Limitation of a reference of generic class with a list argument

When we declare a variable or method parameter with:

- List<**Student**>

Only List subtypes with Student elements can be assigned to this variable or method argument.

We can't assign a list of Student subtypes to this!

The generic method

For a method, type parameters are placed after any modifiers and before the method's return type.

The type parameter can be referenced in method parameters, or as the method return type, or in the method code block, much as we saw a class's type parameter can be used.

A generic method can be used for collections with type arguments, as we just saw, to allow for variability of the elements in the collection, without using a raw version of the collection.

```
public <T> String myMethod(T input) {
    return input.toString();
}
```

The generic method

A generic method can be used for static methods on a generic class, because static methods can't use class type parameters.

A generic method can be used on a non-generic class, to enforce type rules on a specific method.

The generic method type parameter is separate from a generic class type parameter.

In fact, if you've used T for both, the T declared on the method means a different type, than the T for the class.

```
public <T> String myMethod(T input) {
    return input.toString();
}
```

Type Parameters, Type Arguments and using a Wildcard

A **type parameter** is a generic class, or generic method's declaration of the type.

In both of these examples, T is said to be the type parameter.

You can bind a type parameter with the use of the **extends** keyword, to specify an **upper bound**.

Generic class	Generic Method
`public class Team<T> {}`	`public <T> void doSomething(T t) {}`

Type Parameters, Type Arguments and using a Wildcard

A **type argument** declares the type to be used, and is specified in a type reference, such as a local variable reference, method parameter declaration, or field declaration.

In this example, BaseballPlayer is the type argument for the Team class.

Generic class
`Team<BaseballPlayer> team = new Team<>();`

Type Parameters, Type Arguments and using a Wildcard

A **wildcard** can only be used in a **type argument**, not in the type parameter declaration.

A wildcard is represented with the **?** character.

A wildcard means the type is **unknown**.

For this reason, a wildcard **limits what you can do**, when you specify a type this way.

List declaration using a wildcard
`List<?> unknownList;`

Type Parameters, Type Arguments and using a Wildcard

A wild card can't be used in an instantiation of a generic class.

The code shown here is invalid.

Invalid! You can’t use a wildcard in an instantiation expression

```
var myList = new ArrayList<?>();
```

Type Parameters, Type Arguments and using a Wildcard

A wildcard can be unbounded, or alternately, specify either an upper bound or lower bound.

You **can't specify both** an **upper** bound and a **lower** bound, in the same declaration.

Argument	Example	Description
unbounded	`List<?>`	A List of any type can be passed or assigned to a List using this wildcard.
upper bound	`List<? extends Student>`	A list containing any type that is a Student or a **sub type** of Student can be assigned or passed to an argument specifying this wildcard.
lower bound	`List<? super LPAStudent>`	A list containing any type that is an LPAStudent or a **super type** of LPAStudent, so in our case, that would be Student AND Object.

Type Erasure

Generics exist to enforce tighter type checks, at compile time.

The compiler transforms a generic class into a typed class, meaning the byte code, or class file, contains no type parameters.

Everywhere a type parameter is used in a class, it gets replaced with either the type Object, if no upper bound was specified, or the upper bound type itself.

This transformation process is called type erasure, because the T parameter (or S, U, V), is erased, or replaced with a true type.

Why is this important?

Understanding how type erasure works for overloaded methods, may be important.