# 15. Understanding Generics

**Exam Objectives**

1. Candidates are also expected to use Generics, including wildcards.

## 15.1 Introduction to Collections and Generics

The topic of generics is actually entirely independent from collections because generics is a feature of the Java language while collections is just a set of classes in the Java SDK. Generally, however, Java programmers encounter generics for the first time while using collections. Also, making collections type safe was one of the reasons generics were introduced in Java.

Although the OCP exam does not focus on generics directly, it expects you to know a good deal about them in relation with Java collections. For this reason, I will discuss the basics of collection and generics at the same time in this chapter.

### 15.1.1 Using ArrayList instead of arrays

Processing multiple objects of the same kind in the same way is often a requirement in applications. Printing the names of all the Employees in a list of Employees, computing interest for a list of accounts, or something as simple as computing the average of a list of numbers, require you do deal with a list of objects instead of one single object. You have already seen a data structure that is capable of holding multiple objects of the same kind - array. An array lets you hold references to multiple objects of a type and pass them around as a bunch. However, arrays have a couple of limitations. First, an array cannot grow or shrink in length after it is created. If you create an array of 10 employees and later find out you have 11 employees to hold, you will need to create a new array with 11 slots. You can't just add one more slot in an existing array. Second, inserting a value in the middle of an array is a bit difficult. If you want to insert an element just before the last element, you will have to first shift the last element to the next position and then put the value. Imagine if you want to insert an element in the middle of the list!

Although both of the limitations can be overcome by writing some extra code, these requirements are so common that writing the same code over and over again is just not a good idea. `java.util.ArrayList` is a class that provides these, and several other, features straight out of the box. The following is an example that shows how easy it is to manage a list of objects using an `ArrayList`:

```java
import java.util.ArrayList;
public class TestClass{
  public static void main(String[] args){
      ArrayList al = new ArrayList();

      al.add("alice"); //[alice]
      al.add("bob"); //[alice, bob]
      al.add("charlie"); //[alice, bob, charlie]

      al.add(2, "david"); //[alice, bob, david, charlie]

      al.remove(0); //[bod, david, charlie]

      for(Object o : al){  //process objects in the list
         String name = (String) o;
```

```
            System.out.println(name+" "+name.length());
        }

        //dump contents of the list
        System.out.println("All names: "+al);
    }
}
```

You will get a few warning messages when you compile the above program. Ignore them for now. It prints the following output when run:

```
bob 3
david 5
charlie 7
All names: [bob, david, charlie]
```

The above program illustrates the basics of using an `ArrayList`, i.e., adding objects, removing, iterating over them, and printing the contents of the `ArrayList`. I suggest you write a program that does the same thing using an array of strings instead of an `ArrayList`. This will give you an appreciation for the value that `ArrayList` adds over a raw array.

I showed you the above code just to give you a glimpse of one of the readymade classes in the Java library. There is a lot more that you can do with it but before we get to that, you need to understand the bigger picture, the grand scheme of things, into which classes such as `ArrayList` fit.

### Collections framework and Collections API

`ArrayList` belongs to a category of classes called "**collections**". A "**collection**" is nothing but a group of objects held up together in some form. You can visualize a collection as a bag in which you put several things. Just like you use a bag to take various grocery items from a store to your home, you put the objects in a collection so that you can pass them around to other objects and methods. A bag knows nothing about the items themselves. It merely holds the items. You can add items to it and take items out of it. It doesn't care about the order in which you put the items or whether there are any duplicates or how they are stored inside the bag. The behavior of a bag is captured by an interface called `Collection`, which exists in the `java.util` package.
Now, what if you want a collection that ensures objects can be retrieved from it in the same order in which they were added to it? Or what if you want a collection that does not allow duplicate objects? You can think of several features that can be added on top of a basic collection. As you have already learnt previously, subclassing/sub-interfacing is the way you extend the behavior of an existing class or interface. The Java standard library takes the same route here and extends `java.util.Collection` interface to provide several interfaces with various features. In fact, the Java standard library provides a huge tree of interfaces along with classes that implement those interfaces. These classes and interfaces are collectively known as the "**collections framework**" and also as the "**Collections API**".

So what has `ArrayList` got to do with this, you ask? Well, `ArrayList` is a class that implements one of the specialized collection interfaces called `List`. `java.util.List` **extends** `java.util.Collection` to add the behavior of ordering on top of a simple collection. A `List` is supposed to remember the order in which objects are added to the collection. Okay, so that takes care of the "List" part of ArrayList, what about the "Array" part? Again, as you have learnt in previous chapters, an interface merely describes a behavior. It doesn't actually implement that behavior. You need a class to implement the behavior. In this case, `ArrayList` is the class that implements the behavior described by `java.util.List` and to implement this behavior, it uses an array. Hence the name **ArrayList**. Remember I talked about writing code to overcome the limitation of an array that it is not flexible enough to increase in size automatically? `ArrayList` contains that code. When you add an object to an `ArrayList`, it checks whether there is space left in the array. If not, it allocates a new array with a bigger length, copies the elements from the old array to this new array, and then adds the passed object to the new array. Similarly, it contains code for inserting an object in the middle of the array. All this is transparent to the user of an `ArrayList`. A user simply invokes methods such as `add` and `remove` on an `ArrayList` without having any knowledge of the array that is being used inside to store the object.

## Generics

Now, regarding the warning messages that I asked you to ignore. Observe the `for` loop in the above code. The type of the loop variable `o` is `Object` (and not `String`). To invoke the `length` method of `String` on `o`, we need to cast `o` to `String`. This is because, in the same way that a bag is unaware of what is inside of it, so too does the `ArrayList` object have no knowledge about the type of the objects that have been added to it. It is the responsibility of the programmer to cast the objects that they retrieve from the list to appropriate types. In this program, we know that we have added `String`s to this list and so we know that we can cast the object that we retrieve from this list to `String`. But what if we were simply given a pre-populated list as an argument? Since the list would have been populated by another class written by someone else, what guarantee would we have about the type of objects we find in the list? None, actually. As you are aware, if we try to cast an object to an inappropriate type, we get a `ClassCastException` at run time. Getting a `ClassCastException` at run time would be a coding mistake that will be discovered only at run time. Discovering coding mistakes at run time is not a good thing and the compiler is only trying to warn us about this potential issue by generating warning messages while compiling our code.

The first warning message that it prints out is, "`Warning: unchecked call to add(E) as a member of the raw type java.util.ArrayList`" for the code `al.add("alice");` at line 6. It prints the same warning every time an object is added to the list. The compiler is trying to tell us that it does not know what kind of objects this `ArrayList` is supposed to hold and that it has no way of checking what we are adding to this list. By printing out the warning, the compiler is telling us that it will not be held responsible for the objects that are being put in this list. Whoever wants to retrieve objects from this list must already know what type of objects are present inside the list and must cast those objects appropriately upon retrieval at their own risk. In other words, this list is basically **"type unsafe"**. It is unsafe because it depends on assumptions made by humans about the contents of the list. These assumptions are not checked or validated by the compiler for their truthfulness. One can put any kind of object in the list and others will not realize until they are hit with a ClassCastException at run time when they try to cast the retrieved object to its expected type.

Java solves this problem by allowing us to specify the type of objects that we are going to store in a list while declaring the type of the list. If, instead of writing ArrayList al = new ArrayList();, we write ArrayList<String> al = new ArrayList<String> (); all the warning messages go away.

This statement does two things: it declares a reference variable al of type ArrayList<String> and it creates an object of type ArrayList<String> . By declaring al to be of type ArrayList<String> , we are telling the compiler that the ArrayList object pointed to by al is supposed to hold only Strings. Here, it is crucial to recall the distinction between the type of a variable and the type of an object pointed to by that variable that I discussed in the "Reusing Implementations Through Inheritance" chapter. Remember that it is the type of the variable that determines what methods the compiler will allow to be invoked using that variable. Therefore, the compiler will now verify that any method call made using al is consistent with what the list pointed to by alis supposed to contain and if the compiler sees any statement that attempts to add anything that is not a String, or tries to retrieve anything other than a String using the al reference variable, it will refuse to compile that statement. The error message generated by the following code illustrates this point:

```
ArrayList<String> al = new ArrayList<String>();
al.add(new Object());
```

The error message is:

```
Error: no suitable method found for add(java.lang.Object)
    method java.util.Collection.add(java.lang.String) is not applicable
```

Since al has now been declared to be of type ArrayList<String> , the compiler does not let you call al.add(new Object()). Similarly, the compiler will not let you make incorrect assignments either. Here is an example:

```
List<String> al = new ArrayList<String>(); //al points to a List of Strings
al.add("hello"); //valid
String s = al.get(0); //valid, no cast required
Object obj = al.get(0); //valid because a String is-a Object
Integer in = al.get(0); //Invalid. Will not compile
```

The error message is:

```
Error: incompatible types: java.lang.String cannot be converted to java.lang.Integer
```

Observe that we supplied information about the type of object the list is supposed to keep by appending <String> to the variable declaration and the ArrayList creation. This way of specifying the type information is a part of a feature introduced in Java 5 known as "**generics**". The preexisting classes in the collections framework, such as Collection, List, and ArrayList, were retrofitted in Java 1.5 to make use of generics.

The above discussion should tell you that the topics of List/ArrayList, collections, and generics are interrelated. All three of them combine together to provide you a powerful mechanism to write type safe and bug free code. Although the exam does not ask questions specifically on generics, basic understanding of generics is required to decipher the questions related to the collections API. For this reason, I will take a diversion to discuss Generics in more detail now and go into the details of the collections framework after that.

### 15.1.2 Quiz

Assume the following variable declarations:

```
ArrayList<Integer> listI = new ArrayList<Integer>();
ArrayList<Number> listN = new ArrayList<Number>();
ArrayList<Object> listO = new ArrayList<Object>();
Double d = 10.0;
Number n = 10;
String str = "hello";
```

**Q1.** Which of the following method calls will compile without any error?

Select **3** correct options.

**A.** `listI.add(n);`
**B.** `listI.add(10);`
**C.** `listN.add(10); listN.add(10.0); listN.add(n);`
**D.** `listO.add(d); listO.add(str); listO.add(new Object()); listO.add(listI);`

**Correct answer is B, C, D.**
You may see similar code snippets in the exam. Remember that the compiler doesn't care about the actual objects pointed to by references while type checking method calls. It only checks the declared type of the variable on which the method is invoked and the declared type of the variable that is being passed as an argument to that method. Applying this logic, you can see that the declared type of `listI` is `ArrayList<Integer>` , which means, you can only pass an `Integer` reference to the `add` method using the `listI` variable. Even though `n` points to an `Integer` object, the declared type of `n` is `Number` and `Number` is not an `Integer`. `Integer` is-a `Number`. `Double` is-a `Number`. Hence, **A** is incorrect while **B** and **C** are correct.
Since the declared type of `listO` is `List<Object>` , you can pass any object to the `add` method. Since every class is a subclass of `Object`, every object is-a `Object` and so, **D** is also correct.

**Q2.** Which of the following assignments will compile without any error?

Select **2** correct options.

**A.** `n = listI.get(0);`
**B.** `Integer i = listI.get(0);`

**C.** `d = listN.get(0);`
**D.** `str = listO.get(0);`

**Correct answer is A, B.**
Again, compiler doesn't care about the actual objects pointed to by references while type checking assignment expressions. It only checks whether the declared type of the variable or the object on the right side of `=` is type compatible with the declared type of the variable on the left. Since the declared type of `listI` is `ArrayList<Integer>` , the compiler will assume that its `get` method will return a reference to an `Integer`. An `Integer` is a `Number`, therefore, options **A** and **B** are correct. The declared type of `listN` is `List<Number>` and so, the compiler will assume that its `get` method will return a reference to a `Number`. You cannot assign a `Number` to an `Integer` variable because a `Number` reference could actually be pointing to something other than an `Integer` at run time. It could be pointing to a `Double` too, for example, and assigning a `Double` object to an `Integer` variable is exactly the type of thing that the compiler intends to prevent from happening. Therefore, option **C** is incorrect. Option **D** is incorrect for the same reason.

### 15.1.3 Creating generic types

The whole purpose of introducing generics in Java is to enhance type safety of the code. Type safety is important because it helps prevent a category of bugs that occur due to wrong assumption about the type of objects at run time. For example, if the developer assumes that an object pointed to by a variable will be of type CheckingAccount and writes the code accordingly, and if at run time it turns out that the object is actually of type SavingsAccount, the code will fail. Had the compiler warned the developer that their assumption will turn out to be wrong at run time, the developer would have fixed the code right then and there.

Here, "making an assumption" essentially means casting a variable that was declared to be of one type to another type. You have seen cases where the compiler will not allow you to make such a cast. For example, the compiler will refuse to compile the cast in this code - `Integer i = 10; String s = (String) i;`. The compiler knows that the declared type of `i` is `Integer` and since `String` and `Integer` classes do not have a parent-child relationship, there is no way `i` can ever point to an object of type `String`. The compiler correctly points out that this cast is a programming mistake. But, as you saw with the code snippet involving an `ArrayList` in the previous section, there are situations where the compiler does not have enough information to prevent the developer from making such incorrect assumptions. In that code, the compiler had no knowledge about the type of the objects contained in the list and so, it had no option but to allow the compilation to succeed. It meekly generated a warning to let the programmer know that the code may run into issues at run time.

#### Generic classes

The first challenge in enhancing the type safety in a piece of code is to add type information to a class, which the compiler can use to prevent inappropriate casts. One could achieve that by creating a class with more specific types instead of `Object`. For example, one could copy the code from `ArrayList`, make some changes to it and create a `StringList` class that would use `String`, instead of Object, as the type of input parameters and return types in its methods. This solution has an obvious shortcoming. You will need to define a new class every time you need to create a list of a new type. Just imagine the horror if you find out a bug in the code. You would have to fix it in all the cloned classes. It is an impractical solution. The second challenge, therefore, is to prevent duplication of the code. So, what we really need is a way to write a class in a generic fashion, without hardcoding it to any particular type, and also allow that class to be typed to any type as per the requirement, at the time of using it.

This is exactly what generic classes in Java allow you to do. Instead of using an actual type while coding a class, you use a placeholder. Whoever uses the class must specify the value for that placeholder so that the class can be typed or "**parameterized**" to that type. I know it sounds complicated but the following example will make this clear:

```
public class DataHolder<E> {
    private E data;
    public E getData(){ return data;  }
    public void setData(E e){ data = e;  }
    public void useData(){
       System.out.println(data.toString());
    }
}
```

In the above code, instead of using any specific class as the type of data variable, I have used a placeholder named E. Observe the part where it says <E> in the class declaration. This tells the compiler that E is just a placeholder and that E will be replaced by the actual type when this class is actually used by another piece of code. The code in DataHolder is written as if the type of the data variable is E. The code in this class is generic enough to not care about the type with which E is replaced. In technical parlance, DataHolder is a **generic class** because it uses a **type parameter** (the part between <and > , which, in this example, is just E, is called type parameter). The placeholder itself is called **type variable**, which, in this example, is also E.

The useData method illustrates an important point here. Since E is not a real type, we do not know anything about its properties. Specifically, we don't know what methods or fields does E have. Neither does the compiler. Therefore, we can't invoke any method on the data variable except the ones that are defined in the Object class. This is a serious limitation and I will show you how to overcome it soon.

Let us now see how this class can be used by other people. Let us say there is a class that consumes a DataHolder object containing String. Here is the code for such a class:

```
public class SomeClass {
    public static void consumeStringData(DataHolder<String> stringData){
        String s =  stringData.getData(); //no cast required
        //do something with s
        //Integer i =  (Integer) stringData.getData(); //will not compile
    }
}
```

Observe the usage of `<String>` while declaring `stringData` parameter. Here, `String` is a **type argument** that is being passed to the generic class `DataHolder`. That makes `DataHolder<String>` a **parameterized type**. In technical parlance, we call it "typing" or "parameterizing" `DataHolder` to `String`. It tells the compiler that only a `DataHolder` object containing a `String` can be passed to this method. With this declaration, the compiler gets all the information that it needs to prevent a programmer from making a wrong assumption about what the argument contains. Note that this typing applies only to the `stringData` variable and not to all `DataHolder` objects in the program. Only the `DataHolder` object referred to by the `stringData` variable is being guaranteed to contain `String` here. A program could certainly use another variable of type `DataHolder` that is parameterized differently.

There are three points to understand in the above code.

**First**, no cast is required to assign the value returned by `getData` to `s`. The compiler knows that the `DataHolder` object pointed to by `stringData` has been typed to `String`. So, `stringData.getData()` will always return a `String`.

**Second**, the compiler will not let you cast the value returned by `stringData.getData()` to any type that is not a `String`. The reason is the same as above. The compiler knows that `stringData.getData()`can never return anything other than a `String` and so, casting the return value to `Integer` is a programming mistake.

Finally, **third**, the compiler will not let you invoke the `consumeData` method with anything other than a `DataHolder` typed to `String`. For example, the following code in yet another class will not compile:

```
public class TestClass {
    public static void main(String[] args){
        DataHolder<Integer> dh = new DataHolder<Integer>();
        SomeClass.consumeStringData(dh); //will not compile
    }
}
```

The compiler knows that `SomeClass.consumeStringData` expects a `DataHolder` containing `String`. So, it will not let the programmer make the mistake of invoking this method with a `DataHolder` typed to `Integer`.

Observe that the compiler is able to guarantee that the call to `stringData.getData()` inside `consumeData()` method will return a `String` only because it prevents an invocation of `consumeData` method with a `DataHolder` typed to anything other than `String` from compiling in the first place! Whenever the compiler is not able to prevent this from happening, it generates a warning. That, essentially, is the complete mechanism behind the generics magic.

You may wonder at this point how the above code is better than declaring the `data` variable as `Object`. Like this:

```
public class DataHolder{
    private Object data;
    //other methods
}
```

After all, the above code also allows a DataHolder to contain any type of object. But therein lies the problem. We don't know the type of the object that a given DataHolder object contains. In other words, we do want the DataHolder object to be able to contain any type of object but not without us knowing what kind of object it contains at any given point in the code. Furthermore, we don't want a DataHolder object to contain any other type of object once we decide what type of object it should contain.

**Note**

You will see the usage of generic classes and interfaces mostly, but records can also be generic. For example: `record Wrapper<E> (E e) { }`. However, Java does not allow enums to be generic.

### Generic Methods and Constructors

Generic methods and constructors are similar to generic classes. They may have type parameters of their own and the type variables introduced in such type parameters are scoped only to those methods and constructors. Here is an example:

```
public class ListProcessor<E>{
  //T t; //will not compile because T is undefined here
  E data; //fine
  public <E> ListProcessor(E e){
    System.out.print("E is "+e);
    //data = e; //will not compile because the E defined in this scope is different
   from the E defined for the class
  }
  public static <T> T processList(List<T> listOfT){
    return listOfT == null || listOfT.isEmpty() ? null:listOfT.get(0);
  }
}
```

In the code above, E is used as a placeholder for an actual type for the class as well as for the constructor and T is used as a placeholder for an actual type in the processListmethod. Unlike the placeholder E that is defined for the class, the E defined for the constructor and T are valid only within the constructor and the method respectively. Since they are defined only within their respective scopes, there is no issue even if you use the same name for these type variables. Furthermore, if name of the type variable matches with the name of a type variable specified for the class, the one defined in the method/constructor will hide the one defined in the class within that method as illustrated by the above code.

### Syntax for using generic types

You will not be tested on the syntax of defining a generic class or a generic method but you should be aware of it because such code may appear in exam questions. Furthermore, even though defining generic types is not on the exam, using a generic type or a method is. The code snippets presented in exam questions use parameterized types quite liberally. So, you need to know the following ways in which you can declare type safe variables and instantiate parameterized types to understand what the code presented in a question is trying to do:

1. A type safe variable can be declared by specifying a type argument within `<` and `>` . For example,

   ```
   ArrayList<String> stringArray;
   List<Integer> iList;
   ArrayList<com.acme.Person> personList;

    //I will discuss wildcards in the next section
   List<?> iList; //using unbounded wildcard
   List<? extends Integer> iList; //using upper bounded wildcard
   List<? super Integer> iList; //using upper bounded wildcard
   ```

   ```
   List<> iList; //invalid, a type must be specified inside <> while declaring a
       variable
   ```

2. A parameterized class can be instantiated similarly by specifying the type argument within `<` and `>` . For example,

   ```
   new ArrayList<String>();
   new ArrayList<Integer>();
   new ArrayList<com.acme.Person>();
   ```

   Wildcards, however, cannot be used while instantiating generic class objects. So, the following lines will not compile:

   ```
   new ArrayList<?>();
   new ArrayList<? extends Integer>();
   new ArrayList<? super com.acme.Person>();
   ```

3. If you are instantiating a type safe `ArrayList` while assigning it to a type safe variable in the same statement, then you may omit the type name from the instantiation as shown below:

   ```
   ArrayList<String> al = new ArrayList<> ();
   ```

   Upon encountering `<>` , the compiler infers the type of the `ArrayList` object from the type of the variable it is being assigned to. The `<>` operator is known as the "**diamond operator**". and was introduced in Java 7. It can be used only in a `new` statement. Thus, the statement `ArrayList<> al;` will not compile because it has nothing from which the compiler can infer the type of `al`. Although surprisingly, `var al = new ArrayList<> ();`, which too doesn't have much information for the compiler to infer the type, compiles fine. The compiler infers the type of `al` as `ArrayList<Object>` .

Similarly, the compiler can also infer the type argument in method invocations. For example,

```
Object oValue = ListProcessor.processList(new ArrayList<>()); //Object inferred
Integer iValue1 = ListProcessor.processList(new ArrayList<>()); //Integer inferred
Integer ivalue2 = ListProcessor.<Integer>processList(new ArrayList<>());
   //Integer inferred
```

The last method invocation above shows how to specify the type argument explicitly but this is seldom required in practice. In most cases, the compiler is able to figure out the type argument from the context. So, here, since the return value of `processList` is being assigned to an `Integer` variable, the compiler correctly infers the type argument as `Integer`.

Remember that you cannot apply the `<>` syntax to any random class. You can apply it only to a generic class.

Note

**Inferencing the type from <>**

The diamond operator makes the compiler infer the type argument for the generic class using using the type information available in the context. The exact process of inferring this type is not easy to comprehend from the available documentation. Fortunately, as a Java programmer, you do not have to worry about it at all. You will never need to know what the inferred type is because your view into the object is determined solely by the type of the reference variable that you are using, which you already know. For example, you can safely assume that the type inferred by the compiler in all of the following statements, including the fourth, is `Number`, even if that assumption turns out to be incorrect (as it will for the fourth line).

```
ArrayList<Number> list1 = new ArrayList<>();
ArrayList<? super Number> list2 = new ArrayList<Number>();
ArrayList<? extends Number> list3 = new ArrayList<Number>();
ArrayList<?> list4 = new ArrayList<>();
```

In none of the statements above does it matter to the programmer what type the compiler actually uses, as long as it satisfies the constraints imposed by the type argument specified in the variable declaration. For this reason, it is recommended to use the diamond operator as much as possible instead of specifying the type explicitly while instantiating generic type objects.

### Using multiple type parameters

A class may use any number of type parameters. How many type parameters you need in a class depends on your requirements and class design. For example, the following class uses two type parameters:

```
class PairHolder<E1, E2> {
    E1 part1;
    E2 part2;
}
...
//using the above class in another place
PairHolder<String, Integer> ph1 = new PairHolder<>();
String s = ph1.part1;
PairHolder<String, String> ph2 = new PairHolder<>();
```

I have used `E1` and `E2` as the names of the type parameters only to illustrate that you may name them anything. But conventionally, the following letters are used to name type parameters:
`E` for Element (used extensively by the Java collections framework)
`K` for Key
`N` for Number
`T` for Type
`V` for Value
`S`,`U`,`V` etc. - 2nd, 3rd, 4th types

Classes in the collections framework and the Stream API that use multiple type parameters are on the exam as well. These classes often use `T` for the type of the objects in the collection or stream, `A`, for the type of the accumulator, and `R` for the return type.

## 15.1.4 Runtime behavior of generics

I explained in the Arrays chapter that arrays are reified and are covariant. I also alluded to the fact that generics are diametrically opposite to arrays in this respect. Let me show you how.

### Generics and Type Erasure

Generics were added quite late to Java (in version 1.5) and so, backward compatibility was an important concern. Furthermore, since it was a fairly complicated feature, it was expected that their wide spread usage will take some time. Thus, it was deemed necessary that code that uses Generics must work with code that doesn't use Generics. Basically, there are two scenarios that had to be supported. Generic aware Java source code should be able use preexisting classes that did not use generics and non-generics source code should be able to use new generic aware libraries.

To achieve this design goal, the mechanism of **type erasure** was used, where all of the generic information is conceptually stripped away from a class. In other words, even though the generic type information written in Java code is present in the class file, the JVM does not make use of any of that information. It is as if this type information does not exist in the class file as far as the JVM is concerned. This information is used only by the compiler for analyzing type safety of the source code. Thus, even if you write `ArrayList<String> al = new ArrayList<> ();`in your code, the JVM will only see `ArrayList al = new ArrayList();`. This implies that the Java code `String str = al.get(0);` gets translated to `String str = (String) al.get(0);` in the class file. Type erasure happens for the generic class definition as well. For example, the type parameter `E` used in the `DataHolder` class defined earlier is replaced by `Object`. As you will soon learn, if the type parameters have a bound, they are replaced by those bounds in the class file.

Type erasure eliminates the need to create separate class files for differently typed usages, i.e., differently parameterized instantiations of a generic class. This ensures backward compatibility because there is essentially no difference between generics aware code and regular code from the JVM's perspective.

At this point, one may wonder how do generics promote type safety if the JVM has no knowledge of them at runtime. The answer is that generics are meant to promote type safety of the code at compile time only. The usage of parameterized types enhances the type checking capability of the compiler. It is possible to break the type checking done by the compiler by casting a parameterized type to its non-parameterized form as shown by the following example:

```
ArrayList<Integer> al = new ArrayList<>();
ArrayList rawList = (ArrayList) al;
rawList.add("string");
Integer i = al.get(0); //ClassCastException at runtime
```

The above code will fail at runtime with a `ClassCastException`. To the compiler's credit, it does generate a warning message saying the code uses unchecked or unsafe operations, which conveys to the programmer that something fishy is going on in this code and if you compile it with `-Xlint:unchecked` option, it will provide even more details:

```
warning: [unchecked] unchecked call to add(E) as a member of the raw type ArrayList
rawList.add("string");
           ^
  where E is a type-variable:
    E extends Object declared in class ArrayList
1 warning
```

The message makes it clear that `ArrayList` uses a type variable `E` but it is not being set in the given code, which is why the compiler is not able to ensure that adding a `String` to this list will not cause trouble at runtime!

## Generics Invariance

As a result of how generics were implemented in Java, the process of establishing a supertype-subtype relation between two parameterized forms of the same generic type is not as straightforward as it is in the case of arrays, where, if type `B` is a subtype of `A`, then `B[]` is automatically deemed a subtype of `A[]`. For example, since `Integer` is a subclass of `Number`, `Integer[]` is a subtype of `Number[]`. But it is not so, with parameterized types.

From a purely English language perspective, a list of integers does seem like a special case of a list of numbers but from the Java language perspective, `ArrayList<Integer>` cannot be deemed a subtype of `ArrayList<Number>` . As you have learnt in the inheritance chapter, you should be able to substitute a subtype object wherever a supertype object is needed. For example, you can assign an `Integer` array to a variable of type `Number` array as in, `Number[] na = new Integer[10];`. This substitutability makes any superclass-subclass pair **covariant**. But substituting seemingly covariant parameterized types can easily break type safety as shown by the following code:

```
ArrayList<Integer> listI = new ArrayList<Integer>();
ArrayList<Number> listN = listI; //not really valid but let's say it is
listN.add(1.0); //adding a Double to a list of Integers!
Integer iValue = listI.get(0); //ClassCastException
```

The above code pollutes the `ArrayList` object that was meant to store `Integer`s by adding a `Double` to it. Whoever then tries to take out an `Integer` from this list, will encounter a `ClassCastException` at run time. If `List<Integer>` were deemed a subtype of `List<Number>` , the compiler would have to accept the above code as valid and that would completely defeat the purpose of generics. Note that one might expect the JVM to complain by throwing an `ArrayStoreException` as soon as an attempt is made to add a `Double` to a list of `Integer`s just like it did when we tried to store a `Double` in an array of `Integer`s but, because of type erasure, it can't! The JVM doesn't know anything about the type of objects this list is meant to store. For this reason, generics in Java are considered **invariant**.

It is not to say that there is never a supertype-subtype relation between parameterized types. You will soon see that such a relationship can exist, at least for the compiler, when type arguments have bounds and in such cases, you will see parameterized types exhibiting covariance.

Type erasure has a serious impact on overloading and overriding of methods. Although not mentioned explicitly, the exam has questions that touch upon this aspect. So, let's dig a little deeper into it.

### Impact of Type Erasure on Method Overloading

As explained previously, overloaded methods are methods that differ in their signature. Recall that a method signature is made up of method name plus parameter types. Since, due to type erasure, generic information is removed at run time, it follows that the generic information cannot be a part of the signature. Thus, two methods that differ only in their type arguments are not valid overloads. The compiler will refuse to compile such code because it knows that the JVM will not be able to distinguish between them at run time. For example, the following code won't compile:

```
public class TestClass{
   public void processData(DataHolder<String> stringData){
      String s = stringData.getData();
      System.out.println(s);
   }
   public void processData(DataHolder<Integer> intData){
      Integer i = intData.getData();
      System.out.println(i);
   }
}
```

The error message will be `error: name clash: processData(DataHolder<Integer> ) and processData(DataHolder<String> ) have the same erasure`. Indeed, once the generic information is removed, signature of both the methods are same and there is no way for the JVM to bind an invocation of this method to any one version.

### Impact of Type Erasure on Method Overriding

Overriding a method that uses generic types has the same issue that I talked about above. The following code fails to compile:

```
class Base{
   public void processData(DataHolder<String> stringData){
      String s = stringData.getData();
      System.out.println(s);
   }
}
public class SubClass extends Base{
   public void processData(DataHolder<Integer> intData){ //not ok
      Integer i = intData.getData();
      System.out.println(i);
   }
}
```

The error message is: `error: name clash: processData(DataHolder<Integer> ) in SubClass and processData(DataHolder<String> ) in Base have the same erasure, yet neither overrides the other`.

In this case, the overriding method is trying to break the contract promised by the base class. The base class's `processData` version expects a `DataHolder` containing `String`, while the subclass's version expects a `DataHolder` containing `Integer`. Although after erasing the type information, the subclass method seems to correctly override the base class's method. However, the compiler notices that the cast to `Integer` in the subclass's method will fail at run time if `processData` is invoked with a `DataHolder` containing a `String` because `intData.getData()` will return a `String`! Therefore, it refuses to compile the code.

So, to summarize, the type arguments of the parameters of the overriding method must **also** match exactly with the type arguments of the parameters of the overridden method.

### Covariance with parameterized return types

As discussed in the chapter on Inheritance, an overriding method is allowed to return a covariant return type. Meaning, if a base class's method returns `List`, the overriding method in the subclass may return `ArrayList` because `ArrayList` is a subtype of `List`. However, since generics are invariant, you cannot count on the type argument to determine covariance. The following example, illustrates this point:

```
class Base{
  List<Number> getList(){ return null; }
}
class Sub extends Base{
  //List<Integer> getList(){ return null; } //invalid
  ArrayList<Number> getList(){ return null; } //valid
}
```

The above code shows that an overriding method may return a subtype if its type argument matches exactly with the type argument of return type of the overridden method.

Seems too restrictive, right? Fortunately, this is not the end of it. There is an exception to this rule and to understand that exception, you will need to first learn about bounds and wildcards. So, I will revisit this aspect of overriding again soon.

**Exam Tip**

The OCP exam does not focus on generics specifically. It uses generics in code snippets only incidentally and does not have questions that you can't answer without having complete knowledge of bounds and wildcards. So, if you are short on time, you may skip the advanced part of this discussion. However, a good understanding of wildcards will be useful on the job as well as for the interviews. So, I strongly suggest you not to skip it.

### 15.1.5 Quiz

Given the following two classes:

```
class Base{
   public <T> List<T> processList(List<T> list){ return null; }
}
```

and

```
public class SubClass extends Base{
   //add method here
}
```

Which of the following options are true if the method is added to `SubClass`?

**A.** `public <T> List<T> processList(List<T> list){ return null; }` correctly **overrides** `processList` of `Base`.

**B.** `public <T> Collection<T> processList(List<T> list){ return null; }` correctly **overrides** `processList` of `Base`.

**C.** `public List<String> processList(List<String> list){ return null; }` correctly **overrides** `processList` of `Base`.

**D.**. `public <T> ArrayList<T> processList(List<T> list){ return null; }` correctly **overrides** `processList` of `Base`.

**E.** `public <T> List<T> processList(ArrayList<T> list){ return null; }` correctly **overloads** `processList` of `Base`.

**F.** `public <T> List<T> processList(List<T> list){ return null; }` correctly **overloads** `processList` of `Base`.

**G.** `public <T> List<T> acceptList(List<T> list){ return null; }` correctly **overloads** `processList` of `Base`.

**Correct answer is A, D, E**

Remember that when a subclass contains a method with the same signature as a method in the super class, it is called an **override** and when a class contains (or inherits) multiple methods with the same name but different parameter types, it is called an **overload**.

Option **A** is obviously **correct** because the method signature as well as the return type of the method in the subclass matches exactly to the method signature and the return type of the method in the super class. It is a valid override.

Option **B** could have been a valid override but the return type of the overriding method cannot be a wider/super type. Recall the rule of **covariant returns**. So, it is **incorrect**.

Option **C** is **incorrect** because of a mismatch in the type arguments of the method parameters and the return type as explained in this section.

Observe that in this option, after erasure (i.e. at runtime), the base class and the subclass methods will have the same signature and that should make it a valid override. However, that is exactly what confuses the compiler. The compiler understands that the signatures of the given methods are the same after erasure (which means it is an override at runtime) but not the same at compile time (which means it is not an override but an overload at compile time). This contradiction causes the compiler to reject the code. There is one exception to this rule, which is not too important for the exam but is good to know anyway. I have explained it in the note below.

Option **D** is **correct** because the method signatures match and the subclass method follows the rule of covariant returns. `ArrayList` is a narrower type than `List` because `ArrayList` implements `List`.

Option **E** is **correct** because the parameter type of the method in the subclass is different.

The subclass, therefore, has two methods (one of its own and one inherited) with the same name but different parameter types. Thus, it is a valid overload.

Option **F** is **incorrect** because it is a valid override and not a valid overload.

Option **G** is **incorrect** because the method name doesn't match. It is neither an overload nor an override. It is a valid method though.

Note

The general rule for a successful override is that the signature of the subclass method must match exactly to that of the signature of the base class method at compile time (i.e. without applying type erasure). An exception to this rule is that it is valid for a non-generified method to override a generified method. In other words, if the signature of the subclass method without type erasure matches with the signature of the base class method after type erasure, then it is a valid override. For example, the following will compile without an error:

```
class Base{
  public List<String> processList(List<String> list){
     System.out.println("In Base");
     return null;
  }
}
class SubClass extends Base{
  public List processList(List list){  //valid override
    System.out.println("In Sub");
    return null;
  }
  public static void main(String[] args) {
    Base b = new SubClass();
    System.out.println(b.processList(null)); //will print In Sub
  }
}
```

Java language designers have allowed this exception so that library providers (i.e. the ones supplying the base classes) are able to generify their methods without affecting their existing users (i.e. the ones using and extending the base classes). In other words, if you have created a non-generified method, and if other people have overridden this method in their classes, you can still go ahead and generify your method without impacting them.

### 15.1.6 Collection and List API

#### Collection API

**java.util.Collection** is the root interface in the collections hierarchy. It abstracts the concept of a collection in general. It declares only the methods that are applicable to all sorts of collections and leaves the methods that deal with specialized features such as ordering or uniqueness of elements to sub-interfaces such as `List` and `Set`. For example, `Collection` declares `add(Object e)` and `remove(Object e)` methods but does not declare `add(int index, Object e)` and `remove(int index)` because the concept of ordering is not applicable to all collections. We use the `Collection` interface when we don't want to make any assumption about the characteristics of the group of objects. In other words, if we get a `Collection` object, all we know is that it contains a bunch of elements. We don't know whether the elements are ordered in any way, whether the group contains nulls or duplicate elements, or what is the data structure used to keeps the elements in that collection.

At this point, I would encourage you to go through the JavaDoc API description of the `Collection` interface and check out the methods declared in this interface. The method names are quiet descriptive and the methods do what their names suggest. Some of the methods that you should pay attention to are: `add`, `addAll`, `remove`, `removeAll`, `removeIf`, `contains`, `containsAll`, `isEmpty`, and `clear`.

### List API

**java.util.List** interface defines the behavior of collections that keep objects in an order. Elements of a list can be accessed using an index. It also allows elements to be inserted in or removed from any given index. Duplicate elements as well as nulls are permitted. Since it is just an interface, it doesn't specify how the functionality is implemented. Actual implementation classes such as `ArrayList` and `LinkedList` use different mechanisms to implement the behavior. As you will see later with `ArrayList`, implementation classes may provide additional features on top of the features defined in `List`.

Again, you should go through all of the methods of the `List` interface from the JavaDoc. The exam doesn't trick you on method names and doesn't require you to recall whether a given method exists in `List` or not. The exam questions are based on the understanding of how a given method behaves. You are expected to know what happens to the existing list of objects after the invocation of a method. The trick is in knowing "boundary conditions". For example, what happens if you try to remove a non-existing object or a `null` from a list or what happens if you try to add null.

I will list the important methods of `List` interface below. Remember that index is zero based, which means, the index of the first element is zero.

1. `E get(int index)`: Returns the element at index.

2. `E set(int index, E e)`: Replaces the existing element at index with the passed object and returns the original element that was replaced.

3. `boolean add(E e)`: Adds an object at the end of the list. Returns `true`.

4. `void add(int index, E e)`: Adds an object at the specified index and shifts the existing elements at a higher index to the right.

5. `boolean addAll(Collection<? extends E> c)`: Adds all the elements of the given collection to this list at the end. Returns `true` if this list changed as a result of the call.

6. `void addAll(int index, Collection<? extends E> c)`: Adds all the elements of the given collection at the given index. Returns `true` if this list changed as a result of the call.

7. `E remove(int index)`: Removes an object from the specified index and shifts the existing elements at a higher index to the left. It returns the original element that was removed from the list.

8. `boolean remove(Object obj)`: Removes the first occurrence of given object from the list. Returns `true` if the object was found. The passed object is matched with the objects in the list using the equals method.

9. `boolean removeAll(Collection<?> c)`: Removes from this list all the elements that are present in the passed collection. Returns `true` if this list changed as a result of the call.

10. `boolean retainAll(Collection c)`: Removes from this list all the elements that are not in the passed collection. Returns `true` if this list changed as a result of the call.

11. `void clear()`: Removes all the elements from this list.

12. `int size()`: Returns the number of elements in this list.

13. `default void forEach(Consumer<? super T> action)`: Inherited from `java.lang.Iterable`. Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
    You will see this method used with lambda expressions a lot.

Remember that for the `get` and the `remove` methods, the `index` must be between `0` and the list's `size` minus `1` (both inclusive) otherwise an `IndexOutOfBoundsException` is thrown but in the case of the `add` method, the highest `index` value is the same as the list's `size`, which implies that you are adding an element after the last element, i.e., at the end of the list.

A typical question that tests your understanding of the above methods would be to predict the output of the following code:

```java
List<Integer> list =  new ArrayList<Integer>(); //observe that list's size is 0 at this
   point
list.add(0, 1);
list.add(0, 2);
list.add(0, 3);
System.out.println(list);
```

If you think the output is `[1, 2, 3]`, you have been tricked. The output is `[3, 2, 1]` because the elements are being inserted at the `0`th index.

Here are few methods that help you inspect a list:

1. `boolean isEmpty()`: Returns true if the list has no elements.

2. `boolean contains(Object o)`: Returns true of the list contains the given object.

3. `boolean containsAll(Collection c)`: Returns true of the list contains all of the elements present in the given collection.

4. `List subList(int fromIndex, int toIndex)`: Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive. If fromIndex and toIndex are equal, the returned list is empty.

5. `int indexOf(Object o)`: Returns the index at which the given object is found in the list. Returns -1 if it is not found.

6. `int lastIndexOf(Object o)`: Returns the last index at which the given object is found in the list. Returns -1 if it is not found.

7. `Object[] toArray()`: Returns a array of Objects containing all the elements of this list.

8. `<T> T[] toArray(T[] a)`: Returns an array of the type specified in the argument containing all the elements of this list. The objects of the list must be of the type passed in the argument. This method is helpful when you want to convert a list into an array of a specific type. For example, `String[] strArray = listOfString.toArray(new String[0]);` will return a String[] containing all the elements of the list.

The `subList(int fromIndex, int toIndex)` method in the above list of methods is particularly interesting. It does not return a new independent list but just a view into the original list. Thus, any change you do to the view is reflected in the original list as shown the example below:

```
List<String> al = new ArrayList<String>();
al.addAll(Arrays.asList( new String[]{"a", "b", "c", "d", "e" } ));
List<String> al2 = al.subList(2, 4);
System.out.println(al2); //prints [c, d]
al2.add("x");
System.out.println(al2); //prints [c, d, x]
System.out.println(al); //prints [a, b, c, d, x, e]
al.add("y"); //structural modification to the original list
System.out.println(al2); //throws java.util.ConcurrentModificationException
```

Observe the output of the last two lines of the output. `x` was added to the end of the list pointed to by `al2`. However, since `al2` is just a view of a portion of the original list pointed to by `al`, `x` is visible in the middle of this list. Furthermore, all structural modifications must be done through the view. If you structurally modify the backing list and then try to use the view, the results will be unpredictable. Structural modification means any change in size of the list. That is why, the last line of the above code throws a `java.util.ConcurrentModificationException` exception.

### List.of and List.copyOf methods

Java has had the `java.util.List` interface ever since the Collections API was introduced in Java 1.2. It was enhanced in Java 9 and 10 with several static methods and the exam expects you to know them. These are the twelve overloaded `of` (added in Java 9) methods and the `copyOf` method (added in Java 10).

The overloaded `of` methods are merely convenience methods that take 0 to 10 parameters and return an **unmodifiable** List. For example, if you want to create a list of 3 Strings, you could do `List<String> list1 = List.of("a", "b", "c");`. Similarly, the `copyOf` method is a convenience method to create an **unmodifiable** list using the elements of an existing collection. For example, `List<String> list2 = List.copyOf(list1);`.

You will see the usage of these methods all over exam questions even when the question is not about lists. Since understanding the behavior of the lists returned by these methods is very important for the exam, let me first jot down their characteristics as given in the JavaDoc:

1. They are **unmodifiable**. Elements cannot be added, removed, or replaced. Calling any mutator method on the List will always cause `UnsupportedOperationException` to be thrown. However, if the contained elements are themselves mutable, this may cause the List's contents to appear to change.

2. They disallow **null** elements. Attempts to create them with `null` elements result in `NullPointerException`.

3. They are serializable if all elements are serializable.

4. The order of elements in the list is the same as the order of the provided arguments, or of the elements in the provided array.

5. They are value-based. Callers should make no assumptions about the identity of the returned instances. Factories are free to create new instances or reuse existing ones. Therefore, identity-sensitive operations on these instances (reference equality (`==`), identity hash code, and synchronization) are unreliable and should be avoided.

Here is the same code that I showed above but with a small modification:

```
List<Integer> list =  List.of();
list.add(0, 1);
list.add(0, 2);
list.add(0, 3);
System.out.println(list);
```

Before jumping on to answer `[3, 2, 1]` this time, you need to appreciate the fact that the list returned by `List.of` is unmodifiable! So, the code will compile but the first call to `list.add` will throw an `java.lang.UnsupportedOperationException` exception at run time.

### 15.1.7 ArrayList API

As discussed earlier, `ArrayList` is one of the implementation classes of the `List` interface. It has three constructors:

1. `ArrayList()`: Constructs an empty list with an initial capacity of 10. Just like you saw with the `StringBuilder` class, capacity is simply the size of the initial array that is used to store the objects. As soon as you add the 11th object, a new array with bigger capacity will be allocated and all the existing objects will be transferred to the new array.

2. `ArrayList(Collection c)`: Constructs a list containing the elements of the specified collection.

3. `ArrayList(int initialCapacity)`: Constructs an empty list with the specified initial capacity. This constructor is helpful when you know the approximate number of objects that you want to add to the list. Specifying an initial capacity that is greater than the number of objects that the list will hold improves performance by avoiding the need to allocate a new array every time it uses up its existing capacity. It is possible to increase the capacity of an `ArrayList` even after it has been created by invoking `ensureCapacity(int n)` on that `ArrayList` instance. Calling this method with an appropriate number before inserting a large number of elements in the `ArrayList` improves performance of the add operation by reducing the need for incremental reallocation of the internal array. The opposite of `ensureCapacity` is the `trimToSize()` method, which gets rid of all the unused space by reducing its capacity to the match the number of elements in the list.

Here are a few declarations that you may see on the exam:

```
List<String> list = new ArrayList<>(); //ok because ArrayList implements List

var al = new ArrayList<Integer>(50); //initial capacity is 50

ArrayList<String> al2 = new ArrayList<>(list); //copying an existing list, observe the
    diamond operator

var list1 = new ArrayList<>(); //ok, list1 is of type ArrayList<Object>, observe the
    combination of var declaration and the diamond operator
```

```
List list2 = new List(list); //will not compile because List is an interface, it cannot
    be instantiated
```

## Important methods of ArrayList

`ArrayList` has quite a lot of methods. However, since `ArrayList` implements `List` (which, in turn, extends `Collection`), several of `ArrayList`'s methods are declared in `List` and `Collection` interfaces. The exam does not expect you to make a distinction between the methods inherited from `List`/ `Collection` and the methods declared in `ArrayList`.

The following are the ones that you need to know for the exam:

1. `String toString()`: Well, `toString` is not really the most important method of `ArrayList` but since we will be depending on its output in our examples, it is better to know about it anyway. `ArrayList`'s `toString` first gets a string representation for each of its elements (by invoking `toString` on them) and then combines into a single string that starts with `[` and ends with `]`. For example, the following code prints `[a, b, c]`:

```
var al = new ArrayList<String>();
al.add("a");
al.add("b");
al.add("c");
System.out.println(al);
```

Observe the order of the elements in the output. It is the same as the order in which they were added in the list. Calling `toString` on an empty `ArrayList` gets you `[ ]`. I will use the same format to show the contents of a list in code samples.

Methods that add elements to an `ArrayList`:

1. `boolean add(E e)`: Appends the specified element to the end of this list. As discussed in the Generics section, `E` is just a place holder for whichever type you specify while creating the `ArrayList`. For example, if you create an `ArrayList` of Strings, i.e., `ArrayList<String>` , `E` stands for `String`.

   This method is actually declared in `Collection` interface and the return value is used to convey whether the collection was changed as a result of calling this method. In case of an `ArrayList`, the `add` method always adds the given element to the list (even if the element is null), which means it changes the collection every time it is invoked. Therefore, it always returns `true`.

2. `void add(int index, E element)`: Inserts the specified element at the specified position in this list. The indexing starts from `0` and the maximum value of `index` can be `size`. Therefore, if you call `add(0, "hello")` on an list of `Strings`, `"hello"` will be inserted at the first position.

3. `boolean addAll(Collection<? extends E> c)`: Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's `Iterator`. Don't worry about the `? extends E` part right now. You just need to know that you can add all the elements of one list to another list using this method. For example,

```
ArrayList<String> sList1 = new ArrayList<>(); //observe the usage of the diamond
    operator
sList1.add("a"); //[a]
ArrayList<String> sList2 = new ArrayList<>();
sList2.add("b"); //[b]
sList2.addAll(sList1); //sList2 now contains [b, a]
```

4. `boolean addAll(int index, Collection<? extends E> c)`: This method is similar to the one above except that it inserts the elements of the passed list in the specified collection into this list, starting at the specified position. For example,

```
ArrayList<String> sList1 = new ArrayList<>();
sList1.add("a"); //[a]
ArrayList<String> sList3 = new ArrayList<>();
sList3.add("b"); //[b]
sList3.addAll(0, sList1); //sList3 now contains [a, b]
```

Methods that remove elements from an `ArrayList`:

1. `E remove(int index)`: Removes the element at the specified position in this list. For example,

```
ArrayList<String> list = ... // an ArrayList containing [a, b, c]
String s = list.remove(1); //list now has [a, c]
```

   It returns the element that has been removed from the list. Therefore, `s` will be assigned the element that was removed, i.e., `"b"`. If you pass an invalid int value as an argument (such as a negative value or a value that is beyond the range of the list, i.e., `size-1`), an `IndexOutOfBoundsException` will be thrown.

2. `boolean remove(Object o)`: Removes the first occurrence of the specified element from this list, if it is present. For example,

```
ArrayList<String> list = ... // an ArrayList containing [a, b, a]
list.remove("a"); //[b, a]
```

   Observe that only the first `a` is removed.
   You have to pay attention while using this method on an `ArrayList` of `Integers`. Can you guess what the following code will print?

```
ArrayList<Integer> list = new ArrayList<>(Arrays.asList( new Integer[]{1, 2, 3 }
   ));
list.remove(1);
System.out.println(list);
list = new ArrayList<>(Arrays.asList( new Integer[]{1, 2, 3 } ));
list.remove(new Integer(1));
System.out.println(list);
```

   The output is:

```
[1, 3]
[2, 3]
```

   Recall the rules of method selection in the case of overloaded methods. When you call `remove(1)`, the argument is an `int` and since a remove method with `int` parameter is available, this method will be preferred over the other remove method with `Object` parameter because invoking the other method requires boxing `1` into an `Integer`.

   This method returns `true` if an element was actually removed from the list as a result of this call. In other words, if there is no element in the list that matches the argument, the method will return `false`.

3. `boolean removeAll(Collection<?> c)`: Removes from this list all of its elements that are contained in the specified collection. For example the following code prints `[ c ]`:

```
ArrayList<String> al1 = new ArrayList<>(Arrays.asList( new String[]{"a", "b",
    "c", "a" } ));
ArrayList<String> al2 = new ArrayList<>(Arrays.asList( new String[]{"a", "b" } ));
al1.removeAll(al2);
System.out.println(al1); //prints [ c ]
```

Observe that unlike the `remove(Object obj)` method, which removes only the first occurrence of an element, `removeAll` removes all occurrences of an element. This method returns `true` if an element was actually removed from the list as a result of this call.

4. `void clear()`: Removes all of the elements from this list.

Methods that replace an element in an `ArrayList`:

1. `E set(int index, E element)`: Replaces the element at the specified position in this list with the specified element. It returns the element that was replaced. Example:

```
ArrayList<String> al = ... // create a list containing [a, b, c]
String oldVal = al.set(1, "x");
System.out.println(al); //prints [a, x, c]
System.out.println(oldVal); //prints b
```

Methods that read an `ArrayList` without modifying it:

1. `boolean contains(Object o)`: The object passed in the argument is compared with each element in the list using the equals method. A `true` is returned as soon as a matches is found, a `false` is returned otherwise. Here are a couple of examples:

```
ArrayList<String> al = new ArrayList<>();
al.addAll(Arrays.asList( new String[]{"a", null, "b", "c" } ));
System.out.println(al.contains("c")); //prints true
System.out.println(al.contains("z")); //prints false
System.out.println(al.contains(null)); //prints true
```

Observe that it does not throw a `NullPointerException` even if you pass it a `null`. In fact, a `null` argument matches a `null` element.

2. `E get(int index)`: Returns the element at the specified position in this list. It throws an `IndexOutOfBoundsException` if an invalid value (i.e. a value less than `0` or greater than `size-1`) is passed as an argument.

3. `int indexOf(Object o)`: The object passed in the argument is compared with each element in the list using the equals method. The index of the first element that matches is returned. If this list does not contain a matching element, -1 is returned. Here are a couple of examples:

```
ArrayList<String> al = new ArrayList<>();
al.addAll(Arrays.asList( new String[]{"a", null, "b", "c", null } ));
System.out.println(al.indexOf("c")); //prints 3
System.out.println(al.indexOf("z")); //prints -1
System.out.println(al.indexOf(null)); //prints 1
```

Observe that just like `contains`, `indexOf` does not throw a `NullPointerException` either even if you pass it a `null`. A `null` argument matches a `null` element.

4. `boolean isEmpty()`: Returns `true` if this list contains no elements.

5. `int size()`: Returns the number of elements in this list. Recall that to get the number of elements in a simple array, you use the variable named `length` of that array.

The examples that I have given above are meant to illustrate only a single method. In the exam, however, you will see code that uses multiple methods. Here are a few points that you should remember for the exam:

1. **Adding nulls**: `ArrayList` supports `null` elements.

2. **Adding duplicates**: `ArrayList` supports duplicate elements.

3. **Exceptions**: None of the `ArrayList` methods `except toArray(T[] a)` throw `NullPointerException`. They throw `IndexOutOfBoundsException` if you try to access an element beyond the range of the list.

4. **Method chaining**: Unlike `StringBuilder`, none of the `ArrayList` methods return a reference to the same `ArrayList` object. Therefore, it is not possible to chain method calls.

Here are a few examples of the kind of code you will see in the exam. Try to determine the output of the following code snippets when they are compiled and executed:

1. -–

```
var al = new ArrayList<Integer>(); //observe that the type specification is on
    the right side
al.add(1).add(2);
System.out.println(al);
```

2. -–

```
ArrayList<String> al = new ArrayList<>(); //observe the usage of the diamond
    operator
if( al.add("a") ){
  if( al.contains("a") ){
    al.add(al.indexOf("a"), "b");
  }
}
System.out.println(al);
```

3. -–

```
ArrayList<String> al = new ArrayList<>();
al.add("a"); al.add("b");
al.add(al.size(), "x");
System.out.println(al);
```

4. -–

```
var list1 = new ArrayList<String>();
var list2 = new ArrayList<String>();
list1.add( "a"); list1.add("b");
list2.add("b"); list2.add("c"); list2.add("d");
list1.addAll(list2);
list1.remove("b");
System.out.println(list1);
```

5. -–

```
ArrayList<String> list1 = new ArrayList<>();
ArrayList<String> list2 = new ArrayList<>();
list1.add( "a"); list1.add("b");
list2.add("b"); list2.add("c"); list2.add("d");
list1.addAll(list2);
System.out.println(list1);
list1.remove("b");
System.out.println(list1);
```

6. -–

```
ArrayList<String> list1 = new ArrayList<>();
ArrayList<String> list2 = new ArrayList<>();
list1.add( "a"); list1.add("b");
list2.add("b"); list2.add("c"); list2.add("d");
list1.addAll(list2);
list1.removeAll("b");
System.out.println(list1);
```

### 15.1.8 ArrayList vs array

You have already seen all that we can do with ArrayLists and arrays, so, I am just going to summarize their advantages and disadvantages here.

#### Advantages of ArrayList

1. **Dynamic sizing** - An ArrayList can grow in size as required. The programmer doesn't have to worry about the length of the ArrayList while adding elements to it.

2. **Type safety** - An ArrayList can be made type safe using generics.

3. **Readymade features** - ArrayList provides methods for searching and for inserting elements anywhere in the list.

### Disadvantages of ArrayList

1. **Higher memory usage** - An ArrayList generally requires more space than is necessary to hold the same number of elements in an array.

2. **No type safety** - Without generics, an ArrayList is not type safe at all.

3. **No support for primitive values** - ArrayLists cannot store primitive values while arrays can. This disadvantage has been mitigated somewhat with the introduction of autoboxing in Java 5, which makes it is possible to pass primitive values to various methods of an ArrayList. However, autoboxing does impact performance.

### Similarities between ArrayLists and arrays

1. **Ordering** - Both maintain the order of their elements.

2. **Duplicates** - Both allow duplicate elements to be stored.

3. **nulls** - Both allow nulls to be stored.

4. **Performance** - Since an ArrayList is backed by an array internally, there is no difference in performance of various operations such as searching on an ArrayList and on an array.

5. **Thread safety** - Neither of them are thread safe. You will learn about thread safety later but for now, be aware of the fact that accessing either of them from multiple threads may produce incorrect results in certain situations.

## 15.1.9 Map and HashMap

Map is data structure that allows you to lookup one piece of information, i.e., value, using another piece of information, i.e., key. For example, a map of country codes and country names will allow you to look up a country name using the country code, a map of student id and student name will allow you to look up a student name using the student id, and so on. Here, countrycode and student id are the keys and country name and student name are the values. In that sense, a Map is a collection of key-value pairs.

The `java.util.Map` interface of the Collections API captures this behavior. There are several implementation classes that implement the Map interface. The one that is used the most is `java.util.HashMap`.

It is important to know that `Map` does not extend `Collection` interface. `Map` is the root of a separate hierarchy of classes and interfaces that is unrelated to the `Collection` hierarchy. You need to know the following methods of `Map`:

1. `V get(Object key):` Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

2. `V put(K key, V value)`: Associates the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced by the specified value and the old value is returned. It returns null if there was no mapping for key.

3. `V remove(Object key)`: Removes the mapping for a key from this map if it is present. Returns the value or null, if not present.

4. `Set<K> keySet()`: Returns a Set of keys present in the map. Observe that keys are unique, so the return type is Set (not Collection or List).

5. `Collection<V> values()`: Returns a Collection of the values present in the map. Observe that values may be duplicate, so, the return type is Collection.

6. `void clear()`: Removes all entries stored in the map.

7. `int size()`: Returns the number of entries stored in the map.

8. `default void forEach(BiConsumer<? super K, ? super V> action)`: Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.

The above methods and their descriptions are enough for the exam but there are several interesting and useful methods in `Map` and it would be a good idea to go through the JavaDoc to learn more.

The following code illustrates how a map is typically used:

```
import java.util.*;
public class TestClass{
  static Map<String, String> idNameMap = new HashMap<>();
  static List<String> keys = new ArrayList(List.of("0", "1", "2"));
  static List<String> values = List.of("a", "b", "c");

  public static void buildMap(){
    //add key-value pairs in the map
    for(int i=0; i<keys.size(); i++){
       idNameMap.put(keys.get(i), values.get(i));
    }
    System.out.println(idNameMap); //prints {0=a, 1=b, 2=c}
  }
  public static void main(String[] args) {
    buildMap();
    String value = idNameMap.get("2");
    System.out.println(value); //prints c
    //Integer ivalue = idNameMap.get("2"); //will not compile
    keys.clear(); //removes all elements from keys List, doesn't affect the map
    System.out.println(keys.size()); //prints 0, keys is now empty
    System.out.println(idNameMap.size()); //prints 3, map still has the key value
   entries
    idNameMap.remove("1"); //remove the key 1 and its associated value from the map
    idNameMap.clear(); //remove all entries from the map
    System.out.println(idNameMap); //prints {}
  }
}
```

Following are a couple of points that you should observe in the above code.

1. **Map** and **HashMap** are parameterized classes. However, unlike a single type parameter that you saw in the case **List** and **ArrayList**, they use two type parameters. One for the type of key (denoted by K) and one for the type of value (denoted by V). In the above program, the type of the key and the value is the same, i.e., `String`.

2. Since the Map's key and value have been typed to String, the compiler will neither allow you to put anything else in it nor will allow you to retrieve anything other than a String from it.

Note

Map and HashMap are used a lot in Java development and are, therefore, a favorite topic in technical discussions. You should definitely read about them if you are going to face technical interviews.

### 15.1.10 Quiz

Given the following two classes:

```
class Base{
   public void process(Collection c){  }
}
```

and

```
public class SubClass extends Base{
   public void process(Map<Integer, String> map){
     //...code here
   }
}
```

Which of the following options are correct?

**A.** The `process` method in `SubClass` correctly overrides the `process` method in `Base`.

**B.** The `process` method in `SubClass` correctly overloads the `process` method in `Base`.

**C.** A call to `super.process(map.values());` can be inserted in the `process` method of `SubClass`.

**D.** A call to `super.process(map.keySet());` can be inserted in the `process` method of `SubClass`.

**E.** A call to `super.process(map);` can be inserted in the `process` method of `SubClass`.

**Correct answer is B, C, D.**

Since the type of the arguments of the `process` methods do not match, this is not a case of overriding. It is overloading. Thus, option **A** is wrong and option **B** is correct. Since `values()` and `keySet()` methods of `Map` return `Collection` and `Set` respectively (`Set` is a `Collection`), options **C** and **D** are correct. Since `Collection` and `Map` do not have a parent child relationship, a `Map` is not a `Collection` and so, option **E** is incorrect.

## 15.2 Advanced concepts in Generics

### 15.2.1 Using bounded type parameters

All of the generic classes that I have shown so far use single worded type parameters, i.e., there is a single word within `<` and `>` such `E`, `E1`, or `T`. At the time of using these generic classes, you can parameterize them with any type. For example, I parameterized `DataHolder<E>` to `DataHolder<String>` and `DataHolder<Integer>` . But there is no limitation on what types you can use at the time of parameterization of `DataHolder`. In technical parlance, we say that the type variable `E` is "unbounded" i.e. there are no bounds for `E`. It can be anything.

Since there is no limitation on what a type argument can be passed while parameterizing a generic class, the only assumption we could make about that type within the generic class was that it is at least an `Object` and that limited us from making any method call on the variable of that type except the ones present in the `Object` class. As promised earlier, I will now show you how to overcome this limitation in this section.

Let me redefine `DataHolder` as follows:

```
public class DataHolder<E extends Number> {
    //data field and getter and setter methods as before
    public void useData(){
       System.out.println(data.intValue());
    }
}
```

By using `E extends Number` instead of just `E` as the type parameter, I am telling the compiler that `E` is always going to be a type that extends `Number`. In other words, I am saying that `E` will always be **at least** a `Number`. Thus, `E` is now a "bounded" type variable and `E extends Number` is a bounded type parameter. Since the compiler now knows that `E` is at least a `Number`, it can safely replace `E` with `Number` in the generated class, which means, I can now call methods defined in the `Number` class such as `intValue()`, on the `data` variable.

Consequently, while using `DataHolder`, I cannot now parameterize it with just any random class. I can only parameterize it with a class that is-a `Number`. So, `DataHolder<Number>` or `DataHolder<Integer>` are fine but `DataHolder<String>` will fail to compile.

By using the `extends` keyword, we are essentially setting a limit on the super type of `E`. Since in the UML diagram the superclass is drawn above the subclass and the arrow points upwards from subclass to superclass, we call this the "upper bound of `E`", i.e., `Number` is the **upper bound** of `E`. Recall that I used the same analogy while explaining upcast and downcast (casting a variable to a super type is called upcasting). Another way to look at it is, the upper most type in the type hierarchy that matches the pattern `E extends Number` is `Number`. Any type that is lower in the type hierarchy than `Number`, such as `Integer` and `Double`, in the hierarchy is also a match but any class above `Number`, which is just `Object` in this case, is not.

It is worth mentioning here that the keyword used to specify an upper bound is always `extends` even if we want to specify an interface name as the upper bound. So, for example, `<E extends CharSequence>` is a valid upper bounded type parameter and `String` matches it because `String` **implements** `CharSequence`.

Okay, moving on, can we apply a **lower bound** on the type parameter? Can we put a limitation on `E` such that `E` can never be a subtype of `Number`? Something like `<E super Number>` ? It wouldn't be completely absurd to do so but no, Java does **not** allow setting a lower bound on a type parameter for reasons not officially documented. The most likely reason in my opinion that it is not too useful and it would have made the language grammar too complex to implement.

Note

**Having multiple upper bounds**

As discussed previously, Java allows multiple inheritance of type through the use of interfaces. So, if a class implements two unrelated interfaces, it will belong to two lineages of ancestors. It is, therefore, possible to apply an upper bound on each of its lineages. For example, the `DataHolder` class can be defined as follows:

```java
public class DataHolder<E extends Number & Comparable & Runnable > {
    //data field and getter and setter methods as before
    public void useData(){
       //can invoke methods available in Number, Comparable, as well as Runnable
       //because E is all of the three
    }
}
```

Besides getting the ability to invoke methods defined in the upper bound, applying an upper bound to a type parameter also helps us improve the design of our code. Pay close attention to this part because it creates the ground for the next section. Let me start with a example of a utility method that computes average of a list of numbers:

```java
class Averager{
  static <E extends Number> double averageExN(List<E> list){
    double sum = 0.0;
    for(Number n : list){ sum += n.doubleValue(); }
    return sum/list.size();
  }

  static double averageN(List<Number> list){
    //contains EXACTLY the same code as above
  }
}
```

Both the methods above contain exactly the same code. The only difference is in their parameter types. The `averageExN` method declares a type parameter `E` with an upper bound set to `Number`, while the `averageN` method just takes a `List` of `Numbers`. Both the methods can be invoked as shown below:

```
List<Number> listN = new ArrayList<>();
listN.add(1); listN.add(1.0); listN.add(1.0f);

double avg = Averager.averageKxN(listN);
avg = Averager.averageN(listN);
```

Observe that our list contains different kinds of `Numbers`, i.e., `Integer`, `Double`, as well as `Float`. Now, the question is, which version is preferable from a design perspective and why?

The answer is, `averageExN` is preferable because other people won't be able to invoke `averageN` if they have a list of a subclass of `Number`, which makes `averageN` too restrictive. For example, the following won't compile:

```
List<Integer> listI = new ArrayList<>();
Averager.averageN(listI); //won't compile
```

As I explained earlier, generics are invariant and so, `List<Integer>` is not a subtype of `List<Number>`. Therefore, the compiler will not let you pass a `List<Integer>` where a `List<Number>` is expected. This puts an undue restriction on the users of the `averageN` method because the ability to compute average for all kinds of numbers is a reasonable expectation from the method. Defining `E` as `E extends Number` removes that restriction. The compiler now knows that `E` could be any subtype of `Number` and so, it permits calling `averageExN(listI)`.

Although less compelling but there is one more reason to prefer the first version. What if the `averageN` method is coded by the developer as follows?

```
static double averageN(List<Number> list){
  double sum = 0.0;
  list.add(10); //introducing a bug in the code
  for(Number n : list){ sum += n.doubleValue(); }
  return sum/list.size();
}
```

Yes, the method has only four lines of code and the bug is easy to spot but the point is, if this bug is not caught during code walkthrough, it will not be caught until functional testing. Ideally, the method has no business modifying the input list but the compiler cannot object to `list.add(10)` because 10 is an Integer and a List<Number> can certainly hold an Integer.

Now, observe the averageExN method. Here, E is not just a Number. It has been upper bounded to Number. Which means, the compiler knows that at run time the list variable could point to a list of Numbers or a list of any subclass of Numbers such as a list of Integers or a list of Doubles. It also knows that it is possible to add an Integer to a list of Numbers but not to a list of Doubles. Since the compiler is not sure about the exact type of the list that the list variable will point to at run time, it will reject the list.add(10); statement with an error message saying, "

```
incompatible types: int cannot be converted to E where E is a type-variable:
    E extends Number
```

".

Note that applying an upper bound to E has not suddenly made the object pointed to by the list variable immutable. You can still do list.add(null); and list.remove(0); which modify the List object pointed to by list. But applying an upper bound to E did prevent at least one kind of bug from appearing in our code and that is better than having no protection at all. Ideally, if the caller wants to make sure that the callee does not modify the input list, it should pass in an unmodifiable list. I will show you how to do that later while going through the Collections API. The important point to understand here is that applying an upper bound to the variable has only enabled to compiler to prevent the developer from making method calls that compromise type safety **using that variable**. It has no impact on the properties of the actual object.

The general principle expounded in the above discussion is that your method should be as flexible as possible for use and should require only as much information as is absolutely necessary to do its job.

### 15.2.2 Using bounded type arguments

In the previous section, you saw that applying bounds to a generic **type parameter** while coding a generic class makes the type parameter more useful without compromising on the functionality. In this section, we will apply the same concept while parameterizing generic types.

#### Using upper bounded wildcard

You may have noticed that inside the averageExN method, we are not actually using of the type parameter E at all. We did use the fact that contents of the list are instances of Number or its subclass but the business logic of the method did not require the use of E. This is unlike the DataHolder class that we defined earlier, where we used E to define the instance field data. So, why do we need to define the type parameter for the averageExN method? Well, we don't. If we don't have any use for the type variable, we don't have to define it. At least while defining methods, we can use the "wildcard" to define the type argument, as follows:

```
static double averageExN(List<? extends Number> list){
  //same code as before
}
```

The question mark sign is the **wildcard**. It means, "any type". So, `? extends X` means "any type that extends or implements X". In other words, we are directly passing `? extends Number` as the type argument to parameterize `List` instead of defining `E` first and then passing `E` as the type argument. There is no difference between the two approaches. Using the wildcard reduces code size a little bit and is, therefore, the preferred approach if the type parameter is not needed.

Although using the wildcard may be a matter of convenience while defining a generic method, the wildcard is **required** if you want to apply bounds to the type arguments while declaring variables of generic types, so `List<E extends Number> list;` will not compile but `List<? extends Number> list;` will.

### Using lower bounded wildcard

Applying a lower bound on a type argument is allowed and is useful to prevent unnecessary access to members of an object. Here is an example:

```
void loadValuesFromDB(List<? super Number> targetList){
  //for(Number value : targetList){ ... } //will not compile
  //values.add("hello"); //will not compile
  values.add(1.0); //fine
}
void loadValuesFromFile(List<? super Number> targetList){
  for(Object value : targetList){ print(value); } //fine
  targetList.add(1); //fine
}
void loadAllNumbers(){
   ArrayList<Number> listOfNumbers = new ArrayList<>();
   loadValuesFromDB(listOfNumbers);
   loadValuesFromFile(listOfNumbers);

   ArrayList<Object> listOfObjects = new ArrayList<>();
   loadValuesFromDB(listOfObjects);
   loadValuesFromFile(listOfObjects);

   for(Number n : listOfNumbers){
      System.out.println("Value = "+n.doubleValue());
   }
}
```

Observe the usage of `<? super Number>` as the type argument to `List`. The "**? super Number**" pattern matches `Number` or any of its super types. Since the lowest subtype that matches this pattern is `Number`, we call `Number` the "**lower bound**" of this type argument. Thus, `List<?super Number> targetList` means that `targetList` points to a `List` of type `Number` or of any super type of `Number`. Since `Number` extends `Object`, the list could be either of `Number`s or of `Object`s. This is also evident from the fact that inside the `loadAllNumbers()` method, we are able to call `loadValuesFromDB()`and `loadValuesFromFile()` with `ArrayList<Number>` as well as `ArrayList<Object>` as arguments.

Let us now see how this plays out while putting objects and taking out objects from the list pointed to by the `targetList` variable.

If `targetList` points to a list of `Object`s, we can put any object in it. Let us say we try to put a String in it. But what if the list turns out to be of `Number`s? Putting a String in a list of `Number`s is precisely the type of situation we want to avoid! It will cause a `ClassCastException` when the `for` loop written further down in the above code is executed. A `Number`, however, will not cause any issue in either case. A `Number` can be safely added to a list of `Object`s (because a `Number` is-a `Object`) as well as to a list of `Number`s (because a `Number` is-a `Number`). This implies that we cannot write code assuming that `targetList` will always point to a list of `Object`s but we can write code assuming that `targetList` will always be able to contain `Number`s. For this reason, the only type of objects that the compiler will allow us to put in the list pointed to by `targetList` is `Number` and that is exactly why, in the above code, we are able to call `targetList.add(1)` and `targetList.add(1.0)`. Since both an `Integer` and a `Double` are `Number`s, the compiler is satisfied that these method invocations will never cause a violation of type safety irrespective of the kind of list `targetList` points to at run time.

When we take an object out of the list pointed to by `targetList`, we know that it is either of type `Object` or of type `Number` but we don't know exactly which one. Thus, we can't assign it to a variable of type `Number`. What if we try to assign it to a `Number` variable and it turns out to be of type `Object`? In fact, the only type of variable we can assign this object to is `Object` (because every object **is-a** `Object`). That is why the for loop written in `loadValuesFromDB()` fails to compile but the one written in `loadValuesFromFile()` compiles fine. This shows that when a lower bound is applied to a type argument, the type information associated with the generic type variable is hidden. This prevents the programmer from accessing any type specific method or property **using that variable**. Again, this is a not a fool proof security mechanism. That is not even the objective of generics. The objective is only to reduce the opportunities for writing brittle and buggy code.

### Using unbounded wildcard

Just like we used the unbounded type parameter `E` while defining the `DataHolder<E>` class earlier, we can use the unbounded wildcard as an argument while parameterizing a generic type to signify that the exact type argument is unknown. For example: `DataHolder<?> dh = new DataHolder<> ();` Here, `dh` is supposed to point to a `DataHolder` object containing a type that is completely unknown to the us. That type could by anything but whatever it is, we don't know it. The problem in dealing with an unknown type is that we can't do anything interesting with it and the reason is the same as before - what if the `DataHolder` object that `dh` is pointing to contains something different from whatever our assumption is? Since it has no lower bound, no type will satisfy the type parameter `E` required for the `setData` method and so, for example, we can't do `dh.setData("string");`, and since it has no upper bound we cannot assign the return value of the `getData` method to a variable of any type except `Object`. Effectively, `<?>` is the same as `<? extends Object>`. All that we can do is `Object obj = dh.getData();`

So, what is it good for? Well, if the business logic of your method doesn't depend on knowing anything about the actual type of the argument, this is the recommended option because it doesn't restrict your method to using a particular parameterized type and it doesn't force your users to disclose anything about the type of the argument that they are passing to your method. For example, if all you want to do is to write a utility method that logs the contents of a `DataHolder` object, you can define it as follows:

```
class Utility{
    public static void logDH(DataHolder<?> dh){
        System.out.println(dh.getData());
    }
}
```

Anyone can call the `logDH` method and pass a `DataHolder` parameterized to any type like this:

```
public class TestClass {
   public static void main(String[] args){
      DataHolder<Number> dhN = new DataHolder<>();
      DataHolder<String> dhS = new DataHolder<>();
      Utility.logDH(dhN);
      Utility.logDH(dhS);
   }
}
```

If you change the type parameter of the `logDH` method from `DataHolder<?>` to `DataHolder<Object>`, the calls in the main method will not compile because, as explained before, `DataHolder<Number>` and `DataHolder<String>` are not subtypes of `DataHolder<Object>`.

### 15.2.3 Covariance and Contravariance in Generics

We discussed that due to type erasure, the JVM cannot make out the difference between any two parameterizations of the same generic type. However, the compiler has access to all the type information present in the parameterizations and it can use this information to establish fictional supertype-subtype relationships among various parameterizations of generic types. Establishing these relationships is important because it allows the compiler to apply the **principle of substitutability** while validating the type safety of method calls and that allows us to use a subtype object where ever an object of a supertype is needed, such as in **assignment statements**, **method arguments**, and **return values**. Without such relationships, covariance would also not be possible in generics and they would become too rigid to be useful.

#### Establishing supertype-subtype relationships among parameterized types

There are only the following three rules that you need to apply while determining whether a parameterized type is a subtype of another parameterized type:

1. If the type arguments of the two parameterized types are an exact match, then they have the same relationship as their generic types.
   For example, `ArrayList<?>` is a subtype of `List<?>` and `ArrayList<? extends Number>`

is a subtype of `List<? extends Number>` because `ArrayList` is a subtype of `List` and their type arguments are an exact match.

2. If the generic types of the two parameterized types are an exact match, then the relationship is determined by their type arguments. The type argument that matches a subset of the types matched by the other is considered a subtype of the other.
   For example, since an unbounded wildcard (i.e `<?>` ) matches a superset of the types matched by an upper bounded wildcard (i.e. `? extends X`) and also matches a superset of the types matched by a lower bounded wildcard (i.e. `? super X`), `List<?>` is a supertype of `List<? extends Number>` and is also a supertype of `List<? super Number>` . But `List<? extends Number>` and `List<? super Number>` have no relationship because `? extends Number` and `? super Number` match a disjoint set of types. Similarly, `List<? extends Number>` is a subtype of `List<? extends Object>` and `List<? super Number>` is a subtype of `List<? super Integer>` .

3. If neither of the two are an exact match, then both of the above rules must be satisfied individually to establish a relationship, i.e., their generic types as well as their type arguments must have the same supertype-subtype relationship.
   For example: `ArrayList<Integer>` is a subtype of `List<? extends Number>` because `ArrayList` is a subtype of `List` and its type argument `Integer` also matches only a subset of the types matched by `? extends Number`, but `ArrayList<? extends Number>` has no relationship with `List<Integer>` .

### Covariance in generics

Once you understand how to establish a supertype-subtype relation between parameterized types, applying the rule of **covariant returns** is easy, as illustrated by the following code:

```
class Base{
   List<? extends Number> getList(){
      return new ArrayList<Integer>(); //returning a subtype
   }
}
class Sub extends Base{
   @Override
   List<? extends Integer> getList(){ //using a subtype as the return type
      return new ArrayList<Integer>(); //returning a subtype
   }
}
```

Observe the following points in the above code:

1. In `Sub`, I am able to specify a subtype as the return type of the `getList()` method, which overrides the `getList()` method of `Base` ( `List<? extends Integer>` is a subtype of `List<? extends Number>` , as per rule 2 explained above).

2. In `Base` as well as in `Sub`, I am able to return a subtype object from the `getList()` method (`ArrayList<Integer>` is a subtype of `List<? extends Integer>` as per rule 3 explained above).

### Contravariance in generics

Technically speaking, contravariance refers to the situation where you are able to use a super type in place of a subtype. Some programming languages allow method parameters to exhibit contravariance while overriding a super class's method. It is called "contra" because while you are going down the class hierarchy from supertype to subtype, the parameter type of the method goes up from subtype to supertype, i.e., in the opposite direction. But Java, and most other languages including C++ and C#, do not allow this. Therefore, the following code does not compile:

```
class Base{
   void processList(ArrayList<? super Number> list){  }
}
class Sub extends Base{
   void processList(ArrayList<? super Integer> list){ } //does not compile
   //void processList(List<? super Integer> list){ } //will compile fine
}
```

The error message is:

```
error: name clash: processList(ArrayList<? super Integer>) in Sub and
   processList(ArrayList<? super Number>) in Base have the same erasure, yet neither
   overrides the other
```

From the JVM's perspective, both the methods have exactly the same signature due to type erasure and the method in `Sub` should override the method in `Base`. Yet, from the compiler's perspective, the method in `Sub` has a contravariant parameter type. This not allowed. However, the second method in `Sub` will compile fine, because its signature is different from the signature of the method in `Base`. Thus, although it is not a valid override, it is a valid overload.

### The Get/Put principle

You may see the usage of the term contravariance in articles while talking about things that you can do with upper bounded and lower bounded wildcards. As discussed previously, a list with an upper bounded wildcard allows you to take things out of it while a list with a lower bounded wild card allows you to add things to it. In other words, use `? extends X` to "get" things and use `? super X` to "put" things.

You may also use the mnemonic PECS, which stands for **P**roducer **E**xtends, **C**onsumer **S**uper to remember this rule. If you want a create a producer, i.e., the one that produces values for you, use `extends` and if you want to create a consumer, i.e., the one that takes values from you, use `super`.

```
List<? extends Number> producer = new ArrayList<>();
Number n = producer.get(0); //produces numbers

List<? super Number> consumer = new ArrayList<>();
consumer.add(n); //consumes numbers
```

### 15.2.4 Quiz

**Q1.** Given the following variable declarations:

```
List<?> v1 = new ArrayList<>();
List<Object> v2 = new ArrayList<>();
var v3 = new ArrayList<>();

List<? extends CharSequence> v4 = new ArrayList<>();

ArrayList<? super CharSequence> v5 = new ArrayList<>();
ArrayList<String> v6 = new ArrayList<>();
ArrayList<? super String> v7 = new ArrayList<>();
```

Identify the variables that can be passed as arguments to the following method definitions:

```
void m1(List<? super CharSequence> list){ }
void m2(List<? extends String> list){ }
void m3(List<?> list){ }
```

**Correct answer is**

**m1**: v2, v3, and v5

**m2**: v6

**m3**: All variables

Remember the principle of substitutability, which allows you to use a subtype object where ever a super type object is needed. So, all you need to do here is to see if the declared type of the variable is a subtype of the type of the method parameter. The actual object pointed to by the variable has no role to play here.

So, let's apply the rules of subtyping to `m1` and the given variables. To do so, first write down the set of matching types for the wildcards used in the method parameter.

The wildcard in method m1 is `? super CharSequence`, which matches only two types - `CharSequence` and `Object`. Let's call this **M1 Set**.

1. The type of the variable `v1` is `List`, which is the same as the parameter type in `m1`. But the type argument in `v1` is `?`, which matches all types. A set of all types is not a subset of M1 Set. Hence, `List<?>` is not a subtype of `List<? super CharSequence>` and so, `v1` cannot be passed to `m1`.

2. The type argument `<Object>` matches only one type, i.e., `Object`. It is indeed a subset of M1 Set. Therefore, `List<Object>` is a subtype of `List<? super CharSequence>`, and so, `v2` can be passed to `m1`.

3. The case of v3 is interesting. The usage of var causes the compiler to infer the type of v3 using the type of the object that is being assigned to it, which is ArrayList. Furthermore, since the statement does not include any information for determining the parameterization of ArrayList, the compiler infers the type argument as Object. So, the type of v3 is ArrayList<Object> . Now, this becomes similar to v2. ArrayList is a subtype of List and the set of the types matched by <Object> is a subset of M1 Set. Therefore, ArrayList<Object> is a subtype of List<? super CharSequence> and so, v3 can be passed to m1.

4. The variable v4 uses ? extends CharSequence, which matches CharSequence and all of its subtypes (i.e. String, StringBuilder, and StringBuffer). You can see that this set is disjoint to M1 Set, which means, List<? extends CharSequence> and List<? super CharSequence> have no relationship. Hence, v4 cannot be passed to m1.

5. The generic type of v5 is ArrayList, which is a subtype of List and its type argument is the same as the one used in m1. Thus, ArrayList<? super String> is a subtype of List<? super String> . Hence, v5 can be passed to m1.

6. The generic type of v6 is ArrayList, which is a subtype of List but the set of matches produced by its type argument contains just String, which is not a subset of M1 Set. Hence, v6 cannot be passed to m1.

7. The generic type of v7 is ArrayList, which is a subtype of List but the set of matches produced by its type argument contains String, CharSequence, and Object, which is not a subset of M1 Set. Hence, v7 cannot be passed to m1.

You may work out the possibilities for m2 in the same manner. Don't get confused by m3. Just apply subtyping rules. I have already explained what List<?> means in the first point above.

### 15.2.5 Extending generic types

Although not important for the OCP exam, it is good to know the ways in which you can extend a generic class or implement a generic interface. Assuming the following interface, there are three ways in which you can extend/implement it.

```
interface Processor<T>{
   void process(T t);
}
```

1. Declare a type parameter and pass the same to the base type:

   ```
   interface AdvancedProcessor<U> extends Processor<U>{
     void processFast(U u);
   }
   ```

   Here, AdvancedProcessor declares a type parameter U and applies the same to Processor. So, for example, if you parameterize AdvancedProcessor to Integer, i.e., AdvancedProcessor<Integer> , you would be parameterizing Processor to Integer as well.

2. Eliminate the type parameter by parameterizing the base type:

   ```
   class StringProcessor implements Processor<String>{
     public void process(String s){ System.out.println(s); };
   }
   ```

   Here, `StringProcessor` has parameterized `Processor` to `Processor<String>` . There is no need to declare a type parameter in `StringProcessor` now.

3. Eliminate generics altogether:

   ```
   class RawProcessor implements Processor{
     public void process(Object o){ System.out.println(o); };
   }
   ```

   As discussed earlier, the type parameter is replaced by an actual type during compilation anyway due to type erasure. So, it is possible for the subtype to eliminate generics altogether by implementing the generic method of the base type using the same real type that satisfies the type parameter used in base type. In this case, `Object` satisfies the type parameter used in `Processor`.

Of course, a subtype may declare new type parameters for its own use as well. For example:

```
interface DualProcessor<T, U> extends Processor<U>{
    void processDual(T t);
}
```

You could parameterize `DualProcessor` like this:

```
DualProcessor<Integer, String> justForFun = new DualProcessor<>(){
  public void process(String s){ System.out.println("Processing String "+s); }
  public void processDual(Integer i){ System.out.println("Processing dual Integer"+i); }
};
```

Observe that we are using two type parameters here - `T` and `U`. `T` is used only by `DualProcessor` while `U` is passed on to `Processor`. While instantiating `DualProcessor`, `T` is set to `Integer` and `U` is set to `String`. Therefore, we must implement the `process` method of `Processor` with a `String` parameter and the `processDual` method of `DualProcessor`with an `Integer` parameter.

### 15.2.6 Limitations and prohibitions in Generics

Here a few things that are either not possible or are prohibited in generics, listed in the decreasing order of their importance from the exam perspective.

1. Because a type variable is just a placeholder and not a name of an actual type, you cannot instantiate it using the new operator like other regular classes, i.e., you can't do `new E();`, if `E` is a type variable.

2. For the same reason as above, you cannot use a type variable in an `instanceof` operator, i.e., `objRef instanceof E` is invalid. However, you can use a parameterized type in an `instanceof` operator as long as its type argument matches the type argument specified for the reference variable.
   For example, the following is fine:

   ```
   List<? extends Number> listInts = new ArrayList<Integer>();
   if(listInts instanceof ArrayList<? extends Number>){ } //fine
   ```

   But the following is not:

   ```
   if(listInts instanceof ArrayList<Integer>){ } //will not compile
   ```

   This is because the type argument of the actual object pointed to by `listInts` is not known to the JVM.

3. Because arrays are reified and generics are type erased, you cannot create an array of a parameterized type. For example, `new DataHolder<String> [10];` will not compile. Although, declaring an array variable of a generic type such as `DataHolder<String> [] dh;` is valid because you can create an array of the unparameterized generic type. So, `DataHolder<String> [] dh = new DataHolder[10];` is valid, although it will generate an

   ```
   "unchecked
          or unsafe operations"
   ```

   warning at compile time.

4. You cannot use primitive types in type parameters or type arguments. So, something like `<E extends int>` or `<long>` is not allowed.

5. Type variables are associated with non-static contexts and are not allowed in static contexts. This point is too complicated to explain in detail here and since it is way beyond the scope of the exam, I will leave it at that. It is sufficient to know here that you cannot define a static variable of the type of the type variable. For example, `static E e;` is invalid if `E` is a type variable.

6. I haven't discussed exceptions yet, but it is worth mentioning here that it is not permitted to extend a generic class from `java.lang.Throwable` directly or indirectly. There are a few more rules on this topic but since there is rarely a need to mix exceptions and generics, I will not go into it any further.