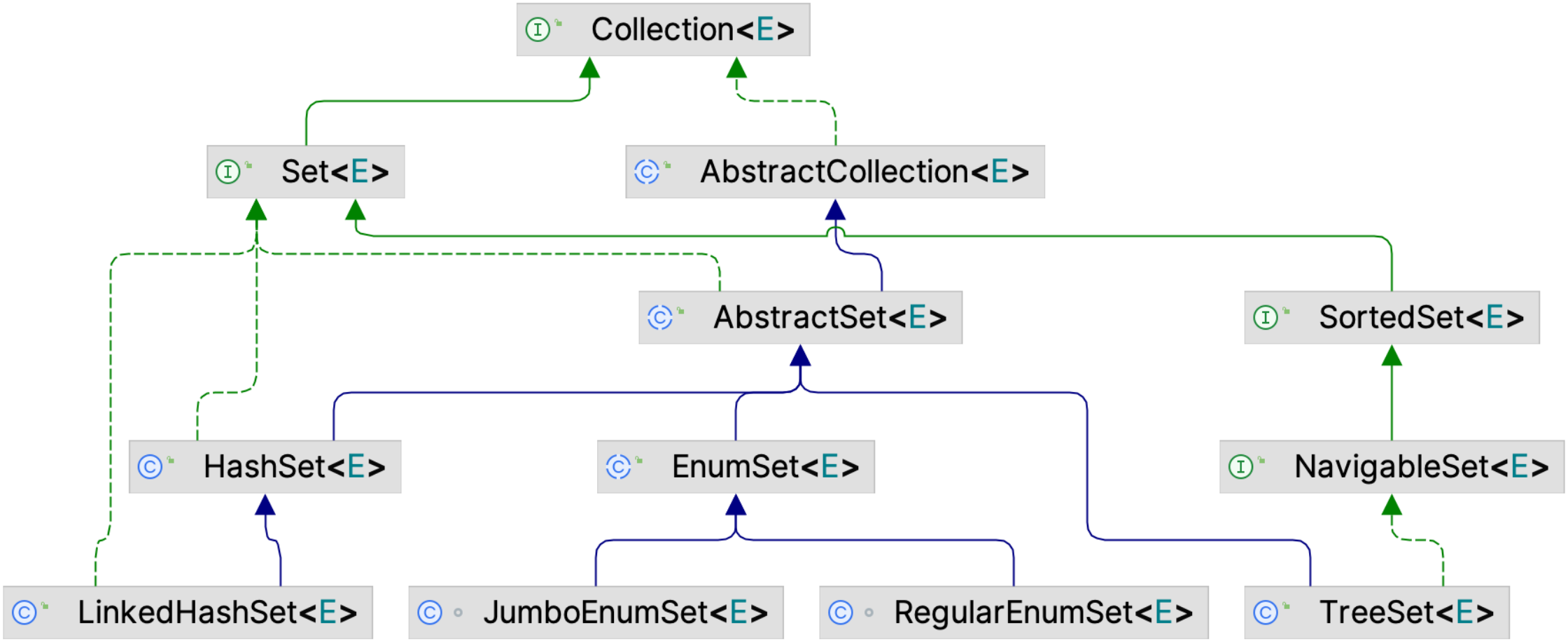
ITERABLE AND MAP INTERFACES



THE MAP HIERARCHY



THE SET HIERARCHY



WHAT CAN YOU DO WITH A SET?

- You have an idea of what you might want to do with a Set and a Queue.
- The actual method names are different, but the operations are there.
- Both extend Collection, which is unordered.

The next several slides are taken from the Collections Framework Overview.

HASHSET DOCUMENTATION

HashSet

“This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element.

This class offers constant time performance for the basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets.”

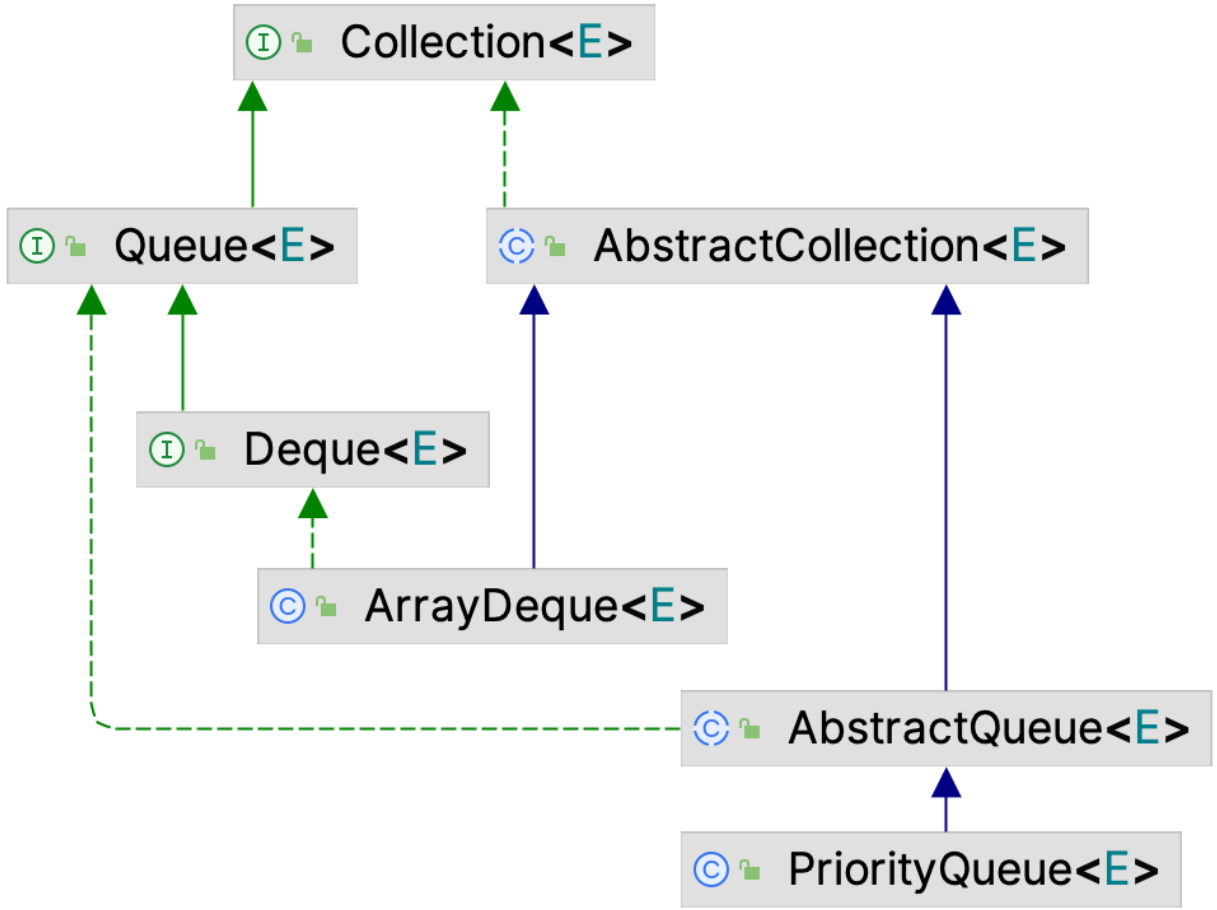
TREESET DOCUMENTATION

TreeSet

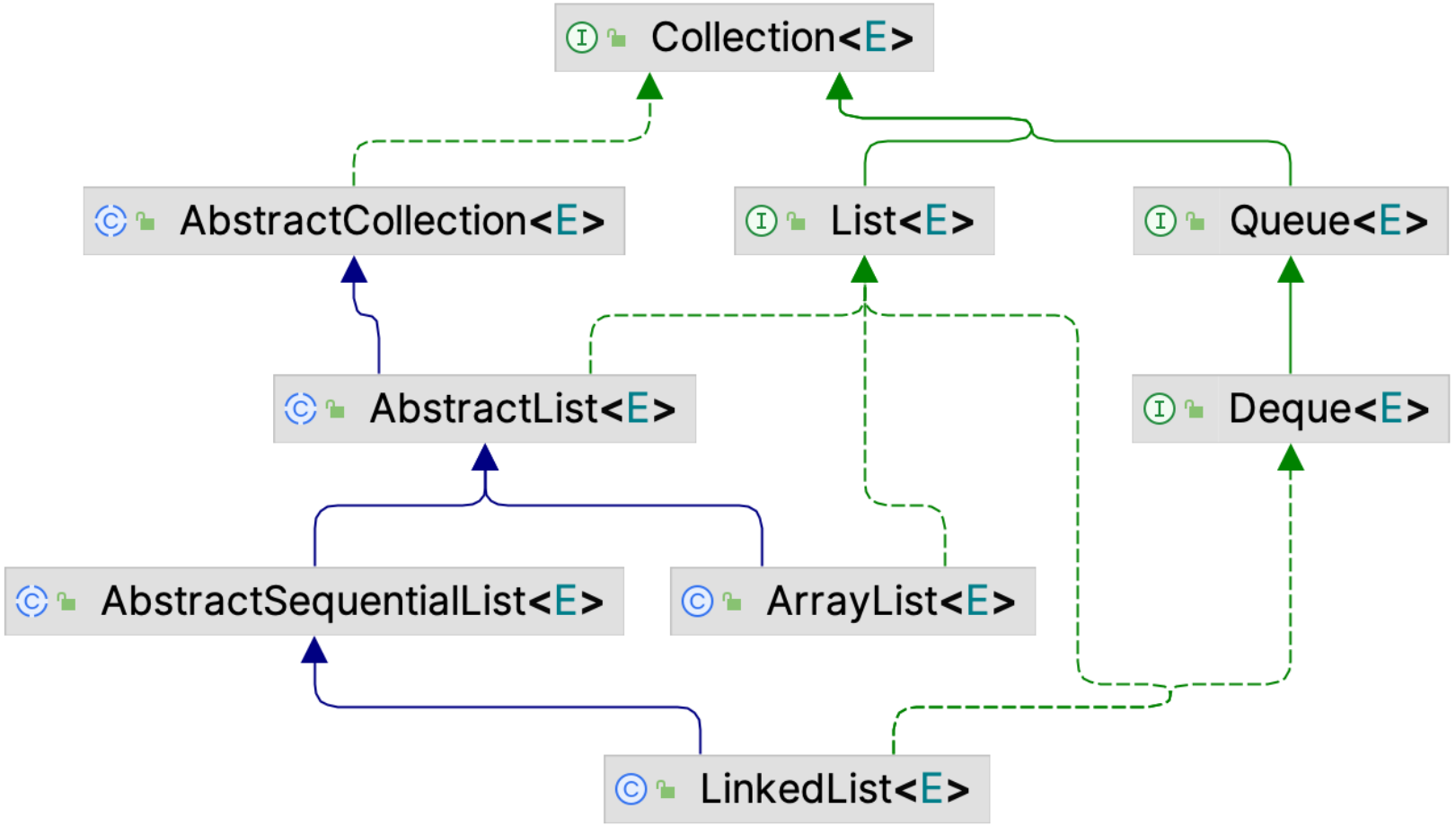
“A NavigableSet implementation based on a TreeMap. The elements are ordered using their natural ordering, or by a Comparator provided at set creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the basic operations (add, remove and contains).”

THE QUEUE HIERARCHY



THE LIST HIERARCHY



ARRAYLIST VS. LINKEDLIST

- ArrayList uses an array to store the contents. When it runs out of room, it makes a new, bigger array and copies over the items. For add(e), get(i) and remove(i), what are the best case running times? Worst? When does it perform well?
- LinkedList uses a doubly-linked list. For add(e), get(i) and remove(i), what are the best case running times? Worst? When does it perform well?
- When would you choose LinkedList over ArrayList?

PRIMARY ADVANTAGES OF DEFINING A FRAMEWORK

- “Reduces programming effort by providing data structures and algorithms so you don't have to write them yourself.
- Increases performance by providing high-performance implementations of data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be tuned by switching implementations.
- Provides interoperability between unrelated APIs by establishing a common language to pass collections back and forth.”

PRIMARY ADVANTAGES OF DEFINING A FRAMEWORK

- “Reduces the effort required to learn APIs by requiring you to learn multiple ad hoc collection APIs.
- Reduces the effort required to design and implement APIs by not requiring you to produce ad hoc collections APIs.
- Fosters software reuse by providing a standard interface for collections and algorithms with which to manipulate them.”

WHAT WERE THEY THINKING?

“The main design goal was to produce an API that was small in size and, more importantly, in "conceptual weight." It was critical that the new functionality not seem too different to current Java programmers; it had to augment current facilities, rather than replace them. At the same time, the new API had to be powerful enough to provide all the advantages described previously.”

WHO CARES ABOUT SOLID?

“To keep the number of core interfaces small, the interfaces do not attempt to capture such subtle distinctions as mutability, modifiability, and resizability. Instead, certain calls in the core interfaces are optional, enabling implementations to throw an UnsupportedOperationException to indicate that they do not support a specified optional operation. Collection implementers must clearly document which optional operations are supported by an implementation.”

Once we learn about the SOLID design principles, come back and reread this slide.

WHAT'S IN THE FRAMEWORK (1)

- "Collection interfaces. Represent different types of collections, such as sets, lists, and maps. These interfaces form the basis of the framework.
- General-purpose implementations. Primary implementations of the collection interfaces.
- Legacy implementations. The collection classes from earlier releases, Vector and Hashtable, were retrofitted to implement the collection interfaces.
- Special-purpose implementations. Implementations designed for use in special situations. These implementations display nonstandard performance characteristics, usage restrictions, or behavior."

WHAT'S IN THE FRAMEWORK (2)

- "Concurrent implementations. Implementations designed for highly concurrent use.
- Wrapper implementations. Add functionality, such as synchronization, to other implementations.
- Convenience implementations. High-performance "mini-implementations" of the collection interfaces.
- Abstract implementations. Partial implementations of the collection interfaces to facilitate custom implementations."

WHAT'S IN THE FRAMEWORK (3)

- "Algorithms. Static methods that perform useful functions on collections, such as sorting a list.
- Infrastructure. Interfaces that provide essential support for the collection interfaces.
- Array Utilities. Utility functions for arrays of primitive types and reference objects. Not, strictly speaking, a part of the collections framework, this feature was added to the Java platform at the same time as the collections framework and relies on some of the same infrastructure."

DISCUSSION

- “Increases performance by providing high-performance implementations of data structures and algorithms. Because the various implementations of each interface are interchangeable, programs can be tuned by switching implementations.”
  - To most easily encourage and take advantage of that, you should choose a Collections interface for (almost) every parameter type, return type, and variable type.”

- Question: When might you *not* want to make a parameter’s type be List?