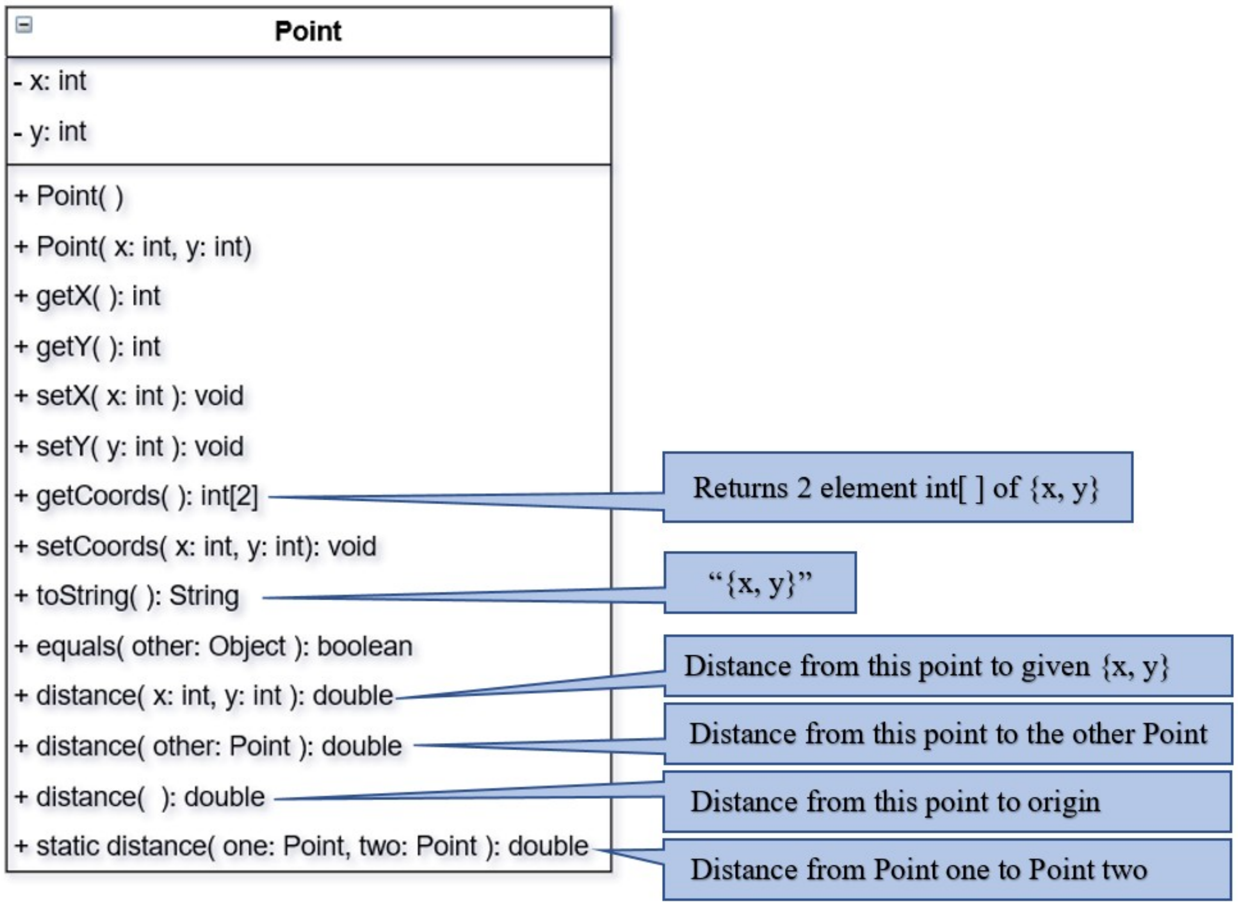
CSCI165 Computer Science II
Lab Assignment
Inheritance and Composition and a glimpse into Polymorphism

Objectives:

- Practice the concepts of inheritance by creating a simple inheritance hierarchy
- Demonstrate that inherited attributes and behaviors can be used correctly
- Demonstrate proper use of the **super** keyword
- Demonstrate and utilize method overloading and method overriding
- Generalize objects into collections of base types

Task One: The `Point` *base class*

We will use this class as the base of our inheritance hierarchy. We will then extend this class via the inheritance mechanism to quickly achieve specialized behavior.


Point
- x: int
- y: int
+ Point( )
+ Point(x: int, y: int)
+ getX( ): int
+ getY( ): int
+ setX(x: int): void
+ setY(y: int): void
+ getCoords( ): int[2]
+ setCoords(x: int, y: int): void
+ toString( ): String
+ equals(other: Object): boolean
+ distance(x: int, y: int): double
+ distance(other: Point): double
+ distance( ): double
+ static distance(one: Point, two: Point): double
Returns 2 element int[ ] of {x, y}
"{x, y}"
Distance from this point to given {x, y}
Distance from this point to the other Point
Distance from this point to origin
Distance from Point one to Point two

Define a class called `Point`, which models a 2D point with `x` and `y` coordinates. The Point class should contain each of the following features:

- Two instance variables `x` (`int`) and `y` (`int`).
- A no-argument constructor that constructs a point at the default location of `(0, 0)`.
- An overloaded constructor that constructs a point with the given `x` and `y` coordinates.
- Also add a copy constructor for use when privacy is a concern. (not in the UML)
- Getter and setter for the instance variables `x` and `y`.
- A method `setXY()` to set both `x` and `y`.
- A method `getXY()` which returns the x and y in a 2-element `int` array.
- A `toString()` method that returns a string description of the instance in the format "`(x, y)`".

Distance Formula: Given the two points (x_1, y_1) and (x_2, y_2), the distance d between these points is given by the formula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- A method called **`distance(int x, int y)`** that returns the distance from *this* point to another point at the given `(x, y)` coordinates, e.g.,

```
Point p1 = new Point(3, 4);
System.out.println(p1.distance(5, 6));
```

- An overloaded **`distance(Point other)`** that returns the distance from *this* point to the given `Point` instance (called `another`), e.g.,

```
Point p1 = new Point(3, 4);
Point p2 = new Point(5, 6);
System.out.println(p1.distance(p2));
```

- Another overloaded **`distance()`** method that returns the distance from `this` point to the origin `(0,0)`, e.g.,

```
Point p1 = new Point(3, 4);
System.out.println(p1.distance());
```

- Another overloaded **`static double distance(Point a, Point b)`** method that returns the distance from two points. This one is just to serve as another example of ***static vs instance*** ownership and isn't truly necessary, just demonstrate that you can call this through the class name e.g.,

```
Point p1 = new Point(3, 4);
Point p2 = new Point(5, 6);
```

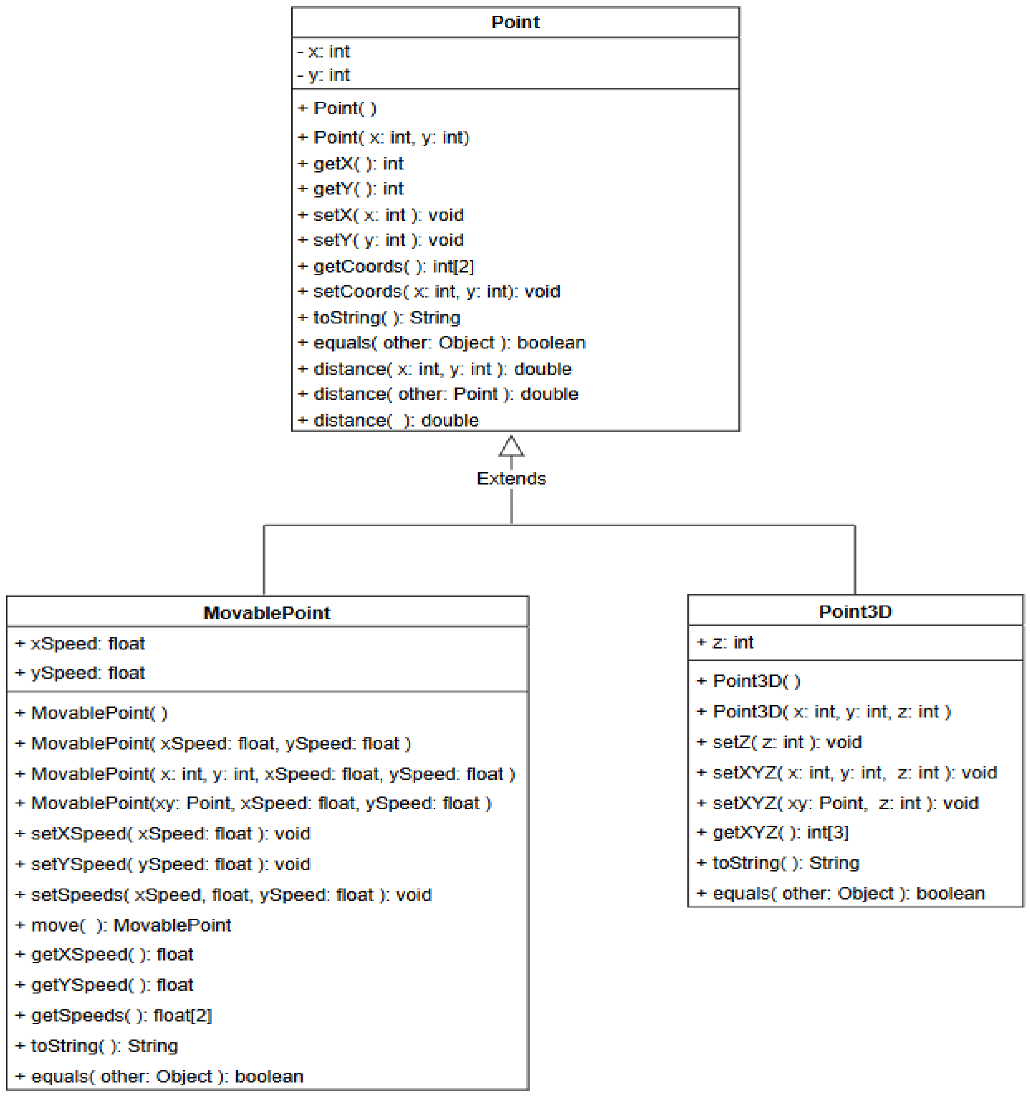
```
System.out.println(Point.distance(p1, p2));
```

- The toString and equals must have the ***@Override*** annotation. Check the readings for an example. We'll be discussing this further in lecture.

Extending `Point` with subclasses `Point3D` and `MovablePoint`

Two example specialized cases of a Point object could be a ***Three Dimensional Point*** with an added Z coordinate and a ***Movable Point*** whose x and y coordinates could be moved at a certain speed. Both of these classes are perfect candidates for extension from the existing Point class because they both possess the general characteristics of being a Point: ***The X and Y coordinates***. This relationship could be summed up in the phrase: ***a point3D object is a Point object.*** The subclasses will add some new behaviors and data by extending the base class definition. Examine the UML diagram below.

Task Two: Define the class `MovablePoint` as an extension of Point. Use the UML diagram as a guide.

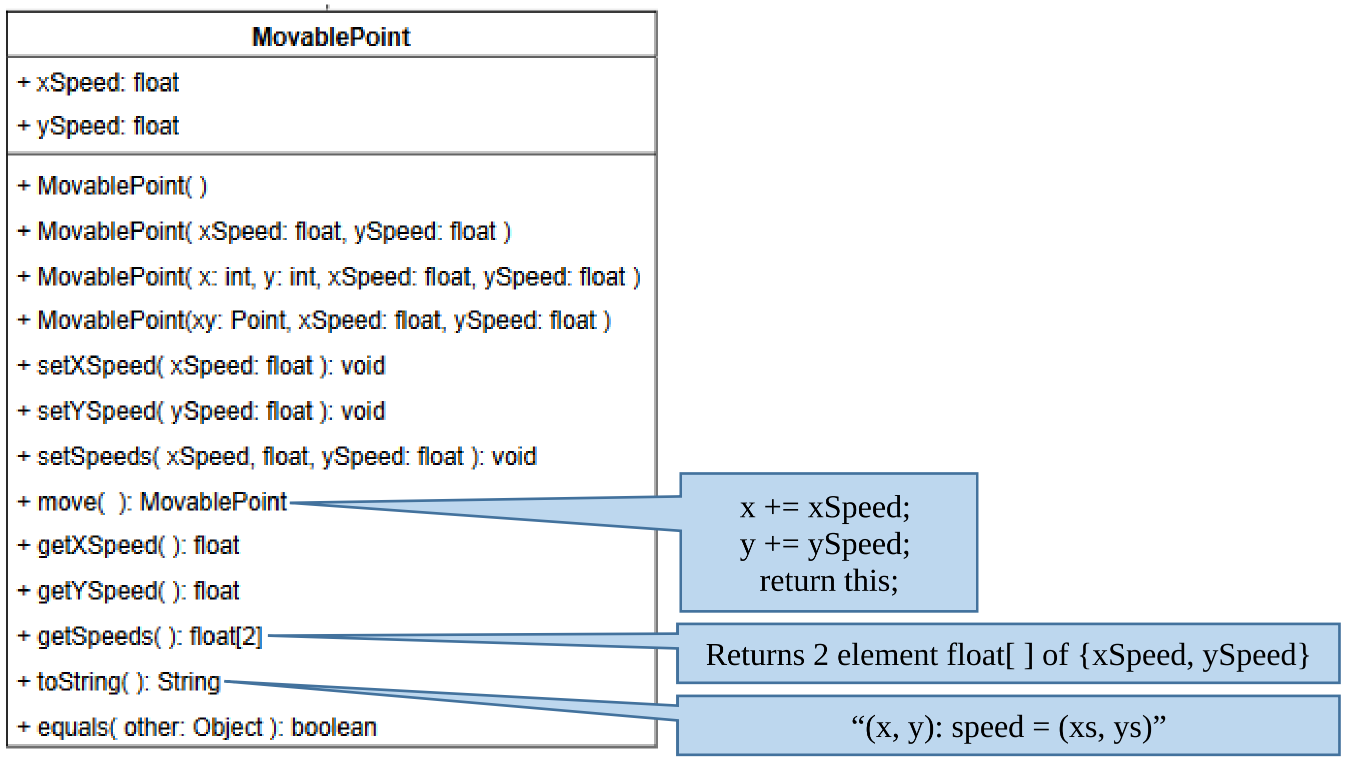


Point
- x: int
- y: int
+ Point( )
+ Point(x: int, y: int)
+ getX( ): int
+ getY( ): int
+ setX(x: int): void
+ setY(y: int): void
+ getCoords( ): int[2]
+ setCoords(x: int, y: int): void
+ toString( ): String
+ equals(other: Object): boolean
+ distance(x: int, y: int): double
+ distance(other: Point): double
+ distance( ): double
Extends
MovablePoint
+ xSpeed: float
+ ySpeed: float
+ MovablePoint( )
+ MovablePoint(xSpeed: float, ySpeed: float)
+ MovablePoint(x: int, y: int, xSpeed: float, ySpeed: float)
+ MovablePoint(xy: Point, xSpeed: float, ySpeed: float)
+ setXSpeed(xSpeed: float): void
+ setYSpeed(ySpeed: float): void
+ setSpeeds(xSpeed, float, ySpeed: float): void
+ move( ): MovablePoint
+ getXSpeed( ): float
+ getYSpeed( ): float
+ getSpeeds( ): float[2]
+ toString( ): String
+ equals(other: Object): boolean
Point3D
+ z: int
+ Point3D( )
+ Point3D(x: int, y: int, z: int)
+ setZ(z: int): void
+ setXYZ(x: int, y: int, z: int): void
+ setXYZ(xy: Point, z: int): void
+ getXYZ( ): int[3]
+ toString( ): String
+ equals(other: Object): boolean

The float type is used here on purpose for you to practice with Java's syntax and with differing numeric types in a strictly typed language. Stick to that design and make appropriate decisions. Include comments

Mark all the overridden methods with annotation **`@Override`.**

Also add a copy constructor


MovablePoint
+ xSpeed: float
+ ySpeed: float
+ MovablePoint( )
+ MovablePoint(xSpeed: float, ySpeed: float)
+ MovablePoint(x: int, y: int, xSpeed: float, ySpeed: float)
+ MovablePoint(xy: Point, xSpeed: float, ySpeed: float)
+ setXSpeed(xSpeed: float): void
+ setYSpeed(ySpeed: float): void
+ setSpeeds(xSpeed, float, ySpeed: float): void
+ move( ): MovablePoint
+ getXSpeed( ): float
+ getYSpeed( ): float
+ getSpeeds( ): float[2]
+ toString( ): String
+ equals(other: Object): boolean
x += xSpeed; y += ySpeed; return this;
Returns 2 element float[ ] of {xSpeed, ySpeed}
"(x, y): speed = (xs, ys)"

Hints

1. You cannot assign a floating-point literal say `1.1` (which is a `double`) to a `float` variable, you need to add a suffix f, e.g. `0.0f`, `1.1f`. This is due to Java treating all floating point types as default **double**

2. The instance variables `x` and `y` are `private` in `Point` and cannot be accessed directly in the subclass `MovablePoint`. You need to access via the `public` getters and setters, which are available via inheritance.

   Example: you cannot write **`x+=xSpeed`**, you need to write **`setX(getX() + xSpeed)`**.

3. The constructors will need to invoke the appropriate *super class constructors* also, due to the privacy of the inherited fields `x` and `y`.

4. The `equals` method will need to ensure super class equality as well. Re-use the super class equals method for this.

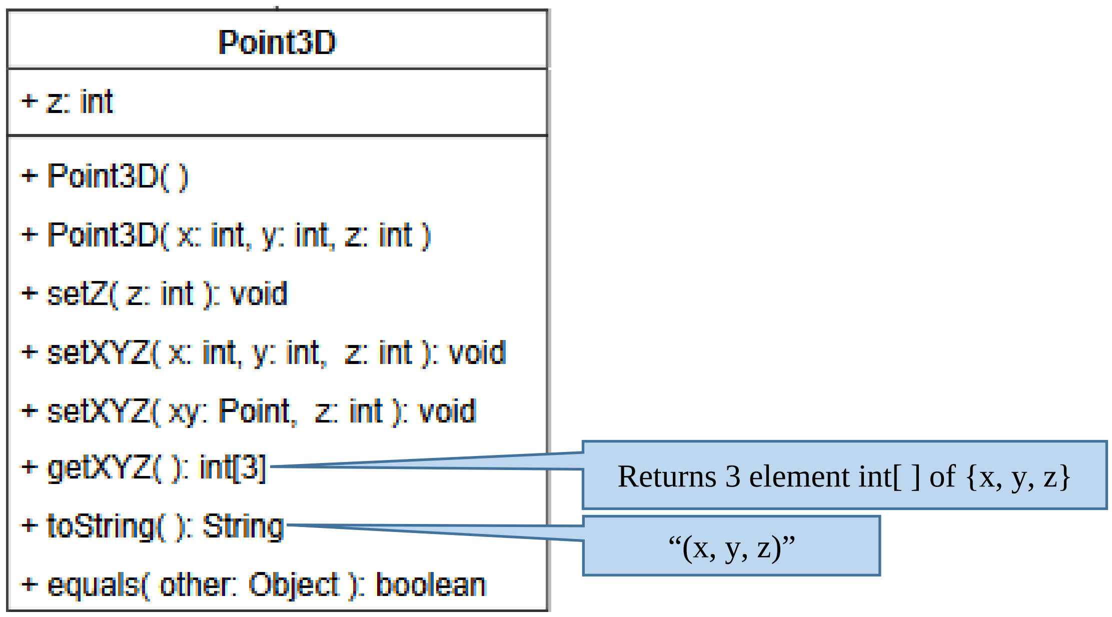
5. The method **`getSpeeds()`** returns a `float` array:

```
public float[] getSpeeds() {
       float[] result = new float[2];  // construct an array
       result[0] = ...
       result[1] = ...
       return result;  // return the array
    }
```

Task Three: Define the class `Point3D` as an extension of `Point`. Use the UML diagram as a guide.

Mark all the overridden methods with annotation `@Override`.

Also add a copy constructor


Point3D
+ z: int
+ Point3D( )
+ Point3D(x: int, y: int, z: int)
+ setZ(z: int): void
+ setXYZ(x: int, y: int, z: int): void
+ setXYZ(xy: Point, z: int): void
+ getXYZ( ): int[3]
+ toString( ): String
+ equals(other: Object): boolean
Returns 3 element int[ ] of {x, y, z}
"(x, y, z)"

Hints:

1. You cannot assign floating-point literal say `1.1` (which is a `double`) to a `float` variable, you need to add a suffix f, e.g. `0.0f`, `1.1f`.

2. The instance variables `x` and `y` are `private` in `Point2D` and cannot be accessed directly in the subclass `Point3D`. You need to access via the inherited getters and setters.

3. The constructors will need to invoke the appropriate super class constructors also, due to the privacy of the inherited fields `x` and `y`.

4. The `equals` method will need to ensure super class equality as well. Re-use the super class equals method for this.

5. The method **`getXYZ()`** shall return a `float` array:

```
public float[] getXYZ() {
        float[] result = new float[3]; // array of 3 coords
        result[0] = ...
        result[1] = ...
         . . .
        return result;  // return the array
    }
```

Task Four: Proof of concept (Driver)

- Instantiate a few Point, Point3D and MovablePoint objects, demonstrating their various constructors.
- Demonstrate that the toString methods all work according to the descriptions above and can chain together calls to ***super.toString( )***.
- Demonstrate that you can call various ***inherited Point*** methods through Point3D and MovablePoint instances.
- Show that you can determine the distances between the various objects. Understand that you can pass an instance of a subclass into a method that accepts an argument of super class type. So you can perform operations like this.

```
MovablePoint mp = new MovablePoint( new Point(1, 2), 3, 4 );

Point3D p3d     = new Point3D( new Point(12, 34), 6 );

double dis      = mp.distance( p3d );
```

- Be sure you understand how this works. Remember the ***is a*** relationship

Task Five: Unit Tests . . . Write ***thorough*** unit tests for the following methods

- **The various distance methods**. Prove that the overloaded methods can handle any of the objects in the hierarchy ***in any combination***. The most interesting implementation here is the distance method that accepts a Point. ***Because it accepts a Point, it can also accept any subclass of Point***. Use an online distance calculator to compute your expectations. Pay close attention to any rounding that may occur.

- **The various equals methods.** Prove that overridden equals methods can work correctly on any combination of objects from the hierarchy.

  Call equals through a Point instance and pass a MovablePoint instance, call equals through a Point3D instance and pass a Point instance . . . etc, etc.

  Use this as an opportunity to understand how these inheritance hierarchy references are related to one another. Also understand that a properly overridden ***equals*** method can accept an instance ***of any class*** and will simply return false if the classes are different.

Task Six: Polymorphism Foreshadowing

- Using the same driver file, create a 9 element array of ***superclass type*** Point

- Because of the inheritance relationship "***Point3D is a Point"*** and "***MovablePoint is a Point"*** you can place ***Point3D and MovablePoint*** objects into a collection of superclass type Point. This ability to group related objects into collections of superclass types is a foundational concept of polymorphism.

- Illustrate this by placing 3 Points, 3 MovablePoints and 3 Point3D instances into this array.

- **Polymorphic behavior:** Using an ***enhanced for loop***,

  ```
  for( Point p : points ){
      // call methods on p
  }
  ```

  loop through the array and call toString on each object. Analyze the output and notice that the appropriate toString was called for each object, even though the array is of type Point. This is polymorphic behavior. The JRE knows which version of toString to call.

- Determine which of the various Point objects are furthest from origin. Display the distance and the toString for this object.

- In the same loop, using the for loop variable, try to call a method that is ***just defined*** in a subclass and watch the compiler complain. Why is this? Do you see any requirements for polymorphic behavior? ***Comment the line out and add some comments in your code with your thoughts.***

- Let's take this one step further. Define an array of ***type Object*** of size 9. Notice the inheritance relationships

  - A Point is an Object
  - A Point3D is a Point, therefore by extension, it is also an Object

  - A MovablePoint is a Point, therefore by extension, it is also an Object

- Place 3 instances of each of those classes into the Object array. Loop through the array calling toString. Analyze the output and notice that the appropriate toString is being called for each instance. We just generalized the collection one step higher in the inheritance hierarchy . . . all the way to the ultimate ancestor.

- Try to call a method that is unique to the subclasses. Why does the compiler complain? ***Comment the line out and include comments in your code communicating your thoughts on this issue.***

- **Experiment:** Add instances of ***ANY*** class you have defined or that exists in the Java API into this same Object array with the Points.

  - Customer
  - MenuItem
  - OrderItem
  - Date
  - String
  - WHATEVER

- Loop through the array and call toString on each instance. How does Java know which method to call? How is this even happening? We are starting to dig into the real power of object orientation.

Submission: Push whatever files you have. I will need everything to run your code ***with no modifications***