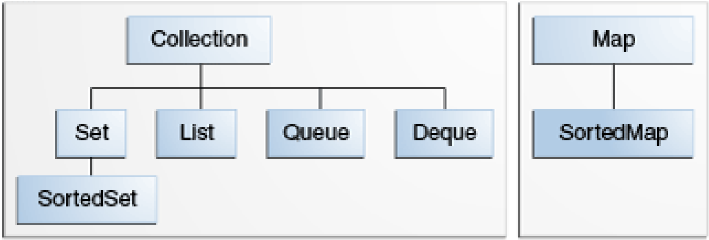
Collections Algorithms

The Collections Framework



- 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.

`Collections.sort(List<T> list)`

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:

```
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 classs): 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`."

For now just think of it this way: 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

```
public interface 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.

Implementing java.lang.Comparable<T>

Here's a Person class whose natural ordering is based on age:

```
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;
    }
}
```

Analyzing <T extends Comparable<? super T»

Given the Collections static method:

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

And the classes:

```
public class Person implements Comparable<Person>
public class GtStudent extends Person { ... }
```

Can we sort a List<GtStudent>?

Type checker "proves" that a type argument satisfies a type bound.

Prove by substituting without causing contradictions:

```
[GtStudent/T, Person/?]<T extends Comparable<? super T>>
⇒
<GtPerson extends Comparable<Person super GtStudent>
```

Yes, we can sort a List<GtStudent> because

- GtStudent extends Person,
- Person implements Comparable<Person>, so
- GtStudent is a subtype of Comparable<Person> and
- Person is a supertype of GtStudent

Using Collections.sort(List<T> list)

Given the Collections static method:

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

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

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

And if we have a class:

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

We could also sort a List<GtStudent> because

- GtStudent extends Person,
- Person implements Comparable<Person> and
- Person is a supertype of GtStudent

Using `Collections.sort(List<T>)` on Raw Lists

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:

```
List rawPeeps = new ArrayList();
rawPeeps.add(new Person(...));
...
Collections.sort(rawPeeps);
```

Using Collections.sort(List<T>) on Raw Lists

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:

```
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;
    }
}
```

Using java.util.Comparator<T>

```
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 `GtStudent~s 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-gnerated `GtStudent~s 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)?