CST8132 OBJECT ORIENTED PROGRAMMING

ArrayList
UML 1

Professor : Dr. Anu Thomas

Email: thomasa@algonquincollege.com

Office: T314

What we learned so far?

- Arrays – 1-dim & multi-dim
- Composition
- Inheritance
- Polymorphism
- Abstraction
- Abstract class
- Interface
- Static
- This
- super

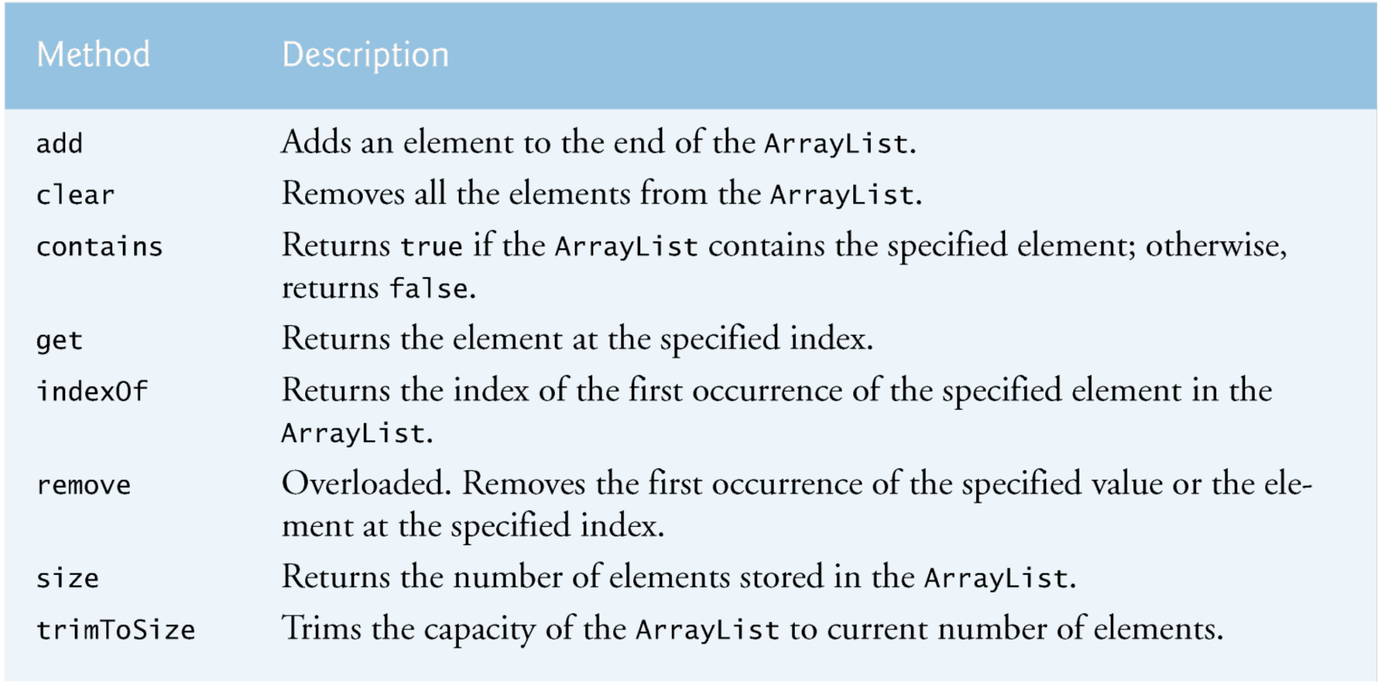
Overriding
Overloading
Access specifiers
Constructors – default, no-arg, parameterized
toString

ArrayList

- **ArrayList** is a resizable, ordered collection of elements.
- Internally, ArrayList implements a dynamically allocated array
- It is expressed as:
  - `ArrayList<T>;` meaning an ArrayList of type T

ArrayList

	array	ArrayList
Size after creation	Fixed	Dynamic
Elements	Primitive or objects	objects
Memory	Values or references are stored in contiguous locations	References are stored in contiguous locations
Get/Set	Assignment using []	Method calls
Added functionality	• length	• Add to end • Insert at position • Clear all elements • Remove element by position or value • Find element • …



Method	Description
`add`	Adds an element to the end of the `ArrayList`.
`clear`	Removes all the elements from the `ArrayList`.
`contains`	Returns `true` if the `ArrayList` contains the specified element; otherwise, returns `false`.
`get`	Returns the element at the specified index.
`indexOf`	Returns the index of the first occurrence of the specified element in the `ArrayList`.
`remove`	Overloaded. Removes the first occurrence of the specified value or the element at the specified index.
`size`	Returns the number of elements stored in the `ArrayList`.
`trimToSize`	Trims the capacity of the `ArrayList` to current number of elements.

Fig. 7.23 | Some methods and properties of class `ArrayList<T>`.

ArrayList (Contd.)

- An `ArrayList`’s capacity indicates how many items it can hold without growing.
- When the `ArrayList` grows, it must create a larger internal array and copy each element to the new array.
  - This is a time-consuming operation. It would be inefficient for the `ArrayList` to grow each time an element is added.
  - An `ArrayList` grows only when an element is added and the number of elements is equal to the capacity—i.e., there is no space for the new element.

ArrayList (Contd.)

- Method add adds elements to the `ArrayList`.
  - One-argument version appends its argument to the end of the `ArrayList`.
  - Two-argument version inserts a new element at the specified position.
  - Collection indices start at zero.
- Method size returns the number of elements in the `ArrayList`.
- Method get obtains the element at a specified index.
- Method remove deletes an element with a specific value.
  - An overloaded version of the method removes the element at the specified index.
- Method contains determines if an item is in the `ArrayList`.

ArrayList (Create and Add)

- Array

```
String studentNames[] = new String[100];
int currentPosition = 0;
studentNames[currentPosition++]=“Peter”;
studentNames[currentPosition++]=“Pauline”;
studentNames[currentPosition++]=“Robin”;
```

- ArrayList

```
List<String> studentNames = new ArrayList<>();
studentNames.add("Peter");
studentNames.add("Pauline");
studentNames.add("Robin");
```

ArrayList (Print names)

- Array

```
for(int i=0; i<=currentPosition; i++)
   System.out.println(studentNames[i]);
```

- ArrayList

```
for(String s : studentNames)
   System.out.println(s);
```

- `System.out.println(studentNames)`

ArrayList (Insert at first position)

- Array

```
for(int i=currentPosition; i>0; i--)
   studentNames[i]=studentNames[i-1];
studentNames[0]=“John";
```

- ArrayList

```
studentNames.add(0, “John");
```

Type-Wrapper Classes

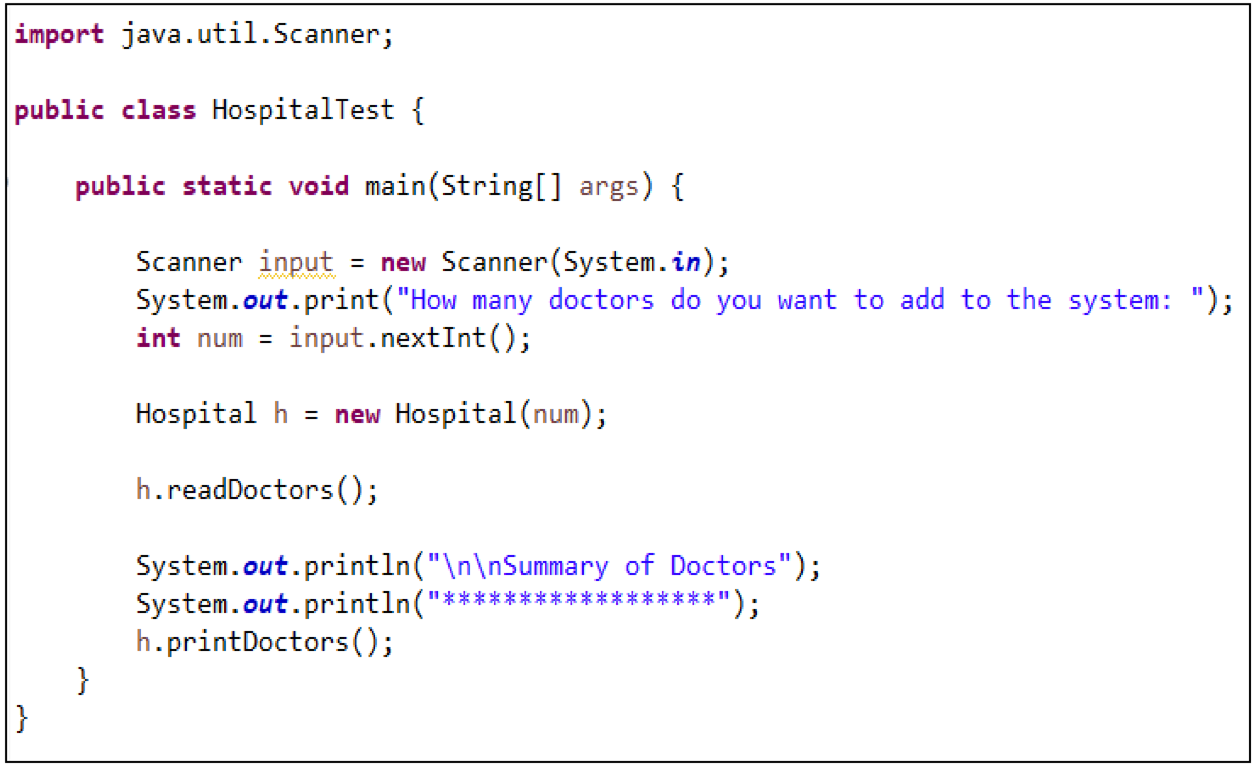
- Each primitive type has a corresponding type-wrapper class (in package `java.lang`).
  - Boolean, Byte, Character, Double, Float, Integer, Long and Short.
- Each type-wrapper class enables you to manipulate primitive-type values as objects.
- Collections cannot manipulate variables of primitive types.
  - They can manipulate objects of the type-wrapper classes, because every class ultimately derives from `Object`.
- Each of the numeric type-wrapper classes—`Byte`, `Short`, `Integer`, `Long`, `Float` and `Double`—extends class `Number`.
- The type-wrapper classes are `final` classes, so you cannot extend them.
- Primitive types do not have methods, so the methods related to a primitive type are in
- the corresponding type-wrapper class

Autoboxing and Auto-Unboxing

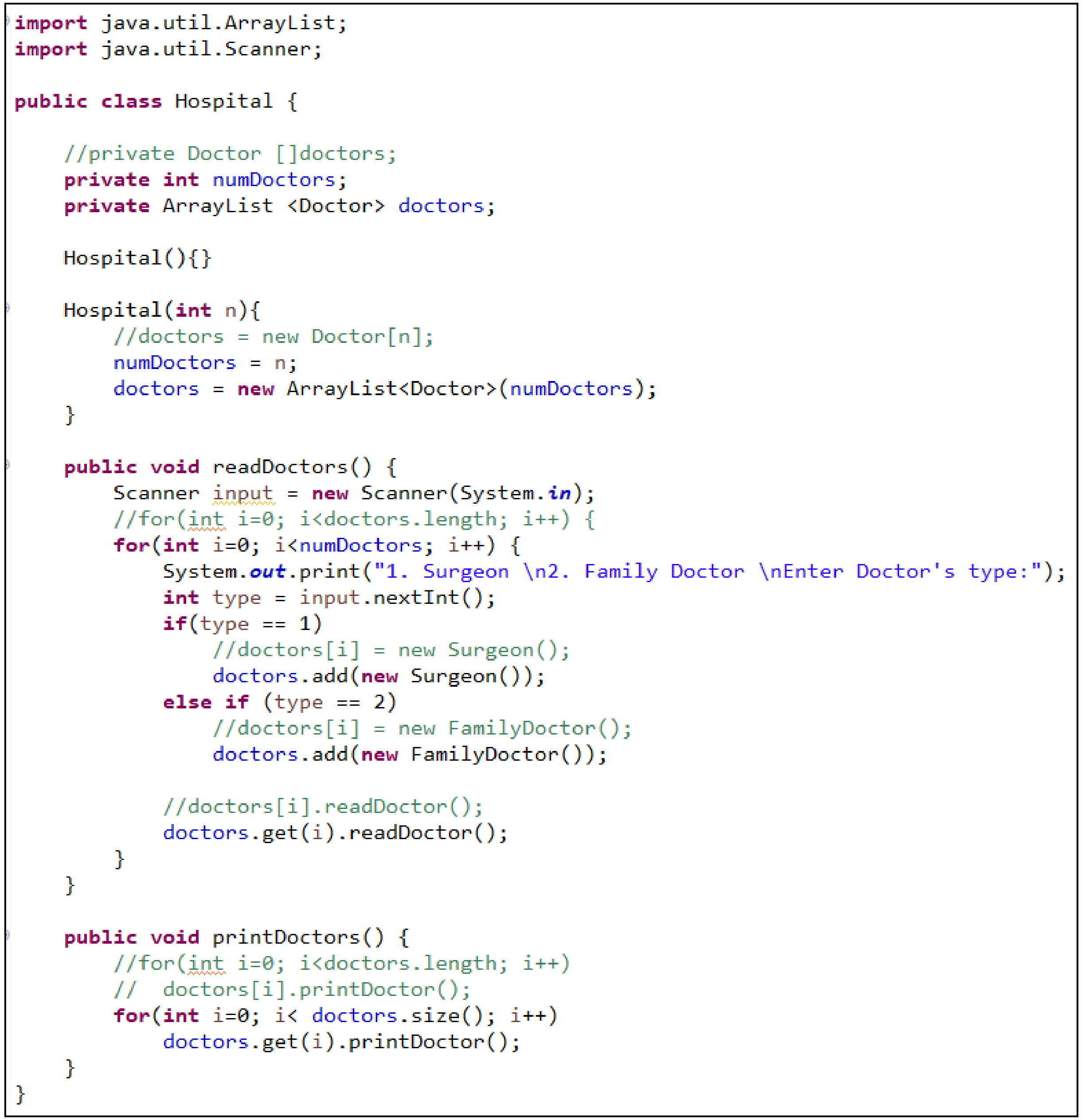
- A boxing conversion converts a value of a primitive type to an object of the corresponding type-wrapper class.
- An unboxing conversion converts an object of a type-wrapper class to a value of the corresponding primitive type.
- These conversions are performed automatically—called autoboxing and auto-unboxing.
- Example:

  -
    ```
    // create integerArray
    Integer[] integerArray = new Integer[5];

    // assign Integer 10 to integerArray[ 0 ] integerArray[0] = 10;

    // get int value of Integer
    int value = integerArray[0];
    ```

Example – Hospital System



```java
import java.util.Scanner;

public class HospitalTest {

    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);
        System.out.print("How many doctors do you want to add to the system: ");
        int num = input.nextInt();

        Hospital h = new Hospital(num);

        h.readDoctors();

        System.out.println("\n\nSummary of Doctors");
        System.out.println("******************");
        h.printDoctors();
    }
}
```



```java
import java.util.ArrayList;
import java.util.Scanner;

public class Hospital {

    //private Doctor []doctors;
    private int numDoctors;
    private ArrayList <Doctor> doctors;

    Hospital(){}

    Hospital(int n){
        //doctors = new Doctor[n];
        numDoctors = n;
        doctors = new ArrayList<Doctor>(numDoctors);
    }

    public void readDoctors() {
        Scanner input = new Scanner(System.in);
        //for(int i=0; i<doctors.length; i++) {
        for(int i=0; i<numDoctors; i++) {
            System.out.print("1. Surgeon \n2. Family Doctor \nEnter Doctor's type:");
            int type = input.nextInt();
            if(type == 1)
                //doctors[i] = new Surgeon();
                doctors.add(new Surgeon());
            else if (type == 2)
                //doctors[i] = new FamilyDoctor();
                doctors.add(new FamilyDoctor());

            //doctors[i].readDoctor();
            doctors.get(i).readDoctor();
        }
    }

    public void printDoctors() {
        //for(int i=0; i<doctors.length; i++)
        //  doctors[i].printDoctor();
        for(int i=0; i< doctors.size(); i++)
            doctors.get(i).printDoctor();
    }
}
```

Final modifier

- With variable – becomes constant
- With method – cannot be overridden in subclasses
- With class – cannot be sub-classed

Eclipse demo

UML – Unified Modeling Language

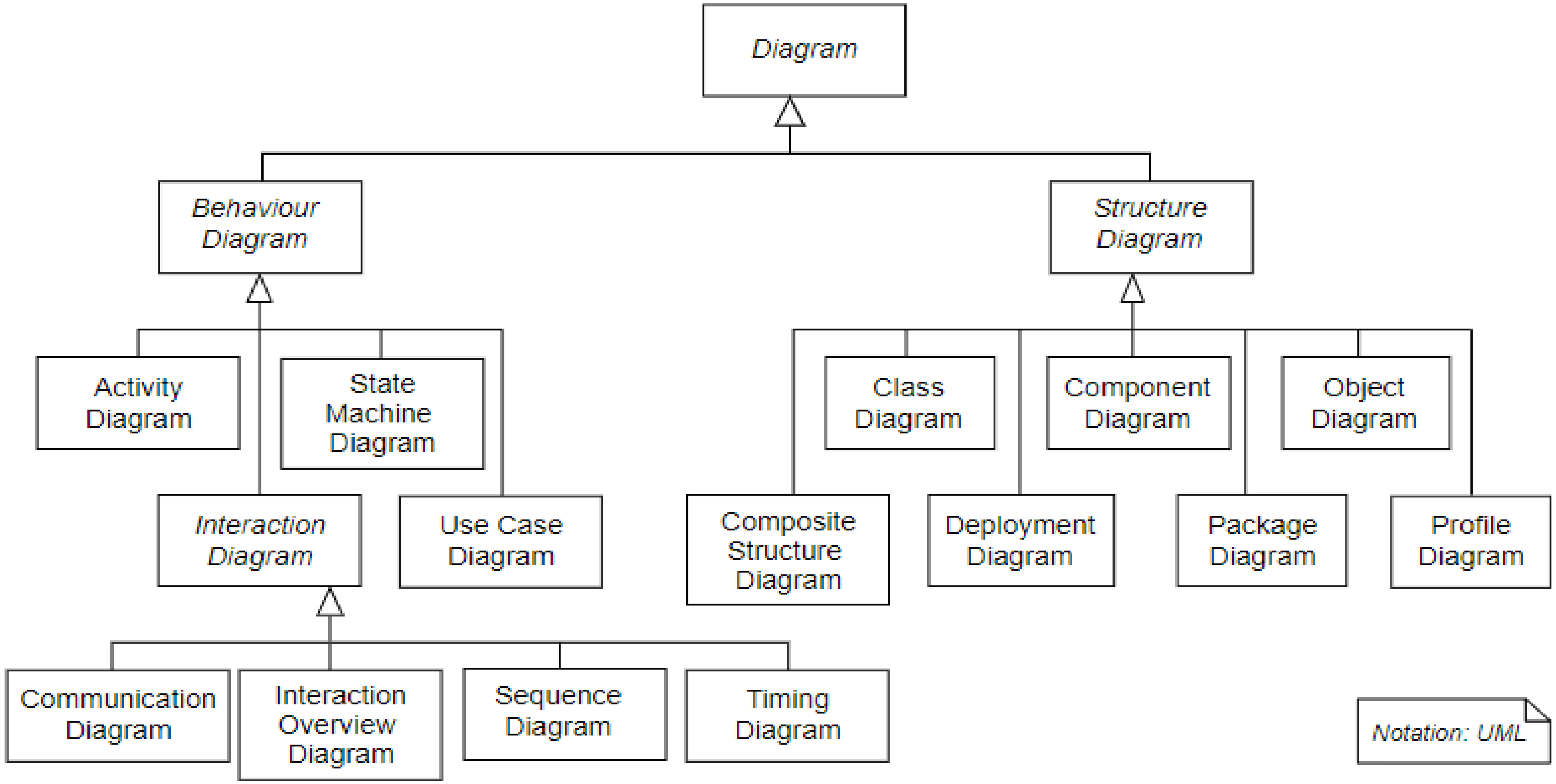
- General-purpose developmental modeling language
- Standard way to visualize the design of a system

- References:
- https://www.oracle.com/technetwork/developer-tools/jdev/gettingstartedwithumlclassmodeling-130316.pdf
- https://en.wikipedia.org/wiki/Unified_Modeling_Language
- https://creately.com/blog/diagrams/class-diagram-tutorial/

Uses of UML

- As a sketch: to communicate aspects of the system
  - Forward design: create UML before coding
  - Backward design: create UML after coding (for documentation)
  - Often done on whiteboard or on paper
  - Used in brainstorming
- As a blueprint: a complete design to be implemented
  - Done with professional tools like Visio
- As a programming language: tools available to auto-generate the structure of the code from the UML

UML diagrams



Structural : Class Diagram

- Top compartment – class name – Centered and **Bold**
- Middle compartment – State (attributes, properties, instance or class variables)
- Bottom compartment – Behaviors (methods)
- Access Level Modifiers:
  - + means public
  - # means protected
  - ~ means package protected
  - - means private
- Notice that in UML, types come after the member name, rather than before in Java

In UML:

```
-myInt: int
+factorial(n: int): int
```

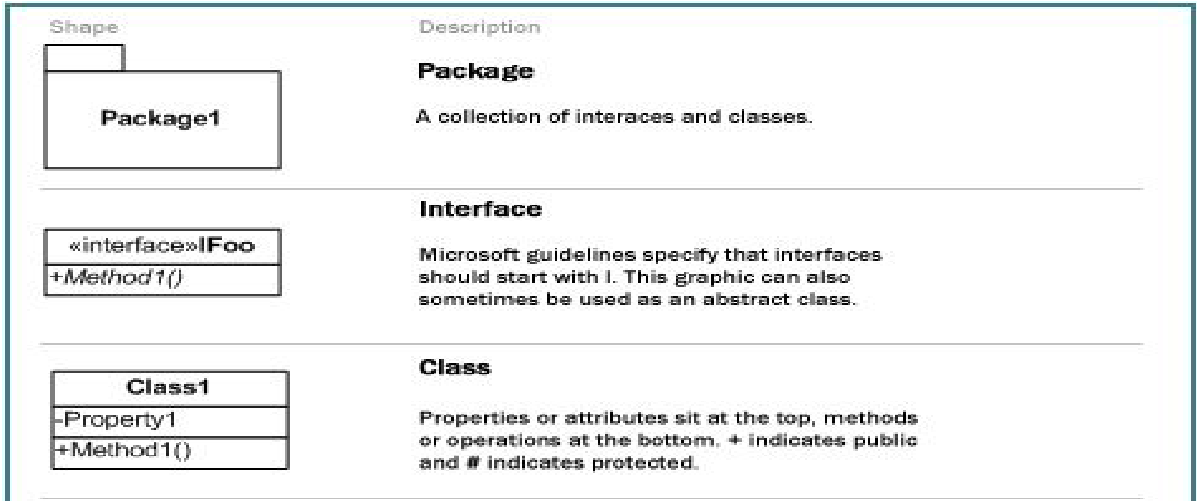
In Java:

private int myInt and
public int factorial (int n)

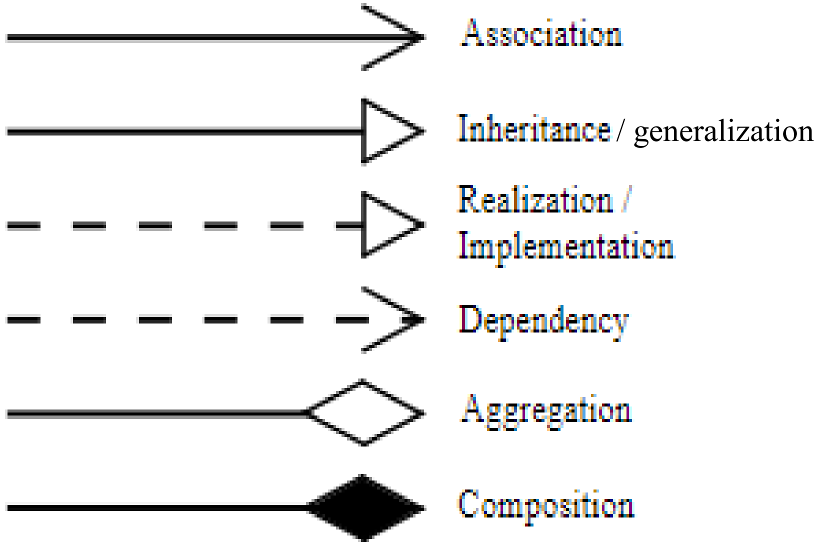
UML Relationships

- Lines drawn between class boxes indicate relationships between objects of the classes
  - Multiplicity: numbers at either end of an association line represent how many of each are involved
    - 0..1 – no instances or one instance - optional
    - 1 or 1..1– one and only one
    - 1..n – one to a specific limit
    - 1..* - one or more
    - 0..* - zero or more
    - * - zero or more
    - 0..n – zero to a specific limit

UML Symbols



Shape	Description
Package1	**Package** A collection of interaces and classes.
«interface»IFoo *+Method1()*	**Interface** Microsoft guidelines specify that interfaces should start with I. This graphic can also sometimes be used as an abstract class.
Class1 -Property1 +Method1()	**Class** Properties or attributes sit at the top, methods or operations at the bottom. + indicates public and # indicates protected.



Association

Inheritance / generalization

Realization / Implementation

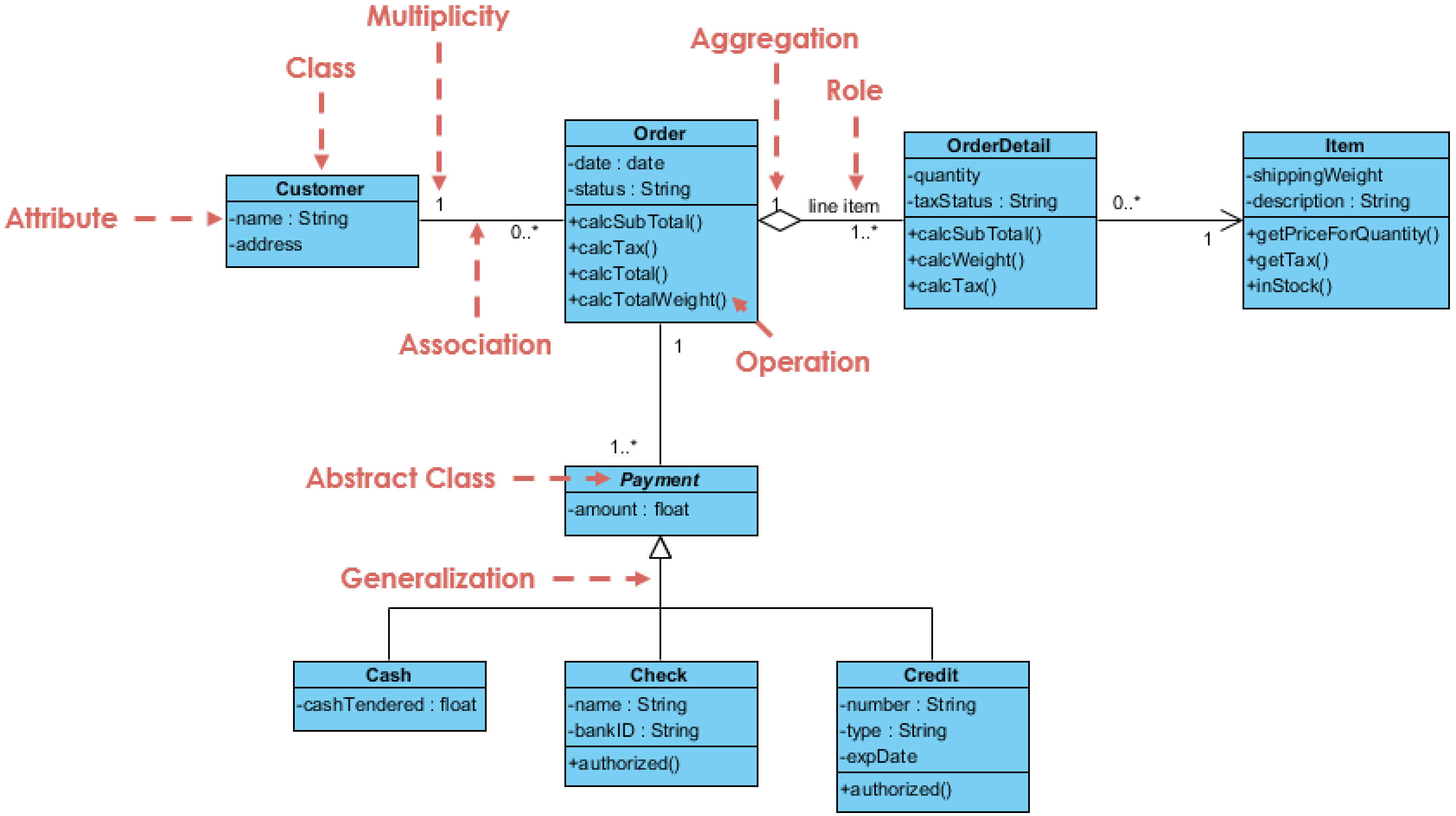
Dependency

Aggregation

Composition

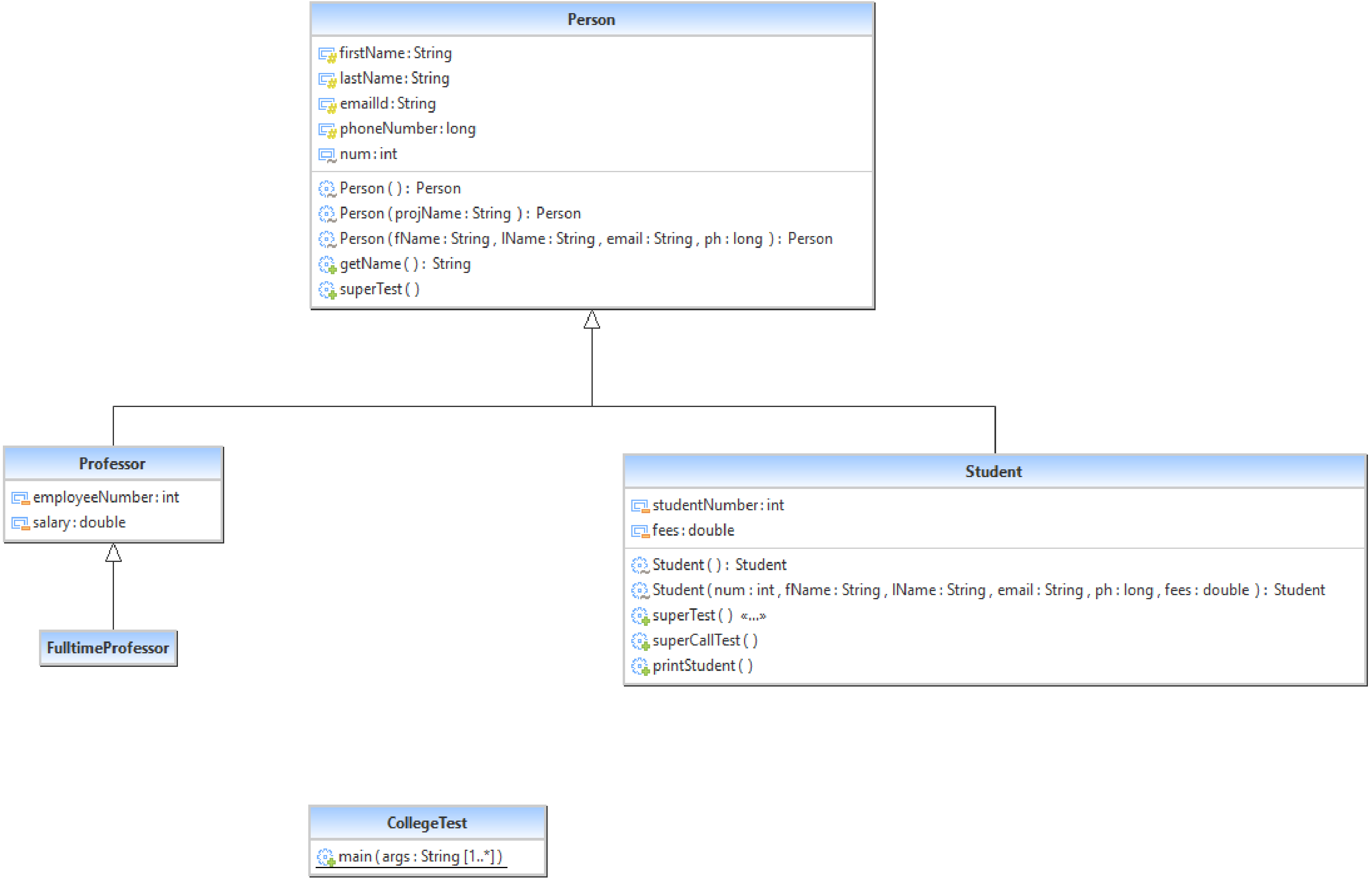
Research more!

Example



Taken from https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/

UML Class Diagram – College System



Person

- firstName : String
- lastName : String
- emailId : String
- phoneNumber : long
- num : int

- Person ( ) : Person
- Person (projName : String) : Person
- Person (fName : String , lName : String , email : String , ph : long) : Person
- getName ( ) : String
- superTest ( )

Professor

- employeeNumber : int
- salary : double

FulltimeProfessor

Student

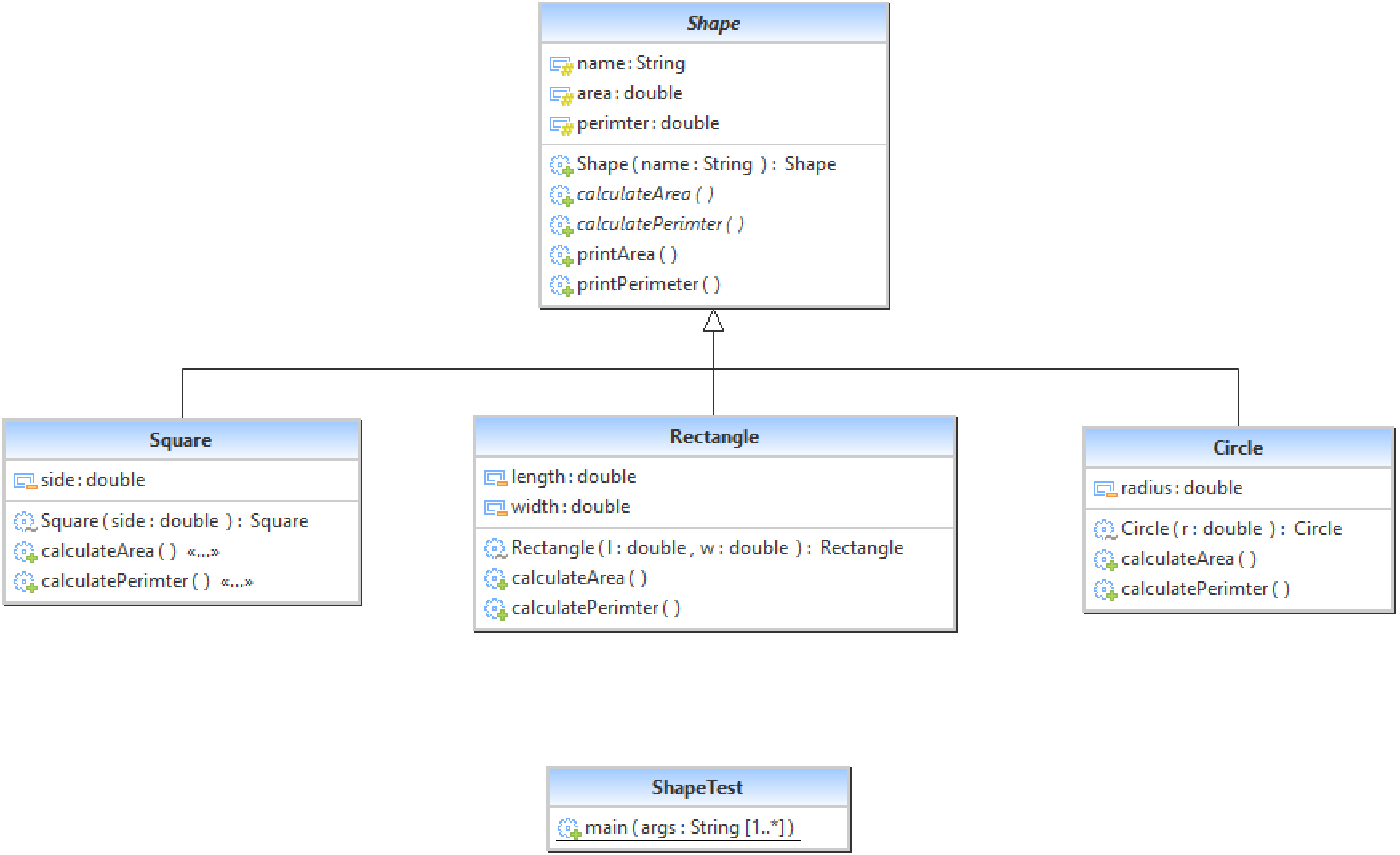
- studentNumber : int
- fees : double

- Student ( ) : Student
- Student (num : int , fName : String , lName : String , email : String , ph : long , fees : double) : Student
- superTest ( ) «...»
- superCallTest ( )
- printStudent ( )

CollegeTest

- main (args : String [1..*])

UML class diagram – Shape (abstract class)



Shape

name : String
area : double
perimter : double

Shape (name : String) : Shape
calculateArea ( )
calculatePerimter ( )
printArea ( )
printPerimeter ( )

Square

side : double

Square (side : double) : Square
calculateArea ( ) «...»
calculatePerimter ( ) «...»

Rectangle

length : double
width : double

Rectangle (l : double , w : double) : Rectangle
calculateArea ( )
calculatePerimter ( )

Circle

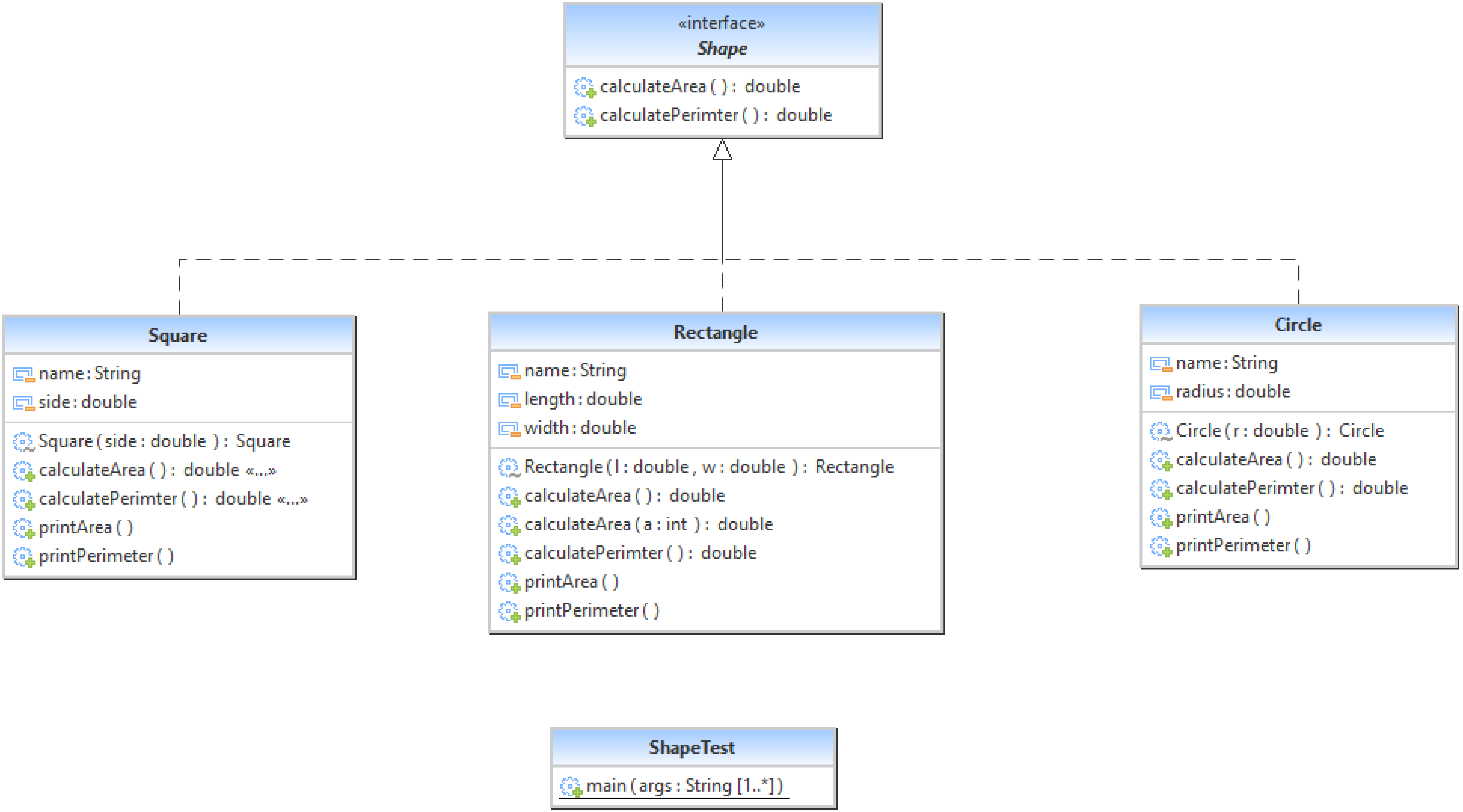
radius : double

Circle (r : double) : Circle
calculateArea ( )
calculatePerimter ( )

ShapeTest

main (args : String [1..*])

Shape (interface)



«interface»
Shape

calculateArea ( ) : double
calculatePerimter ( ) : double

Square

name : String
side : double

Square (side : double) : Square
calculateArea ( ) : double «...»
calculatePerimter ( ) : double «...»
printArea ( )
printPerimeter ( )

Rectangle

name : String
length : double
width : double

Rectangle (l : double , w : double) : Rectangle
calculateArea ( ) : double
calculateArea (a : int) : double
calculatePerimter ( ) : double
printArea ( )
printPerimeter ( )

Circle

name : String
radius : double

Circle (r : double) : Circle
calculateArea ( ) : double
calculatePerimter ( ) : double
printArea ( )
printPerimeter ( )

ShapeTest

main (args : String [1..*])

- Final
  - Variables – makes them constant
  - Methods – cannot be overridden
  - Classes – cannot be subclassed