Generics

All we know is "*All instances of a any class shares the same java.lang.Class object of that type of class*"

e.g)

Student a = new Student();

Student b = new Student();

Then a.getClass() == b.getClass() is true.

Now assume

Teacher t = new Teacher();

without generics the below is possible.

Class studentClassRef = t.getClass();

But this is wrong now ..?

e.g) public void printStudentClassInfo(Class studentClassRef) {} can be called with Teacher.class

This can be avoided using generics.

Class<Student> studentClassRef = t.getClass(); //Compilation error.

Now what is T ?? T is type parameters (also called type variables); delimited by angle brackets (<>), follows the class name.  
T is just a symbol, like a variable name (can be any name) declared during writing of the class file. Later that T will be substituted with  
valid Class name during initialization (HashMap<String> map = new HashMap<String>();)

e.g) class name<T1, T2, ..., Tn>

So Class<T> represents a class object of specific class type 'T'.

Assume that your class methods has to work with unknown type parameters like below

/\*\*

\* Generic version of the Car class.

\* @param <T> the type of the value

\*/

public class Car<T> {

// T stands for "Type"

private T t;

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

public T get() { return t; }

}

Here T can be used as String type as **CarName**

OR T can be used as Integer type as **modelNumber**,

OR T can be used as Object type as **valid car instance**.

Now here the above is the simple POJO which can be used differently at runtime.  
Collections e.g) List, Set, Hashmap are best examples which will work with different objects as per the declaration of T, but once we declared T as String  
e.g) HashMap<String> map = new HashMap<String>(); Then it will only accept String Class instance objects.

**Generic Methods**

Generic methods are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Static and non-static generic methods are allowed, as well as generic class constructors.

The syntax for a generic method includes a type parameter, inside angle brackets, and appears before the method's return type. For generic methods, the type parameter section must appear before the method's return type.

class Util {

// Generic static method

public static <K, V, Z, Y> boolean compare(Pair<K, V> p1, Pair<Z, Y> p2) {

return p1.getKey().equals(p2.getKey()) &&

p1.getValue().equals(p2.getValue());

}

}

class Pair<K, V> {

private K key;

private V value;

}

Here <K, V, Z, Y> is the declaration of types used in the method arguments which should before the return type which is boolean here.

In the below; type declaration <T> is not required at method level, since it is already declared at class level.

class MyClass<T> {

private T myMethod(T a){

return a;

}

}

But below is wrong as class-level type parameters K, V, Z, and Y cannot be used in a static context (static method here).

class Util <K, V, Z, Y>{

// Generic static method

public static boolean compare(Pair<K, V> p1, Pair<Z, Y> p2) {

return p1.getKey().equals(p2.getKey()) &&

p1.getValue().equals(p2.getValue());

}

}

**OTHER VALID SCENARIOS ARE**

class MyClass<T> {

//Type declaration <T> already done at class level

private T myMethod(T a){

return a;

class MyClass<T> {

//Type declaration <T> already done at class level

private T myMethod(T a){

return a;

}

//<T> is overriding the T declared at Class level;

//So There is no ClassCastException though a is not the type of T declared at MyClass<T>.

private <T> T myMethod1(Object a){

return (T) a;

}

//Runtime ClassCastException will be thrown if a is not the type T (MyClass<T>).

private T myMethod1(Object a){

return (T) a;

}

// No ClassCastException

// MyClass<String> obj= new MyClass<String>();

// obj.myMethod2(Integer.valueOf("1"));

// Since type T is redefined at this method level.

private <T> T myMethod2(T a){

return a;

}

// No ClassCastException for the below

// MyClass<String> o= new MyClass<String>();

// o.myMethod3(Integer.valueOf("1").getClass())

// Since <T> is undefined within this method;

// And MyClass<T> don't have impact here

private <T> T myMethod3(Class a){

return (T) a;

}

// ClassCastException for o.myMethod3(Integer.valueOf("1").getClass())

// Should be o.myMethod3(String.valueOf("1").getClass())

private T myMethod3(Class a){

return (T) a;

}

// Class<T> a :: a is Class object of type T

//<T> is overriding of class level type declaration;

private <T> Class<T> myMethod4(Class<T> a){

return a;

}

}

And finally Static method always needs explicit declaration; It wont derive from class level Class. This is because of Class level T is bound with instance.

Also read [Restrictions on Generics](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#createStatic)

[Wildcards and Subtyping](https://docs.oracle.com/javase/tutorial/java/generics/subtyping.html)

[type argument for a generic method](https://stackoverflow.com/questions/4829631/unusual-generic-syntax-arrays-stringaslist)