GEOG 178/258
Week 6:

Interfaces, UML

mike johnson

Set up:

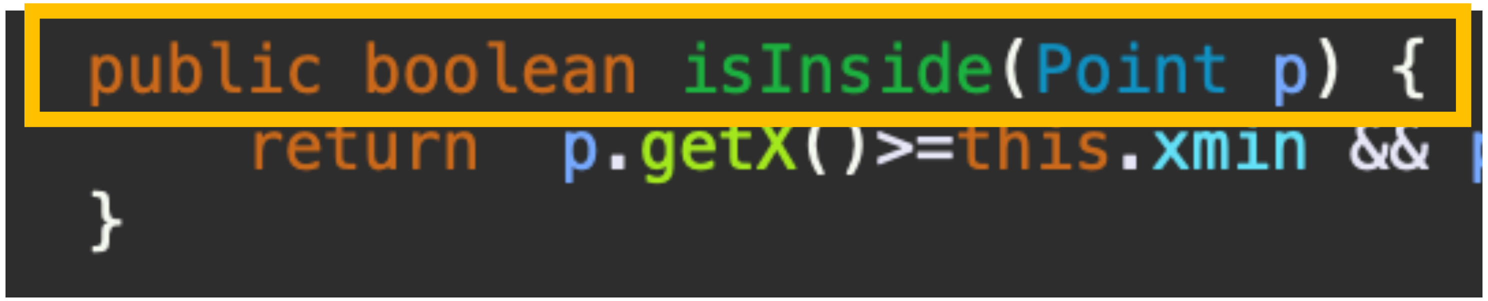
Before we get started let’s set up for this weeks lab:

Create a new project (week6) and copy over your:

- Point Class
- Bbox Class
- Polyline Class
- Polygon Class

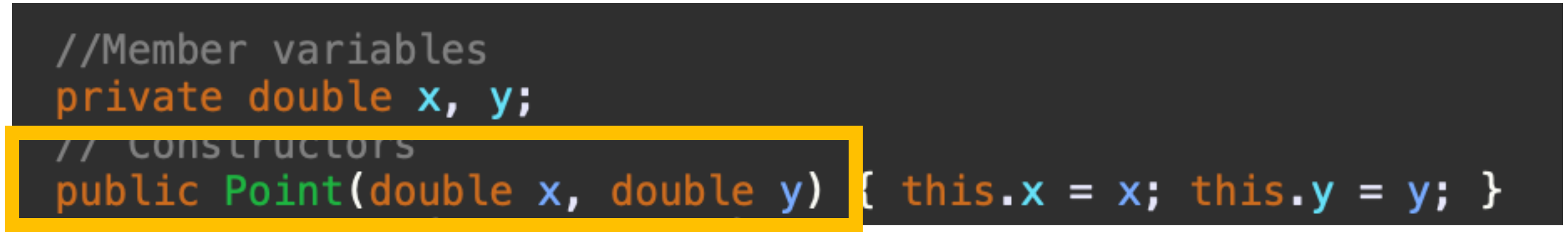
Recap

- Signatures
- Delegation
- Inheritance (extending a class)
- Overriding



```
public boolean isInside(Point p) {
        return  p.getX()>=this.xmin &&
}
```

Visibility - return type – name - inputs



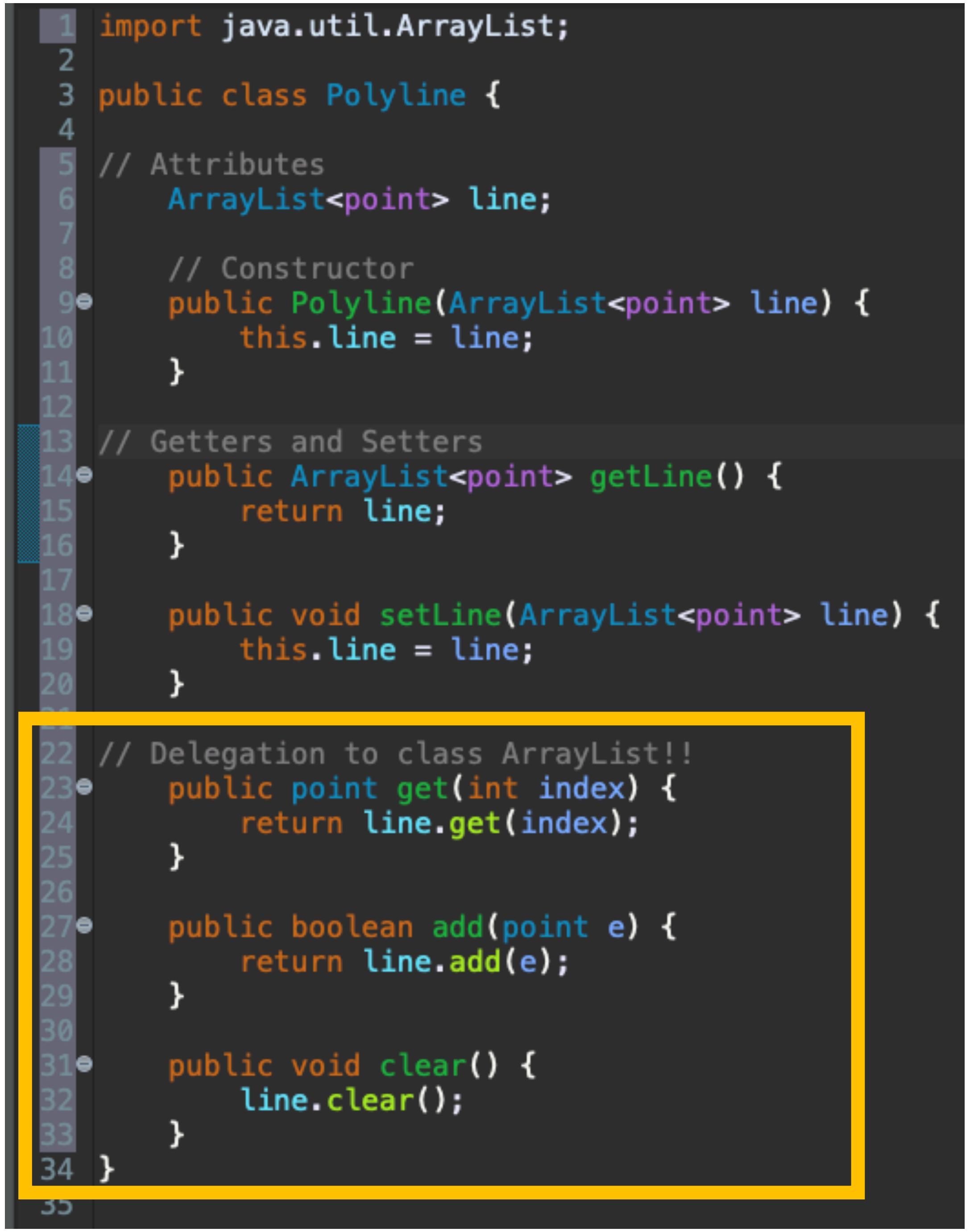
```
//Member variables
private double x, y;
// Constructors
public Point(double x, double y) { this.x = x; this.y = y; }
```

Visibility – Name that matches class -- Input

Recap

- Signatures
- Delegation
- Inheritance (extending a class)
- Overriding

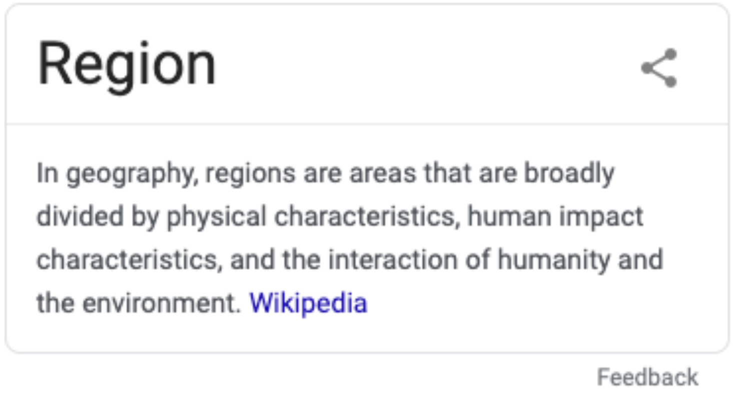
- Passing your work (a duty) over to someone/something else (anther class!!).
- When you delegate, **you are simply calling up some class which knows what must be done**. You do not really care how it does it, all you care about is that the class you are calling knows what needs doing.



```
import java.util.ArrayList;

public class Polyline {

// Attributes
    ArrayList<point> line;

    // Constructor
    public Polyline(ArrayList<point> line) {
        this.line = line;
    }

// Getters and Setters
    public ArrayList<point> getLine() {
        return line;
    }

    public void setLine(ArrayList<point> line) {
        this.line = line;
    }

// Delegation to class ArrayList!!
    public point get(int index) {
        return line.get(index);
    }

    public boolean add(point e) {
        return line.add(e);
    }

    public void clear() {
        line.clear();
    }
}
```

Recap

- Signatures
- Delegation
- Inheritance (extending a class)
- Overriding


Region

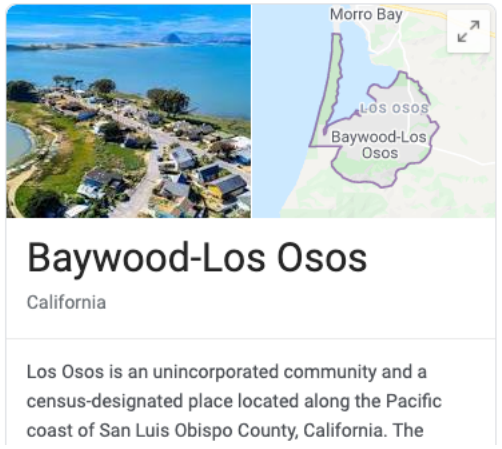
In geography, regions are areas that are broadly divided by physical characteristics, human impact characteristics, and the interaction of humanity and the environment. Wikipedia

Feedback

San Luis Obispo

City in California

San Luis Obispo is a city in California's Central Coast region. On Mission Plaza, the Mission San Luis



Baywood-Los Osos

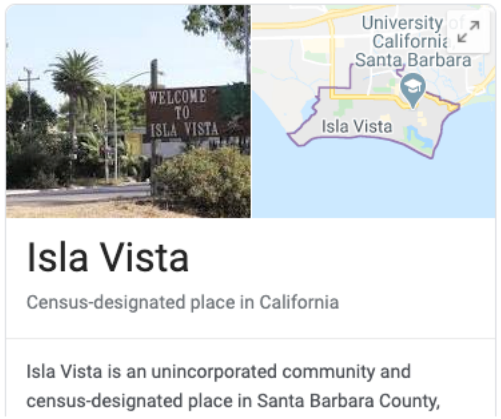
California

Los Osos is an unincorporated community and a census-designated place located along the Pacific coast of San Luis Obispo County, California. The

Goleta

City in California

Goleta is a city in southern Santa Barbara County, California, United States. It was incorporated as a city



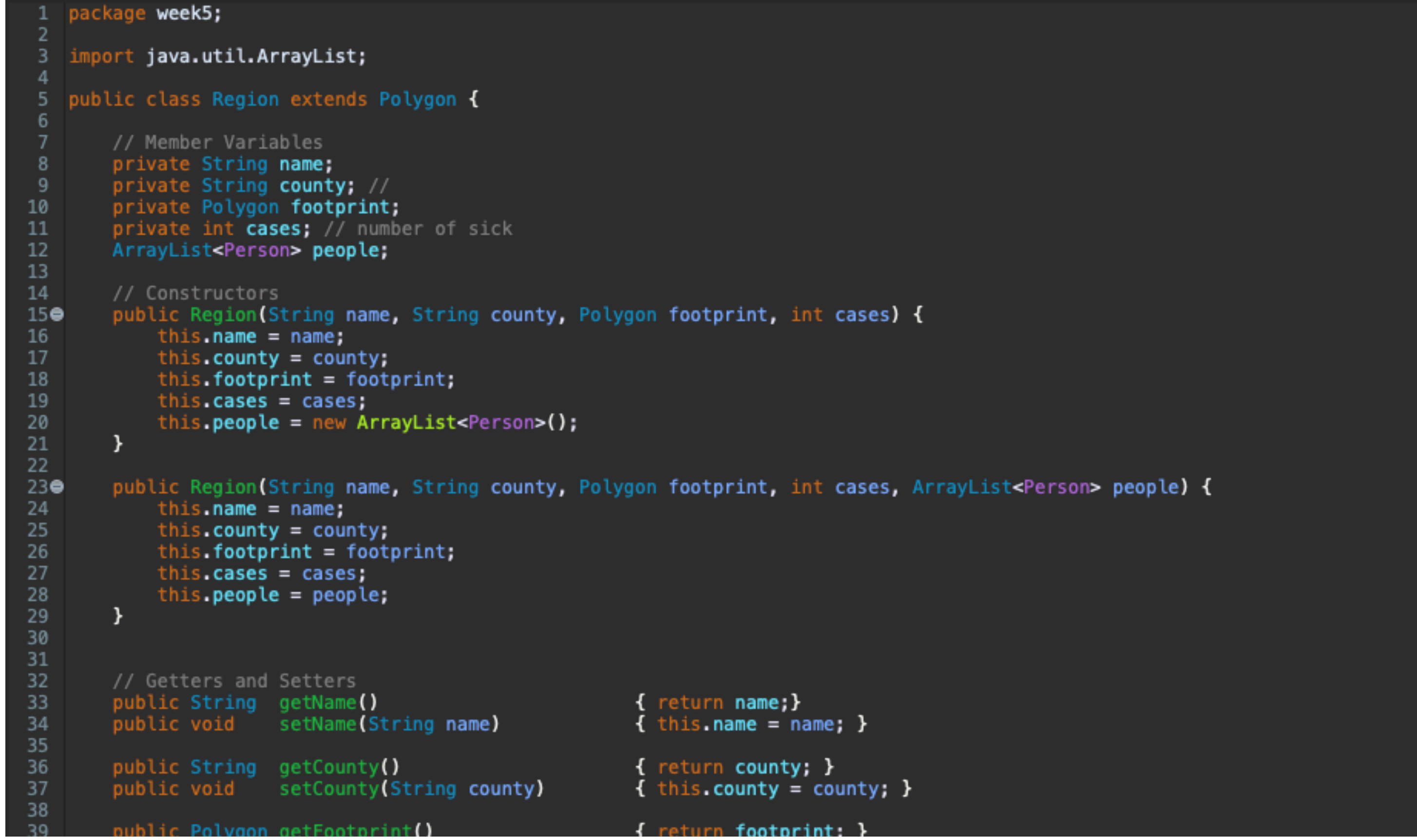
Isla Vista

Census-designated place in California

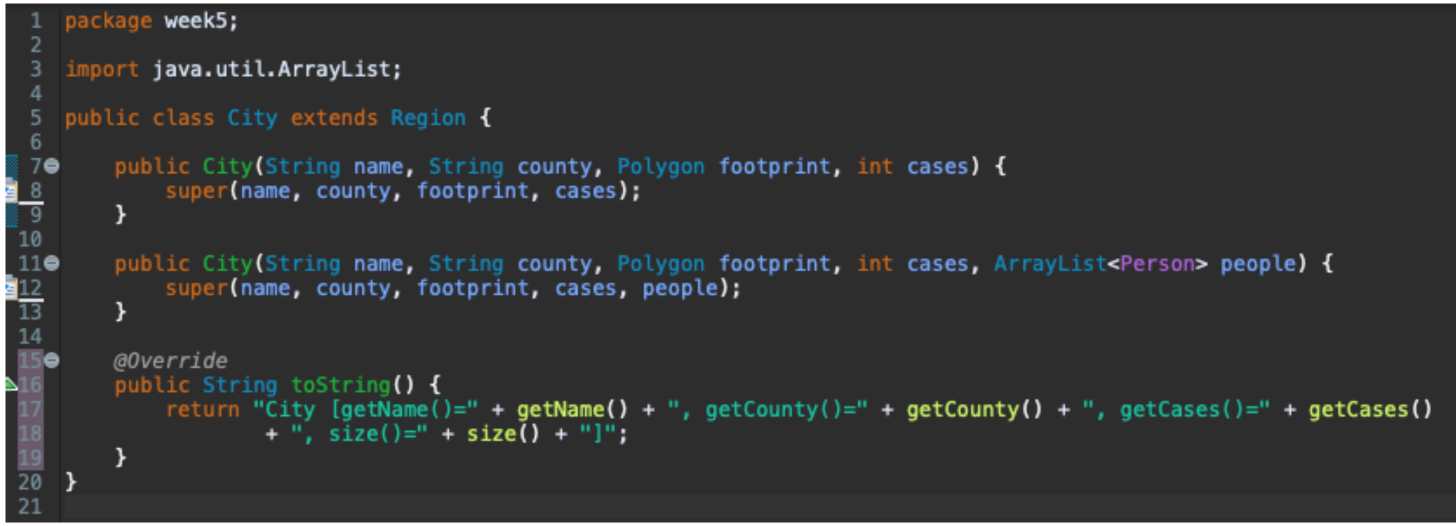
Isla Vista is an unincorporated community and census-designated place in Santa Barbara County,

Recap

- Signatures
- Delegation
- Inheritance (extending a class)
- Overriding



```java
package week5;

import java.util.ArrayList;

public class Region extends Polygon {

    // Member Variables
    private String name;
    private String county; //
    private Polygon footprint;
    private int cases; // number of sick
    ArrayList<Person> people;

    // Constructors
    public Region(String name, String county, Polygon footprint, int cases) {
        this.name = name;
        this.county = county;
        this.footprint = footprint;
        this.cases = cases;
        this.people = new ArrayList<Person>();
    }

    public Region(String name, String county, Polygon footprint, int cases, ArrayList<Person> people) {
        this.name = name;
        this.county = county;
        this.footprint = footprint;
        this.cases = cases;
        this.people = people;
    }


    // Getters and Setters
    public String  getName()                    { return name;}
    public void    setName(String name)         { this.name = name; }

    public String  getCounty()                  { return county; }
    public void    setCounty(String county)     { this.county = county; }

    public Polygon getFootprint()               { return footprint; }
```



```java
package week5;

import java.util.ArrayList;

public class City extends Region {

    public City(String name, String county, Polygon footprint, int cases) {
        super(name, county, footprint, cases);
    }

    public City(String name, String county, Polygon footprint, int cases, ArrayList<Person> people) {
        super(name, county, footprint, cases, people);
    }

    @Override
    public String toString() {
        return "City [getName()=" + getName() + ", getCounty()=" + getCounty() + ", getCases()=" + getCases()
                + ", size()=" + size() + "]";
    }
}
```

Recap

- Signatures
- Delegation
- Inheritance (extending a class)
- Overriding

@Override

In any object-oriented programming language…

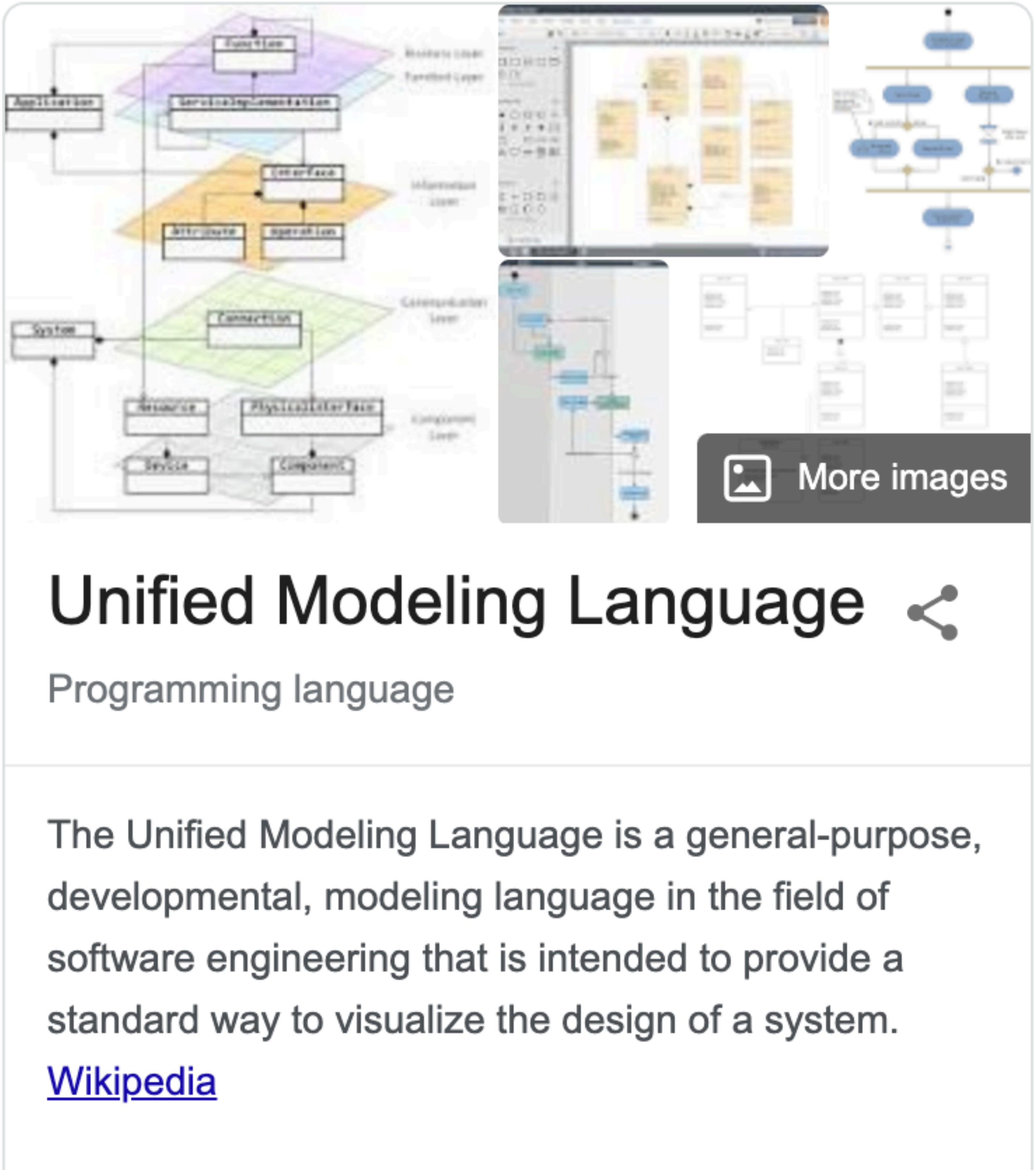
Overriding is a feature

that allows a subclass or child class to provide

a unique implementation of a method that is already provided

by one of its super-classes or parent classes.

UML



More images

Unified Modeling Language

Programming language

The Unified Modeling Language is a general-purpose, developmental, modeling language in the field of software engineering that is intended to provide a standard way to visualize the design of a system. Wikipedia

“standard way to visualize the design of a system…”

Cross Domain (Hydrology Example)

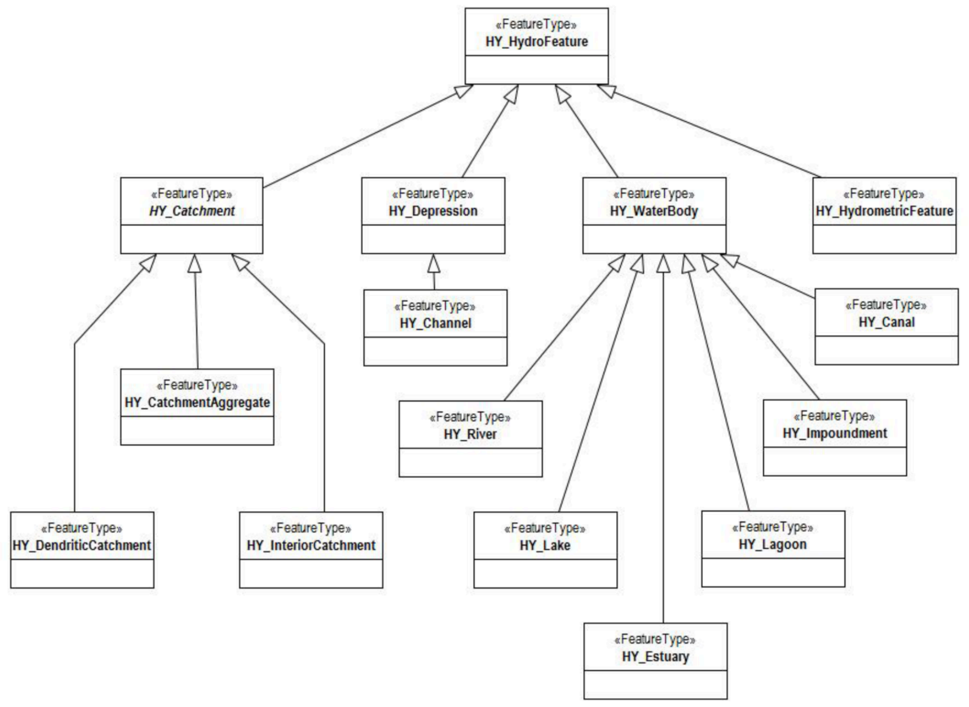

«FeatureType» HY_HydroFeature
«FeatureType» HY_Catchment
«FeatureType» HY_Depression
«FeatureType» HY_WaterBody
«FeatureType» HY_HydrometricFeature
«FeatureType» HY_Channel
«FeatureType» HY_Canal
«FeatureType» HY_CatchmentAggregate
«FeatureType» HY_River
«FeatureType» HY_Impoundment
«FeatureType» HY_DendriticCatchment
«FeatureType» HY_InteriorCatchment
«FeatureType» HY_Lake
«FeatureType» HY_Lagoon
«FeatureType» HY_Estuary

Figure 23. Hydrologic features describing separate aspects of hydrology phenomena

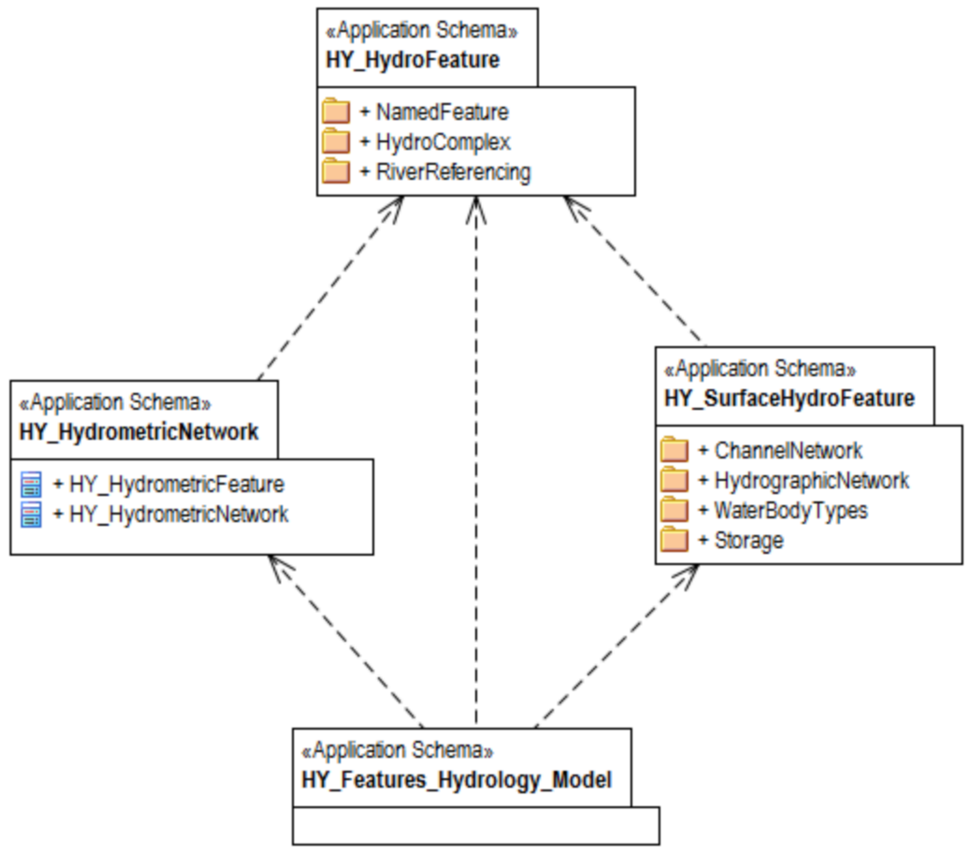
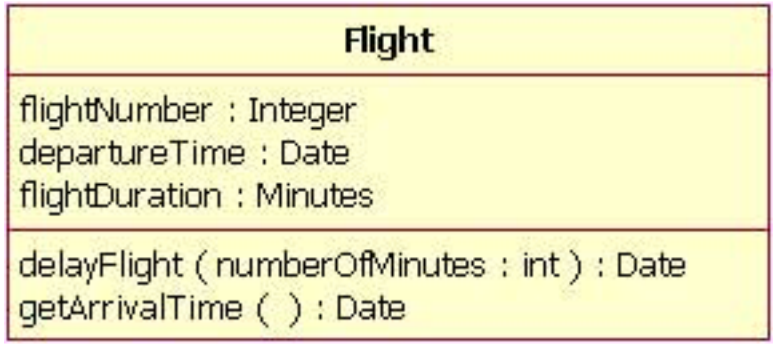

«Application Schema» HY_HydroFeature
+ NamedFeature
+ HydroComplex
+ RiverReferencing
«Application Schema» HY_HydrometricNetwork
+ HY_HydrometricFeature
+ HY_HydrometricNetwork
«Application Schema» HY_SurfaceHydroFeature
+ ChannelNetwork
+ HydrographicNetwork
+ WaterBodyTypes
+ Storage
«Application Schema» HY_Features_Hydrology_Model

Figure 20. HY_Features modules and packages

https://docs.opengeospatial.org/is/14-111r6/14-111r6.html#_the_hy_features_conceptual_model

Classes



Classes are represented as rectangles with stacked compartments:

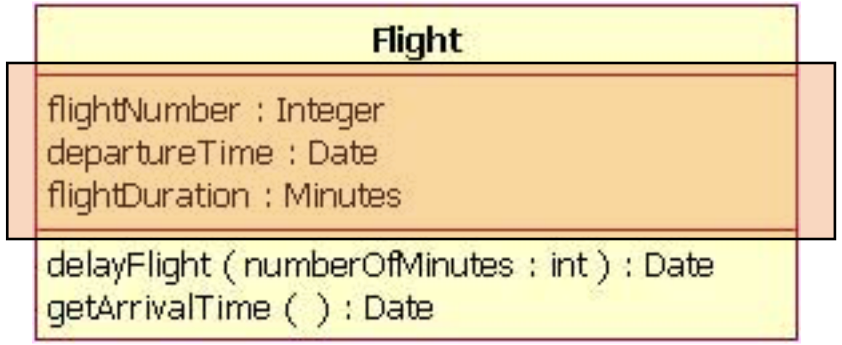
The top compartment shows the **class name** (Flight)

The middle: the **class attributes**

The last: the class operations (aka methods)

Think about how this already mirrors our structure of (**Member variables**, **Constructors**, Getters& Setters, **Methods**)

Member Variables (Attributes)



Attribute lines are optional but if included are written in the following structure:

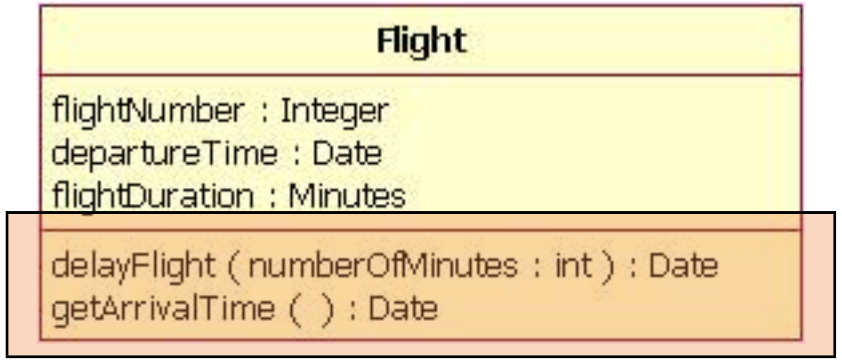
Name : attribute type

In many "everyday" class diagrams, the attribute types usually show units that make sense to readers (i.e., minutes, dollars, etc.). However, a class diagram that will be used to generate code needs classes whose attribute types are limited to the types provided by the programming language, or types included in the model that will also be implemented in the system.

Often default values will be provided as well:

MyBank: double = 0

Operations (Methods)



Operations (methods!) are documented as a list with the following format:

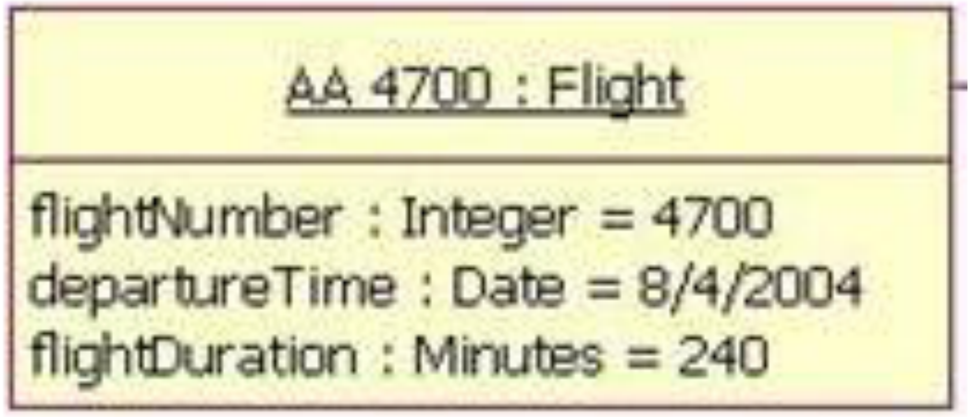
Name(parameter list) : type of value returned

(think to the **signature** of your methods like isInside!)

When parameters are needed the name and type should be explicitly provided:

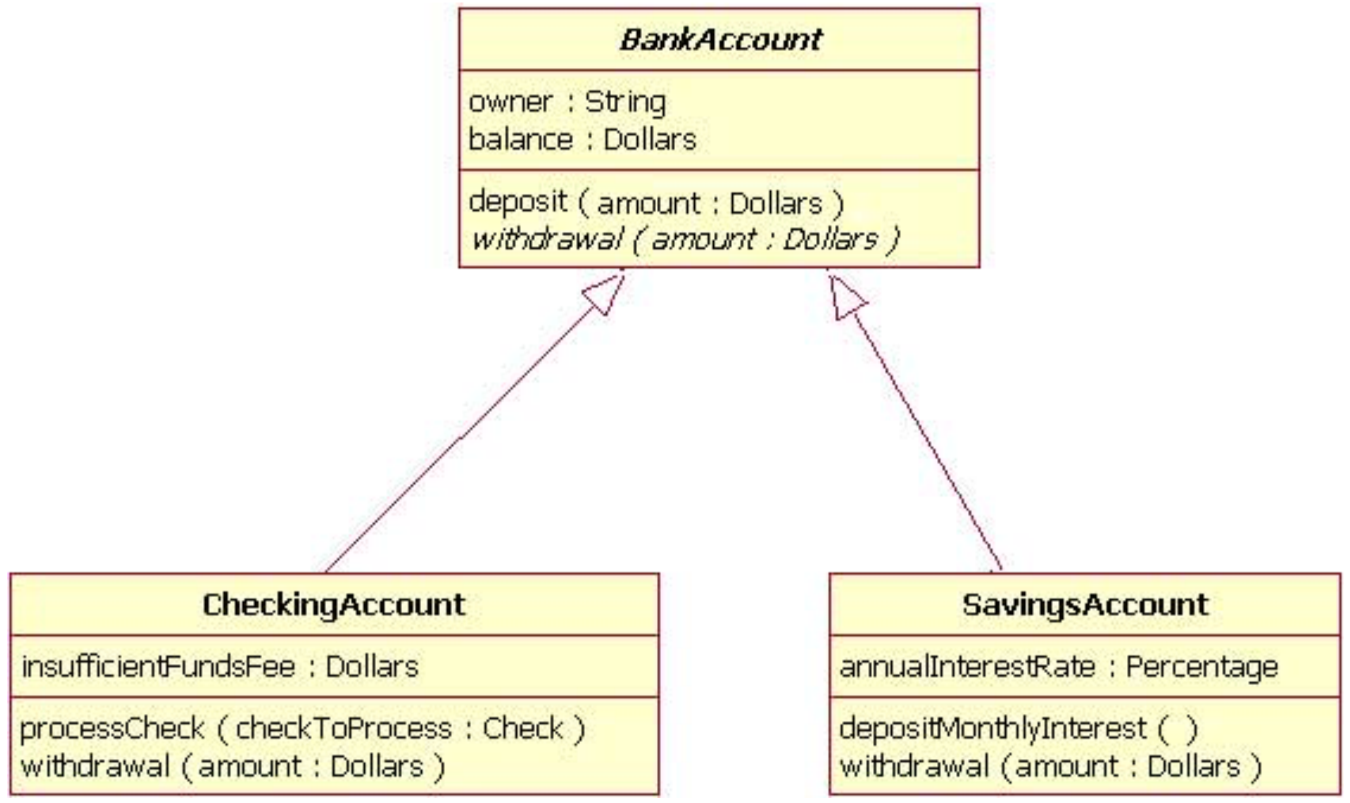
isInside (P1 : Point, P2: Point) : Boolean

Objects (objects)

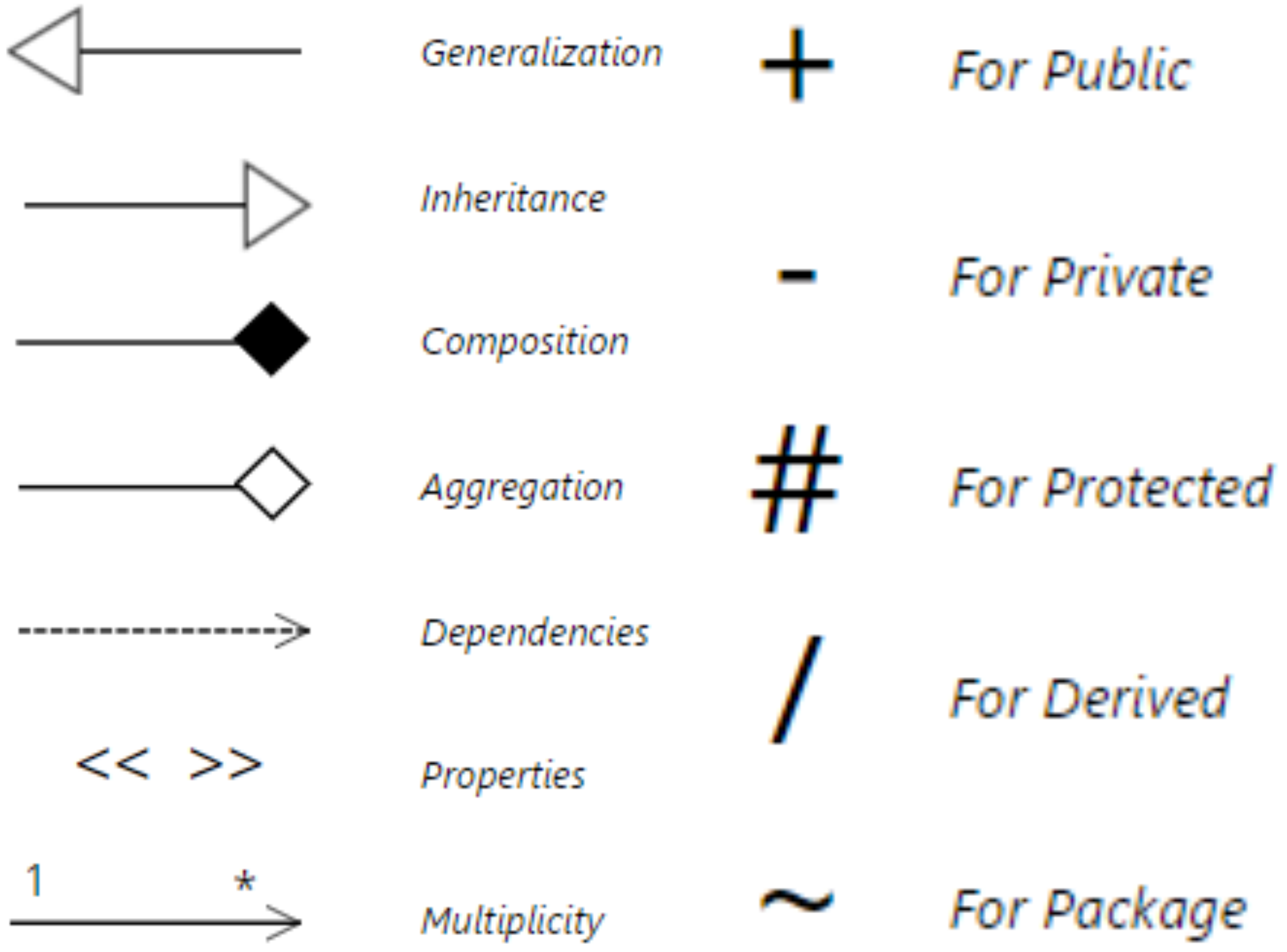

AA 4700 : Flight
flightNumber : Integer = 4700
departureTime : Date = 8/4/2004
flightDuration : Minutes = 240

Relationships (Inheritance)

REVIEW: inheritance refers to the ability of one class (child class) to inherit the identical functionality of another class (super class), and then add new functionality of its own.

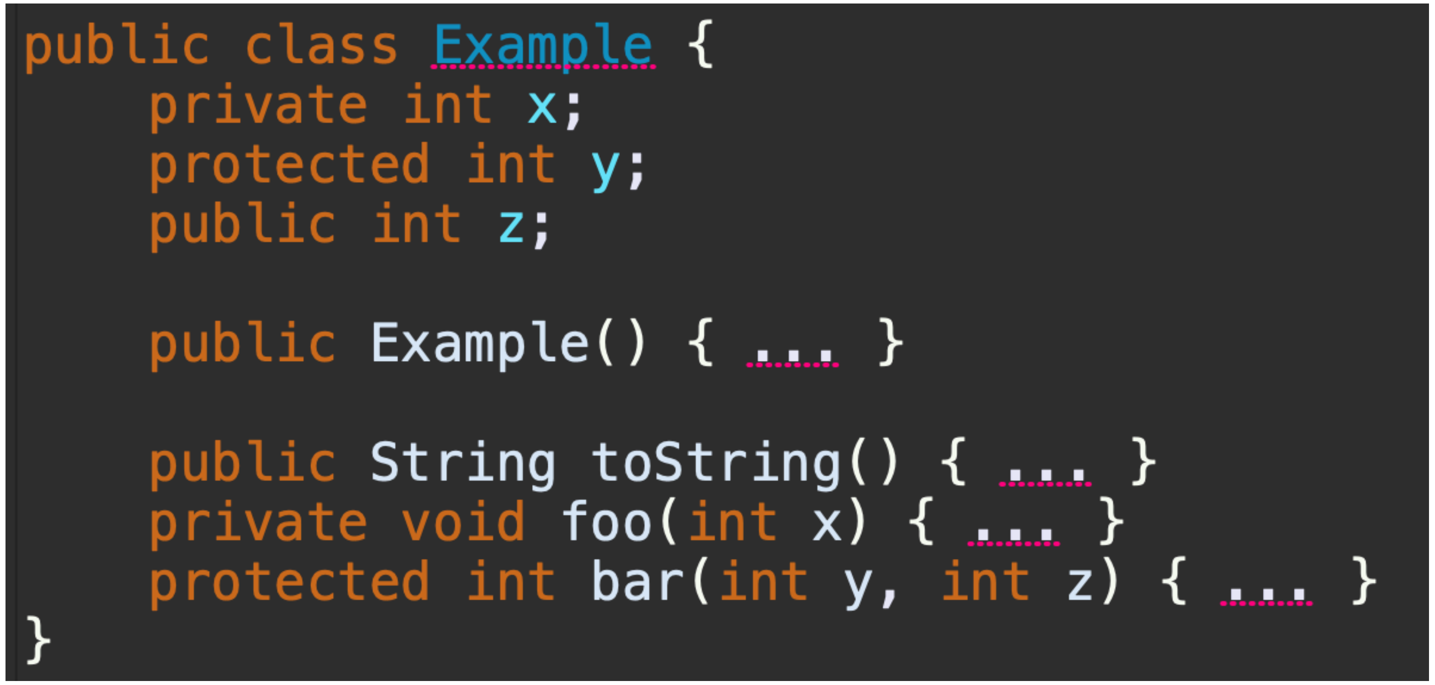

BankAccount
owner : String
balance : Dollars
deposit (amount : Dollars)
withdrawal (amount : Dollars)
CheckingAccount
insufficientFundsFee : Dollars
processCheck (checkToProcess : Check)
withdrawal (amount : Dollars)
SavingsAccount
annualInterestRate : Percentage
depositMonthlyInterest ( )
withdrawal (amount : Dollars)

Icons

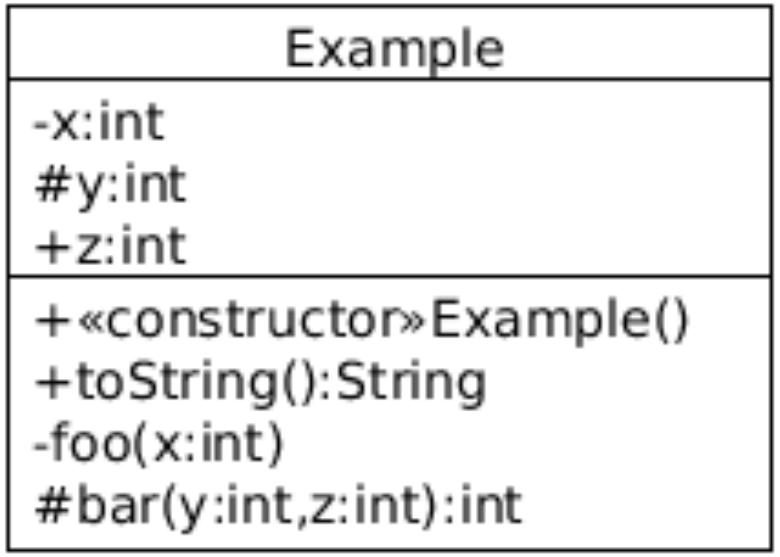


For more look here

Example

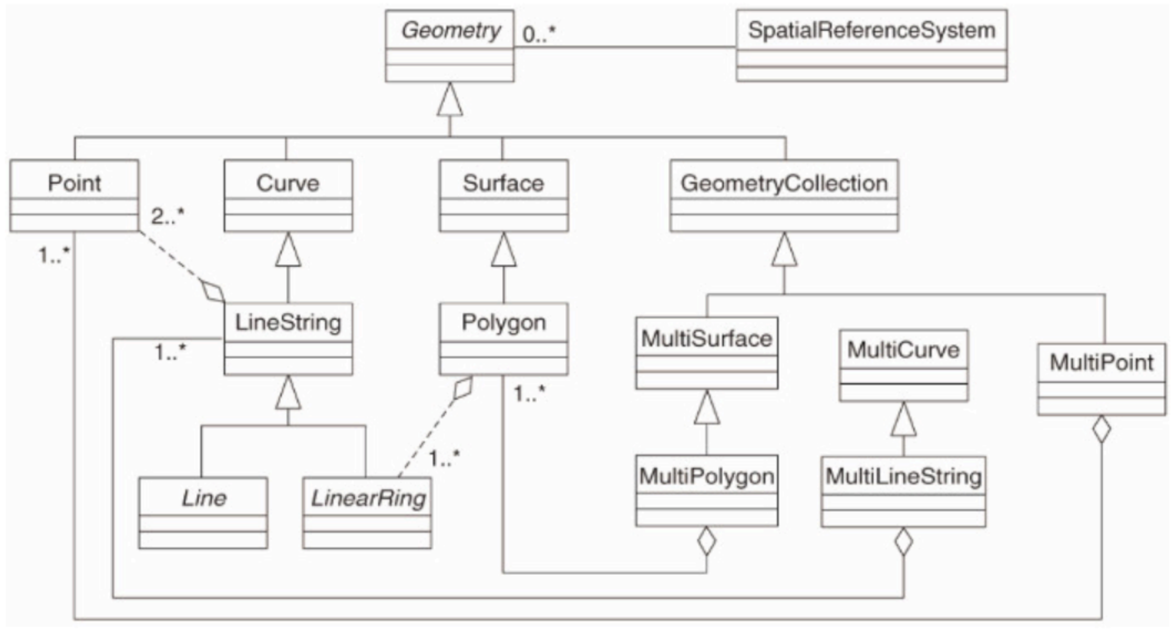


```
public class Example {
    private int x;
    protected int y;
    public int z;

    public Example() { ... }

    public String toString() { ... }
    private void foo(int x) { ... }
    protected int bar(int y, int z) { ... }
}
```



Putting it together

OGC **Simple** Feature Access



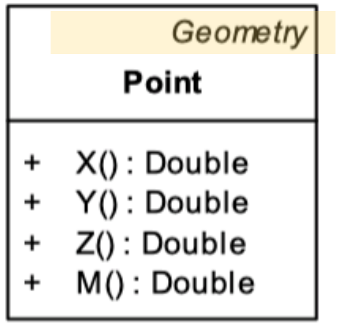


Figure 4: Point

6.1.4.2 Methods

— **X** ( ):Double — The *x*-coordinate value for *this* Point.

— **Y** ( ):Double — The *y*-coordinate value for *this* Point.

— **Z** ( ):Double — The *z*-coordinate value for *this* Point, if it has one. Returns NIL otherwise.

— **M** ( ):Double — The *m*-coordinate value for *this* Point, if it has one. Returns NIL otherwise.

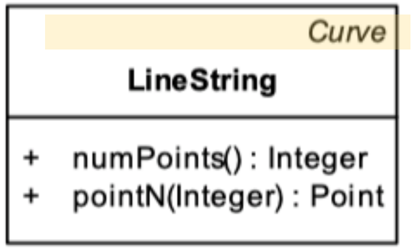


Figure 7: LineString

6.1.7.2 Methods

— **NumPoints** ( ): Integer — The number of Points in *this* LineString.

— **PointN** (N: Integer): Point — Returns the specified Point N in *this* LineString.

Extend vs Implements

Extends

Java allows classes to inherit the **fields** and **methods** of a class. But only one class can be extended!

Example: ArrayList class:

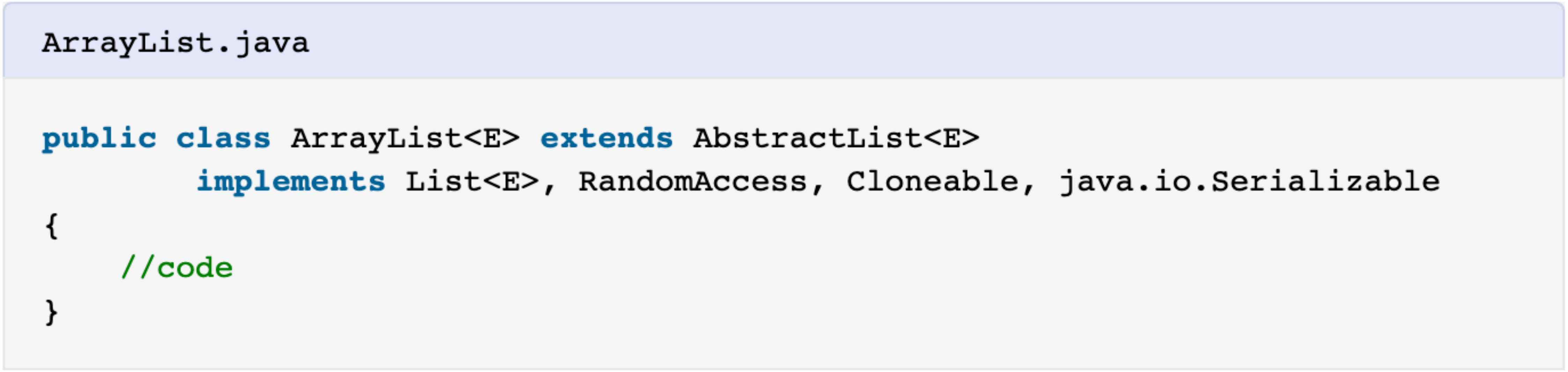
- *ArrayList* **extends** *AbstractList*
- *AbstractList* **extends** *AbstractCollection*.

So *ArrayList(s)* have methods and behaviors of both *AbstractList* and *AbstractCollection*.

- *AbstractCollection* provides methods like contains(Object o), toArray(), remove(Object o)
- *AbstractList* class provides add(), indexOf(), lastIndexOf(), clear() etc.

Some of the methods are ***overridden*** by ArrayList.

ArrayList extends AbstractList



```
ArrayList.java
```

```java
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    //code
}
```

Inheritance Example

ParentClass.java

```java
public class ParentClass {

    public int dataVal = 100;

    public int getDataVal() {
        return this.dataVal;
    }
}
```

ChildClass.java

```java
public class ChildClass extends ParentClass
{

}
```

Main.java

```java
public class Main
{
    public static void main(String[] args)
    {
        ChildClass child = new ChildClass();

        System.out.println( child.dataVal );
        System.out.println( child.getDataVal() );
    }
}
```

What will this print??

implements

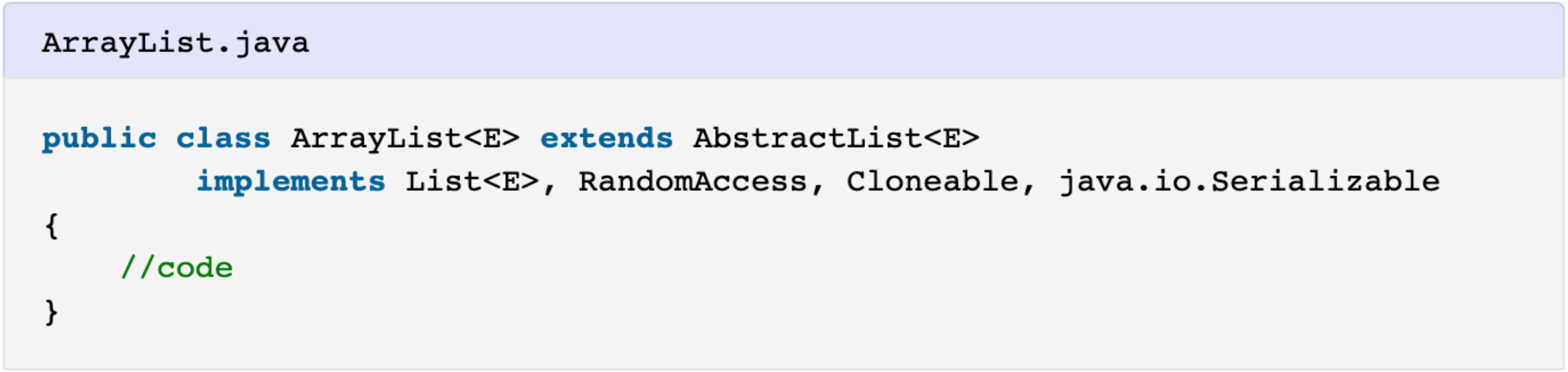
Interfaces enforce a ***contract*** in Java.

They **force** the implementing class to provide a certain behavior.

Java can implement more than one interfaces. In this case, class must implement all the methods from all the interfaces. (**or declare itself abstract**).

Look at the ArrayList class declaration one more time. It implements 4 interfaces i.e. **List**, **RandomAccess**, **Cloneable** and **Serializable**. It has implemented all the methods in given interfaces.

ArrayList implements ....



ArrayList.java

```java
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    //code
}
```

Interface Example

Must contain move, but doesn't specific what move does….

Movable.java

```java
public interface Movable {

    public void move();
}
```

Swimmable.java

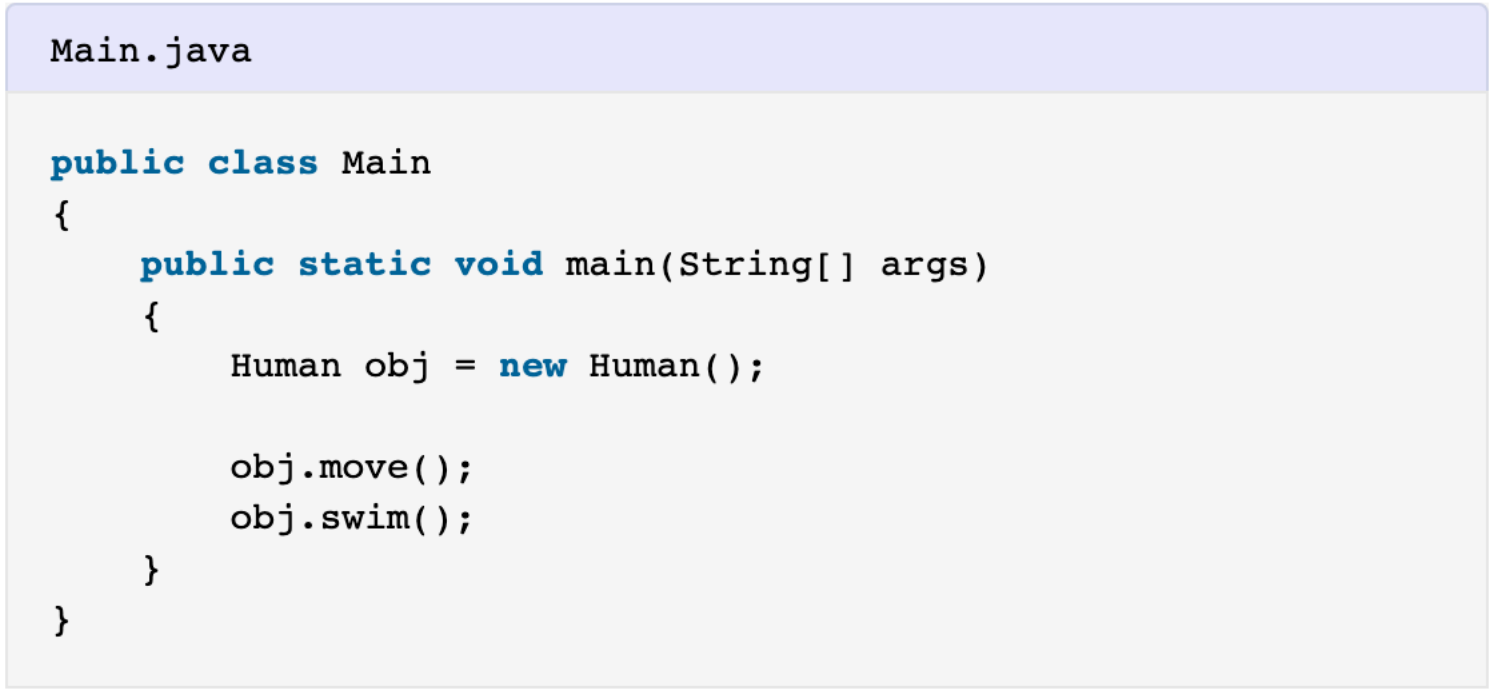
```java
public interface Swimmable
{
    public void swim();
}
```

Makes move concrete, Humans move is a certain way. In this case by saying "I am moving"

Human.java

```java
public class Human implements Movable, Swimmable
{
    @Override
    public void swim() {
        System.out.println("I am swimming");
    }

    @Override
    public void move() {
        System.out.println("I am moving");
    }
}
```

Human objects can swim
and move



Main.java

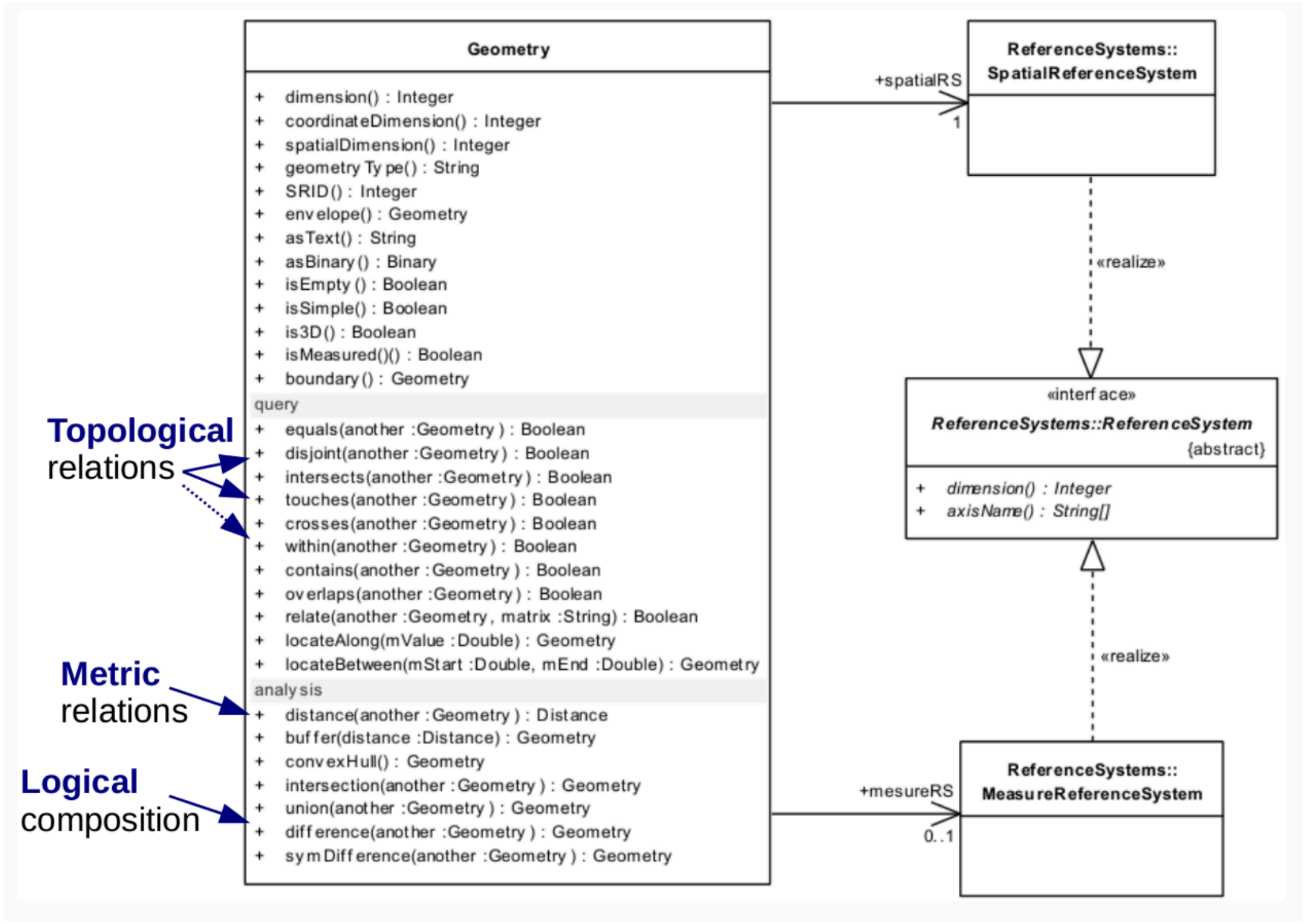
```java
public class Main
{
    public static void main(String[] args)
    {
        Human obj = new Human();

        obj.move();
        obj.swim();
    }
}
```

Recap

- **extends** is used to **inherit** a **class**
- **implements** is used to **inherit** the **interfaces**.
- A class can extend only one class; but can implement any number of interfaces.
- A subclass that extends a superclass may override some of the methods from superclass.
- A class must implement all the methods from interfaces.

Homework

OGC Geometry Interface


Geometry
+ dimension() : Integer
+ coordinateDimension() : Integer
+ spatialDimension() : Integer
+ geometry Ty pe() : String
+ SRID() : Integer
+ envelope() : Geometry
+ asText() : String
+ asBinary () : Binary
+ isEmpty () : Boolean
+ isSimple() : Boolean
+ is3D() : Boolean
+ isMeasured()() : Boolean
+ boundary () : Geometry
query
+ equals(another :Geometry) : Boolean
+ disjoint(another :Geometry) : Boolean
+ intersects(another :Geometry) : Boolean
+ touches(another :Geometry) : Boolean
+ crosses(another :Geometry) : Boolean
+ within(another :Geometry) : Boolean
+ contains(another :Geometry) : Boolean
+ overlaps(another :Geometry) : Boolean
+ relate(another :Geometry , matrix :String) : Boolean
+ locateAlong(mValue :Double) : Geometry
+ locateBetween(mStart :Double, mEnd :Double) : Geometry
analysis
+ distance(another :Geometry) : Distance
+ buffer(distance :Distance) : Geometry
+ convexHull() : Geometry
+ intersection(another :Geometry) : Geometry
+ union(another :Geometry) : Geometry
+ difference(another :Geometry) : Geometry
+ symDifference(another :Geometry) : Geometry
Topological relations
Metric relations
Logical composition
+spatialRS
1
ReferenceSystems:: SpatialReferenceSystem
«realize»
«interface»
ReferenceSystems::ReferenceSystem
{abstract}
+ dimension() : Integer
+ axisName() : String[]
«realize»
+mesureRS
0..1
ReferenceSystems:: MeasureReferenceSystem

We are going to create 2 interfaces

BOUNDINGAREA

GEOMETRY

BoundingBox and **pointBuffer** will implement *BoundingArea*

Point, **Polyline**, and **Polygon** will implement *Geometry*

Our BoundingArea contract will ensure all boundingarea objects have:

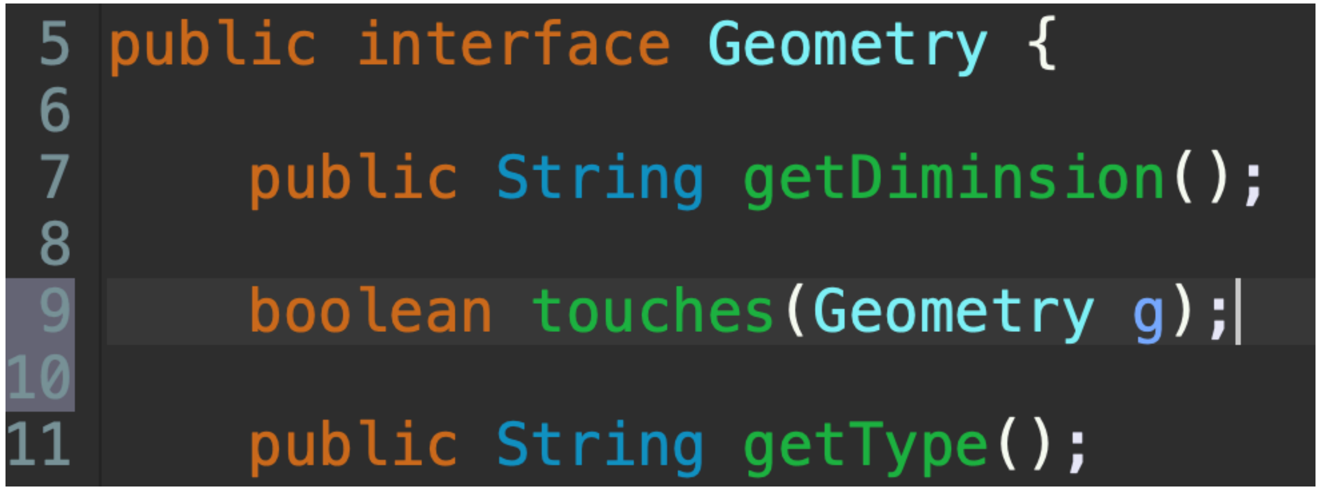
- isInside
- ...

Our Geometry contract will ensure all geometry objects have:

- getDiminsion
- getType
- getEnvelope
- isEmpty
- Equals
- getArea
- getLength
- touches
- numPoints
- ...

Remember, these are the method name. We need to build the contractual method signatures !!

Interface Example:



```
public interface Geometry {

    public String getDiminsion();

    boolean touches(Geometry g);

    public String getType();
```

We need to be explicit in the methods we expect each geometry to have. Since methods can be overloaded

Notes

- This homework is very ”easy” coding-wise.
- But much trickery conceptually.
- There is no right and wrong way to do things (but there are better ways ☺)
- You will reach “success” when all of you geometry types (point, polyline and polygon) can implement meaningful geometry methods and all of you BoundingAreas (pointBuffer and BoundingBox) can implement meaningful BoundingAreas methods.

Group Work time

- Sean Reid is inviting you to a scheduled Zoom meeting.

  Topic: Geog 178/258 Work Session
  Time: May 5, 2020 07:00 PM Pacific Time (US and Canada)

  Join Zoom Meeting
  https://ucsb.zoom.us/j/96237624575?pwd=UnhUSEV6c1BOWHlCY3l
  Db2VOU2swUT09

Code together