- You cannot make arrays of parameterized type.
- You cannot cast to parameterized type.
- You cannot use an instance of a parameterized type.

It almost always relieves you of the need to explicitly pass types to the method parameter.

This may fail, for example, when the method takes no argumenets.

```
Collections.<String>emptySet()
```

The declaration of a generic is never actually expanded - not anywhere.

```
List<Object> l = newArrayList<Object>();
```

Then you can refer to it as `List<?>`.

Note that you cannot parameterize constructor call with ?.

`List<?>` guarantees `get()` will have a return type of parameterized type, even though it is unknown to the compiler. Raw `List` does not.

`List<?>` effectively read-only because the compiler will not allow invocation of methods that have a parameter of the unknown type, because if can't do type checking on what you pass it.

It's an exception to the general rule that objects of unknown type cannot use generic methods.

Sometimes the compiler can infer the unknown type of a wildcard as a type argument to a generic method.

Only when a single type variable is used to express a relationship between two parameters, or between a parameter and a return value.

```
name extends type 1 & type 2 & ...
name super type 1 & type 2 & ...
```

Name is either wildcard or parameter name like T.

`typeN` can be parameterized, too.

- Raw (unparameterized) types are allowed for pre-1.5 compatibility.
- Parameterized types can be passed to methods with raw type arguments, also for pre-1.5 compatibility.
- Because of type erasure, array types cannot be parameterized. No runtimecheck for `ArrayStoreException` is possible.

`varargs` implicitly create an array, so one method invocation may succeed and another fail depending on whether the `vararg` type passed is parameterized.

`public static T foo(T...vals)` will fail for parameterized `T`.