Introduction to Object-Oriented Programming
Collections Algorithms

Christopher Simpkins
chris.simpkins@gatech.edu
A collection is an object that represents a group of objects.

The collections framework allows different kinds of collections to be dealt with in an implementation-independent manner.
The collections framework includes algorithms that operate on collections. These algorithms are implemented as static methods of the `Collections` class. A good example is the (overloaded) `sort` method:

```java
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

This method signature demonstrates how to declare a generic method (so far we’ve seen only generic classes): put a type parameter before the return type.

- **This `sort` uses the “natural ordering” of the list, that is, the ordering defined by** `Comparable`.
- `<? super T>` is a **type bound**. It means “some superclass of `T`.”
- **The type parameter** `<T extends Comparable<? super T>>` means that the element type `T` or some superclass of `T` must implement `Comparable`. 
The `java.lang.Comparable` Interface

```java
class Comparable<T>
{
    public int compareTo(T o);
}
```

`compareTo(T o)` compares this object with the specified object for order. Returns

- a negative integer if this object is less than the other object,
- zero if this object is equal to the other object, or
- a positive integer if this object is greater than the other object.
Here’s a `Person` class whose natural ordering is based on age:

```java
public class Person implements Comparable<Person> {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String toString() {
        return name;
    }

    public int compareTo(Person other) {
        return this.age - other.age;
    }
}
```
Using `Collections.sort(List<T> list)`

Given the `Collections` static method:

```java
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

We could sort a `List<Person> because Person implements Comparable<Person>`:

```java
List<Person> peeps = new ArrayList<>();
peeps.add(new Person(...));
...
Collections.sort(peeps);
```

And if we have a class:

```java
public class GtStudent extends Person { ... }
```

We could also sort a `List<GtStudent>` becuase

- GtStudent extends Person,
- Person implements Comparable<Person> and
- Person is a supertype of GtStudent
Java uses *type erasure* to implement generics, meaning that the compiled code is nearly identical to non-generic code. Type erasure allows for compile-time type checking while preserving the ability to work with legacy code. So you can sort a raw `List` of `Person` using the `compareTo(Person)` method:

```java
List rawPeeps = new ArrayList();
rawPeeps.add(new Person(...));
...
Collections.sort(rawPeeps);
```
Overriding only happens when methods have identical signatures. To allow generic classes to work in non-generic settings, the compiler inserts *bridge* methods. So `Person` looks like:

```java
public class Person implements Comparable<Person> {
    // ...

    // This is a bridge method inserted by the compiler to allow this
    // class to work with legacy non-generic code
    public int compareTo(Object other) {
        return compareTo((Person) other);
    }

    public int compareTo(Person other) {
        return this.age - other.age;
    }
}
```
public interface Comparator<T> {

    int compare(T o1, T o2);

    boolean equals(Object obj);
}

Comparator<T> is an interface with two methods:

- int compare(T o1, T o2) - same contract as o1.compareTo(o2)

- boolean equals(Object obj)

It’s always safe to use the inherited equals method, so the one you need to implement is compare.

See SortTroopers.java and Trooper.java for examples using Comparable, Comparator and Collections.sort(...).
Programming Exercise

Write a class to represent Georgia Tech students called, say, GtStudent.

- Give GtStudent name, major, GPA, and year fields/properties.
- Have GtStudent implement Comparable<T> with some ordering that makes sense to you – perhaps some majors are harder than others, so GPAs are adjusted in comparisons.
- Add instances of GtStudents to an ArrayList<E>.
- Sort the ArrayList of GtStudents using Collections.sort(List<E>).
- Write a Comparator<GtStudent> and sort your list with Collections.sort(List<E>, Comparator<E>).

Extra: add thousands of randomly-generated GtStudents to an ArrayList and a LinkedList and time Collections.sort(List<E>) method invocations for each of them. Is one faster? Why (or why not)?