2018 T2 Sept Q2:

```
interface P{

    String p = "PPPP";

    String methodP();

}

interface Q extends P{

    String q = "QQQQ";

    String methodQ();

}

class R implements P, Q{

    public String methodP(){

        return q+p;

    }

    public String methodQ(){

        return p+q;

    }

}



public class Practice {

    public static void main(String[] args){

        R r = new R();

        System.out.println(r.methodP());

        System.out.println(r.methodQ());

    }

}
```

Warning:

Redundant superinterface P for the type R, already defined by Q

Output:

QQQQPPPP

PPPPQQQQ

2019 T2 Nov Q3

```
class Test{

    int a=10;

    static int b=20;



    public static void main(String[] args){

        Test t1 = new Test();

        t1.a = 100;

        t1.b = 200;



        Test t2 = new Test();



        System.out.println("t1.a="+t1.a+" t1.b="+t1.b);

        System.out.println("t2.a="+t2.a+" t2.b="+t2.b);

    }

}
```

Output:

t1.a=100 t1.b=200

t2.a=10 t2.b=200

If a variable is declared as static, only one copy of the variable exists. Memory space for a static variable is created as soon as the class in which it is declared is loaded. All objects created from the class share access to the static variable. Changing the value of a static variable in one object changes it for all others.

```
interface CanFight{

    void fight();

}



interface CanSwim{

    void swim();

}



interface CanFly{

    void fly();

}



class ActionCharacter{ //abstract class

    void fight();

}



class Hero extends ActionCharacter implements CanFight, CanSwim, CanFly{

    public void fight(){ System.out.println("Hero fight");}

    public void swim(){ System.out.println("Hero Swim");}

    public void fly(){ System.out.println("Hero fly");}

}

class Adventure{

    static void t(CanFight x){

        System.out.println("t(CanFight x)");
```

```
        x.fight();}
    static void u(CanSwim x){
        System.out.println("u(CanSwim x)");
        x.swim();}
    static void v(CanFly x){
        System.out.println("v(CanFly x)");
        x.fly();}
    static void w(ActionCharacter x){
        System.out.println("w(ActionCharacter x)");
        x.fight();}

    public static void main(String[] args){
        Hero i = new Hero();
        t(i);
        u(i);
        v(i);
        w(i);

    }
}
```

Output:

t(CanFight x)

Hero fight

u(CanSwim x)

Hero Swim

v(CanFly x)

Hero fly

w(ActionCharacter x)

Hero fight

Can we override static method in Java?

No, you cannot override static method in Java because method overriding is based upon dynamic binding at runtime. Usually static methods are bonded using static binding at compile time before even program runs.

When super class and sub class contains same method including parameters and if they are static.

The method in the super class will be hidden by the one that is in the sub class. This mechanism is known as **method hiding**.

```
public class ClassA {

public void methodOne(int i) { ... }

public void methodTwo(int i) { ... }

public static void methodThree(int i) { ... }

public static void methodFour(int i) { ... }

}

public class ClassB extends ClassA {

public static void methodOne(int l) {... }

public void methodTwo(int l) {.. }

public void methodThree(int r) {...}

public static void methodFour(int r) {...}

}
```

1. Which method overrides a method in the superclass?

>> methodTwo

2. Which method hides a method in the superclass?

>> methodFour

3. What do the other methods do?

>> methodOne : `Static is added here (Causes Compiler Error)`

>> methodThree : `Static is removed here (Causes Compiler Error)`

Can we overload static methods?

Yes, we can have two or more static methods with the same name, but differences in input parameters

```
public class Test {

    public static void foo() {

        System.out.println("Test.foo() called ");

    }

    public static void foo(int a) {

        System.out.println("Test.foo(int) called ");

    }

    public static void main(String args[])

    {

        Test.foo();

        Test.foo(10);

    }

}
```

```
Test.foo() called
Test.foo(int) called
```

Can we overload methods that differ only by static keyword?

We cannot overload two methods in Java if they differ only by static keyword (number of parameters and types of parameters is the same).

```
public class Test {
    public static void foo() {
        System.out.println("Test.foo() called ");
    }
    public void foo() { // Compiler Error: cannot redefine foo()
        System.out.println("Test.foo(int) called ");
    }
    public static void main(String args[]) {
        Test.foo();
    }
}
```

```
Compiler Error, cannot redefine foo()
```

```
class Practice{


    public static void main(String[] args) throws ArithmeticException{
        try{
            throw new ArithmeticException("new AE");
        }
        catch(ArithmeticException x){
            //throw new ArithmeticException("new AE 2");
            throw x;
        }
        finally{
            System.out.println("Did this print?");
        }
        System.out.println("Did this print too?");
    }
}
```

Unreachable code

```
1. //Java Program to understand the working of string concatenation operator
2. public class StringToIntExample{
3. public static void main(String args[]){
4. //Declaring String variable
5. String s="200";
6. //Converting String into int using Integer.parseInt()
7. int i=Integer.parseInt(s);
8. System.out.println(s+100);//200100, because "200"+100, here + is a string
   concatenation operator
9. System.out.println(i+100);//300, because 200+100, here + is a binary plus
   operator
10.}}
```

```
200100

300
```

Ese 2017

```
public class Main
{
   public static void main(String[] args) {
      try{
            int a[]= {1,2,3,4,5};
            for (int i =0; i<7 ; ++i){
               System.out.println(i);
            System.out.println(a[i]) ;
            }

      }
      catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("e");
            }
   }
}
```

OUTPUT:

0

1

1

2

2

3

3

4

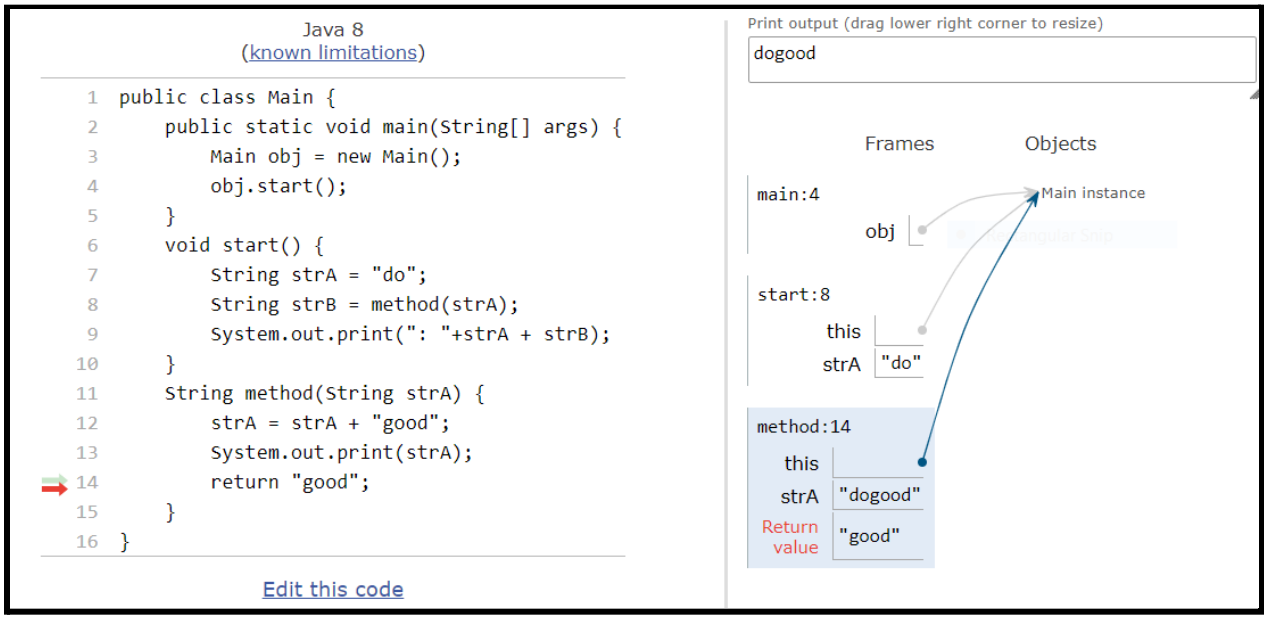
4

5

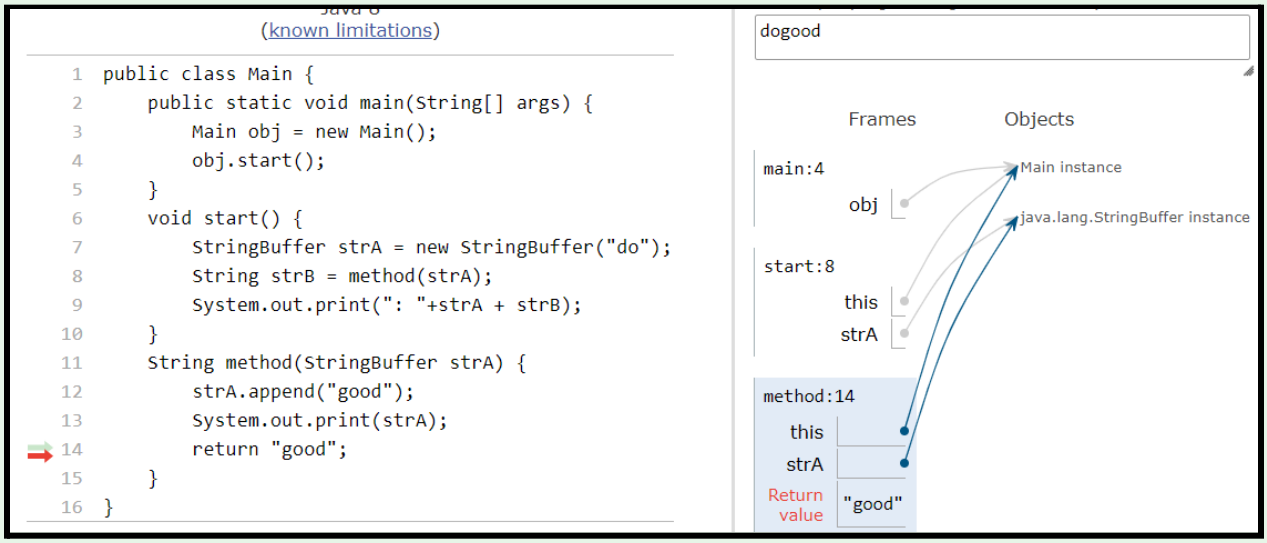
5

e

Class Main

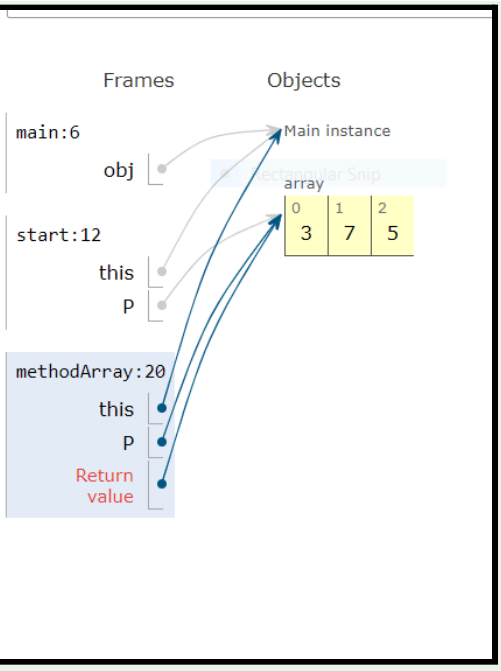


dogood : dogood



dogood: dogoodgood

```
1  public class Main
2  {
3    public static void main(String[ ] args)
4    {
5      Main obj = new Main( );
6      obj.start( );
7    }
8
9    void start( )
10   {
11     long [ ] P= {3, 4, 5};
12     long [ ] Q= methodArray (P);
13     System.out.print (P[0] + P[1] + P[2]+":");
14     System.out.print (Q[0] + Q[1] + Q[2]);
15   }
16
17   long [ ] methodArray (long [ ] P)
18   {
19     P[1]=7;
20     return P;
21   }
```



Output:

15:15

• When an object is passed to a method, the actual parameter and the formal parameter become aliases of each other.

MultiLevel inheritance

```
class A {
   float i;
   A(){
      System.out.println("Class A ()");
   }
   A(float a){
      i = a;
      System.out.println("Class A (i=a) :"+i); // i in A
   }
}

class B extends A {
   float i;
   B(){
      System.out.println("Class B ()");
   }
   B(float b){
      super(1.1F);
      System.out.println("Class B (++super.i) :"+(++super.i)); // i in A
      i = b;
      System.out.println("Class B (i=b) :"+i); //i in B
   }
}

class C extends B {
```

```
    float i;
    C(){
        System.out.println("Class C ()");
    }

    C(float c){
        super(1.2F);
        System.out.println("Class C (super.i) :"+super.i); //i in B
        i = c;
        System.out.println("Class C (i=c) :"+i); //i in C
    }
}
public class Main {
        public static void main(String[] args) {
                A a1 = new A();
                System.out.println("-------------");
                A a2 = new A(1.0F);
                System.out.println("_____________");
                B b1 = new B();
                System.out.println("-------------");
                B b2 = new B(2.0F);
                System.out.println("_____________");
                C c1 = new C();
                System.out.println("-------------");
                C c2 = new C(3.0F);
        }
```

}

Output:

```
Class A ()
----------
Class A (i=a) :1.0
__________
Class A ()
Class B ()
----------
Class A (i=a) :1.1
Class B (++super.i) :2.1
Class B (i=b) :2.0
__________
Class A ()
Class B ()
Class C ()
----------
Class A (i=a) :1.1
Class B (++super.i) :2.1
Class B (i=b) :1.2
Class C (super.i) :1.2
Class C (i=c) :3.0
```

Nested Class

```
class A{
   private int a = 20;
   A(){
         B b1 = new B();
      System.out.println("class A:"+a);
   }
   class B {
   public int b = 10;
   B(){
      System.out.println("class A:"+a);
      System.out.println("class B:"+b);
   }
}
}
public class Main
{
         public static void main(String[] args) {
                  A a1 = new A();
         }
}
```

Output:

class A:20

class B:10

class A:20

Note that any method of class A cannot directly access int b (i.e. variable of inner class B)
