Problem 1 (compile-time polymorphism)

a. Create two classes `Point`, `Line` where a point (in 2D) is defined by two double fields `x` & `y` and a line can be defined by two points.
b. The `Point` class has only one constructor which takes two decimal values as its arguments, whereas `Line` has two constructors (constructor overloading)- one accepts four decimal values for the two points as arguments and the other one directly accepts two `Point` objects. Reuse the definition of the first constructor by calling `this(...)` with proper arguments to invoke that constructor.
c. The `Line` class has a method to calculate the distance of the line from a given point as argument.
d. The `Line` class also has another two methods each called `doesIntersects()`. One of them takes a `Point` object as an argument and the method returns `true` if the given point is on the line segment (defined by its two constituent end points) and `false` otherwise. Another one takes a `Line` object as its argument and verifies whether the intersection point is on the original line segment (if the two lines intersects at all).
e. Write a `main` method for demonstration.

Problem 2 (run-time polymorphism)

Recall that *abstraction* is a key feature of any Object-Oriented Programming Language and Java is no exception. There are various ways to achieve abstraction in Java. Using access modifiers (`private`, `protected` etc.) is the simplest way hide members of a class thus achieving abstraction. There are two ways to declare an entire class as an abstract entity - abstract class and interface.

In Java an *abstract class* is a class that is declared `abstract`—it may or may not include abstract methods. Abstract classes cannot be instantiated (object variables can be declared but no memory can be allocated via constructor call), whereas they can be inherited/derived (this is in fact purpose of making such a class).

```
public abstract class ABC {
        ...
}
```

An *abstract method* is a method that is declared without an implementation i.e. without braces and body, and followed by a semicolon.

```
abstract void doSomething(...);
```

Ref: https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

a. Create an abstract class `Shape` which have two private fields `area` and `perimeter` and one `public abstract void` method `display()`.
b. Now create another three classes `Square`, `Rectangle` and `Circle` each inheriting from the `Shape` class. Add relevant fields to each of the three classes to calculate area and the perimeter of the corresponding shape. The values of these fields are passed as arguments to the constructor and the calculation is also done inside the constructor method of each class.
c. Implement the unimplemented method `display()` in each of the derived classes so that it prints the type of the shape(i.e. class name) and values of `area` and `perimeter` calculated.
d. Create a `main` method to demonstrate the *runtime polymorphism* by using these classes. Hint: create an object variable of type `Shape` and assign it instances of the derived classes one by one and call the `display()` method.

Problem 3 (abstract class vs interface)

As discussed earlier Interface also allow us to create abstract entity. Now two question arises what is an interface and how is it different from an abstract class! Let us address them one by one. We interact with an object through its available methods. An interface simply describes what these methods are, its like buttons on an electronic gadget. Like these buttons methods of an interface has no actual function rather they are associated with concrete definitions in their implementation in inherited classes similar to background circuitry of those buttons.

In other words, an `interface` is a class like construct which contains a few abstract methods (without the `abstract` keyword) and maybe some fields which must be declared as `static` and/or `final`. Another point to note that the `implements` keyword is used (instead of `extends`) when a class inherits from an interface.

Ref: https://www.javatpoint.com/difference-between-abstract-class-and-interface compares the two.

Now a third question arises why do we need interface if abstract class can serve the purpose! Recall that in Java we are not allowed to inherit from multiple class simultaneously. But we can inherit from multiple interfaces simultaneously. Moreover, we can have a class that `extends` another class as well as `implements` several interfaces. Thus, both them are present in Java to serve different purposes.

a. Create an interface called `Sortable` which has a method called `sort()`. Create another interface called `Printable` which has a method called `print()` (assume it will print to console i.e. standard output stream).
b. Create an abstract class called `DataStore` which is `Printable` and has a field which stores the number of elements in it and two abstract methods to insert and delete elements.
c. Extend this `DataStore` class to create three classes called `DataStoreArray`, `DataStoreList` and `DataStoreBST` of which first two are `Sortable`. Put required fields into these classes and implement relevant methods. You may create other classes/interfaces if required. Assume all data elements are integers.

Problem 4 (generics)

Suppose you have defined a Stack using array of integers for some application. A few days later suppose another application requires a Stack with floating point values. Similarly, another application may require a Stack that handles some complex datatype. Instead of writing new Stack classes for each of the applications Java solves problem by allowing us to define generic types.

A *generic type* is a generic class or interface that is parameterized over types. The following example will help you understand the concept and syntax of using generic types.

Ref: https://docs.oracle.com/javase/tutorial/java/generics/types.html

Ref: https://stackoverflow.com/questions/529085/how-to-create-a-generic-array-in-java

```
public class Stack<T> {
        private T[] arr;
        private int SIZE, top;

        public Stack() {
                this(10);
        }

        public Stack(int SIZE) {
                this.SIZE = SIZE;
                arr = (T[]) new Object[SIZE];
                top = -1;
        }

        void push(T data) throws Exception {
                if(top == SIZE -1)
                        throw new Exception("Stack Overflow");
                arr[++top] = data;
        }

        T pop() throws Exception {
                if(top < 0)
                        throw new Exception("Stack Empty");
                return arr[top--];
        }

        public static void main(String[] args) {
                Stack<Integer> s = new Stack<>();
                //rest of the code
        }
}
```

Now implement the problem 3 using generic types.