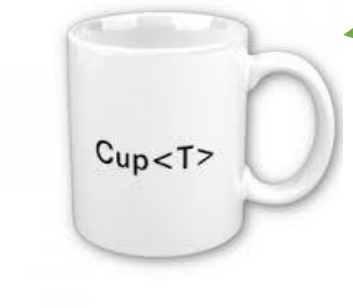
Generics

Generics

Generic programming is a style of computer programming, in which algorithms are written in terms of *types to-be-specified-later*. These Generics types are instantiated when needed, and *specific types* are provided as parameters.



Generics refers to the technique of writing the code for a *class* without specifying the data *type(s)* that the *class* works on. You specify the data type when you declare an instance of a Generic *class*. This allows a Generic *class* to be specialized for many different data *types* while only having to write the *class* once.

Why use Generics?

It would be nice if we could write one single **sort** method that could sort the elements in an Integer *array*, a String *array*, or an *array* of any **type** that supports ordering.
Java Generic *methods*, and Generic *classes* enable programmers to specify, with a single *method* declaration, a set of related *methods*, or with a single *class* declaration, a set of related *types*, respectively.

Generics also provide *compile-time* **type safety** that allows programmers to catch invalid *types* at compile time. Using the Java Generic concept, we might write a Generic *method* for sorting an *array* of objects, then invoke the Generic *method* with Integer *arrays*, Double *arrays*, String *arrays*, and so on, to sort the *array* elements.

Advantages of Java Generics

Type-safety:

The Java Generics programming is introduced in *J2SE 5* to deal with *type-safe* objects. Before Generics, we could store any *type* of *objects* in *collection* i.e. *non-generic*. Now Generics forces the java programmer to store specific *type* of *objects*.

Type casting:

There is no need to typecast the *object*.

Compile-Time Checking:

Generics *types* are checked at *compile time*, so the problem will not occur at *runtime*. The good programming strategy says that it is far better to handle the problem at *compile time* than *runtime*.

Java Generics Class - Example

We can define our own *classes* with Generics *type*. A Generics *type* is a *class* or *interface* that is parameterized over *types*. We use angle brackets (**<>**) to specify the type parameter.
To understand the benefit, let's say we have a simple *class* as:

Notice that while using this *class*, we have to use *type casting* and it can produce ClassCastException at *runtime*.

```
class GenericsTypeOld {
    private Object t;
    public Object get() {
        return t;
    }
    public void set(Object t) {
        this.t = t;
    }
    public static void main(String[] args) {
        GenericsTypeOld type = new GenericsTypeOld();
        type.set("Just testing string object :)");
        // type casting, error prone and can cause ClassCastException,
        // in case "t" didn't match (CASTING) type
        String str = (String) type.get();
    }
}
```

Java Generics Class - Example

We can define our own *classes* with Generics *type*. A Generics *type* is a *class* or *interface* that is parameterized over *types*. We use angle brackets (**<>**) to specify the type parameter.
To understand the benefit, let's say we have a simple *class* as:

Now we will use java Generic *class* to rewrite the same *class*. Notice the use of GenericsType *class* in the main method. We dn't need to do *type-casting* and we can remove ClassCastException at *runtime*. If we don't provide the type at the time of creation, the compiler will give a **warning**: "GenericsType is a raw type. References to generic type GenericsType<T> should be parameterized".
When we don't provide *type*, the *type* becomes Object and hence it's allowing both String and Integer *objects*, but we should always try to avoid this because we will have to use *type casting* while working on raw type that can produce *runtime* errors.

```
class GenericsType<T> {
    private T t;
    public T get() {
        return this.t;
    }
    public void set(T t1) {
        this.t = t1;
    }
    public static void main(String args[]) {
        GenericsType<String> type = new GenericsType();
        type.set("Ali"); //valid

        GenericsType type1 = new GenericsType(); //raw type
        type1.set("Ali"); //valid
        type1.set(123); //valid and autoboxing support
    }
}
```

Java Generic Interface

Comparable *interface* is a great example of Generics in *interfaces*, and it's written as:

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

In similar way, we can create Generic *interfaces* in java. We can also have multiple *type* parameters as in Map *interface*. Again, we can provide parameterized *value* to a parameterized *type*, for example:

```
new HashMap<String, List<String>>(); //is valid
```

Java Generic Type

Java Generic Type naming convention helps us to understand code easily, and having a naming convention is one of the best practices of java programming language. Generics also comes with its own naming conventions. Usually *type* parameter names are **single**, **uppercase** letters to make it easily distinguishable from java *variables*. The most commonly used *type* parameter names are:

- **E** – Element (used extensively by the Java Collections *Framework*, for example ArrayList, Set etc.)
- **K** – Key (Used in Map)
- **N** – Number
- **T** – Type
- **V** – Value (Used in Map)
- **S**,**U**,**V** etc. – 2nd, 3rd, 4th *types*

Java Generic Method

Sometimes we don't want the entire *class* to be parameterized, in that case we can create a single java Generics *method*.

Here is an example showing a java generic method:

The signature of the **areEquals** *method* shows how to use Generics *type* in *methods*. Also notice how to use this *method* in our java program. We can specify *type* while calling this *method,* or we can invoke it like a normal *method*. The Java compiler is smart enough to determine the *type* of the *variable* that is used. This facility is called *type inference*.

```
//Java Generic Method
public static <T> boolean areEquals(T firstValue, T secondValue) {
    return firstValue.equals(secondValue);
}
public static void main(String args[]) {
    // using integers
    boolean _integerAreEquals = areEquals(10, 10);
    // using string
    boolean _stringAreEquals = areEquals("Ali", 10);
    // printing the result
    System.out.println("10 equals 10: " + _integerAreEquals + "\n"
                    + "Ali equals 10: " + _stringAreEquals);
}
```

```
run:
10 equals 10: true
Ali equals 10: false
BUILD SUCCESSFUL (total time: 0 seconds)
```

Java Generics Bounded Type

Suppose we want to restrict the *type* of *objects* that can be used in the *parameterized type*, for example in a *method* that compares two *objects*, and we want to make sure that the accepted *objects* are comparable. In order to declare a *bounded type parameter*, list the *type* parameter's name, followed by the extends *keyword*, followed by its upper bound, similar like below method:

The invocation of this *method* is similar to unbounded *method* **except** that if we try to use any *class* that is not Comparable, it will throw compile time error.

```java
//Java Generic Method
public static <T extends Comparable<T>> int compare(T t1, T t2) {
    return t1.compareTo(t2);
}
```

Bounded type parameters can be used with *methods* as well as *classes* and *interfaces*. Java Generics supports multiple bounds also, i.e. **<T extends A & B & C>**. In this case A can be an *interface* or a *class*. If A is a *class* then B and C MUST be *interfaces*. We can't have more than one *class* in multiple bounds.

Wildcards

<?>

Java Generics Wildcards

A question mark, that is surrounded with angle brackets, <?> is the wildcard in Generics and represents an *unknown type*. The wildcard can be used as the *type* of a parameter, field, or local variable, and sometimes as a *return type*. We can't use wildcards while invoking a Generic *method* or instantiating a Generic *class*.

In the following sections, we will learn about upper bounded wildcards, lower bounded wildcards, and wildcard capture.

Upper Bounded Wildcard

Upper bounded wildcards are used to relax the restriction on the *type* of variable in a *method*. Suppose that we want to write a *method* that will return the sum of numbers in the list. Our implementation can be something like this:

```
public static double sum(List<Number> list) {
    double sum = 0;
    for (Number n : list) {
        sum += n.doubleValue();
    }
    return sum;
}
```

Now, the problem with this implementation is that it won't work with List of Integers or Doubles, because we know that List<Integer> and List<Double> are not related. This is where *upper bounded wildcard* is helpful.

Upper Bounded Wildcard

Upper bounded wildcards are used to relax the restriction on the *type* of variable in a *method*. Suppose that we want to write a *method* that will return the sum of numbers in the list. Our implementation can be something like this:

We use Generics wildcard with extends *keyword*, and the *upper bound class/interface* that will allow us to pass argument of upper bound or it's *subclasses types*.

```
public static double sum(List<? extends Number> list) {
    double sum = 0;
    for (Number n : list) {
        sum += n.doubleValue();
    }
    return sum;
}
}
```

Unbounded Wildcard

Sometimes we have a situation, where we want our Generic *method* to be working with all *types*. In this case *unbounded wildcard* can be used.
It looks like <?>

```
public static void printData(List<?> list) {
    for (Object obj : list) {
        System.out.print(obj + "::");
    }
}
```

We can provide List<String> or List<Integer> or any other *type* of *Object list* argument to the **printData** *method*. Similarly to *upper bound list*, we are not allowed to add anything to the list.

Lower bounded Wildcard

Suppose that we want to add Integers to a list of integers in a *method*. We can keep the argument *type* as List<Integer> but it will be tied up with Integers whereas List<Number> and List<Object> can also hold integers. We can use *lower bound wildcard* to achieve this. We use Generics *wildcard* (?) with super *keyword* and *lower bound class* to achieve this.

```
public static void addIntegers(List<? super Integer> list) {
    list.add(new Integer(50));
}
```

We can pass *lower bound* or any super *type* of lower bound as an argument in this case. **The java compiler allows us to add *lower bound object types* to the list.**

Good to know

Generics doesn't support *sub-typing*, because that would cause issues in achieving *type safety*. That's why List<**T**> is not considered as a *subtype* of List<**S**> where **S** is the *super-type* of **T**.
In order to understand why it's not allowed, let's see what could have happened if it had been supported

```
List<Long> listLong = new ArrayList<Long>();
listLong.add(Long.valueOf(10));
List<Number> listNumbers = listLong; // compiler error
listNumbers.add(Double.valueOf(1.23));
```

As you can see from the above code, **IF** Generics had been supporting *sub-typing*, we could easily have added a Double to the list of Long, and that would have caused ClassCastException at *runtime* while traversing the list of Long.

Good to know

We are not allowed to create Generic *arrays*, because *arrays* carry *type* information of it's elements at *runtime*. This information is used at *runtime* to throw ArrayStoreException when the *type* of the element doesn't match the defined *type*. Since Generics *type* information gets erased at *runtime* by **Type Erasure**, the *array store check* would have accepted the assignment that should NOT have been allowed. Let's understand this with a simple example code

```
List<Integer>[] intList = new List<Integer>[5]; // compile error
Object[] objArray = intList;
List<Double> doubleList = new ArrayList<Double>();
doubleList.add(Double.valueOf(1.23));
// this should fail but it would pass,
// because at runtime intList and doubleList both are just Lists
objArray[0] = doubleList;
```

Arrays are covariant by nature i.e. **S[]** is a *subtype* of **T[]** whenever **S** is a *subtype* of **T**. But Generics doesn't support covariance or *sub-typing* as we saw in the previous slide. If we had been allowed to create Generic *arrays*, the *type* erasure would not cause ArrayStoreException even though both *types* are not related.

Questions?