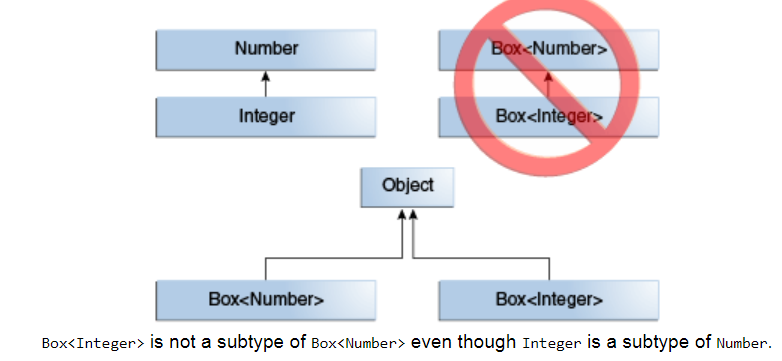
1. **Generics**: With generic we can define an algorithm independent of specific type of data.
   1. Generics let us define parameterized classes, interfaces and methods where type of data they operate upon can be passed as parameter.
   2. Pre-generic code generalized using object references but it can’t guarantee type safety.
2. **Type erasure**: Type erasure can be explained as the process of enforcing type constraints only at compile time and discarding the element type information at runtime. The compiler ensures type safety of our code and prevents runtime errors.
   1. **Bridge Methods**: To solve the edge case above, the compiler sometimes creates a bridge method. This is a synthetic method created by the Java compiler while compiling a class or interface that extends a parameterized class or implements a parameterized interface where method signatures may be slightly different or ambiguous.

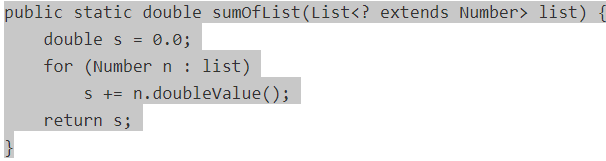
<https://docs.oracle.com/javase/tutorial/java/generics/bridgeMethods.html>

1. **Points**:
   1. Generic works only with reference types not with primitive types.
   2. The references of two different generic types are not compatible with each other.
      1. iob != strobe, but without generic it compiles.
      2. The compiler turns all generic uses into casts to the right type. This is to maintain backwards compatibility with previous JVM runtimes. Primitives won’t work here.

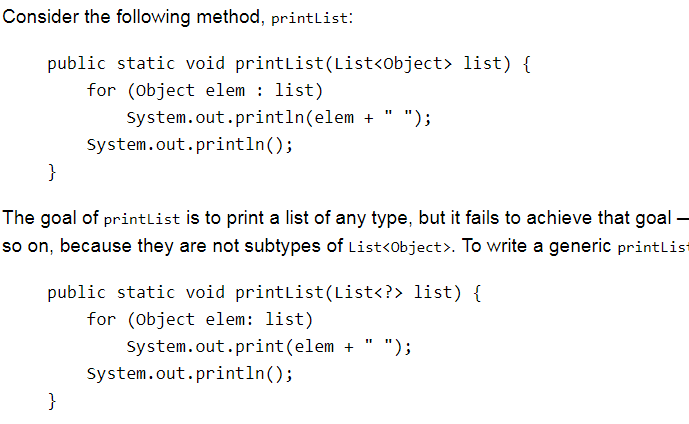


* 1. Using generics we don’t need to explicitly cast objects to ensure type safety.
  2. Casting of instance of generic class works if the two are compatible and their type argument are same.

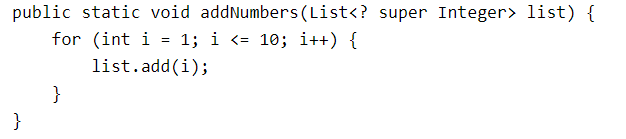
1. **Bounded types**: Bounded type parameters are used to restrict the types that can be passed as parameters.
   1. <T extends superClass>
   2. <T extends superclass & interface> [T can be bounded by multiple interfaces but one class]
2. **Wildcard**: In generic code, the question mark (?), called the wildcard, represents an unknown type. The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable; sometimes as a return type (though it is better programming practice to be more specific). The wildcard is never used as a type argument for a generic method invocation, a generic class instance creation, or a supertype.
   1. **Upper bound**: You can use an upper bounded wildcard 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 achieve this by using an upper bounded wildcard.



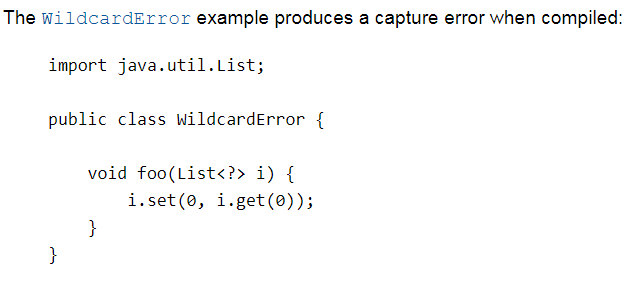
* 1. **Unbounded**: The unbounded wildcard type is specified using the wildcard character (?), for example, List<?>. This is called a list of unknown type. There are two scenarios where an unbounded wildcard is a useful approach:
     1. If you are writing a method that can be implemented using functionality provided in the Object class.
     2. When the code is using methods in the generic class that don't depend on the type parameter. For example, List.size or List.clear. In fact, Class<?> is so often used because most of the methods in Class<T> do not depend on T.



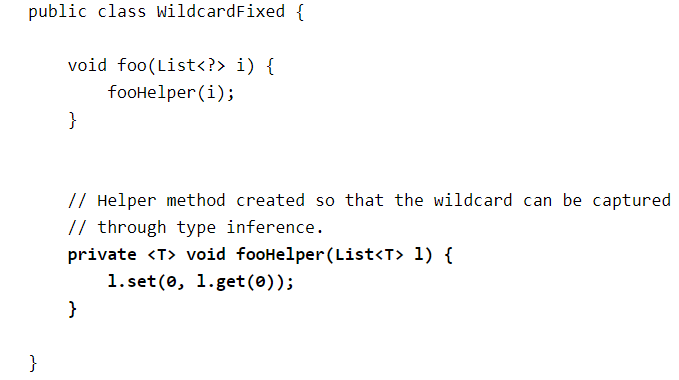
* 1. **Lower bound**: lower bounded wildcard restricts the unknown type to be a specific type or a super type of that type. A lower bounded wildcard is expressed using the wildcard character ('?'), following by the super keyword, followed by its lower bound: <? super A>.
     1. You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.
     2. To write the method that works on lists of Integer and the supertypes of Integer, such as Integer, Number, and Object, you would specify List<? super Integer>. The term List<Integer>is more restrictive than List<? super Integer> because the former matches a list of type Integer only, whereas the latter matches a list of any type that is a supertype of Integer.



* 1. **Wildcard and subtyping:** In some cases, the compiler infers the type of a wildcard. For example, a list may be defined as List<?> but, when evaluating an expression, the compiler infers a particular type from the code. This scenario is known as *wildcard capture*.
     1. the compiler processes the i input parameter as being of type Object. When the foo method invokes [List.set(int, E)](https://docs.oracle.com/javase/8/docs/api/java/util/List.html" \l "set-int-E-" \t "_blank), the compiler is not able to confirm the type of object that is being inserted into the list, and an error is produced.



* + 1. **Helper methods**: You can fix it by writing a *private helper method* which captures the wildcard. In this case, you can work around the problem by creating the private helper method, fooHelper, as shown in [WildcardFixed](https://docs.oracle.com/javase/tutorial/java/generics/examples/WildcardFixed.java" \t "_blank):



1. **Wildcard Guidelines**:
   1. An "in" variable is defined with an upper bounded wildcard, using the extends keyword.
   2. An "out" variable is defined with a lower bounded wildcard, using the super keyword.
   3. In the case where the "in" variable can be accessed using methods defined in the Object class, use an unbounded wildcard.
   4. In the case where the code needs to access the variable as both an "in" and an "out" variable, do not use a wildcard.
2. **Generic methods**: It’s possible to define generic method that defines one or more generic type variable of its own. Also a non-generic class can have generic methods.
   1. Ex: <T extends Comparable<T>, V extends T> Boolean isIn(Tx, V[] y)
   2. Class.<Integer, Integer> isIn(2, nums)
3. **Generic interface**: **interface** MinMax <T **extends** Comparable<T>>
   1. **class** MyMinMax<T **extends** Comparable<T>> **implements** MinMax<T>
   2. Since an interface must be implemented by a concrete class that class must specify the same bound. In general implementing class has to be generic at least to the extend of interface generic parameter.
4. **Ambiguity errors**: Ambiguity error occurs when two different generic declaration resolve to same erased type.
5. **Restrictions**:
   1. Static variables can’t be of generic type.
   2. We can’t declare instantiate an array whose type is a type parameter.
      1. Arrays are said to be covariant which basically means that, given the subtyping rules of Java, an array of type T[] may contain elements of type T or any subtype of T.
      2. The subtyping rules of Java also state that an array S[] is a subtype of the array T[] if S is a subtype of T.
   3. We can’t create an array of type specific generic reference.
   4. Generic class cannot extend Throwable. We cannot create generic exception class.