# CompSci 230 Software Construction

Collections S1 2015

# Syllabus

- Four Themes:
  - The object-oriented programming paradigm
    - Object-orientation, object-oriented programming concepts and programming language constructs – because, for many important problems, OO design is a convenient way to express the problem and its solution in software.
  - Frameworks
    - Inversion of control, AWT/Swing and JUnit – because many important "sub-problems" have already been solved: these solutions should be re-used!
  - Software quality
    - Testing, inspection, documentation – because large teams are designing, implementing, debugging, maintaining, revising, and supporting complex software.
  - Application-level concurrent programming
    - Multithreading concepts, language primitives and abstractions – because even our laptops have multiple CPUs. Dual-core smartphones are now available…

## Learning Goals for this set of slides

- You will have a basic understanding of the syntax and semantics of the Java Collection Framework
  - This provides a *foundation* for you to develop a *working understanding* as you gain practical experience through self-study and in your assignments.
- You will have a basic understanding of the advantages of a well-designed and well-implemented framework.
  - (You will not be exposed to any ill-designed or poorly-implemented frameworks ;-)

## What is a framework?

- In the context of Java, the word "framework" is used loosely.
  - A Java framework is any set of packages whose classes define a unified architecture for an implementation. Examples:
    - The Java Collections Framework (JCF)
    - The Swing Application Framework (SAF)
    - The JUnit testing framework (JUnit)
- Many computer scientists define "framework" narrowly.
  - Swing and JUnit are "frameworks", because they implement "the skeleton of an application that can be customized by the application developer".
  - The Java Collections Framework is a "library", because
    - it is a set of closely-related classes for implementing data structures, but
    - it does not provide a skeleton for an entire application.
- I will try to avoid using the word "framework" in my lectures,
  - except in a proper noun: Java Collections Framework, .NET Framework, etc.

## Collections, in Java

- "A collection — sometimes called a container — is simply an object that groups multiple elements into a single unit.
  - Collections are used to store, retrieve, manipulate, and communicate aggregate data.
  - Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers).
  - If you have used the Java programming language — or just about any other programming language — you are already familiar with collections."

[Lesson: Introduction to Collections, *The Java Tutorials*]

# Collections Framework

- "A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:
  - **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
  - **Implementations:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
  - **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

## Collections in Other Languages

- "Apart from the Java Collections Framework, the best-known examples of collections frameworks are
  - the C++ Standard Template Library (STL) and
  - Smalltalk's collection hierarchy."

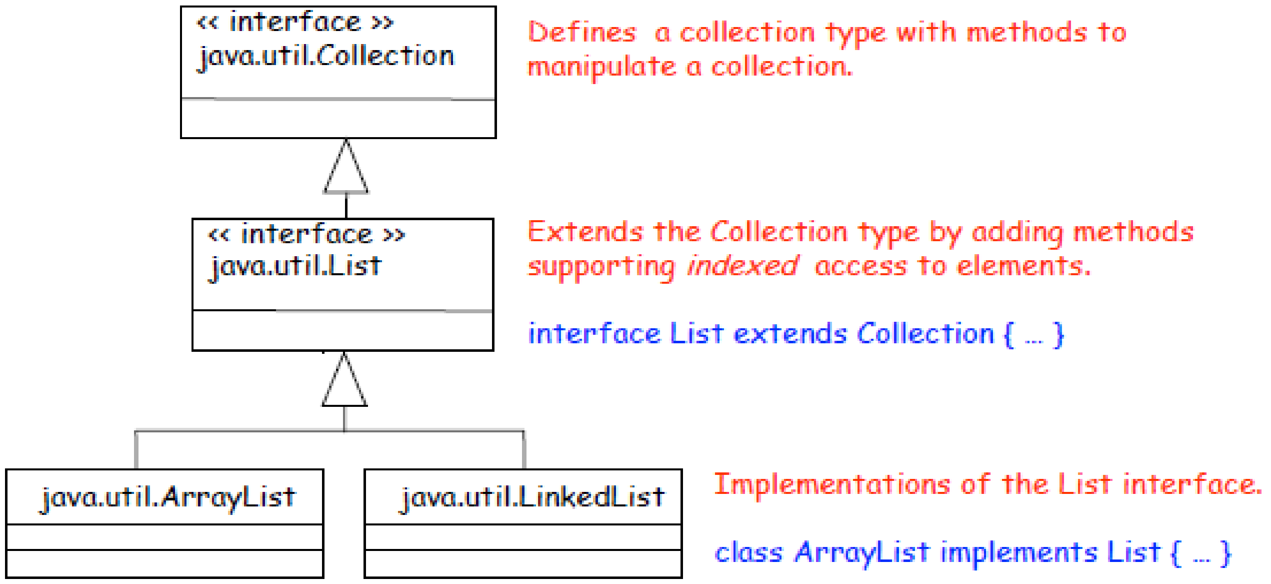
- I wouldn't expect *The Java Tutorials* to discuss a competitor's product! However I'd say:
  - The `System.Collections` and `System.Collections.Generic` namespaces in the Base Class Library of the .NET Framework are very comparable to the JCF.
  - Only a few data structures are standardised in Python. Pythonistas write wrappers to use libraries from other languages.
- If a programming language does not offer well-designed and well-implemented libraries for collections, then programming is much more time-consuming and error-prone.
  - Readability and maintainability are greatly improved by standardised data structures and algorithms.

# Sales Pitch for the JCF

- **Reduces programming effort**:
  - By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work. By facilitating interoperability among unrelated APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.
- **Increases program speed and quality:**
  - This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you're freed from the drudgery of writing your own data structures, you'll have more time to devote to improving programs' quality and performance.
- **Allows interoperability among unrelated APIs:**
  - The collection interfaces are the vernacular by which APIs pass collections back and forth. If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.
- **Reduces effort to learn and to use new APIs:**
  - Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard collection interfaces, the problem went away.
- **Reduces effort to design new APIs:**
  - This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
- **Fosters software reuse:**
  - New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

# Overview of the JCF Class Hierarchy

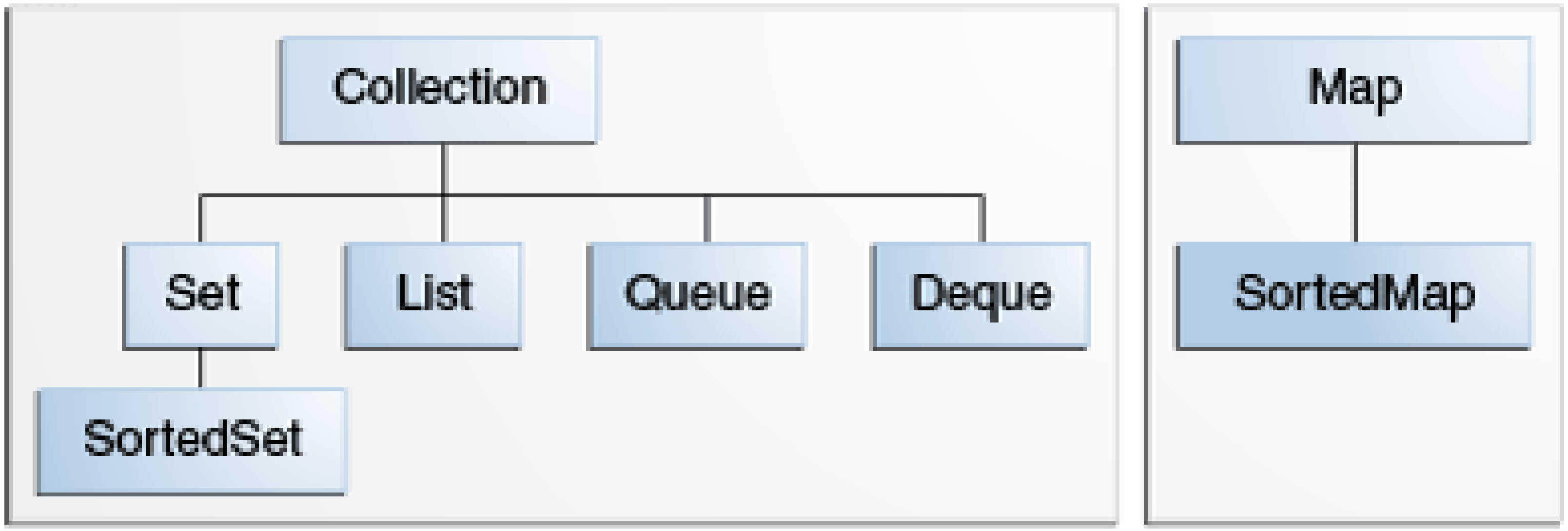
- Collection Interface
  - A collection represents a group of objects, known as its elements
- List Interface
  - An ordered collection. The user can access elements by their integer index (position in the list), and search for elements in the list.
- ArrayList
  - Resizable-array implementation of the List interface
  - Implements the size, isEmpty, get, set, iterator, and listIterator methods
- LinkedList
  - Doubly-linked list implementation of the List.
  - Implements the size, isEmpty, get, set, iterator, and listIterator methods



« interface »
java.util.Collection
Defines a collection type with methods to manipulate a collection.
« interface »
java.util.List
Extends the Collection type by adding methods supporting *indexed* access to elements.
interface List extends Collection { ... }
java.util.ArrayList
java.util.LinkedList
Implementations of the List interface.
class ArrayList implements List { ... }

## JCF: Core Interfaces

- "The *core collection interfaces* encapsulate different types of collections, which are shown in the figure below.
  - These interfaces allow collections to be manipulated independently of the details of their representation."



## Generic Types in JCF

- "… all the core collection interfaces are generic. For example, this is the declaration of the Collection interface.

```
public interface Collection<E> …
```

- The <E> syntax tells you that the interface is generic.
- When you declare a Collection instance you can and should specify the type of object contained in the collection.
  - Specifying the type allows the compiler to verify (at compile-time) that the type of object you put into the collection is correct, thus reducing errors at runtime."

- In this offering of CompSci 230, I will not cover the formal semantics of generic types in Java – you'll learn by example.

## List versus List<T>

- Nice:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

- Nicer (due to better type-checking, and less type-casting):

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

- Note: lists in Java are indexed from `0`. This convention is a legacy from the C language.
  - Arrays in Java are also indexed from `0`.
  - Indexing from 1 may seem more natural.
  - If you're importing multiple packages, do they all have the same conventions?

## The Collection Interface

- "The `Collection` interface is used to pass around collections of objects where maximum generality is desired.
- For example, by convention all general-purpose collection implementations have a constructor that takes a `Collection` argument.
  - This constructor, known as a conversion constructor, initializes the new collection to contain all of the elements in the specified collection, whatever the given collection's subinterface or implementation type.
  - In other words, it allows you to convert the collection's type.
- Suppose, for example, that you have a `Collection<String> c`, which may be a `List`, a `Set`, or another kind of `Collection`.
  - [The following] idiom creates a new `ArrayList` (an implementation of the `List` interface), initially containing all the elements in `c`.

```
List<String> list = new ArrayList<String>(c);
```

# Traversing Collections

- "There are three ways to traverse collections:

1. using aggregate operations
2. with the for-each construct and
3. by using Iterators.

- Aggregate Operations (not examinable!)
  - In JDK 8 and later, the preferred method of iterating over a collection is to obtain a stream and perform aggregate operations on it.
  - Aggregate operations are often used in conjunction with lambda expressions to make programming more expressive, using less lines of code.
  - The following code sequentially iterates through a collection of shapes and prints out the red objects:

```
myShapesCollection.stream()
.filter(e -> e.getColor() == Color.RED)
.forEach(e -> System.out.println(e.getName()));
```

## For-each iteration over a Collection

- "The for-each construct allows you to concisely traverse a collection or array using a for loop…
- The following code uses the for-each construct to print out each element of a collection on a separate line.

```
for (Object o : collection) {
  System.out.println(o);
}
```

[https://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html]

- Stylistic suggestion: use braces on all loops!
  - See e.g. Google's Style Guide, 4.1.1 Braces are used where optional:
    - "Braces are used with `if`, `else`, `for`, do and `while` statements, even when the body is empty or contains only a single statement."

## Iterators

- An **`Iterator`** is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired.
  - You get an Iterator for a collection by calling its iterator method.
  - The following is the Iterator interface.

```
public interface Iterator<E> {
   boolean hasNext();
   E next();
   void remove(); //optional
}
```

  - The **`hasNext`** method returns true if the iteration has more elements, and
  - the **`next`** method returns the next element in the iteration.
  - The **`remove`** method removes the last element that was returned by **`next`** from the underlying **`Collection`**.
    - The **`remove`** method may be called only once per call to **`next`** and throws an exception if this rule is violated.

## Modifying a collection

- Note that **`Iterator.remove()`** is the only safe way to modify a collection during iteration;
  - the behavior [of an iteration] is unspecified if the underlying collection is modified in any other way while the iteration is in progress.
- Use **`Iterator`** instead of the for-each construct when you need to:
  - Remove the current element.  Note: the for-each construct hides the iterator, so you cannot call **remove()** within a for-each loop.
  - Iterate over multiple collections in parallel.

# Filtering a collection

- "The following method shows you how to use an Iterator to filter an arbitrary **`Collection`** — that is, traverse the collection removing specific elements.

```
    static void filter(Collection<?> c) {
        for (Iterator<?> it = c.iterator(); it.hasNext(); )
            if (!cond(it.next()))
                it.remove();
    }
```

- Note the use of the wild-card type **`<?>`** in this Iterator declaration.
  - You'll be programming-by-example when you're using generics.
  - The example above is an "idiom", that is, a common coding pattern.

# Set, List, Queue, Deque

- You should have learned these data structures in COMPSCI 105.
- The Java Collection Framework has very efficient implementations of these data structures.
  - You'll have to choose an implementation!
  - Sets are implemented as **`HashSet`**, **`TreeSet`**, and **`LinkedHashSet`**.
  - Lists are implemented as **`ArrayList`** and **`LinkedList`**.
  - Queues are implemented as **`LinkedList`** and **`PriorityQueue`**.
  - Deques are implemented as **`LinkedList`** and **`ArrayDeque`**.
- I don't expect you to memorise details of these implementations!
  - You will be able to understand simple uses of the JCF, **after** you have worked through some examples on your own. (You won't learn this by listening to me!)
  - You'll make some syntax errors, especially with generics, but you should be able to sort these out by reviewing some idioms, reading the tutorials, and bashing your way through coding via Eclipse.
  - You'll have semantic difficulties with complex implementations and methods, but you should already understand the basic operations on Sets and Lists – so this is mostly review with only a few "surprises" or "quirks" (e.g. indexing from `0`).

# Map

- "A Map is an object that maps keys to values.
- A map cannot contain duplicate keys:
  - Each key can map to at most one value.
  - It models the mathematical *function* abstraction.
- The **Mapinterface** includes methods for
  - basic operations (such as **put**, **get**, **remove**, **containsKey**, **containsValue**, **size**, and **empty**),
  - bulk operations (such as **putAll** and **clear**), and
  - collection views (such as **keySet**, **entrySet**, and **values**).
- The Java platform contains three general-purpose Map implementations: HashMap, TreeMap, and LinkedHashMap.
  - Their behavior and performance are precisely analogous to **HashSet**, **TreeSet**, and **LinkedHashSet**, as described in The Set Interface section.

## Map Interface Basic Operations

- "The following program generates a frequency table of the words found in its argument list.
  - The frequency table maps each word to the number of times it occurs in the argument list.

```
import java.util.*;
public class Freq {
    public static void main(String[] args) {
        Map<String, Integer> m = new HashMap<String, Integer>();
        // Initialize frequency table from command line
        for (String a : args) {
            Integer freq = m.get(a);
            m.put(a, (freq == null) ? 1 : freq + 1);
        }
        System.out.println(m.size() + " distinct words:");
        System.out.println(m);
    }
}
```

- The only tricky thing about this program is the second argument of the **put** statement.
  - That argument is a conditional expression that has the effect of setting the frequency to
    - one if the word has never been seen before or
    - one more than its current value if the word has already been seen.
- Try running this program with the argument…" (I strongly encourage you to do this! ;-)

## Have you achieved these learning goals?

- You will have a basic understanding of the syntax and semantics of the Java Collection Framework
  - This provides a *foundation* for you to develop a *working understanding* as you gain practical experience through self-study and in your assignments.
- You will have a basic understanding of the advantages of a well-designed and well-implemented framework.
  - (You will not be exposed to any ill-designed or poorly-implemented frameworks ;-)