Contents

Generics

Generic are a way to provide a compile time type safe parametrization and generalization of a set of algorithms, In the past to achieve generic behavior in java, one could use the Object type, however that is not safe, it requires type casts and is generally error prone, errors that can be usually only captured at runtime. The generics in java have the following syntax `<T>` every parameter enclosed in angle brackets is called a `type parameter` - since its a parameter and it represents a type.

```
class Generic<T> {
    // then one ca n use T to reference the type in internal class members

    private T value;

    public void method(T input) {
    }

    public T compute() {
    }
}
```

`Generics work only with reference types, on other words one can not create a generic or define a generic class that uses primitives, further more generics can not be assigned willy nilly - one can not assign variable references that point to two different genericly paramterized types - Generic<Integer> != Generic<Double>. The type parameter defined in the generic is part of the class definition in a way`

Erasure

Remember, there is no generic type information available at run time, all information about the type of a generic is erased at runtime, meaning that one can not obtain information about the class type of a generic,

An important constraint that governed the way that generics were added to Java was the need for compatibility with previous versions of Java. Simply put, generic code had to be compatible with preexisting, non-generic code. Thus, any changes to the syntax of the Java language, or to the JVM, had to avoid breaking older code. The way Java implements generics while satisfying this constraint is through the use of erasure. In general, here is how erasure works.

`When your Java code is compiled, all generic type information is removed (erased). This means replacing type parameters with their bound type, which is Object if no explicit`

bound is specified, and then applying the appropriate casts (as determined by the type arguments)to maintain type compatibility with the types specified by the type arguments. The compiler also enforces this type compatibility. This approach to generics means that no type parameters exist at run time. They are simply a source-code mechanism.

Ambiguity

This occurs because of the already mentioned process of type erasure, where two seemingly unrelated types resolve to the same underlying object, in the example below, one can define an instance of the Generic class which causes a compilation error

```
class Generic<T, V> {
    public voi set(V o) {
    }
    public voi set(T o) {
    }
}

Generic<String, String> obj = new Generic<String, String>(); // consider
   what would happen here, the Generic class will have two methods called
   `set` which both have the same type of parameter - String
Generic<String, Integer> obj = new Generic<String, Integer>(); // this
   however is perfectly valid, there is no overlap between the two
   objects, meaning that the `set` method will be overloaded just fine
```

The example above, will cause a compile time error since we are not allowed to overload methods like that, they are identical, due to the way the generics are specified, this is something to keep in mind, had we provided bounding restrictions, where T and V are deriving off of different type hierarchies, then that problem can never occur.

Bounded

Generics are mostly useful when more information is given to the compiler, one can imagine that in the example above having T defined alone as a type parameter is not very useful, because the type `T` is way too generic, there are not many methods one can invoke on that type, besides the one defined in Object class type. However in java one can specify the base or boundary of a generic, meaning that the compiler now has more information on which type of type parameter this generic class will be working on

```
class Sum<T extends Number> {
    private T[] numbers;

    public int sum() {
        int sum = 0;
        for(T num: numbers) {
            sum += num.integerValue();
        }
        return sum;
    }
}
```

With the example above, one can see how providing a specialization for the generic, telling the compiler the generic type is of type Number, which exposes a given interface, in this case `integerValue`, one can use this generic class to sum any type of numbers - `Sum<Double>` or `Sum<Integer>` or `Sum<Short>` etc

Unbound

There are situations where a generic class might want to take advantage of another generic class, which is also parametrized, however unless the other class' type parameter is the same as the one in the main class, one can run into problems. Java provides a way to specify a usage of a generic class without providing the type or using a wildcard

```
class Sum<T extends Number> {
    private T number;

    // provide an unbound unknown type parameter in the specialization,
       allowing the caller to pass any generic object of Sum<>
    public boolean compare(Sum<?> another) {
        return number.equals(another.number);
    }

    // if we had only this method, we would be only able to compare Sum<T>
       with Sum<T> which while useful, is not generic enough
    public boolean compare(Sum<T> another) {
        return number.equals(another.number);
    }

    // this is not overloading the method, it is a compile time error,
       java can not understand and overload generic type parameters
    public boolean compare(Sum<Integer> another) {
        return number.equals(another.number);
    }
}
```

`Based on Wildcards in methods alone, the methods can not be overloaded, there is simply not enough information for the compiler to achieve this`

With the example above, we can compare two Sum objects which have totally different types, that allows the method in sum to be very generic while still retaining type safety, since we know that the other wildcard parameter passed to the function will certainly be extending off of Number

Wildcards can also be bounded, meaning that an upper or lower boundary can be placed on what exactly the wildcard represents,

```
class Base<T extends Number> {
    T value;

    int extract() {
        // some implementation
    }
}

class LevelOne<T> extends Base<T> {
    // more members for child one
}
class LevelTwo<T> extends LevelOne<T> {
    // more members for child two
}
class LevelThree<T> extends LevelTwo<T> {
```

```
    // more members for child two
}

class Usage<T extends Base<Number>> {
    private T entry;
}

Usage<LevelOne<Number>> usageOne;
Usage<LevelTwo<Number>> usageTwo;
Usage<LevelThree<Number>> usageThree;
```

Upper boundary Upper boundary represents the highest class or type in the hierarchy that can be placed in place of the wildcard, every other child class derived from that boundary class is also allowed In the example below, the wildcard represents any derived child class of `LevelTwo`, including that class type as well, meaning that every object that can be replaced with the parent class type LevelTwo can be a parameter to to the `Usage<>` parameter the method.

```
public int method(Usage<? extends LevelTwo> elem) {
    // now one can use any class deriving from the upper boundary of
       LevelTwo, down to the bottom of the hierarchy - LevelTwo, LevelThree
}
```

Lower boundary Wildcards can also specify a lower boundary, meaning that the class specified as boundary is the lowest one in the hierarchy, and every other one must be a super class of the specified one.

```
public int method(Usage<? super LevelTwo> elem) {
    // now one can use any class starting from the lower boundary of
       LevelTwo, up to the very base class, Object - LevelTwo, LevelOne,
       Base and Object
}
```

Methods

As already seen methods can be generalized as well as classes, one can go a step further and have the method be generic without the class it is being defined in. This gives greater flexibility since the method, unlike the class is not linked to instance or object directly

```
class NoGenericClass {

    public <T extends Comparable<T>, V extends T> int inside(T value, V[]
       arr) {
        // the generic method provides two boundary restrictions, first
           the Type T must be comparable, and second the type V must be a
           subclass of T,
        // this implies that V itself is also comparable, however it can
           be of any type which is comparable, thus one can compare T one
           type
        // to V another type as long as both are subclasses and implement
           comparable
    }
}
```

Note how one can call the function inside()where T and V can be completely different types, for example TypeA and TypeB, and still invoke compareTo since both are guaranteed to be compatible, T extending off of comparable (being a child of comparable)and V extending off of T (being a child of T)

Constructors

Just as methods, constructors can also be defined generic even if their enclosing class is not, this provides an easier more flexible way to initialize the object or construct it from a wide variety of targets

```
class NoGenericClass {
    double value;

    public <T extends Number> NoGenericClass(T value) {
        this.value = value.doubleValue();
    }
}

// can be used to construct the instance with any type of identifier
   considered to be a number, and due to autoboxing one can use primitives
NoGenericClass g1 = new NoGenericClass(1);
NoGenericClass g2 = new NoGenericClass(10L);
NoGenericClass g3 = new NoGenericClass(5.5f);
NoGenericClass g4 = new NoGenericClass(6.5);
```

The example above shows how this class can be instantiated with many types of `Numbers` and due to autoboxing one can call it with primitives too to construct the target instance.

Inheritance

Inheriting or implementing a generic class has one very specific requirement, that is when a generic class inherits from another, or implements an interface, the same or more restricted boundaries have to be specified, for example, below one can see that the type of T is defined to be strictly a sub-class of `Comparable`, therefore all classes which inherit from the interface, must at the very least specify a boundary that is the same, there is no restriction on providing a more tight boundary, e.g. in case `Comparable` was a `superclass` of another interface one could specify that one instead, but not a `superclass` of `Comparable` as a boundary

```
interface MinMax<T extends Comparable<T>> {
    T min();
    T max();
}
class MyClass<T extends Comparable<T>> implements MinMax<T> {
    T[] vals;
```

Note the boundary is the same as in the interface above, that is important, if that is not the case the compiler will produce an error

Rawtypes

To accommodate older code, existing before the age of generics Java provides a way to specify a generic type without parameters, which will basically revoke all type safety, and checks, likely forcing the program to terminate at run-time instead.

```
GenericType g1 = new GenericType(); // not an error but the generic type
   is implicitly converted to Object, the top most class in the java class
   hierarchy
GenericType g2 = new GenericType<Integer>(); // this is a compliant
   definition of a variable which does not omit the type parameter and is
   safe to use
```

Inference

`MyClass<Integer, String> mcOb = new MyClass<>(98, "A String")` - Notice that the instance creation portion simply uses <>, which is an empty type argument list. This is referred to as the diamond operator. It tells the compiler to infer the type arguments needed by the constructor in the new expression. The principal advantage of this type-inference syntax is that it shortens what are sometimes quite long declaration statements.

Restrictions

- Type Parameters Can't Be Instantiated - this is due to the fact that the `T` is just a placeholder, there is no information for the compiler to know how to instantiate the generic placeholder

```
// Can't create an instance of T.
class Gen<T> {
    T ob;
    Gen() {
    ob = new T(); // Illegal!!!
    }
}
```

- No static members can be instantiated from the type placeholder declared in the enclosing class - this is actually a special case of the restriction above, there is no information for the compile to know how to construct the static member, since the type place holders are defined and declared `on per instance definition basis`

```
class Wrong<T> {
    // Wrong, no static variables of type T.
    static T ob;
    // Wrong, no static method can use T.
    static T getob() {
        return ob;
    }
}
```

- Creating arrays with generic types is also not possible, there are two cases shown below which are

```
class Gen<T extends Number> {
    T ob;
    T vals[]; // OK
    Gen(T o, T[] nums) {
        vals = new T[10]; // compile time error, not enough information on
           the type to create
        vals = nums; // can assign reference to existent array
    }
}
```

```
Gen<Integer> gens[] = new Gen<Integer>[10]; // compile time error, not
    enough type information
Gen<?> gens[] = new Gen<?>[10]; // this however is valid, due to type
    erasure
```

- A generic class cannot extend `Throwable`. This means that you cannot create generic exception classes.