Chapter 6: Generics

Generics are a way for programmers to generalize the type that a class or method works with. They allow us to reuse the same code for various input types without needing to cast things constantly.

For example, consider the `List<T>` interface built into Java. This interface specifies what it means for a class to store a list of objects of a generic type `T`. When we declare a variable to refer to a list of integers, we specify the type as `List<Integer>`.

Here is a quick example working with generics:

```
// note we don't need <Integer> on the right since it is implicit
ArrayList<Integer> ma = new ArrayList<>();
ma.add(1);
Integer my_item = ma.get(0) + 5;
```

The `<Integer>` specifies that we're working with an `ArrayList` containing only Integers: we don't need to do any casting.

For our purposes, we'll primarily be making use of existing code that is written using generics, like `ArrayList<T>` above. It is also possible to include generics in your own custom classes, as described next, but this typically won't be necessary in this course.

6.1. Custom Generic Classes

Briefly, the syntax to define a generic class is as follows:

```
public class Box<T> {
    private T item;

    public void set(T item) {
        this.item = item;
    }

    public T get() {
        return item;
    }
}
```

In this example:

- `Box<T>` is a generic class where `T` is a type parameter.
- You can create instances of `Box` for different types, like `Box<Integer>` or `Box<String>`.
- The type `T` is replaced with the actual type when the class is instantiated.

Example usage of our generic class might look like:

```
Box<Integer> intBox = new Box<>();
intBox.set(123);
Integer value = intBox.get();

Box<String> strBox = new Box<>();
strBox.set("Hello");
String text = strBox.get();
```

6.2. bounded type parameters

You can also **restrict** the types that can be used with a generic class or method using **bounded type parameters**. This is useful when your code relies on certain behaviors or interfaces being available.

For example, Java's `Collections.sort()` method requires that the elements in the list implement the `Comparable` interface, so that they can be ordered.

Here's how you might define a generic method that sorts a list of items that are guaranteed to be comparable:

```
public static <T extends Comparable<T>> void sortList(List<T> list) {
    Collections.sort(list);
}
```

In this example:

- `<T extends Comparable<T>>` means that `T` must be a type that implements `Comparable<T>`.
- This ensures that the `sortList` method can safely call `compareTo()` on elements of the list.

If you try to call `sortList()` with a type that doesn't implement `Comparable`, you'll get a compile-time error. This is exactly what generics are designed to help prevent.

It turns out that our `sortList()` is actually *slightly* more restrictive than it needs to be. If we look at the signature for `Collections.sort`, it is actually:

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

The `<T extends Comparable<? super T>>` part means that `T` must be a type that can be compared to itself **or any of its supertypes**.

The `?` is a wildcard that stands for an unknown type. In `Comparable<? super T>`, it means we don't care exactly what the type is — only that it is a supertype of T. This allows the method to work with a wider range of types, especially when inheritance is involved.

If the signature were just `<T extends Comparable<T>>`, it would fail in cases like this:

```
class Animal implements Comparable<Animal> { ... }
class Dog extends Animal { ... }

List<Dog> dogs = new ArrayList<>();
Collections.sort(dogs); // works with Comparable<? super T>
```

This works because `Dog` is a subtype of `Animal`, and `Animal` implements `Comparable<Animal>`. Using `? super T` ensures that this kind of relationship is allowed.

This design helps make the `sort()` method compatible with a wider range of types in real-world code.

You can read a much more in-depth explanation of Generics in the Java documentation if you are interested. A tutorial is also available here.

> Under the hood, Java generics use a technique called **type erasure**. This means that generic types are replaced with `Object` at compile time, and the compiler inserts casts where necessary. So while generics give us type safety at compile time, the JVM doesn't retain generic type information at runtime — it just sees regular classes and `Object` references. This is why you can't do things like `new T()` or check `instanceof T` — the type information isn't available at runtime.