As you know, **generics** enable types to be parameters when defining classes (or interfaces) and methods. Parameterized types make it possible to re-use the same code while processing different concrete types.

## Reusing code with generics

Let's consider a generic class named GenericType that stores a value of "some type".

class GenericType<T> {   
   
    private T t;  
   
    public GenericType(T t) {  
        this.t = t;  
    }  
   
    public T get() {  
        return t;  
    }  
}

It is possible to create an object with a concrete type (e.g., String):

GenericType<String> instance1 = new GenericType<>("abc");  
String str = instance1.get();

You can also create instances with other types (Integer, Character) and then invoke the get method to take the internal field. So, generics allow you to use the same class and methods for processing different types.

## Reusing code with Object

But there is another way to reuse code. If we declare the class field as Object, we can assign a value of any reference type to it. This approach has been widely used before Java 5.

The class NonGenericClass demonstrates it.

class NonGenericClass {  
   
private Object val;  
   
    public NonGenericClass(Object val) {  
        this.val = val;  
    }  
   
    public Object get() {  
        return val;  
    }  
}

Now we can create an instance of this type with the same string as in the previous example (see GenericType).

NonGenericClass instance2 = new NonGenericClass("abc");

It is also possible to create an instance passing a value of type Integer or Character.

That is how you can reuse the same class with an Object field to store different types in the same way.

## The advantage of generics: from run-time to compile-time

After an invocation of the method get() we obtain an Object, not a String or an Integer. We cannot get a string directly from the method.

NonGenericClass instance2 = new NonGenericClass("abc");  
String str = instance2.get(); // Compile-time error: Incompatible types

To get the string back, we should perform an explicit type-casting to the String class.

String str = (String) instance2.get(); // "abc"

Of course, it works, but what if the instance does not store a string at all? Since it is declared as an Object, the field value can keep any type. If this is the case, the code throws an exception. Here is an example:

NonGenericClass instance3 = new NonGenericClass(123);  
String str = (String) instance3.get(); // throws java.lang.ClassCastException

Now we can see the main advantage of generics over the class Object to re-use code. There is no need to perform an explicit type-casting and, as a consequence, we never get the runtime exception. If we do something wrong, we can see it at the compile-time.

GenericType<String> instance4 = new GenericType<>("abc");  
          
String str = instance4.get(); // There is no type-casting here  
Integer num = instance4.get(); // It does not compile

A compile-time error is detected by the programmer, not a user of the program. This makes generics both flexible and safe. At the same time, working with Object that requires type-casting is error-prone. It's better to let the compiler take care of it.

## Generics without specifying a type argument

When you create an instance of a generic class, you have a possibility not to specify an argument type at all.

GenericType instance5 = new GenericType("my-string");

In this case, the field of the class is Object, and the method get returns an Object as well.

It is the same as the following line:

GenericType<Object> instance5 = new GenericType<>("abc"); // it is parameterized with Object

Usually, you will not use generics parameterized by Object because it has the same problems as presented above. Just remember that this possibility exists.

## Conclusion

Both generics and Object allows you to write a generalized code. Using Object , however, may need explicit type-casting that is error-prone. Generics provide type-safety by shifting more type checking responsibilities to the Java compiler.