Head First Java: Chapter 16 Notes

September 17, 2019

Generics

Generics in Java is a mechanism that increases type-safety and is often used in collection-related classes due to polymorphism. In Java, most collection classes like `ArrayList` requires consistency in the type of object it will contain as well as the type of object argument its methods will pass. This consistency requirements can be seen in the Java API documentation of various collection classes. For example, the class declaration for ArrayList reads:

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

The notation `<E>` heres denotes a generic type (where E stands for "element") that can be pretty much any objects (`String`, `Integer`, even `Object`), but whatever E is must be consistent within the declared object. This becomes clear when looking at associated constructors and methods for `ArrayList<E>`:

```
get()
public E get(int index)

add()
public boolean add(E e)
```

Both the methods `ArrayList.get()` and `ArrayList.add()` requires an argument of class E object, which is whatever E was during the declaration of the `ArrayList` itself. This means that `ArrayList<String>` can only accept elements of class `String`, and not, for example, of class `Integer`. Polymorphism applies here, so E can be anything that passes the IS-A test. For example, if an `ArrayList<Animal>` was declared, where `E = Animal`, then when calling `ArrayList<Animal>.add()` a argument of object class `Animal` can be passed, so does a `Dog` class object (a subclass of `Animal`, so `Dog` IS-A(N) `Animal`) as well as a `Cat` class object (same as `Dog`; a subclass of `Animal`). However, if `ArrayList<Dog>` was declared, then `ArrayList<Dog>.add()` cannot accept a `Cat` object as argument, because `E = Dog` and a `Cat` fails the IS-A test for `Dog`.

Note that when writing class and their related methods and using generics, by convention `E` is only use to denote generics for collection classes (as `E` stands for element), for all other cases it is generally accepted that `T` should be used instead. When declaring methods with generics, a special parameterization can be declared:

```
public <T extends Animal> void takeThing(ArrayList<T> list)
```

To parameterize a method the generic should be declared just before the return type. Here `<T extends Animal>` means `T` = any class that extends the class `Animal` (including `Animal` itself). Note that in order to utilize the polymorphism aspect of generics the above syntax must be used. If the parameterization wasn't declared:

```
public void takeThing(ArrayList<Animal> list)
```

In this case, the compiler will not allow the code to compile if the argument passed to `takeThing()` is not exactly an `ArrayList<Animal>`. Polymorphism is disabled in this case to prevent type confusion down the road.

Type parameterization during method declaration can accommodate greater degree of complexity. For example in the `sort()` static method available in the `Collections` package:

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

There the generic type `T` is defined as any class that extends (or in this context, implements) the `Comparable` interface. The `?` notation used here is a Java "wild card" character, similar to `E` and `T`, except there is no consistency expectation (that is, whatever `?` is does not have any relation to another `?` within the same declaration statement.) Here, `Comparable<?  super T>` means that the `Comparable` interface must be parameterized to ether `T` or one of `T`'s superclass. The wild card character can be used interchangeably with the `T/E` syntax:

```
public <T extends Animal> void takeThing(ArrayList<T> one, ArrayList<T> two)
```

```
public void takeThing(ArrayList<? extends Animal> one, ArrayList<? extends Animal> two)
```

The two different declaration above are functionally the same. However, using `T` is less verbose.

Collections

The Java `Collections` API has many useful classes that give various functionality similar to `ArrayList`.

- `TreeSet` — The `TreeSet` class limits the elements it can contain to only unique elements, and can only keep them in sorted order. The ordering occurs automatically and there is no way to change it. Also, elements added into a `TreeSet` must be a from a class that implemented `Comparable` somewhere in its inheritance tree.
- `HashMap` — This is similar to Python dictionaries with keys and values. Because of that, the parameterization of `HashMap` takes

two elements in its angle brackets, for example `HashMap<String, Integer>`.

- `LinkedList` — Functionally almost the same as `ArrayList`, but difference in internal structure means `LinkedList` is faster for manipulation but slower for random access.
- `HashSet` — This is similar to Python sets in that it only contains unique elements. However, *unique* has very specific meaning in Java that might cause seemingly unpredictable behavior. Also, there is ordering in a Java `HashSet` unlike a Python set.
- `LinkedHashMap` — List a `HashMap` but with ordering.

`Comparable` and `Comparator`

Collection class objects can utilize its methods to manipulate its elements, for example sorting using the static method `Collection.sort()`. However, many of these methods require the elements are from a class that is either `Comparable` (by extending/implementing the `Comparable` interface) or that the method call is passed a `Comparator` class object instead. The interface `Comparable` has an abstract method that must be declared by any class that implements the interface: `compareTo()`, which accepts the argument `T` which is parameter is must specified during class declaration (*e.g.*, public class Animal implements Comparable<Animal>). Often times while overriding the `compareTo()` method it isn't necessary to write the code from scratch, as the instance variable the user desired to be used as the reference for comparison usually comes from a class that had already implemented `Comparable` (classes such as `String`, `Integer`, etc.) Therefore when overriding in the new class all that is needed to is to call `compareTo()` from that variable. For example:

```
class Song implements Comparable<Song> {

    String title;

    Song(String t) {
        title = t;
    }

    public String getTitle() {
        return title;
    }

    public int compareTo(Song s) {
        return title.compareTo(s.getTitle());
    }
}
```

An alternative way is to use a `Comparator` as an argument for these methods. In this example for the `Collections.sort()` method it is overloaded with two different sets of arguments, either a `List<T>` (`List` has many subclasses, such as `ArrayList` etc., which would all be acceptable because an `ArrayList` IS-A `List`) or a `List<T>` and a `Comparator<?  super T>`. This difference here is for the one-argument version of the method call (the case described above), `T` is parameterized to `<T extends Comparable<?  super T»`, while the two-argument version `T` is not parameterized and can be any object.

To use the two-argument method call then an object that implemented `Comparator` as an interface must be declared (but note that in this case `T` does not need to implement `Comparable`). The `Comparable` interface has two abstract methods that needs to be implemented, `compare()` and `equals()`, but `equals()` is a special case in Java where it is not necessary for it to be implemented. The method `compare()` takes two argument of the desired object, and similar to `compareTo()` above it is often not necessary to rewrite the code body of the method (for the same reason as above):

```
class Song {

    String title;

    Song(String t) {
        title = t;
    }

    public String getTitle() {
        return title;
    }
}

class TitleCompare implements Comparator<Song> {

    public int compare(Song one, Song two) {
        return one.getTitle().compareTo(two.getTitle());
    }
}
```

In this case, a new instance of the `TitleCompare` class would be created and pass with `Collections.sort()` along with the `List<T>` (`T` would be `Song` in this case) argument.

Duplicates and Unique Objects in Java

Unlike in Python, whether or not objects are duplicates or unique from one another might not be self-evident. This is because Java treats *reference equality* and *object equality* as not trivial. Two objects can have the same values (that is, the same instant variable values), and exist are two different object on the heap. In this case, this is referred to as *object equality*, where it is meaningfully equal to each other, yet Java treats them as unique from each other. Internally, Java keeps track of object identify via *hash codes* when putting them in the hash-based collections (*e.g.* `hashMap`, `hashSet`), and unique objects are determined by having different hash codes. This means that objects that has *reference equality* must have the same hash codes (note that having the same hash code does not guarantee reference equality.) This can lead to unintentional behavior when dealing with collections. If a user desired that a collection only contains "unique" objects, they can put them into a hash-based collection such as a `hashSet`. However, more often than not the user actually desires that the `hashSet` only contain objects that are unique in the object equality sense (that is, objects with the same instance variable values be treated as duplicates even those they might be different objects on the heap with different hash codes.) In this case the `hashCode()` and `equals()` methods must be overridden in the class at hand. For example:

```java
class Song implements Comparable<Song> {

    String title;

    Song(String t) {
        title = t;
    }

    public String getTitle() {
        return title;
    }

    public int compareTo(Song s) {
        return title.compareTo(s.getTitle());
    }

    public boolean equals(Object aSong) {
        Song s = (Song) aSong;
        return getTitle().equals(s.getTitle());
    }

    public int hashCode() {
        return title.hashCode();
    }
}
```

With regards to the `equals()` method, overriding this is a similar process as the `compare()` override above as most classes already has its own `equals()` method. The `equals()` override here should return the `equals()` return value of whatever the user desired the uniqueness should be based upon (in this case, the song title.) This is the same with the `hashCode()` override, where instead of returning the hashCode of the entire object, `hashCode()` should return the hash code of the `String title`, so that `Song` objects with the same `title` value will generate the same hash code.