ABSTRACT DATA TYPES: COLLECTIONS, LISTS, SETS, MAP, QUEUES
Lists are Iterable

```java
public static void printList(List<String> strings) {
    for (int idx = 0; idx < strings.size(); idx++) {
        String s = strings.get(idx);
        System.out.print("Idx: "+ idx + ".");
        System.out.println("String: " + s);
    }
}
```

```java
public static void printList2(List<String> strings) {
    // alternative syntax for Iterable object
    for (String s : strings) {
        // with this approach, do not get index
        System.out.println("String: " + s);
    }
}
```
Organizing data
Abstract Data Types in use

- **Collection**
  - Bunch of data (e.g., web pages)

- **Set**
  - Bunch of data, no duplicates (e.g., set of words)

- **Map**
  - A map between collections (words to web pages)

- **Lists**
  - Collection where position matters (e.g. search results, user logs)
Java Collections Framework

- **Collections**: holders that let you store and organize objects in useful ways for efficient access

- Since Java 1.2, the package `java.util` includes interfaces and classes for a general collection framework

- **Goal**: conciseness
  - A few concepts that are broadly useful
  - Not an exhaustive set of useful concepts

- The collections framework provides
  - Interfaces (i.e., ADTs)
  - Implementations
JCF Interfaces and Classes

- **Interfaces**
  - Collection
  - **Set** (no duplicates)
  - **List** (position matters)
  - **Map** (aka, dictionary, index)
  - **Queue** (“first come, first serve”)
  - ...

- **Classes**
  - **HashSet**, **TreeSet**
  - **ArrayList**, **LinkedList**
  - **HashMap**, **TreeMap**
  - **AbstractQueue**, **ArrayDeque**
Collections in Java

- Collection interface
  - Subinterfaces: List, Set, Queue, ...
  - Iterable

- Map
  - A mapping between two collections (keys and values)
  - Not a subinterface of Collection
  - Not Iterable (but keySet is)
When using a collection (e.g., `LinkedList`, `HashSet`, `HashMap`), we generally have a single type `T` of elements that we store in it (e.g., `Integer`, `String`).

Before Java 5, when extracting an element, had to cast it to `T` before we could invoke `T`'s methods.

Compiler could not check that the cast was correct at compile-time, since it didn't know what `T` was.

Inconvenient and unsafe, could fail at runtime.

- Generics in Java 5 provide a way to communicate `T`, the type of elements in a collection, to the compiler.
  - Compiler can check that you have used the collection consistently.
  - Result: safer and more-efficient code.
/** Counts how many students are studying at "CTB" */
public static int studentCount(Collection students) {
    int count = 0;
    for (Object o : students) {
        CornellStudent student = (CornellStudent) o;
        if (student.getStudyLocation().equals("CTB")) {
            count++;
        }
    }
    return count;  }

public static int studentCount(Collection<Student> students) {
    int count = 0;
    for (Student student : students) {
        if (student.getStudyLocation().equals("CTB")) {
            count++;
        }
    }
    return count;  }
Java 5 also has autoboxing and auto-unboxing of primitive types, which can be handy when working with primitive types.

```java
public static int gpa(List<Integer> grades) {
    int avgGrade = 0;
    for (Integer grade : grades) {
        avgGrade += grade; //auto convert Integer to int!
    }
    return avgGrade / grades.size();
}
```

AutoBoxing/Unboxing: converts from “int” to “Integer”, “byte” to “Byte”, etc.
Using Generic Types

- `<T>` is read, “of T”
  - For example: `List<Integer>` is read, “List of Integer”

- The type annotation `<T>` informs the compiler that all extractions from this collection should be automatically cast to T

- Specify type in declaration, can be checked at compile time
  - Can eliminate explicit casts
Advantage of Generics

- Declaring `Collection<String> c` tells us something about the variable `c` (i.e., `c` holds only Strings)
  - This is true wherever `c` is used
  - The compiler checks this and won’t compile code that violates this

- Without use of generic types, explicit casting must be used
  - A cast tells us something the programmer `thinks` is true at a single point in the code
  - The Java virtual machine `checks` whether the programmer is right only at runtime
Using Lists, Sets, Maps