

# AP Computer Science B

## Java Object-Oriented Programming [Ver. 2.0]

### Unit 4: Object-Oriented Design

WEEK 4: CHAPTER 11 OBJECT-ORIENTED THINKING (PART 1: CLASS-TO-CLASS)

DR. ERIC CHOU

IEEE SENIOR MEMBER



# Objectives

---

- Class Loading Time and Data Memory Model
- Design of a Class
- has\_A Relationship
- Conversion of Static Program to Object-Oriented Programs
- UML Editor
- Stack of Integers



# Loading of a Class and Its Objects

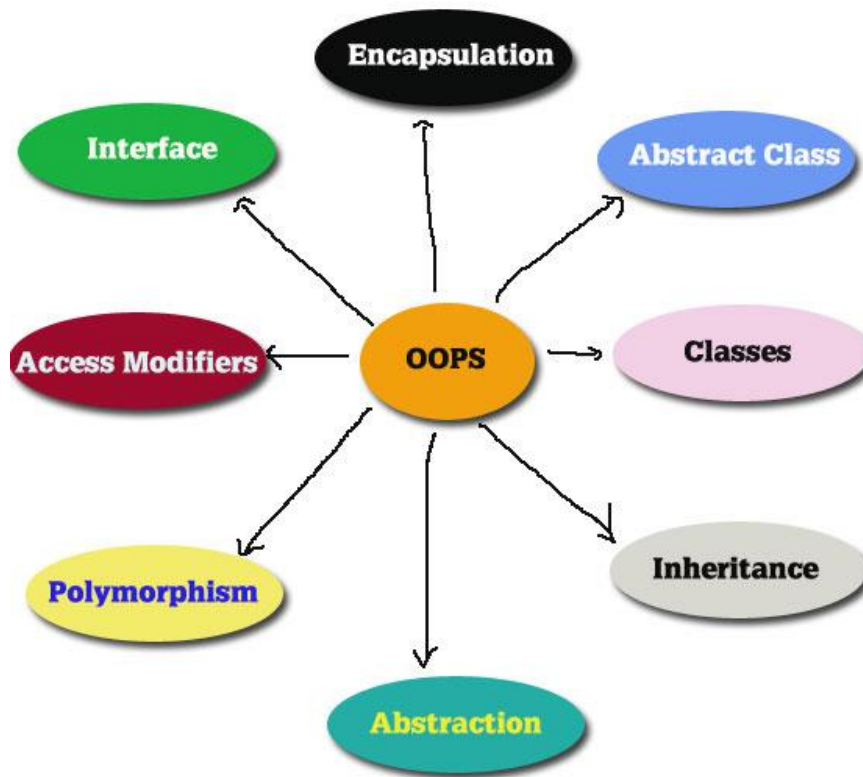
---

LECTURE 1



# Single Class Design Issues

scope, visibility modifiers, static modifiers, and final modifier



## Static Variable/Static Method:

Constants: (final static variables)

Class Variable: (static variables)

Utility Method: (static methods)

## Instance Variable/Instance Method:

encapsulated data fields: (private data)

un-protected data: (public data)

accessor/mutator methods: (public method)

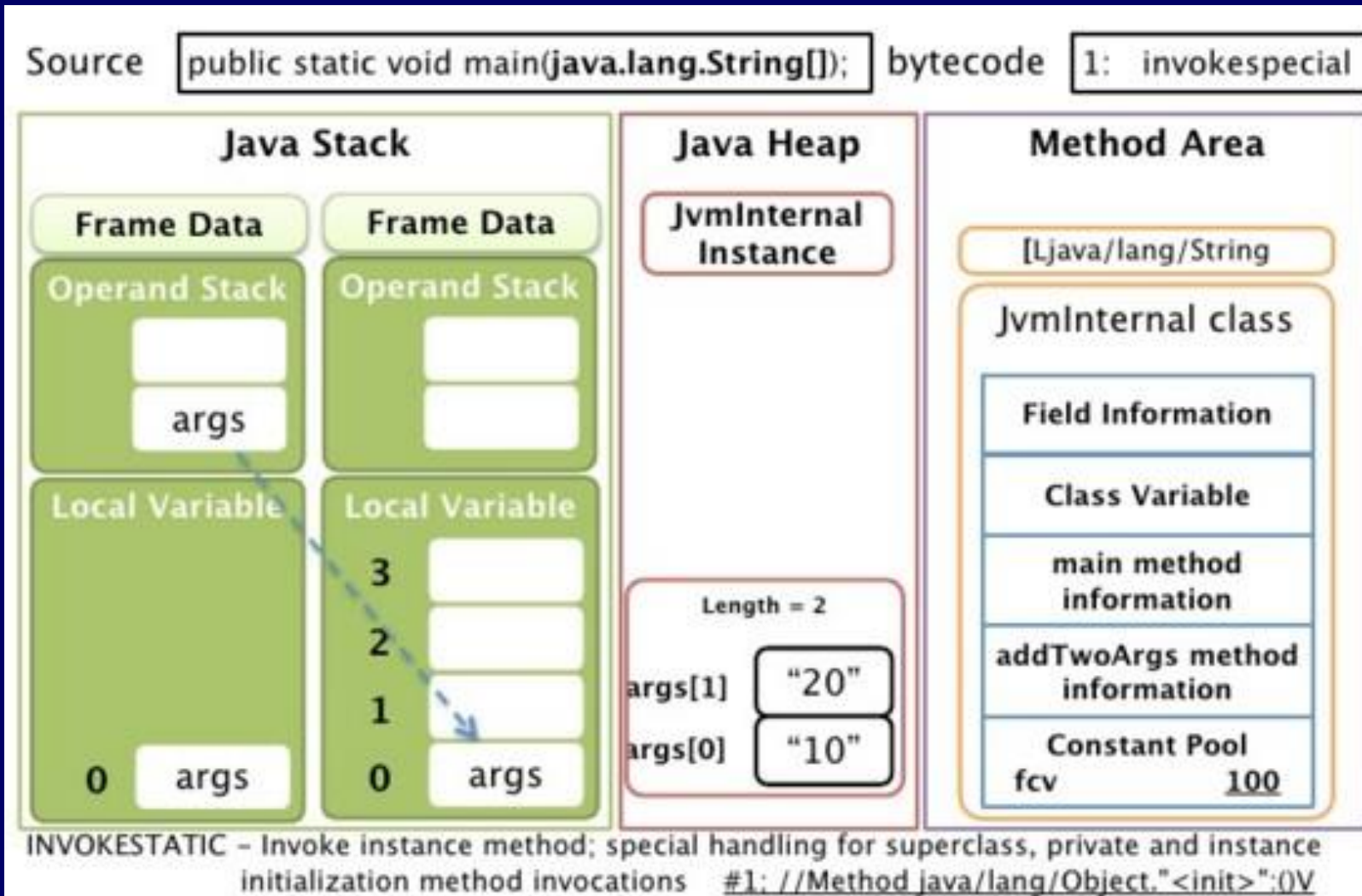
client methods: (private method)

**Encapsulated Data Class:** private data/public method

**Immutable Data Class:** private data/no mutator methods/  
no returned pointer



# Memory Allocation



# Loading a Class to JVM

## Static Variable/Static Method:

Constants: (final static variables)

Math.PI, Integer.MIN\_VALUE

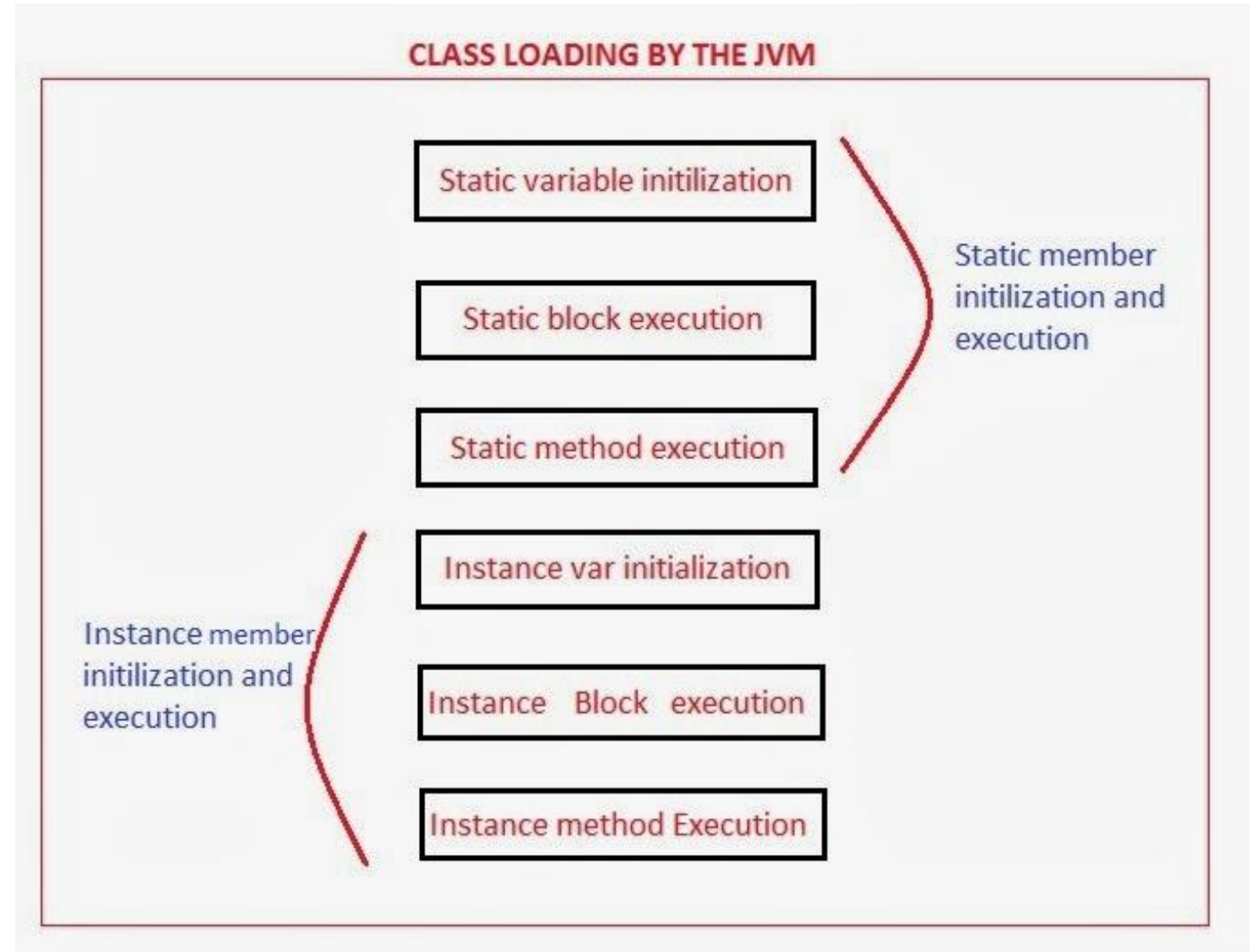
Class Variable: (static variables)

Circle.count

Utility Method: (static methods)

Math.random(), Math.abs(),

Integer.parseInt()



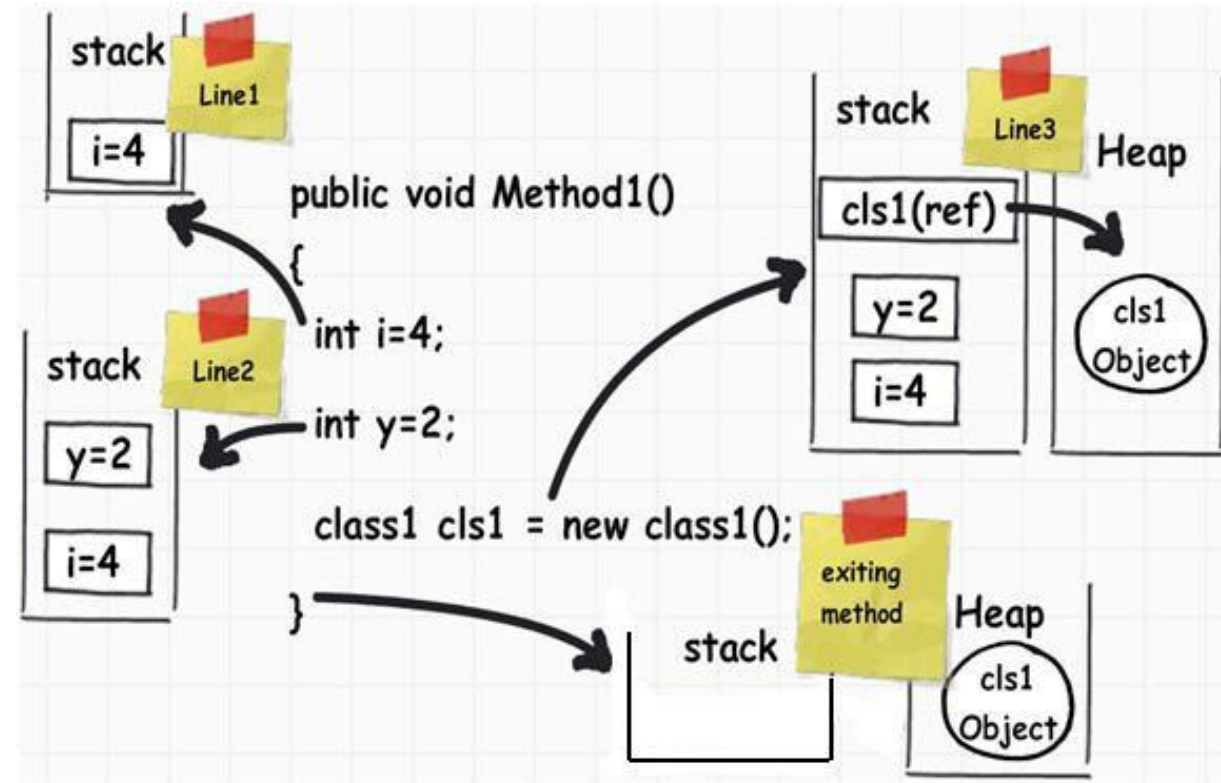


# Object-Oriented Programming

(scope, visibility modifiers, static modifiers, and final modifier)

## Instance Variable/Instance Method:

- encapsulated data fields: (private data)
- un-protected data: (public data)
- accessor/mutator methods: (public method)
- client methods: (private method)





# Object-Thinking: Design of a Class

---

LECTURE 2





# Object-Oriented Thinking

---

- Part 1-Chapters 1-8 introduced fundamental programming techniques for problem solving using loops, methods, and arrays.
- The studies of these techniques lay a solid foundation for object-oriented programming. Classes provide more flexibility and modularity for building reusable software.



# Object-Oriented Thinking

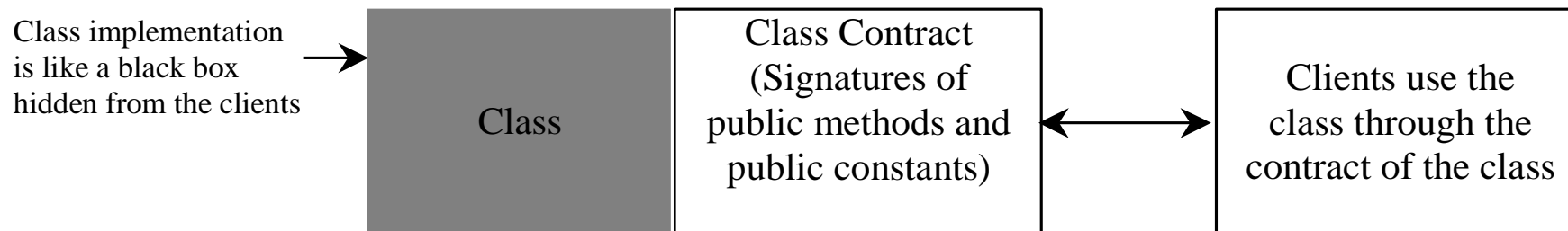
---

- This chapter reviews chapter 9 and improves the solution for a problem introduced in Part-1 using the object-oriented approach.
- From the improvements, you will gain the insight on the differences between the procedural programming and object-oriented programming and see the benefits of developing reusable code using objects and classes.



# Class Abstraction and Encapsulation

**Class abstraction** means **to separate class implementation from the use of the class**. The creator of the class provides a description of the class and let the user know how the class can be used. The user of the class does not need to know how the class is implemented. The detail of implementation is encapsulated and hidden from the user.





# Designing a Class: Coherence

---

- **(Coherence)** A class should describe a **single entity**, and all the class operations should logically fit together to support a coherent purpose. You can use a class for students, for example, but you should not combine students and staff in the same class, because students and staff have different entities.

e.g. Student, Subject, ScoreSheet, Card, Deck, and Hand



# Designing a Class, cont.

---

- **(Separating responsibilities)** A single entity with too many responsibilities can be broken into several classes to separate responsibilities.
- The classes **String**, **StringBuilder**, and **StringBuffer** all deal with strings, for example, but have different responsibilities. The **String** class deals with immutable strings, the **StringBuilder** class is for creating mutable strings, and the **StringBuffer** class is similar to **StringBuilder** except that **StringBuffer** contains synchronized methods for updating strings.



# Designing a Class, cont.

---

- Classes are designed for reuse. Users can incorporate classes in many different combinations, orders, and environments. Therefore, you should design a class that imposes **no restrictions** on what or when the user can do with it, design the properties to ensure that the user can set properties in any order, with any combination of values, and design methods to function independently of their order of occurrence.



# Designing a Class, cont.

---

- Provide a public no-arg constructor and override the equals method and the toString method defined in the Object class whenever possible.
- Overriding standard methods inherited from Object class.



# Designing a Class, cont.

---

- Follow standard Java programming style and naming conventions. Choose informative names for classes, data fields, and methods.
- Always place the data declaration before the constructor, and place constructors before methods.
- Always provide a constructor and initialize variables to avoid programming errors.





# How to Design a Good Class

---

LECTURE 3



# Guidelines for Class Design

---

- Good design of individual classes is crucial to good overall system design. A well-designed class is **more re-usable** in different contexts, and **more modifiable** for future versions of software.
- Here, we'll look at some general class design guidelines, as well as some tips for specific languages, like **C++** or **Java**.

*(Chapter 9's class design guidelines on technical issue, this chapter is on design styles)*



# General goals for building a good class

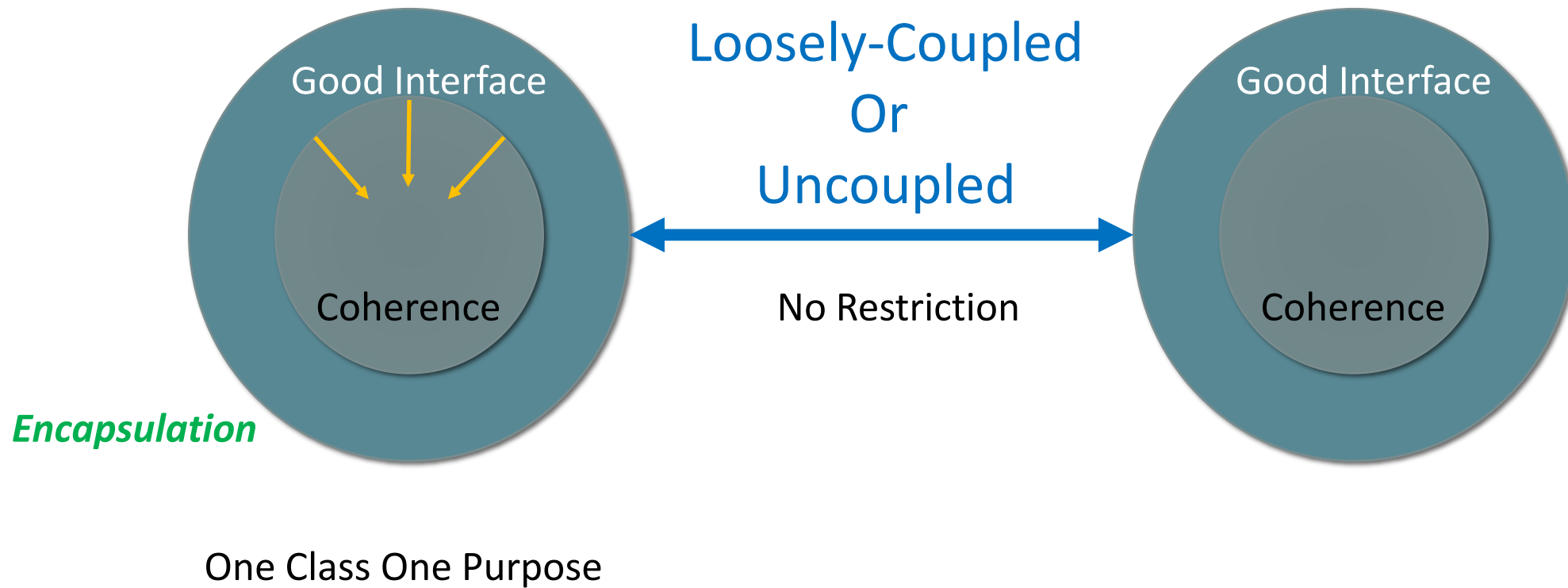
---

- Having a **good usable interface** (**Information Hiding**)
- **Implementation objectives**, like efficient algorithms and convenient/simple coding
- Separation of implementation from interface!! (**Encapsulation**)
- Improves **modifiability** and **maintainability** for future versions (**re-usability**)
- **Decreases coupling between classes**, i.e. the amount of dependency between classes (will changing one class require changes to another?) (**no restrictions**)
- Can re-work a class inside, without changing its interface, for outside interactions
- Consider how much the automobile has advanced technologically since its invention. Yet the basic interface remains the same -- steering wheel, gas pedal, brake pedal, etc.



# Good Class Design

---





# Designing a good class interface

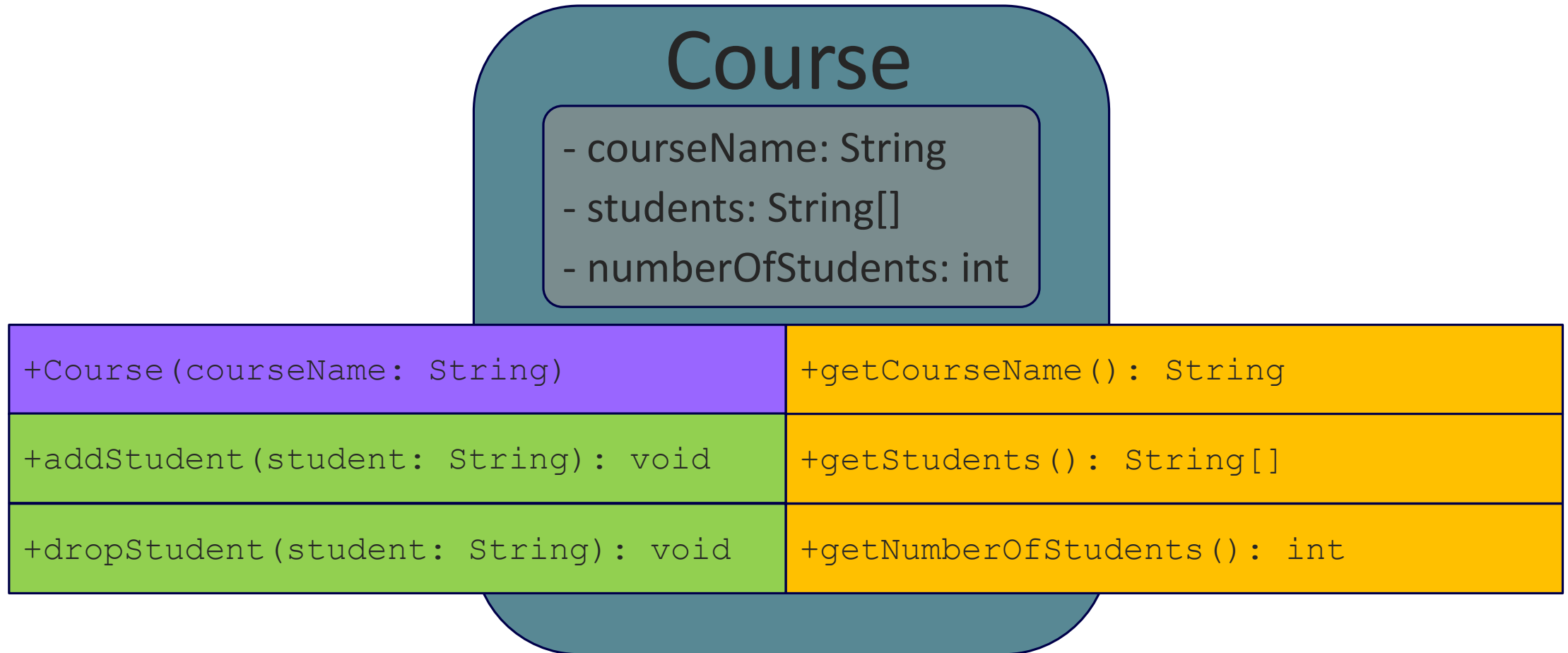
By interface, we are talking about what the class user sees. This is the public section of the class.

- **Cohesion** (or *coherence*): one class single abstraction
- **Completeness**: A class should support all important operations
- **Convenience**:
  1. User-friendly
  2. API-Oriented (written like API)
  3. Systematic
- **Clarity**: No confusion or ambiguous interpretations
- **Consistency**
  1. Operations in a class will be most clear if they are consistent with each other.
  2. Naming conventions (toString, compareTo, equals, isLetter, and etc.)



# Course Class

## One Abstraction: Course Information





# The Course Class

Course
<pre>-courseName: String -students: String[] -numberOfStudents: int</pre>
<pre>+Course(courseName: String) +getCourseName(): String +addStudent(student: String): void +dropStudent(student: String): void +getStudents(): String[] +getNumberOfStudents(): int</pre>

The name of the course.

An array to store the students for the course.

The number of students (default: 0).

Creates a course with the specified name.

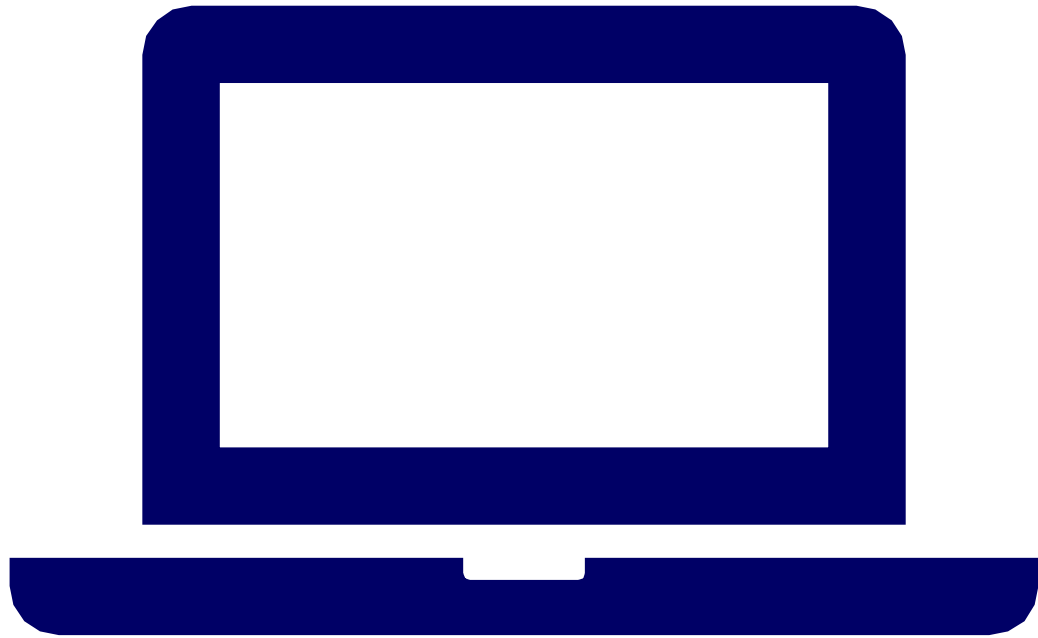
Returns the course name.

Adds a new student to the course.

Drops a student from the course.

Returns the students in the course.

Returns the number of students in the course.



# Demonstration Program

---

`COURSE.JAVA/TESTCOURSE.JAVA`





# Completeness and Design Conventions

---

LECTURE 4



# Class Design

---

## **Class Data Fields:**

- Constants
- Class Variables

## **Member Data Fields:**

- Instant Variables

## **Constructors:**

- no-arg
- Long-form

## **Getters/Setters Method:**

- no-arg
- Long-form

## **Inherited Methods:**

- toString()
- Equals()
- compareTo()
- next(), hasNext(), remove()

## **Derived Data Fields:**

- getDerivedData()

## **Utility Methods:**

- Static Methods



# Class Writing Styles

## (Not Syntax, Design Conventions)

```
public class Counter
{
    private final String name;
    private int count;

    public Counter(String id)
    { name = id; }

    public void increment()
    { count++; }

    public int tally()
    { return count; }

    public String toString()
    { return count + " " + name; }

    public static void main(String[] args)
    {
        Counter heads = new Counter("heads");
        Counter tails = new Counter("tails");

        heads.increment();
        heads.increment();
        tails.increment();

        StdOut.println(heads + " " + tails);
        StdOut.println(heads.tally() + tails.tally());
    }
}
```

Annotations:

- instance variables* → `private final String name;` and `private int count;`
- constructor* → `public Counter(String id)`
- instance methods* → `public void increment()`, `public int tally()`, and `public String toString()`
- test client* → `public static void main(String[] args)`
- create and initialize objects* → `Counter heads = new Counter("heads");` and `Counter tails = new Counter("tails");`
- invoke constructor* → `new Counter("heads")` and `new Counter("tails")`
- object name* → `heads` and `tails`
- invoke method* → `heads.tally()` and `tails.tally()`
- instance variable name* → `name` in `toString()`
- automatically invoke toString()* → `heads + " " + tails`



# Completeness

How to create it? How to get it? How to change it? And, how to show it?

---

**Constructor Parameter Type Manipulation** (Overloading of Constructor Method):

// Make user easier to use.

Complex(), Complex(double r), Complex(double r, double i)

**Accessor/Mutator:**

getR(), getI(), setR(), setI(), set(double r, double i);

**Comparators:**

.equals(), Complex.equals(Complex c1, Complex c2)

**To String Method:**

.toString(), Complex.toString(Complex c)

# BMI calculation

[http://www.nhlbi.nih.gov/health/educational/lose\\_wt/BMI/bmicalc.htm](http://www.nhlbi.nih.gov/health/educational/lose_wt/BMI/bmicalc.htm)

## Calculate Your Body Mass Index

Body mass index (BMI) is a measure of body fat based on height and weight that applies to adult men and women.

- Enter your weight and height using standard or metric measures.
- Select "Compute BMI" and your BMI will appear below.

Español

**STANDARD**


**METRIC**

Your Height:    
(feet) (inches)

Your Weight:   
(pounds)

Compute BMI


Your BMI:



**BMI Categories:**  
Underweight =  $<18.5$   
Normal weight =  $18.5\text{--}24.9$   
Overweight =  $25\text{--}29.9$   
Obesity = BMI of 30 or greater

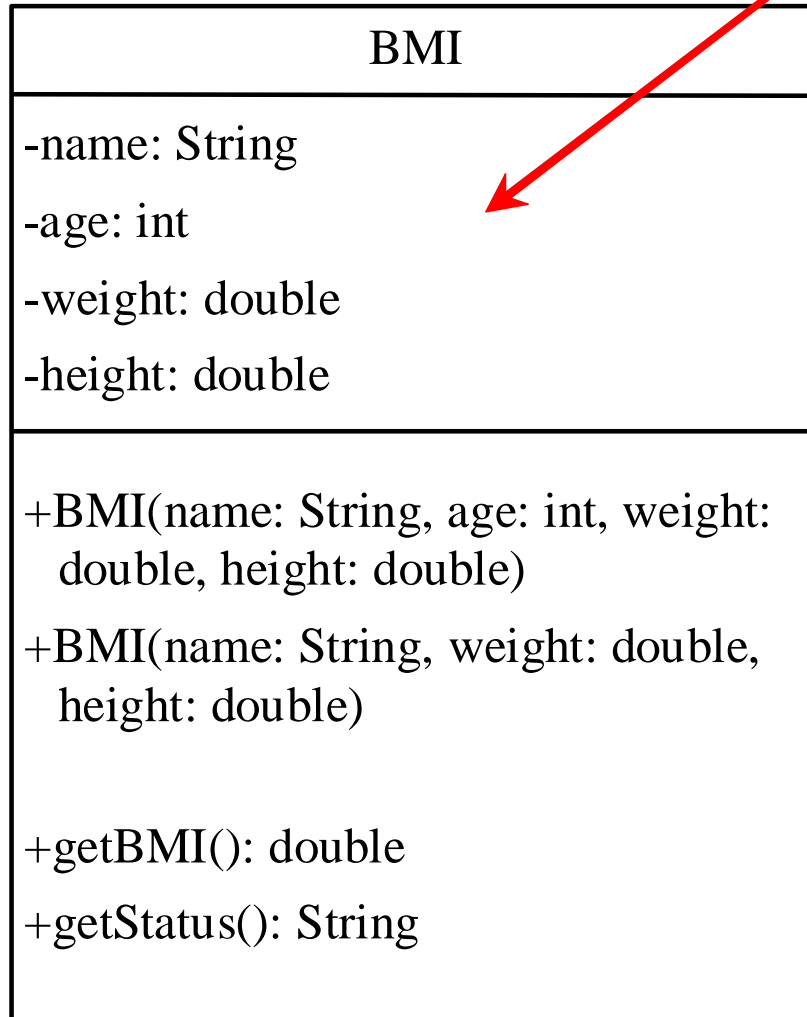
The BMI Tables

Aim for a Healthy Weight:  
Limitations of the BMI  
Assessing Your Risk  
Controlling Your Weight  
Recipes

 Download the BMI Calculator iPhone App

# The BMI Class

## Incomplete Version



The get methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

The name of the person.

The age of the person.

The weight of the person in pounds.

The height of the person in inches.

Creates a BMI object with the specified name, age, weight, and height.

Creates a BMI object with the specified name, weight, height, and a default age 20.

Returns the BMI

Returns the BMI status (e.g., normal, overweight, etc.)

BMI class
<ul style="list-style-type: none"> <li>- name: String</li> <li>- age: int</li> <li>- weight: double</li> <li>- height: double</li> <li>+ final KILOGRAMS_PER_POUND: static double</li> <li>+ final METERS_PER_INCH: static double</li> </ul>
<pre> &lt;&lt; Constructors &gt;&gt; + BMI(String name, int age, double weight, double height) + BMI(String name, double weight, double height) &lt;&lt; Methods &gt;&gt; + getBMI(): double + getStatus(): String + getName(): String + getAge(): int + getWeight(): double + getHeight(): double + setName(String name): void + setAge(int age): void + setWeight(double weight): double + setHeight(double height): double + equals(BMI b): boolean + compareTo(BMI b): double + toString(): String </pre>

# The BMI Class

## Complete Version

**getBMI()** method is an implicit method (implicit variable, or derived variable). The class stores only **weight** and **height**.

The **BMI** number is calculated on method calls. The class does not store it.

The **Status** string is also calculated on method calls. The class does not store it, neither.

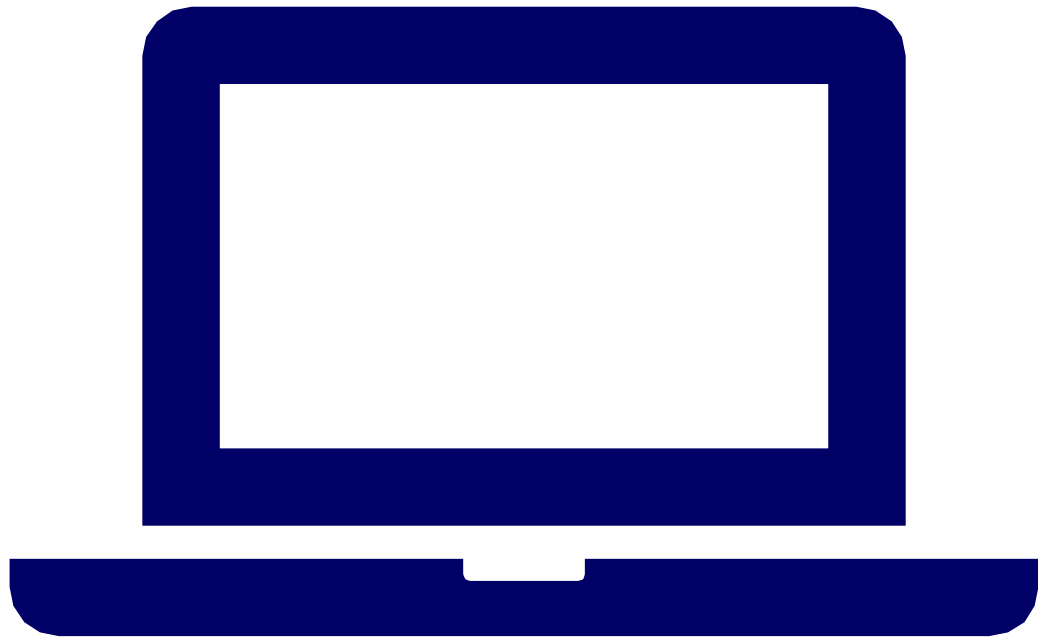


# Review Use of Violet UML Tool

---

- If you did not have violet UML tool, download from here:  
<http://horstmann.com/violet/>
- Try to learn how to design UML class (we need other UML knowledge later. )





# Demonstration Program

---

BMI/TESTBMI CLASS



# Assignment

---

GEOMETRIC 1 ASSIGNMENT

SUBMIT YOUR PROGRAM TO  
MOODLE COURSE UPLOAD LINK



# Lab Project: BMR class

---

LECTURE 5



# What is BMR (Basal Metabolic Rate)

**Basal metabolic rate (BMR) is the amount of energy expended while at rest in a neutrally temperate environment. Find out your BMR with this handy calculator!**

Basal metabolic rate (BMR) is the amount of energy expended while at rest in a neutrally temperate environment, in the post-absorptive state (meaning that the digestive system is inactive, which requires about twelve hours of [fasting](#)).

The release of energy in this state is sufficient only for the functioning of the vital organs, such as the heart, lungs, brain and the rest of the nervous system, liver, kidneys, sex organs, muscles and skin. BMR decreases with age and with the loss of lean body mass. Increasing muscle mass increases BMR.



## BMI Calculator » BMR Formula

If you are unable to use our [BMR Calculator](#), or if you are interested in how BMR is calculated, this page has the mathematical **BMR Formulas**.

The **BMR formula** uses the variables of height, weight, age and gender to calculate the Basal Metabolic Rate (BMR). This is more accurate than calculating calorie needs based on body weight alone. The only factor it omits is lean body mass and thus the ratio of muscle-to-fat a body has. Remember, leaner bodies need more calories than less leaner ones. Therefore, this equation will be very accurate in all but the very muscular (will underestimate calorie needs) and the very fat (will overestimate calorie needs).

### English BMR Formula

**Women:**  $\text{BMR} = 655 + (4.35 \times \text{weight in pounds}) + (4.7 \times \text{height in inches}) - (4.7 \times \text{age in years})$

**Men:**  $\text{BMR} = 66 + (6.23 \times \text{weight in pounds}) + (12.7 \times \text{height in inches}) - (6.8 \times \text{age in year})$

### Metric BMR Formula

**Women:**  $\text{BMR} = 655 + (9.6 \times \text{weight in kilos}) + (1.8 \times \text{height in cm}) - (4.7 \times \text{age in years})$

**Men:**  $\text{BMR} = 66 + (13.7 \times \text{weight in kilos}) + (5 \times \text{height in cm}) - (6.8 \times \text{age in years})$

Once you know your BMR, you can calculate your [Daily Calorie Needs](#) based on your activity level using the [Harris Benedict Equation](#).

### Resources

[Harris Benedict Equation](#)

[Recommended Daily Allowance](#)

[Underweight Treatment](#)

[Overweight Treatment](#)

[Obesity Treatment](#)

### Calculators

[BMI Calculator](#)

[BMR Calculator](#)

[Body Fat Calculator](#)

[Waist to Hip Ratio Calculator](#)

Basal Metabolic Rate  
(BMR)  
<http://www.bmi-calculator.net/bmr-calculator/bmr-formula.php>

---



# Lab

---

BMR.JAVA/TESTBMR.JAVA



# Lab Project:

**BMR.java/TestBMR.java**

---

- Write a BMR class to calculate BMR (Basal Metabolic Rate) number for men and women of different gender, age, weight, and height.
- Need to make this class cohesive, complete and following design convention of the class design guidelines.
- Data fields: name, gender, age, height, weight and their accessors/mutators. And, the derived BMR/rounded BMR methods (using the formula in previous page)  
(Use BMI class as an example)

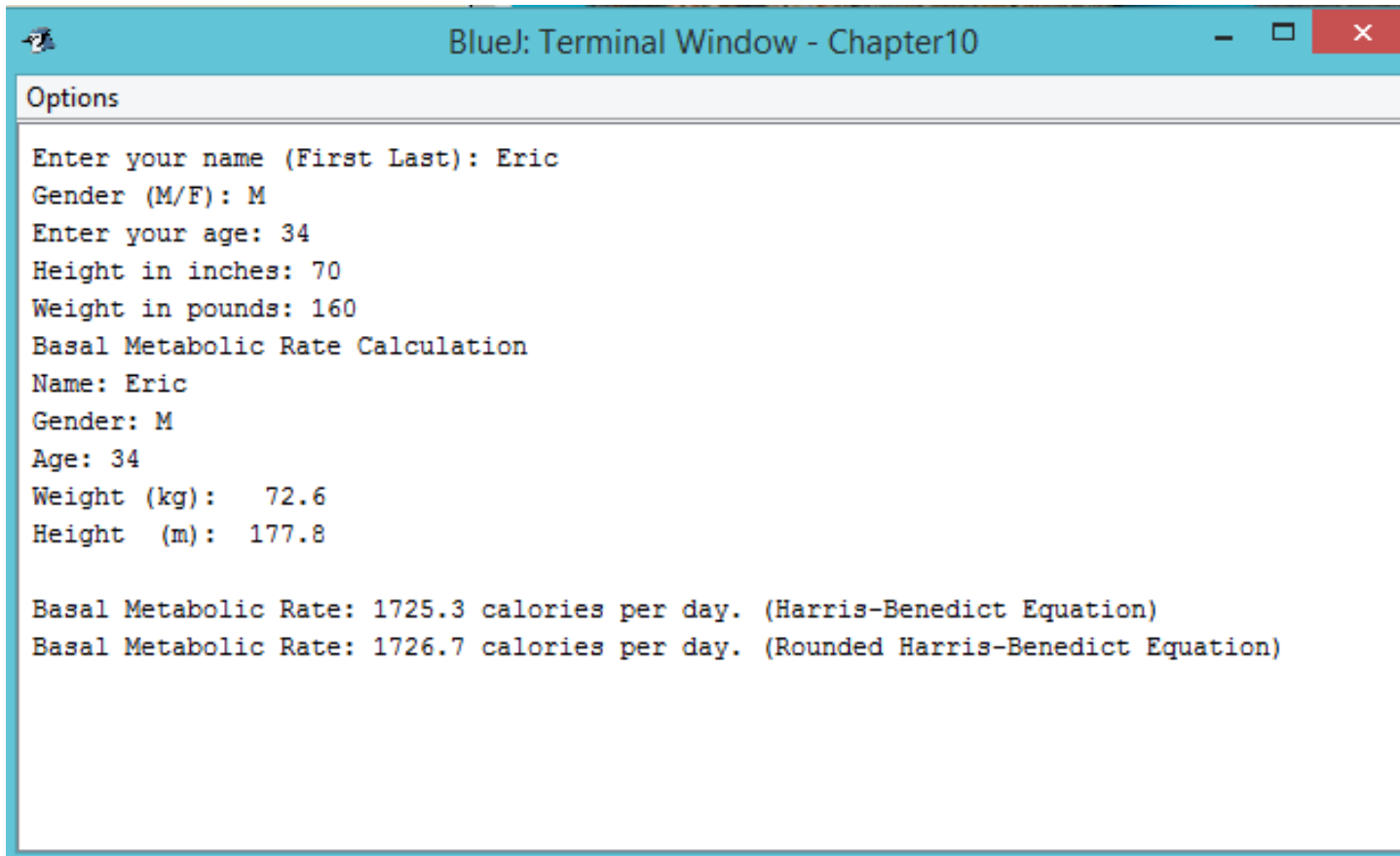


# TestBMR.java class

---

- Read in data from a client about his name, gender, age, height and weight. Create a BMR object and store these read-in data into the object.
- Then, use the BMR instance methods to get name, gender, age, height and weight and then, print these data out.
- At the end, print out the BMR and rounded BMR number (rounded to 1 decimal place).





```
Options
Enter your name (First Last): Eric
Gender (M/F): M
Enter your age: 34
Height in inches: 70
Weight in pounds: 160
Basal Metabolic Rate Calculation
Name: Eric
Gender: M
Age: 34
Weight (kg): 72.6
Height (m): 177.8

Basal Metabolic Rate: 1725.3 calories per day. (Harris-Benedict Equation)
Basal Metabolic Rate: 1726.7 calories per day. (Rounded Harris-Benedict Equation)
```

Expected  
Results:  
(BMR.java  
and  
TestBMR.java)

---



# Demo:

Conversion of Structural  
Program to Object-Oriented  
Program

---

LECTURE 6

## BMI Calculator » BMR Formula

If you are unable to use our [BMR Calculator](#), or if you are interested in how BMR is calculated, this page has the mathematical **BMR Formulas**.

The **BMR formula** uses the variables of height, weight, age and gender to calculate the Basal Metabolic Rate (BMR). This is more accurate than calculating calorie needs based on body weight alone. The only factor it omits is lean body mass and thus the ratio of muscle-to-fat a body has. Remember, leaner bodies need more calories than less leaner ones. Therefore, this equation will be very accurate in all but the very muscular (will underestimate calorie needs) and the very fat (will overestimate calorie needs).

### English BMR Formula

**Women:**  $\text{BMR} = 655 + (4.35 \times \text{weight in pounds}) + (4.7 \times \text{height in inches}) - (4.7 \times \text{age in years})$

**Men:**  $\text{BMR} = 66 + (6.23 \times \text{weight in pounds}) + (12.7 \times \text{height in inches}) - (6.8 \times \text{age in year})$

### Metric BMR Formula

**Women:**  $\text{BMR} = 655 + (9.6 \times \text{weight in kilos}) + (1.8 \times \text{height in cm}) - (4.7 \times \text{age in years})$

**Men:**  $\text{BMR} = 66 + (13.7 \times \text{weight in kilos}) + (5 \times \text{height in cm}) - (6.8 \times \text{age in years})$

Once you know your BMR, you can calculate your [Daily Calorie Needs](#) based on your activity level using the [Harris Benedict Equation](#).

### Resources

[Harris Benedict Equation](#)  
[Recommended Daily Allowance](#)  
[Underweight Treatment](#)  
[Overweight Treatment](#)  
[Obesity Treatment](#)

### Calculators

[BMI Calculator](#)  
[BMR Calculator](#)  
[Body Fat Calculator](#)  
[Waist to Hip Ratio Calculator](#)

Basal Metabolic Rate  
(BMR)  
<http://www.bmi-calculator.net/bmr-calculator/bmr-formula.php>

---



# Steps to convert Structural programs to Object-oriented programs.

---

Step 1: Separate Data from Testing Fixture. (copy the static main method to TestBMR.java file)

(copy the data variables to the BMR.java's data field location)

Step 2: Change the variable to private for data encapsulation

Change static method to instance method in BMR class.

Add accessor/Mutator to BMR class.

Modify accessors for generating derived variables

Step 3: connect the BMR object to Testing fixture.

Step 4: use instance methods to access instance variables.



# Demo: Conversion from a Structural Program to an Object-oriented Program

([BMR.java/TestBMR.java](#))

---

- I used one of my own existing program to create the lab project sample answer.
- In this lecture, I documented down the whole process to show you how to convert a structural program to an object-oriented program step by step.



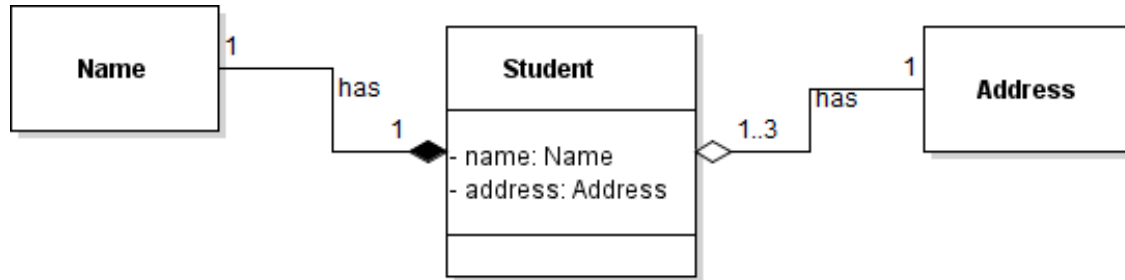
# Class Use Relationship

---

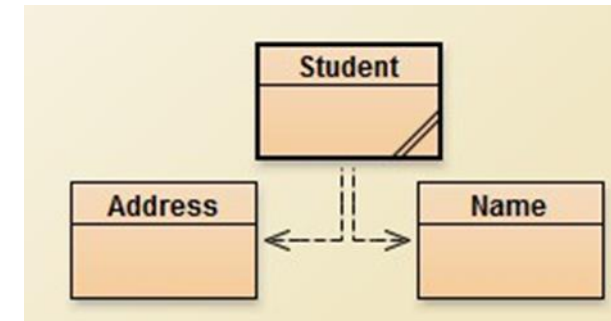
LECTURE 7

# has A Relationship

Violet



BlueJ



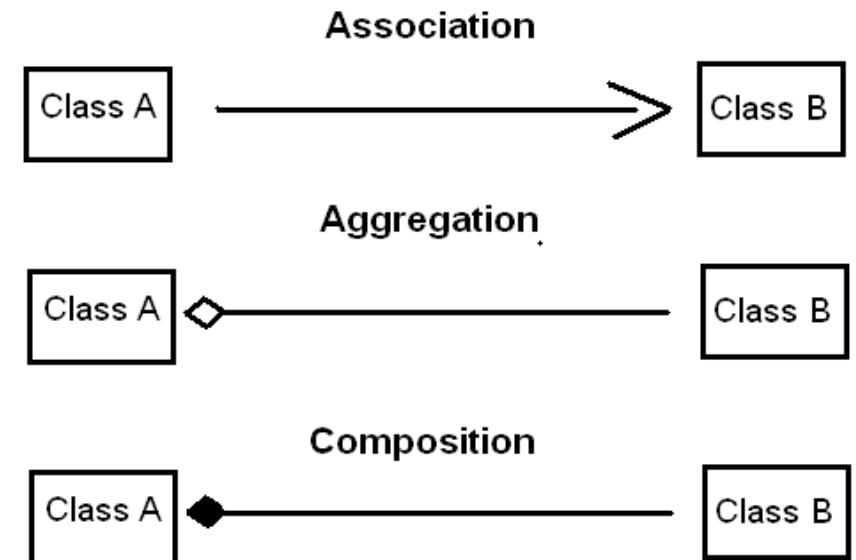


# Class Relationships (use and inherit)

To design classes, you need to explore the relationships among classes. The common relationships among classes are *association*, *composition*, and *inheritance* (later)

**These three relationship is use-relationship in BlueJ.**

- Association is general case. [many(one)-to-many(one)]
- Aggregation is has-a relationship. (many-to-one/one-to-one)
- Composition is exclusive has-a relation. (one-to-one)







# Association

Association is a general binary relationship that describes an activity between two classes.

*Association* is a general binary relationship that describes an activity between two classes. For example, a student taking a course is an association between the **Student** class and the **Course** class, and a faculty member teaching a course is an association between the **Faculty** class and the **Course** class.



An association is illustrated by a **solid line** between two classes with an optional label that describes the relationship. (Sometimes an **arrow line** is used.) In Figure above, the labels are **Take** and **Teach**. Each relationship may have an optional small black triangle that indicates the direction of the relationship. In this figure, the direction indicates that a student takes a course (as opposed to a course taking a student).



# Multiplicity

placed at the side of the class to specify how many of the class's objects are involved in the relationship in UML.

---

- A multiplicity could be a number or an interval that specifies how many of the class's objects are involved in the relationship.
- The character **\*** means an unlimited number of objects, and
- the interval **m..n** indicates that the number of objects is between **m** and **n**, inclusively.
- In Figure of previous page, each student may take any number of courses, and each course must have **at least five and at most sixty** students. Each course is taught by only one faculty member, and a faculty member may teach **from zero to three** courses per semester.

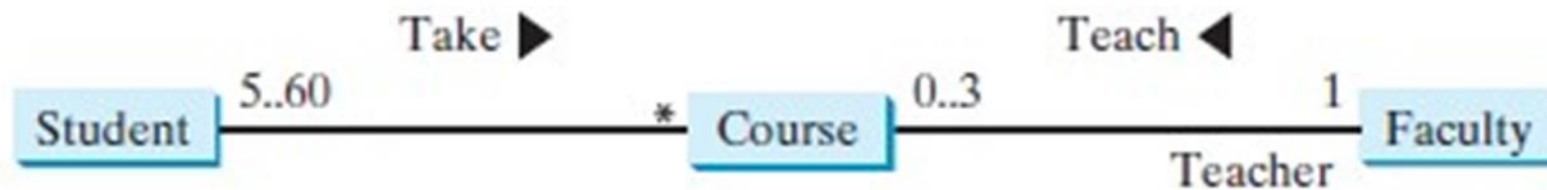


# Association Relationship

```
public class Student {  
    private Course[]  
        courseList;  
  
    public void addCourse(  
        Course s) { ... }  
}
```

```
public class Course {  
    private Student[]  
        classList;  
    private Faculty faculty;  
  
    public void addStudent(  
        Student s) { ... }  
  
    public void setFaculty(  
        Faculty faculty) { ... }  
}
```

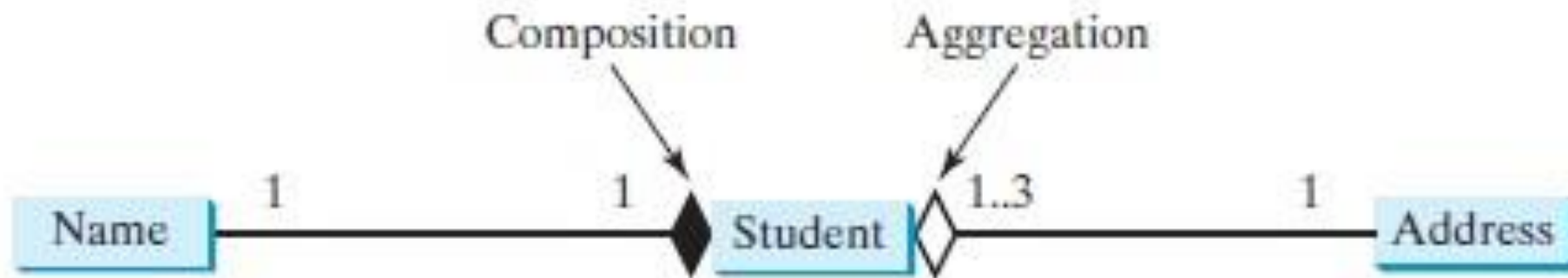
```
public class Faculty {  
    private Course[]  
        courseList;  
  
    public void addCourse(  
        Course c) { ... }  
}
```





# Aggregation or Composition

Since aggregation and composition relationships are represented using classes in similar ways, many texts don't differentiate them and call both compositions.

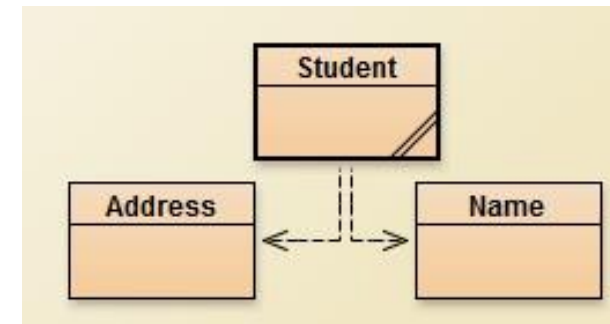


Student has a Name exclusively.  
(Composition)

Student has an address non-exclusively.  
(Aggregation)



# Class Representation



An aggregation relationship is usually represented as a data field in the aggregating class. For example, the relationship in Figure of previous slide can be represented as follows:

```
public class Name {
    ...
}
```

Aggregated class

```
public class Student {
    private Name name;
    private Address address;
    ...
}
```

Aggregating class

```
public class Address {
    ...
}
```

Aggregated class

Aggregating: Using

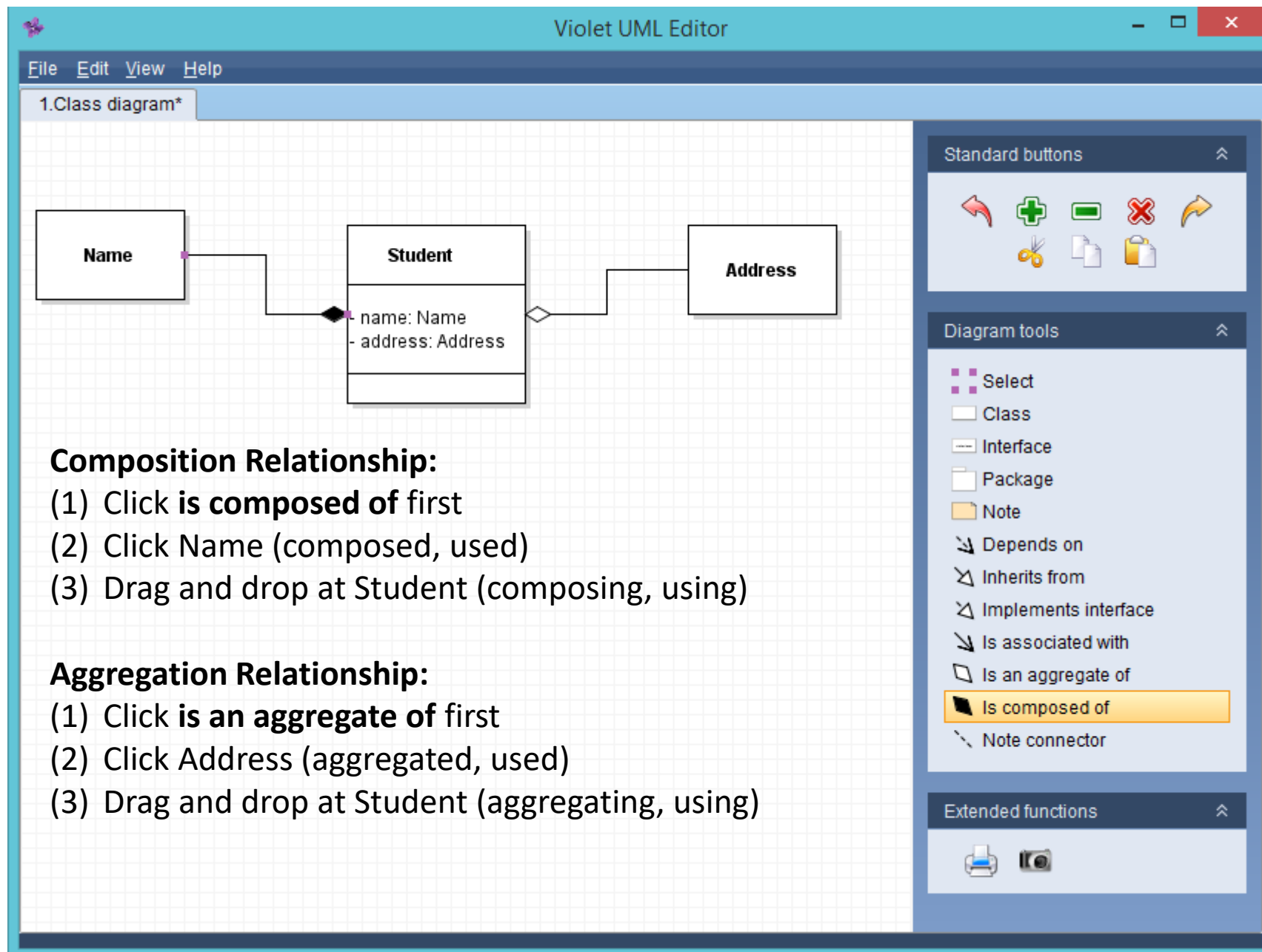
Aggregated: Used



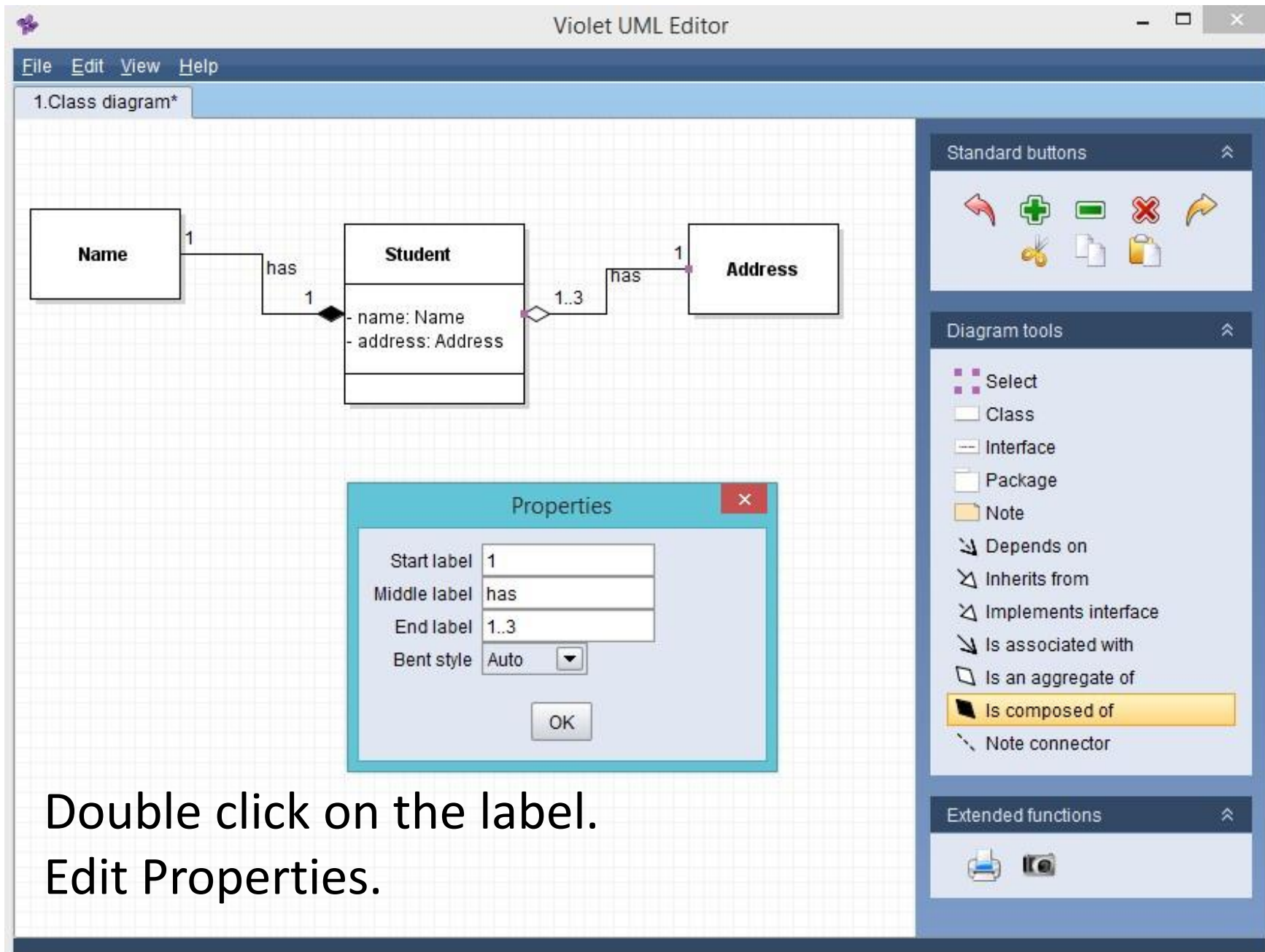
# Aggregation (has-a Relationship)

---

- **Aggregation** is a special form of association that represents an ownership relationship between two objects. Aggregation models **has-a** relationships. The owner object is called an **aggregating object**, and its class is called an **aggregating class**. The subject object is called an **aggregated object**, and its class is called an **aggregated class**.
- An object can be owned by several other **aggregating objects**.
- If an object is exclusively owned by an aggregating object, the relationship between the object and its aggregating object is referred to as a **composition**.







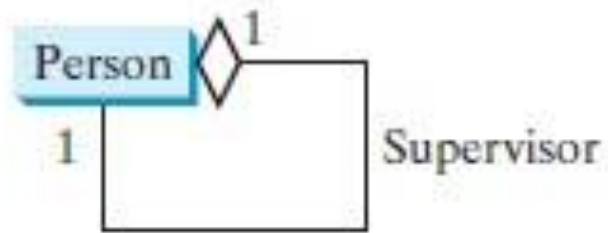
Double click on the label.  
Edit Properties.





# Aggregation Between Same Class

Aggregation may exist between objects of the same class. For example, a person may have a supervisor. **(Using itself once)**



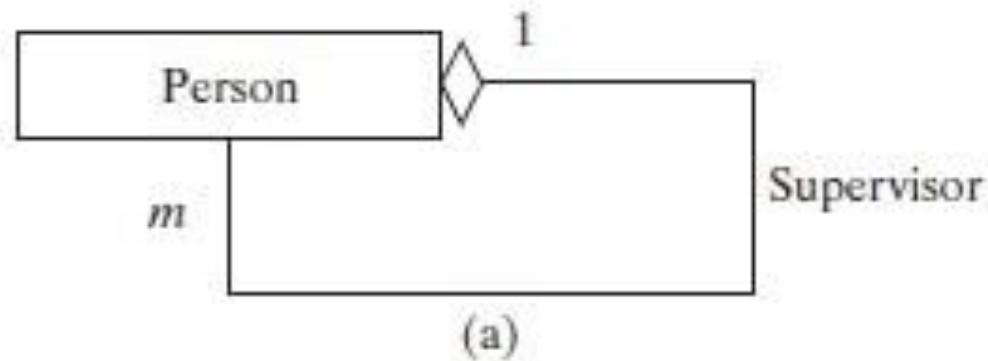
```
public class Person {  
    // The type for the data is the class itself  
    private Person supervisor;  
    ...  
}
```



# Aggregation Between Same Class

What happens if a person has several supervisors?

**(One class using itself many times)**



```
public class Person {  
    ...  
    private Person[] supervisors;  
}
```

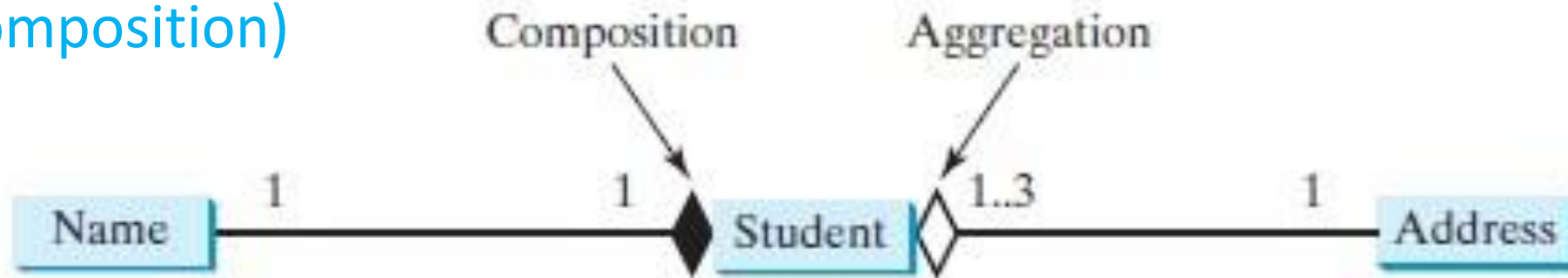
(b)



# Object Composition

Composition is actually a special case of the aggregation relationship. Aggregation models *has-a* relationships and represents an ownership relationship between two objects. The owner object is called an *aggregating object* and its class an *aggregating class*. The subject object is called an *aggregated object* and its class an *aggregated class*.

Student has a Name exclusively.  
(Composition)



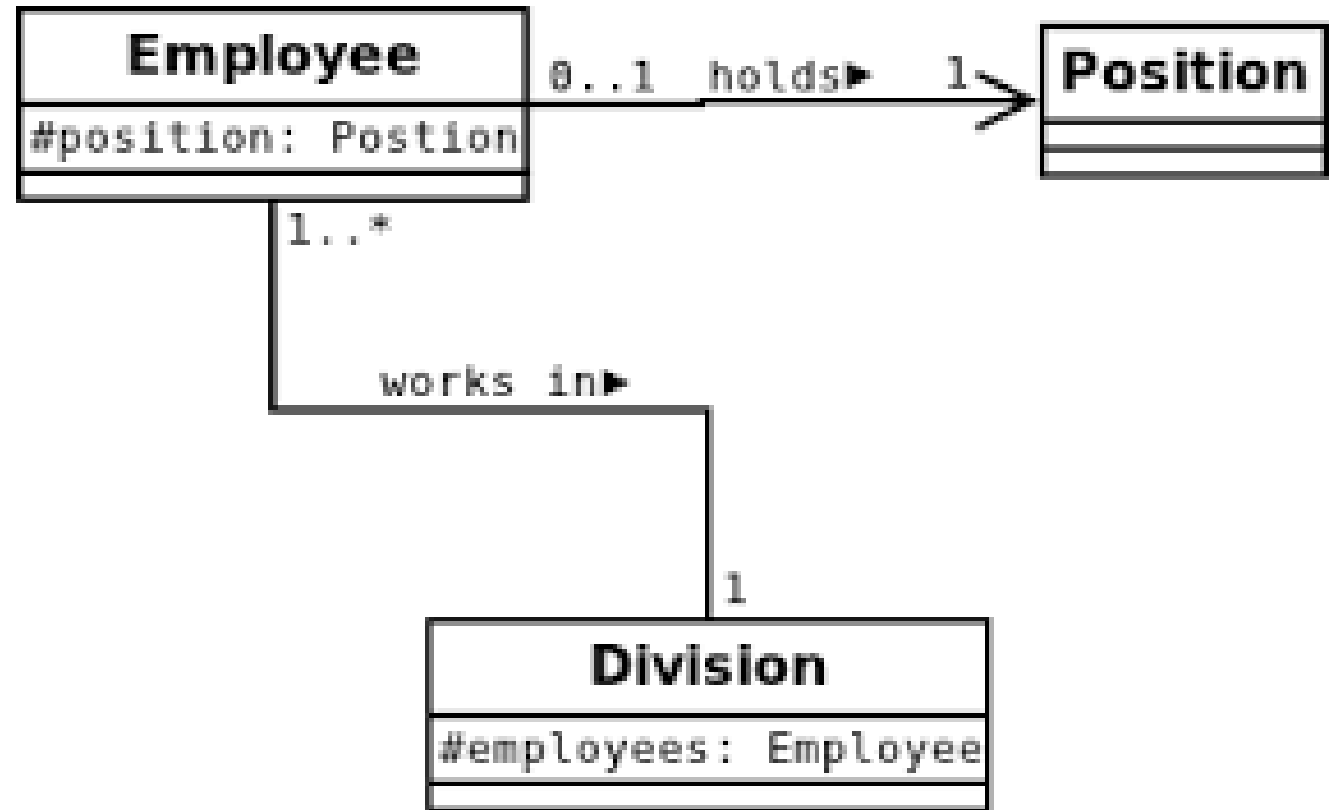


# Why analyze the multiplicity of relationship?

<http://code.tutsplus.com/articles/sql-for-beginners-part-3-database-relationships--net-8561>

Widely used in database design.  
When inner-join, outer-join are to be performed, analysis of these relationship is very essential.

It will be very important for data science.





# Assignment

---

GEOMETRIC 2 ASSIGNMENT

SUBMIT YOUR PROGRAM TO  
MOODLE COURSE UPLOAD LINK



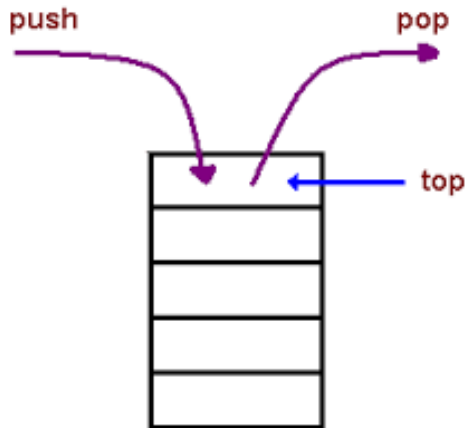
# Lab Project: Stack Of Integers

---

LECTURE 8



# Data Structure: Stack (Last-In-First-Out)

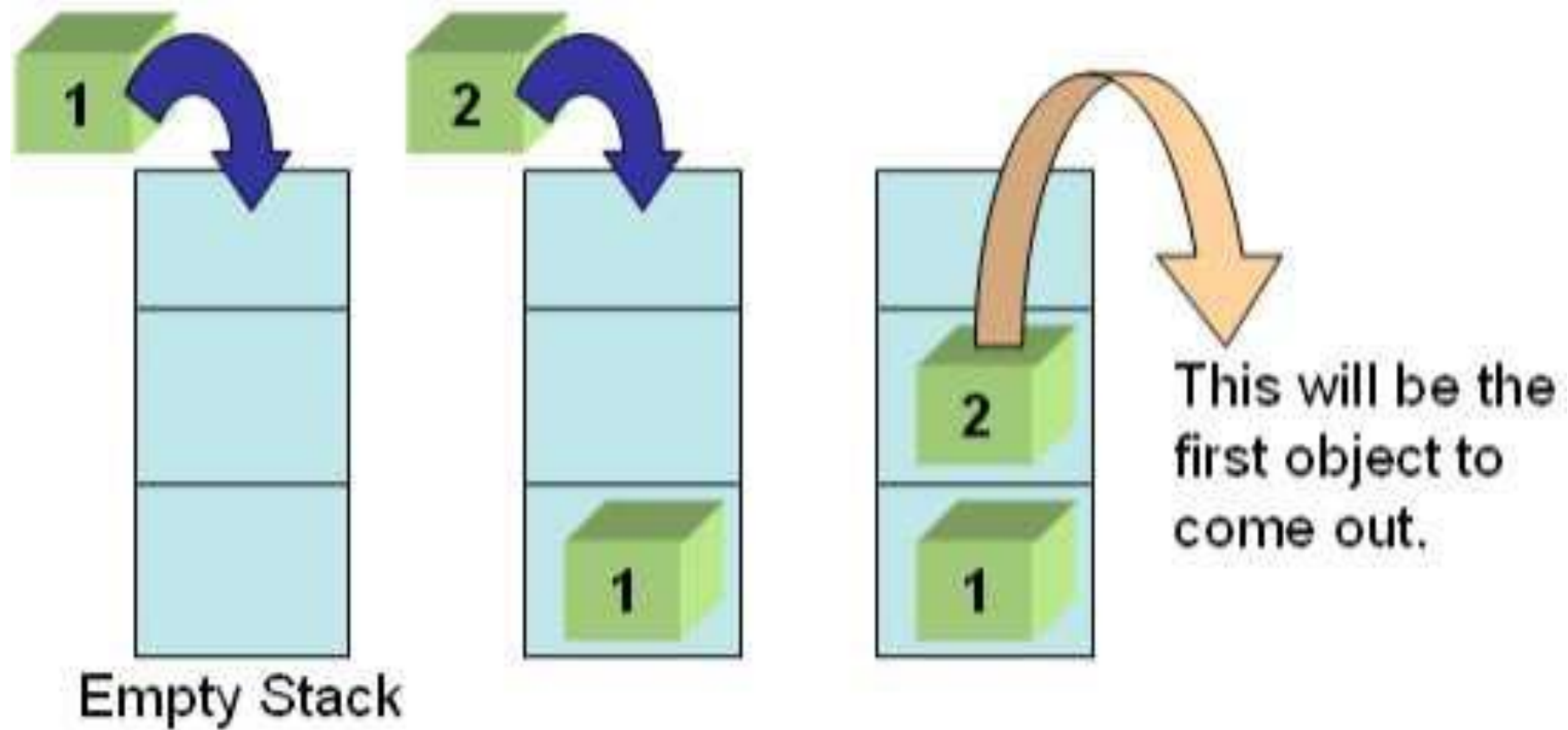


A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: **push** the item into the stack, and **pop** the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. **push** adds an item to the top of the stack, **pop** removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top. A stack is a **recursive** data structure. Here is a structural definition of a Stack:

**a stack is either empty or  
it consists of a top and the rest which is a stack;**



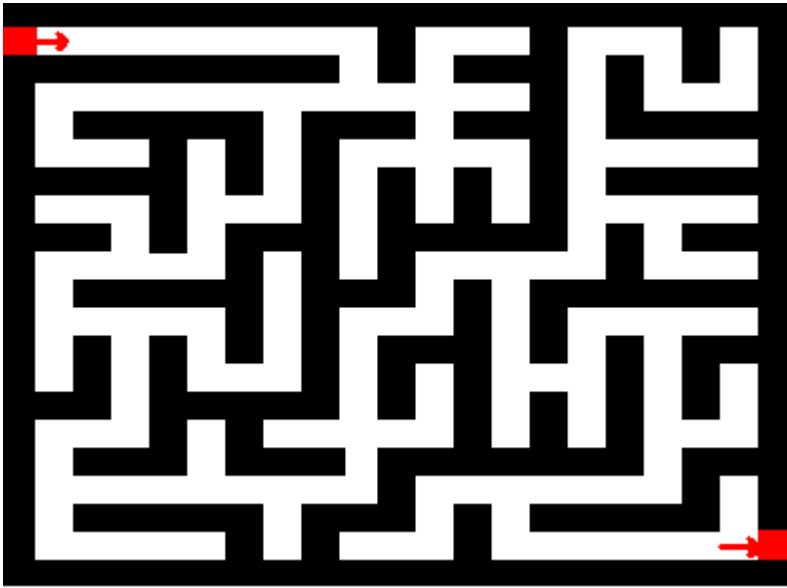
# How Stacks Works?







# Application of Stacks



**Reverse:** The simplest application of a stack is to **reverse** a word. You push a given word to stack - letter by letter - and then pop letters from the stack.

**Undo:** Another application is an "**undo**" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack.

**Backtracking:** This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze - how do you find a way from an entrance to an exit?

Once you reach a dead end, you must backtrack. But backtrack to where? to the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.

## Language processing:

- space for parameters and local variables is created internally using a stack.
- compiler's syntax check for matching braces is implemented by using stack.
- support for recursion. (and **call-stack**)



# The StackOfIntegers Class

## Implementation by Array

StackOfIntegers	
-elements: int[]	
-size: int	
<hr/>	
+StackOfIntegers()	
+StackOfIntegers(capacity: int)	
+empty(): boolean	
+peek(): int	
+push(value: int): int	
+pop(): int	
+getSize(): int	

An array to store integers in the stack.

The number of integers in the stack.

Constructs an empty stack with a default capacity of 16.

Constructs an empty stack with a specified capacity.

Returns true if the stack is empty.

Returns the integer at the top of the stack without removing it from the stack.

Stores an integer into the top of the stack.

Removes the integer at the top of the stack and returns it.

Returns the number of elements in the stack.



# Pseudo Code

---

## Data Fields:

int[] elements;  
int size;

## Constructors:

StackOfInteger():

build an array of **16** elements and assign the reference to elements.

set top to 0 (as empty)

StackOfInteger(int **capacity**):

build an array of **capacity** elements and assign the reference to elements.

set top to 0 (as empty)

## Methods:

**empty()**: return true if size==0, otherwise false

**peek()**:

if size != 0, return elements[size-1];

else print out message ("Stack is empty");

**push(int value)**:

if size>=capacity

{ expand the array size by 2.

copy the old array to the first half.

}

elements[size++] = value;

**pop()**:

if size!=0

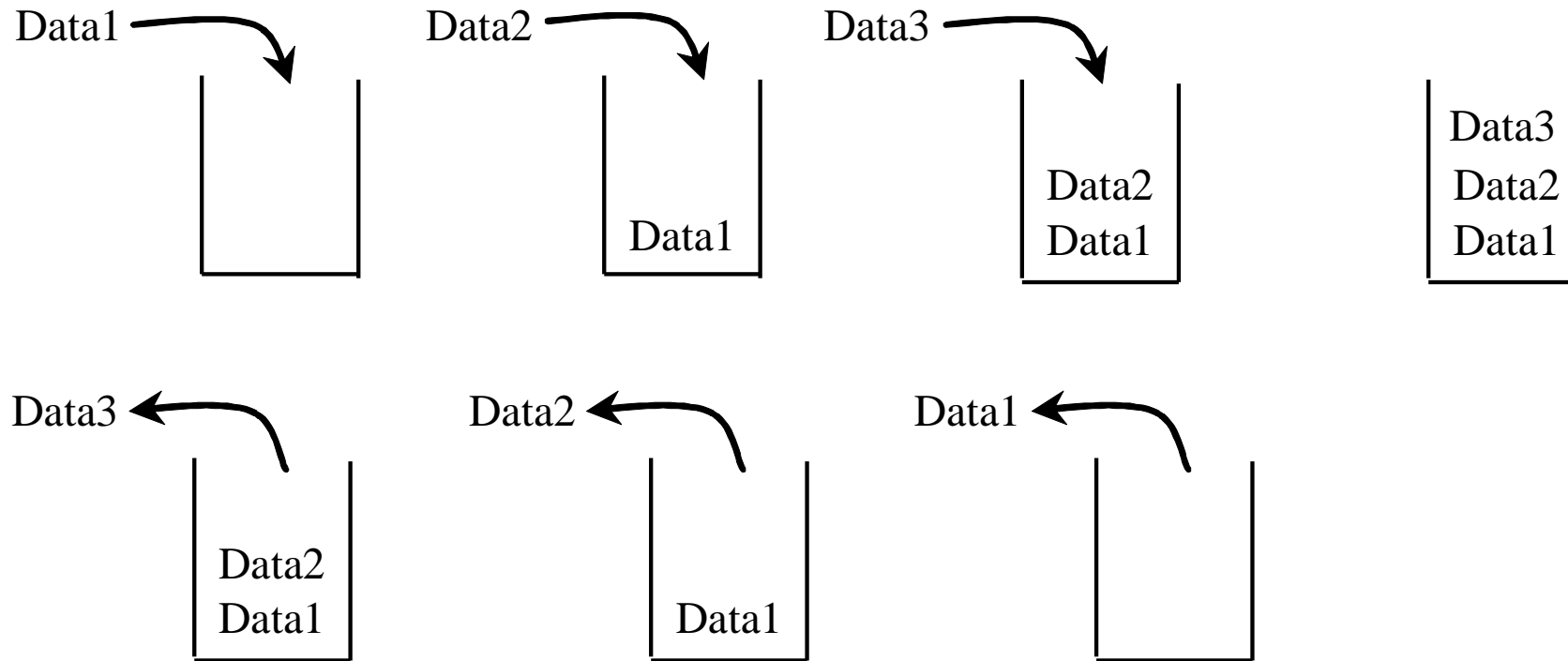
return elements[--size];

**getSize()**:

return size;

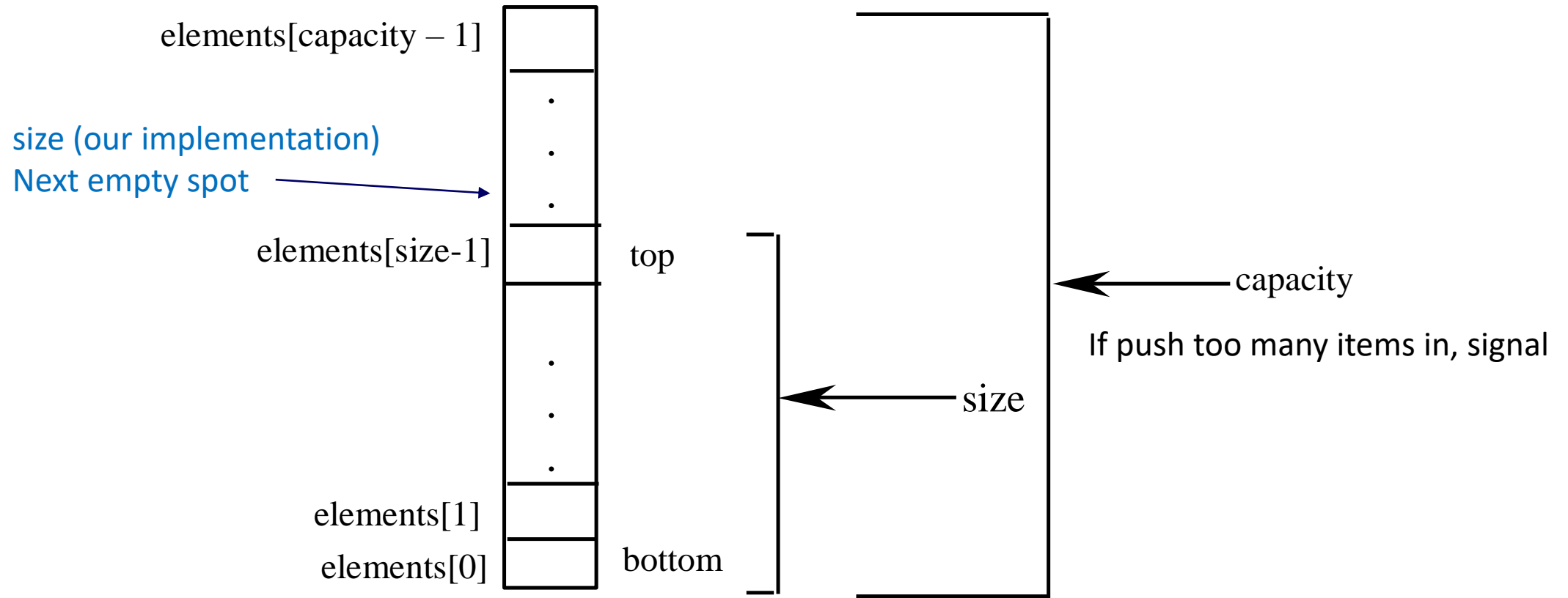


# Designing the StackOfIntegers Class





# Implementing StackOfIntegers Class



Demo Program:  
StackOfInteger.java  
TestStackOfInteger.java

Write

Write a program to implement the StackOfInteger class following the UML and the pseudo code.

Please

For the Test Class, please use TestStackOfInteger.java directly.

Do not  
refer

Please do not refer to StackOfInteger.java before you finish your own version.



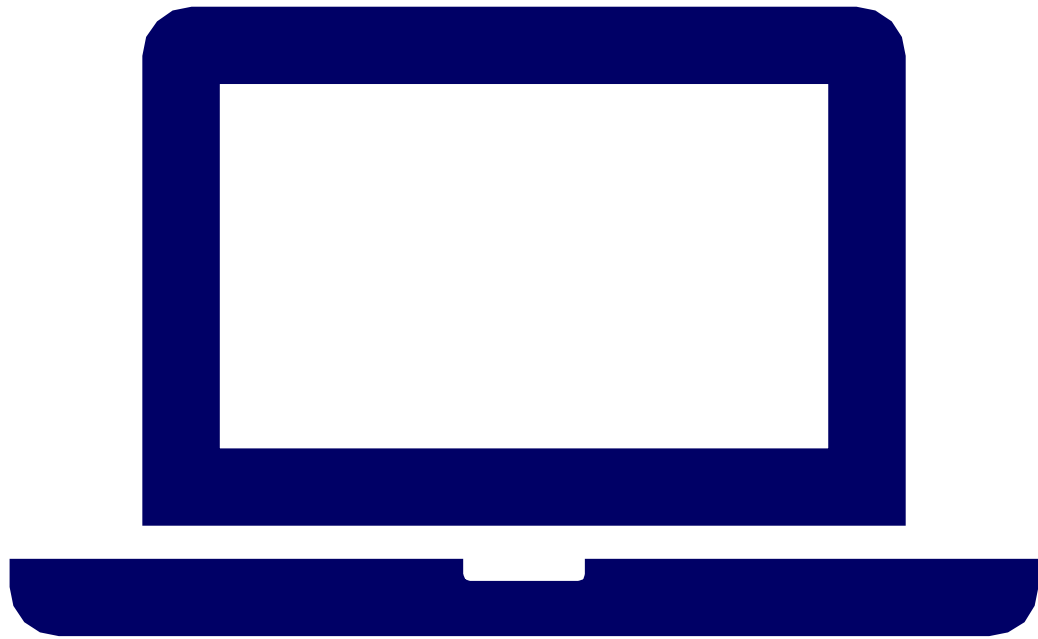
# TestStackOfIntegers.java

---

```
for (int i = 0; i < 10; i++) stack.push(i); // push 10 elements
while (!stack.empty()) System.out.print(stack.pop()+" "); // pop all of them out.
System.out.println();
```

```
System.out.println("Check Underflow Condition");// check underflow condition
int a = stack.pop();
System.out.println();
```

```
System.out.println("Check Overflow Condition"); // check overflow condition.
for (int i=0; i<18; i++){
    stack.push(i*2);
}
for (int i=0; i<18; i++){
    System.out.print(stack.pop() + " ");
}
System.out.println();
```

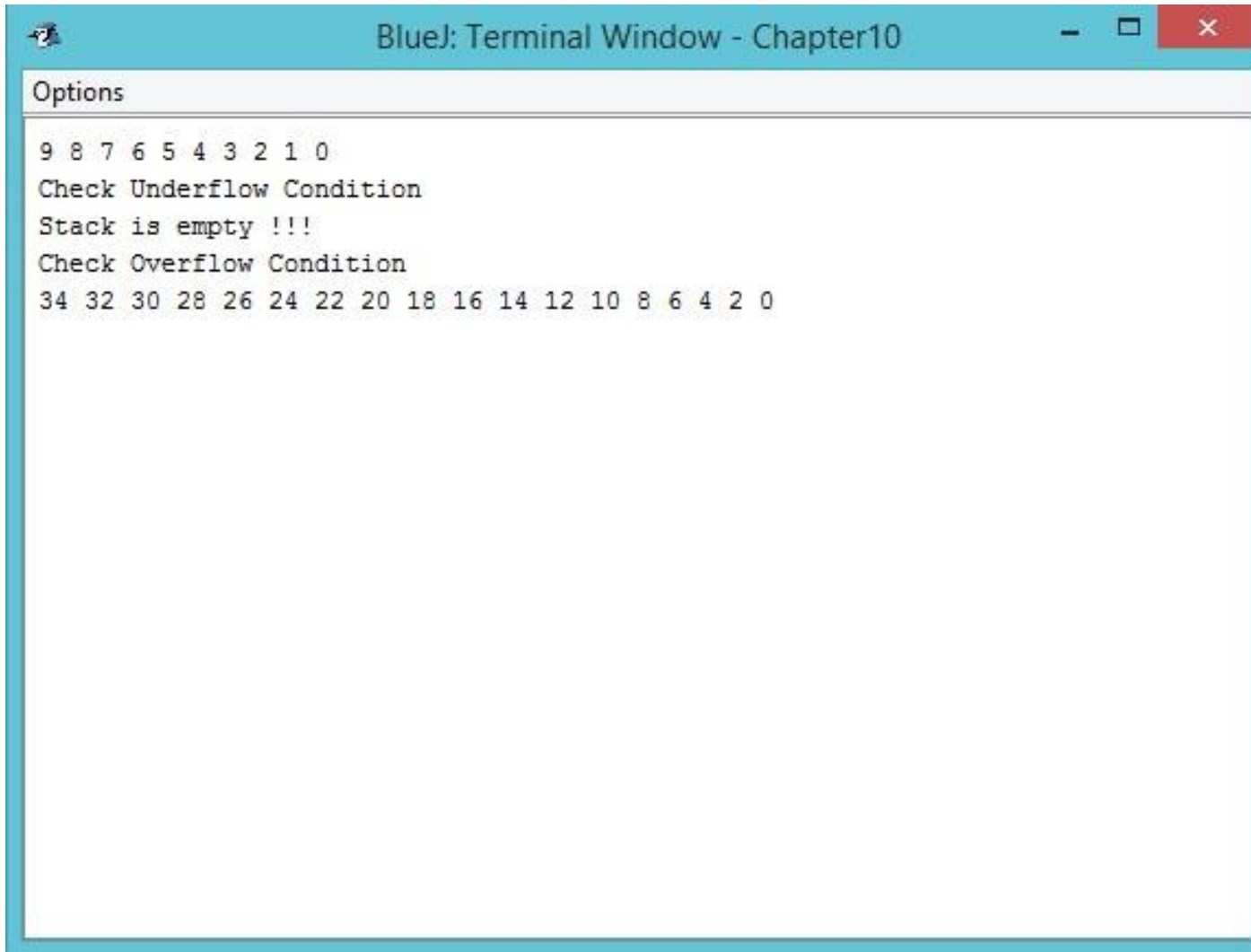


# Demonstration Program

---

STACKOFINTEGER.JAVA  
TESTSTACKOFINTEGER.JAVA

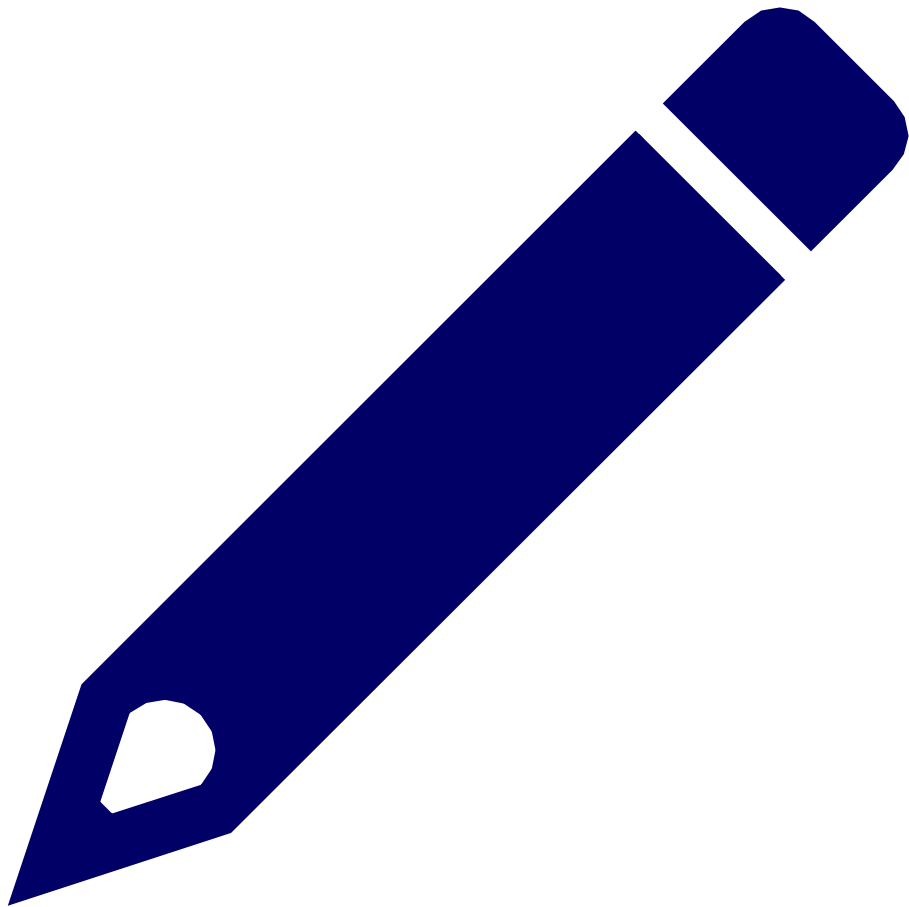




```
Options
9 8 7 6 5 4 3 2 1 0
Check Underflow Condition
Stack is empty !!!
Check Overflow Condition
34 32 30 28 26 24 22 20 18 16 14 12 10 8 6 4 2 0
```

# Expected Results:

---



# Assignment

---

GEOMETRIC 3 ASSIGNMENT

SUBMIT YOUR PROGRAM TO  
MOODLE COURSE UPLOAD LINK