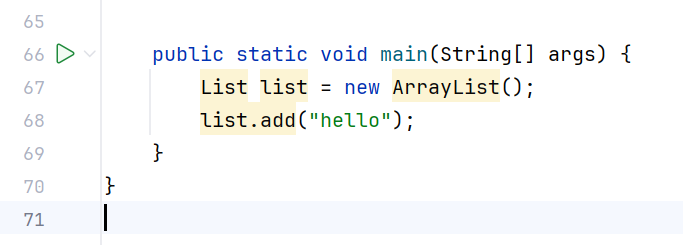
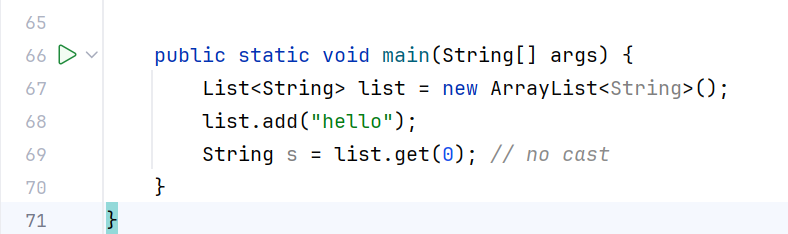
**Generics In Java**

It enables types (classes and interfaces) to be parameters when defining classes, interfaces, and methods. Much like the more familiar formal parameters used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

* Stronger type checks at compile time.  
  A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.
* Helps in providing a parametrized type to interfaces, classes.
* Elimination of casts.  
  The following code snippet without generics requires casting:



When re-written to use generics, the code does not require casting:



The most used type of parameter names is:

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

**Bounded Type Parameters**

There may be times when you want to restrict the types that can be used as type arguments in a parameterized type.

For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.

To declare a bounded type of parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound, which in this example is Number.

Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

**Wildcards in Generics**

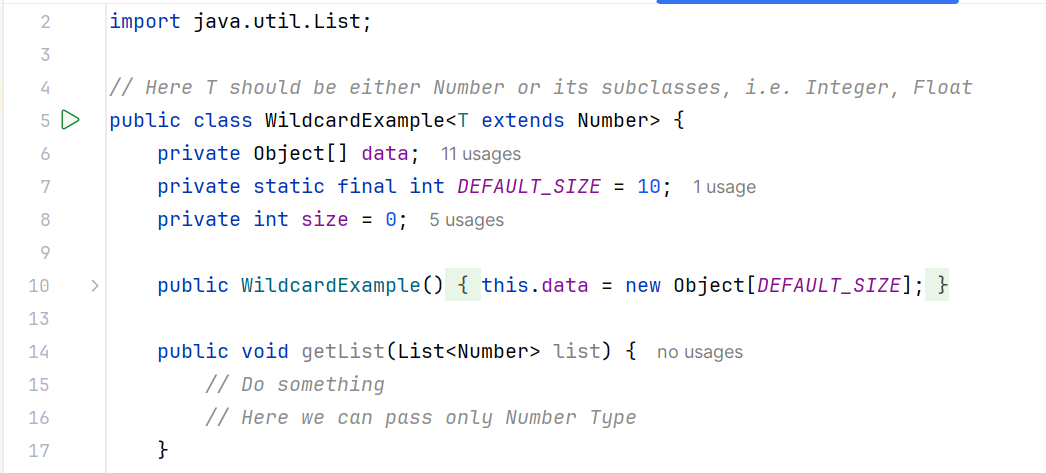
* The question mark (?) is known as the wildcard in generic programming.
* It represents an unknown type.
* The wildcard can be used in a variety of situations such as the type of a parameter, field, or local variable; sometimes as a return type.

**Types of Wildcards**

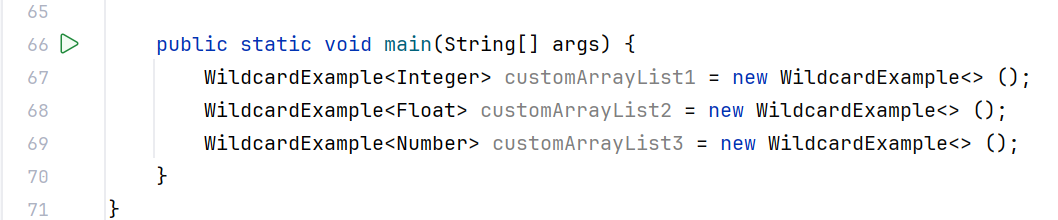
1. **Upper Bounded Wildcards:**

These wildcards can be used when you want to relax the restrictions on a variable. For example, say you want to write a method that works on List < Integer >, List < Double >, and List < Number >, you can do this using an upper bounded wildcard.

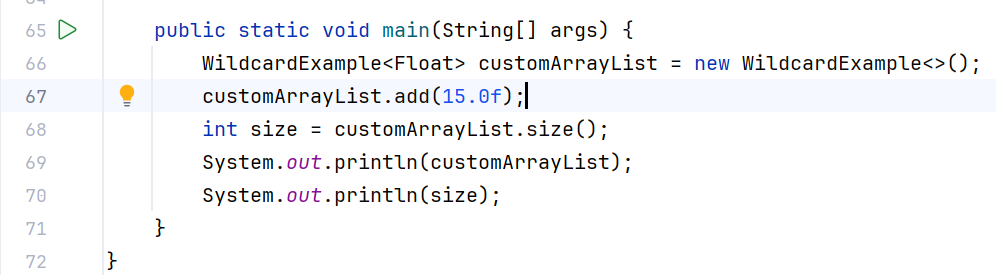
To declare an upper-bounded wildcard, use the wildcard character (‘?’), followed by the extends keyword, followed by its upper bound.



Example: public class WildcardExample<T extends Number> {}

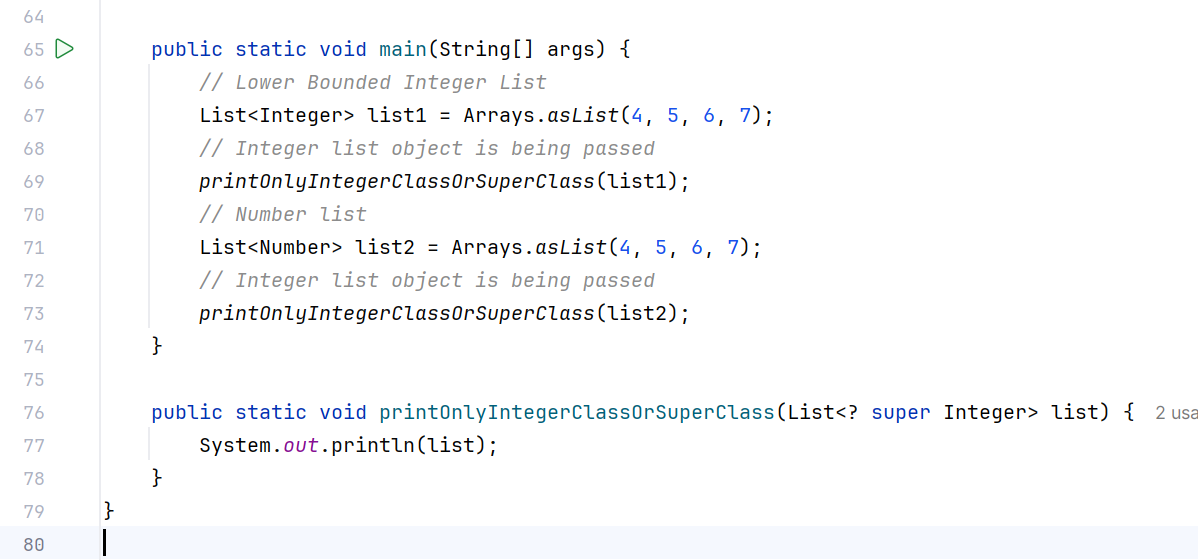


Are all correct.



1. **Lower Bounded Wildcards**

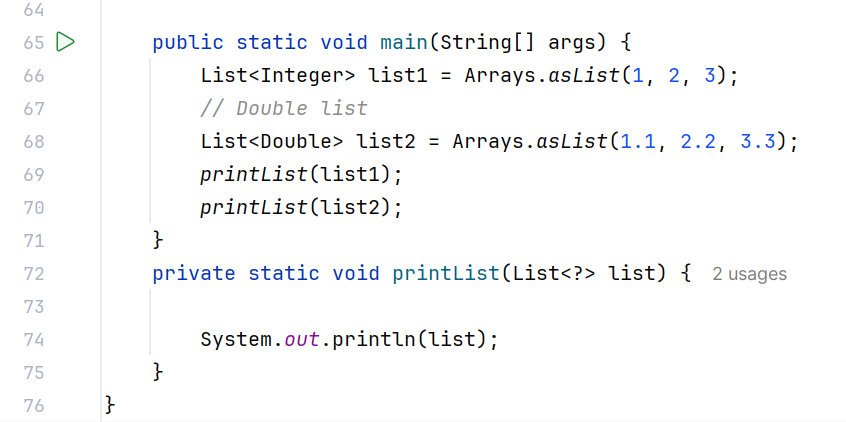
It is expressed using the wildcard character (‘?’), followed by the super keyword, followed by its lower bound: <? super A>.



**Unbounded Wildcards in Generics**

This wildcard type is specified using the wildcard character (?), for example, List. This is called a list of unknown types. These are useful in the following cases:

* When writing a method that can be employed using functionality provided in Object class.
* When the code is using methods in the generic class that does not depend on the type of parameter.

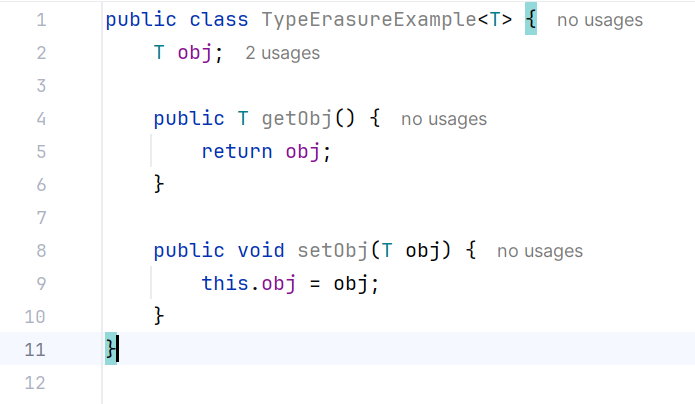


Type Erasure in Generics

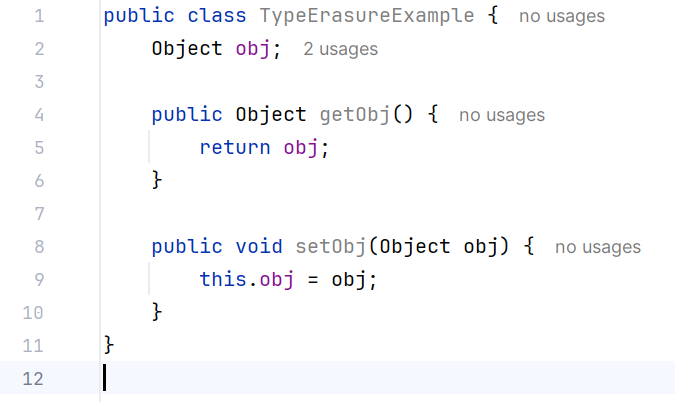
The Java compiler applies type erasure in generics to:

* Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded. It means, if you have an unbounded type of parameter, that will be replaced by Object while compiling the code.
* The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
* Insert type casts if necessary to preserve type safety.
* Generate bridge methods to preserve polymorphism in extended generic types.

Example: Here, type parameter T is unbounded



Thus, Java compiler will replace it with Object.



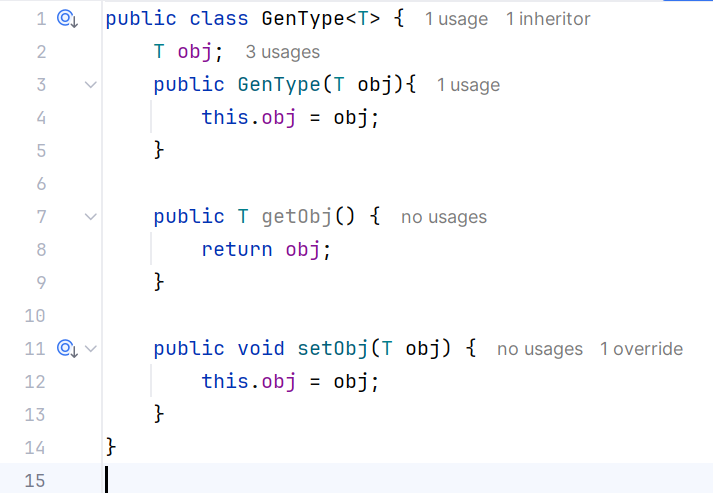
Similarly, if the type of parameter is bounded, it will be replaced by Bounded parameter type after compilation.

Bridge Methods

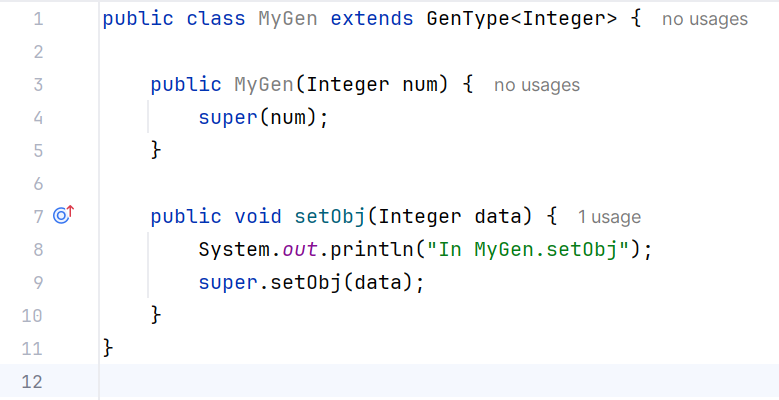
Two classes before and after type erasure:

Before:

CLASS GenType<T>

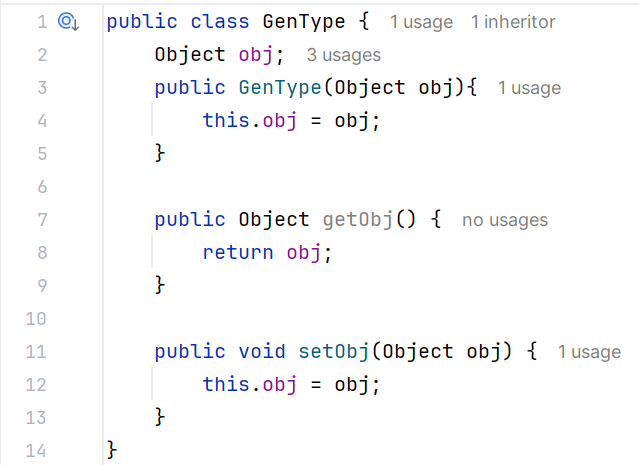


CLASS MyGen<Integer>

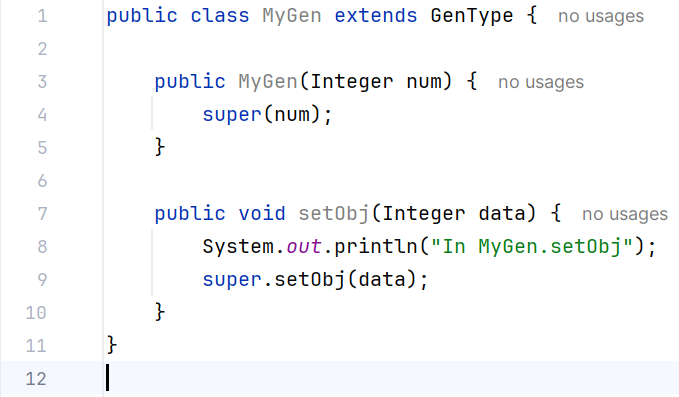


After:

CLASS GenType

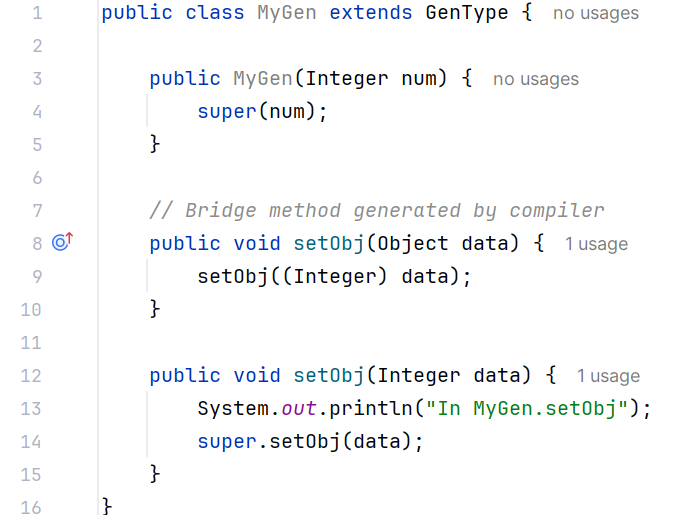


CLASS MyGen



To solve this problem and preserve the polymorphism of generic types after type erasure, JAVA compiler generates a bridge method to ensure that subtyping works as expected.

For MyGen class, the compiler generates the following bridge method for setObj.



Restrictions on Generics

To use Java generics effectively, you must consider the following restrictions:

* Cannot Instantiate Generic Types with Primitive Types
* Cannot Create Instances of Type Parameters
* Cannot Declare Static Fields Whose Types are Type Parameters
* Cannot Use Casts or instanceof With Parameterized Types
* Cannot Create Arrays of Parameterized Types
* Cannot Create, Catch, or Throw Objects of Parameterized Types
* Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type