A white text on a black background

Description automatically generated

1. String result = numbers.stream().filter(n-> n%3==0 || n%2==0).map(n->’A’+(n+1)+’B’).reduce(“”,String::concat);
2. int result = numbers.stream().filter(n->n%4==0).map(n->n+1).reduce(0,(sum,n)->(sum+n)%2)

A white paper with black text

Description automatically generated

int result = numbers.stream().filter(n->n%3==0 || n%4==0).map(n->(n+1)%3).reduce(0,(sum,n)->(sum+n)%2)

A black text on a white background

Description automatically generated

String result = numbers.stream().filter(n->n%5 ==0 || n%4 ==0).map(n->”N”+n+”R”).reduce(‘’,String::concat).

A white paper with black text

Description automatically generated

String result = numbers.stream().filter(n-> n%2==0 || n%5 == 0).map(n->”N”+n+”R”).reduce(“”,String::concat);

A white background with black text

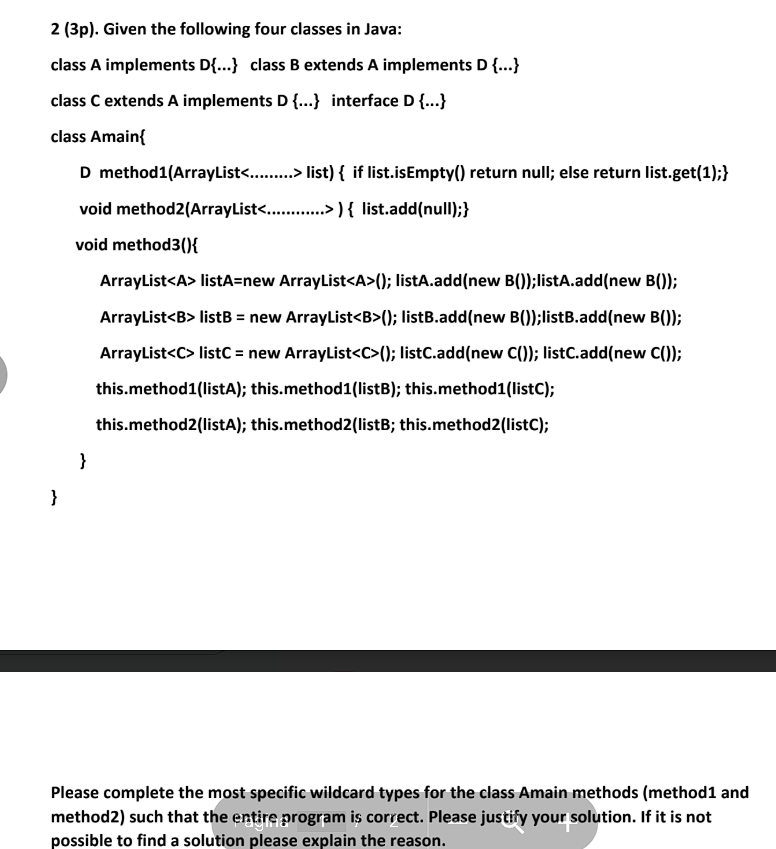
Description automatically generated

int result = numbers.stream().filter(n->n%3==0 || n%7 == 0).map(n->(n-1)\*10).reduce(0,(sum,n)->(sum+n)%5);

A white text on a white background

Description automatically generated

String result = numbers.stream().filter(n->n%2 == 0 || n%3 == 0).map(n->”A”+(n+1)+”B).reduce(“”,String::concat);



Method 1 takes as parameter an ArrayList an returns D.The method accessed the second element of the list via list.get(1).

The return type is D , so the list must cotain elements that are either of instances D or its sublasses.The most restrictive type is ! extends D because the list can gold any type that is a subtype of D.The wildcard guarantees that each element from the list is at least D or a sublass of D.

The method 2 takes as parameter an ArrayList and adds null to the list.The most restrictive type is ? extends A because we can safely assume that all elements from the list are of type A or one of its sublasses.The key part is adding null , because null can be assigned to any reference type.

A screenshot of a computer program

Description automatically generated

In method 1 we use get an Array List and return an element of type D.If the list is empty null will be returned , else the second element of the list will. The return type is D so the list given as a parameter should be of type D or its subclasses. The most restrictive is ? extends D because the list can hold and return any elements of type D or any of its sublasses. This wildcard guarantees that any element contained in the list and returned is at least of type D or any of its sublasses.

In method 2 we get an Array List and an element of type D.For this to be valid , the list must accept elements of type D or any subclass of D. Since elem is of type D and the list contains element of type C or a sublass of C , the wildcard ? super C is the most specific choice. This ensures that lists of type of type ArrayList<A> , ArrayList<B> or ArrayList<C> can all accept elem.

A screenshot of a computer program

Description automatically generated

For method 1 , we need to put a return value as well as a wildcard. Our method takes an element from the list and then returns it. The most specific wildcard type for the method 1 is ? extends A , having as return value the type A . In this case we ensure that any element from the list is at least of type A or any of its subclasses (B,C). This ensures safe assignment since A is a base class.

For method 2 there is no wildcard that can be used. In that methodwe get a list and an element of type A and we add it to the list. To add an element of type A safely , the list must be of type A or a supertype that guarantees compatibility with A.However ArrayList<B> and ArrayList<C> are specific lists that only accept elements of type B or C.

A screenshot of a computer program

Description automatically generated

For method 1 we get as parameter an ArrayList , from which we try to return an element of type D, if list is not null. The most specific wildcard we can use is ? extends D which ensure that the lists contain elements of type D or any of its sublasses such that we can safely return an element from the list as type D or a subtype of D.

For method 2 we get as parameter an ArrayList and an element which will be added in the list. The element is of type C meaning that the list should countain element of at least type C .The most specific wildcard we can use is ? super C . This unsures that you can safely add a C Object into the list because the list will be able to holst at least C and potentially giher-lever objects.

A screenshot of a computer program

Description automatically generated

For method 1 we get an ArrayList from which we extract an element if the list is not null and return it. Our method returns an element of type D , meaning that ArrayList is expected to have element of type D or any of its sublasses. In this context , the most specific wildcard will be ? extends D . This ensures that the ArrayList contains elements of type D or lower , making the return valid for any input array.

For method 2 we get an ArrayList and an element of type C which will be added in the list. The input list must be capable of holding C or any supertype of C,In this case we expect the list to have elements of at least type C making the add valid. In this case , the most specific wildcard is ? super C.

A screenshot of a computer code

Description automatically generated

The given code is partially correct. Some mistakes made in the code is misspelling the word implements as iplments. This is a simple typo but it prevents the code from compilling.The method getS1() in class A does not match the signature required by the interface ln1. The interface defines int getS1(int ) which means that A should accept a parameter. Also the signature is incorrect because we should get a variable as a parameter of type int but instead we get only the type.Another mistake made along the code lies in the static function getS which tries to access a non-static variable. In Java , static methods cannot access not-static fields directly because they do not have an instance of the class.

A screenshot of a computer code

Description automatically generated

The code provided is incorrect due to a few issues. Firstly , the method getS() is static but it calls getS1() which is a non-static function. In Java static methods cannot access non-static fields because they do not have instance of the class. Another mistake is the mismatch between the abstract function implemented in the abstract class and the method implemented in class A. In class A , the method doesn t get any parameters while in the abstract class it does. This causes a mismatch of the signature.

A white paper with black text

Description automatically generated

The code provided is mostly correct but has some issues. Firstly, the static function getS() tries to access the non-static function int getS1(). In Java , static methods don t have direct access to static fields because they don t have instances of the class. Also the method getS1() calls getS(). The issue is that getS() also calls getS1() which leads to unintentional recursion leading to an overflow error because of the infinite recursion.

A screenshot of a white paper with black text

Description automatically generated

The code provided is mostly correct but has some issues. Firstly , the static function getS() tries to access the non-static function in getS1(). In Java, static methods don’t have direct access to non-static fields because they do not have instances of classes. Another problem is the infinite recursion which occurs because the method getS1 calls the function getS() which itself calls again method getS1()