

AP Computer Science A

Java Programming Essentials [Ver.4.0]

Unit 3: Class Creation

CHAPTER 12: CLASS DESIGN AND
CLASS TO CLASS RELATIONSHIP

DR. ERIC CHOU
IEEE SENIOR MEMBER



AP Computer Science Curriculum

- Instance Methods: Passing and Returning References of an Object (T 3.6)
- Class Variable and Methods (Statics) (T 3.7)
- this Keyword (T 3.9)

Objectives:

- Class Design I, II, III, IV
- Data Encapsulation
- Immutable Classes
- Instance Methods: Passing and Returning References of an Object
- this Reference
- Class Loading and Execution Sequence
- Object-Oriented Thinking

Objectives:

- How to design a good Class
- Class Completeness
- Is_A and Has_A relationship (Non-AP topic)
- super keyword (Non-AP Topic)
- Object Class
- Immutable Classes (Big-Integer/BigDecimal, Non-AP Topic)

Object-Oriented Programming

Package

Module

Classes

Interface
s

Abstract
Classes

enum

Statics

Objects

Function
s

Contain
er

Constant
s

Access
Modifiers

Visibility

public

protected

default

private

Encapsulat
ion

Information
Hiding

Wrapper
Classes

Immutabl
e

Relations

has_A
Compositio
n

Many to 1
Aggregation

Many to Many
Association

Coherenc
e

Inheritance

Is_A
Inheritance

this

super

Multiple
Inherita
nce

Polymorphism

Overloading

Overriding

Dynamic
Binding

Polymorphi
c Methods

Generics

Generic
Contain
er

Generic
Method

Object
Generic



Classes and Objects (1): Static Members

Lecture 1

Classes and Objects Design Styles

Classification by Styles

- **All Static Classes:** *Classes and Objects (1)*
- (1) Utility Class: Math, RandomCharacter (Chapter 6))
- (2) Tester Class: TestCircleWithStaticMembers.java (This Lecture)
- (3) Structural Programming: This program design style is equivalent to non-object-oriented programming (*Reduction to Structural Programming*).

Classes and Objects Design Styles

(Classification by Styles)

Data Encapsulation: All Private Data Fields and Public Accessors and Mutators. (**data hiding, information hiding**: first Object-Oriented feature discussed so far.)

Classes and Objects (2)

How to Enter Data Capsules (Accessing Objects):

(1) Using Public Mutators

(2) Passing Objects to Methods.

Classes and Objects (3)

Classes and Objects Design Styles

Classification by Styles

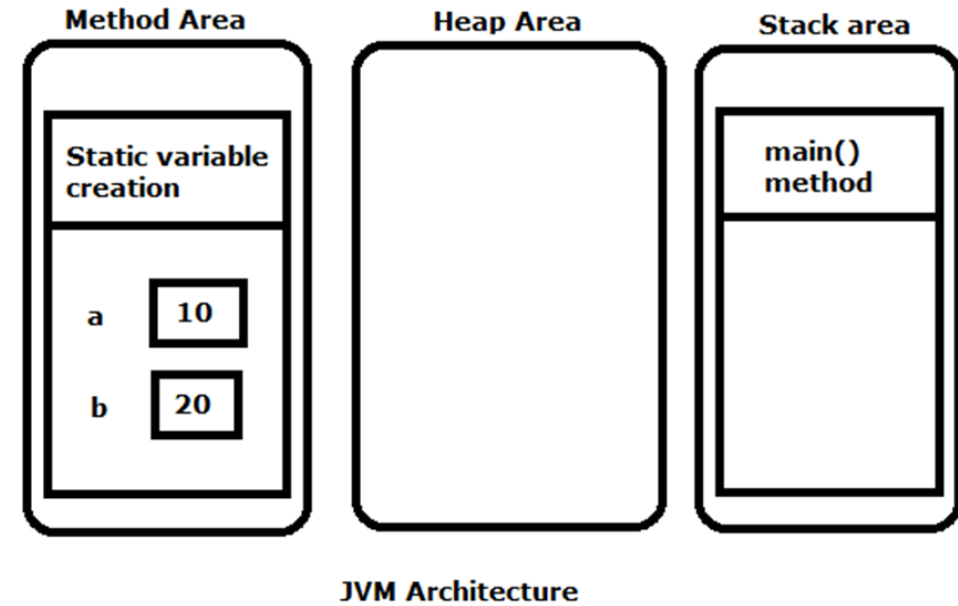
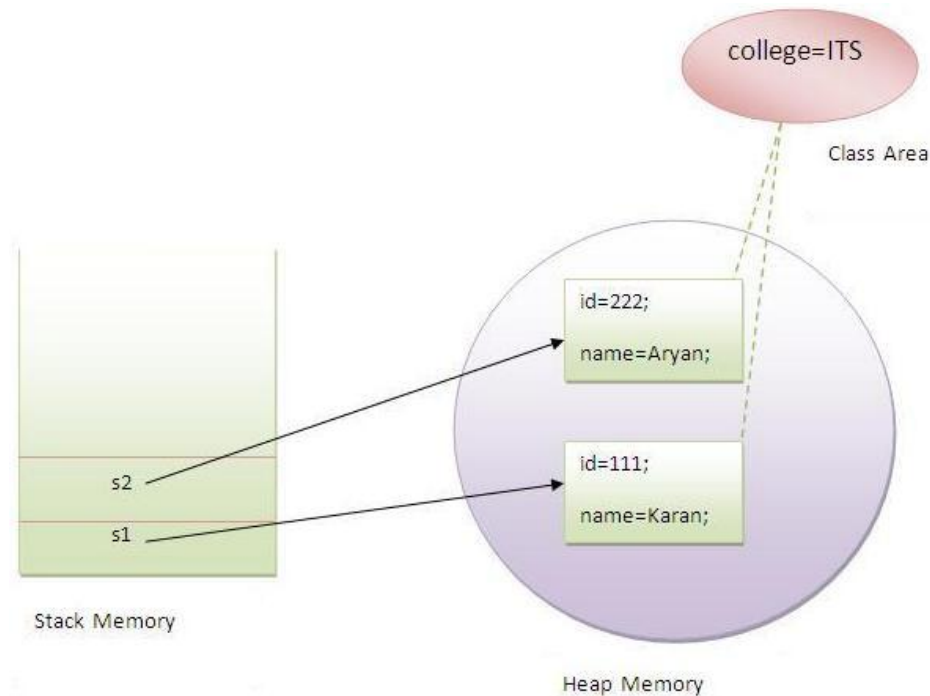
Immutable Classes (Object): All Private Data Fields, No Mutators and No Accessor Method Returning Reference Data.

Classes and Objects (4)

Visibility Modifiers

	public	private
Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the class

Memory Allocation for Class Variable and Instance Variable



Types of Static Members:

- Java supports four types of static members
 - Static Variables
 - Static Blocks
 - Static Methods
 - Main Method (static method)
- All static members are identified and get memory location at the time of class loading by default by JVM in **Method area**.
- Only static variables get memory location, **methods** will **not** have separate memory location like variables. (Their variables will be loaded to call-stack when the code is called at runtime.)
- Static Methods are just identified and can be accessed directly without object creation.

Life time and scope

- Static variable get life as soon as class is loaded into JVM and is available till class is removed from JVM or JVM is shutdown.
- And its scope is class scope means it is accessible throughout the class.

UML Design for Class Header

Easy Description of Classes

Table 4: Marks for UML-supported visibility types

Mark	Visibility type
+	Public
#	Protected
-	Private
~	Package

UML Design for Class Header

Easy Description of Classes

CircleWithStaticMethods	CircleWithStaticMethods
Properties	Properties
~double radius ~static int numberOfObjects	~radius: double ~numberOfObjects: static int
Constructors	Constructors
~CircleWithStaticMembers() ~CircleWithStaticMemembrs(double newRadius)	~CircleWithStaticMembers() ~CircleWithStaticMemembrs(double newRadius)
Methods	Methods
~double getArea() + static int getNumberOfObjects()	~getArea(): double + getNumberOfObjects(): static double

CircleWithStaticMembers

Class Header Only (Body Omitted)...

```
public class CircleWithStaticMembers {  
    double radius;  
    static int numberOfObjects = 0;  
    CircleWithStaticMembers() {...}  
    CircleWithStaticMembers(double newRadius) {...}  
    static int getNumberOfObjects() {...}  
    double getArea() {...}  
}  
// keep track of number of circles created.
```

Static Members (Class Members)
Properties
~static int numberOfObjects
Methods
+ static int getNumberOfObjects()
Non-static Members (Instance Members)
Properties
~double radius
Constructors
~CircleWithStaticMembers() ~CircleWithStaticMemembrs(dou ble newRadius)
Methods
~double getArea()

Class Variable Update

ClassName.variable, ClassName.update()

```
CircleWithStaticMembers() {  
    radius = 1.0;  
    numberOfObjects++;  
}  
/** Construct a circle with a specified radius */  
CircleWithStaticMembers(double newRadius) {  
    radius = newRadius;  
    numberOfObjects++;  
}  
/** Return numberOfObjects */  
static int getNumberOfObjects() {  
    return numberOfObjects;  
}
```

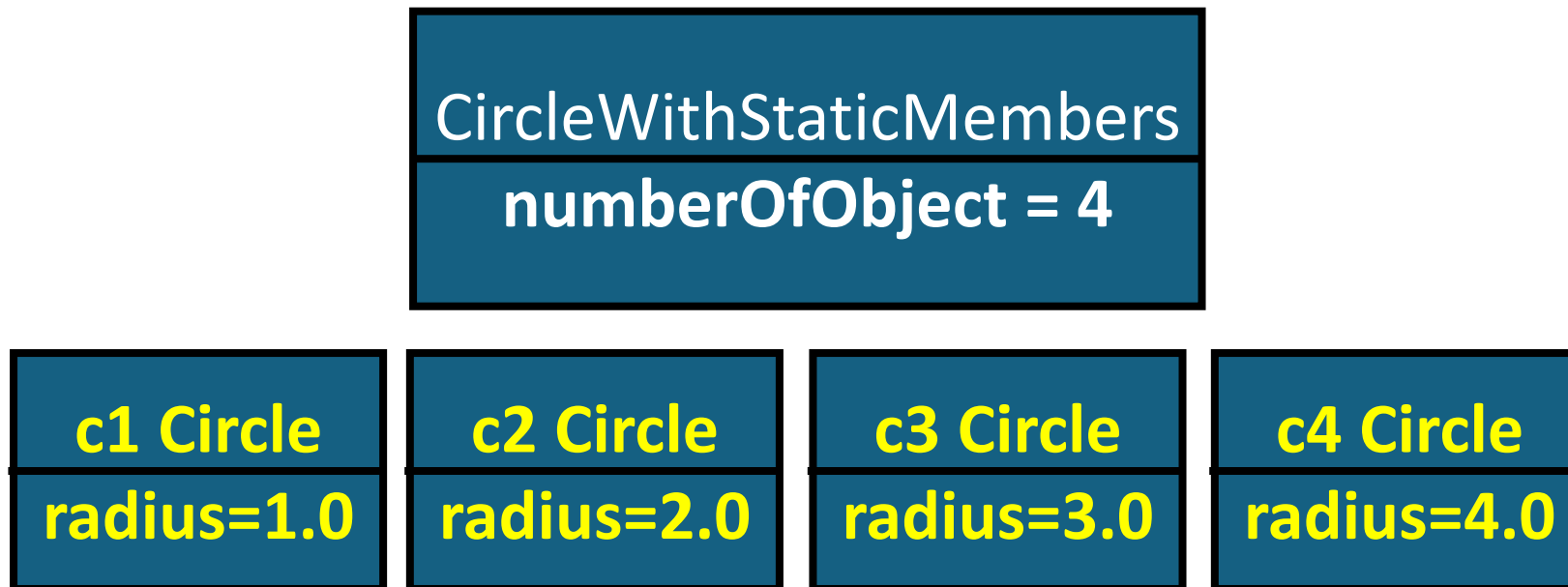

Class Variable Update

ClassName.variable, ClassName.update()

Static Variables/Methods Stay With Class:

Available when JVM load the class.

(Instance variable stays in heap temporarily)





Options

Before creating objects

The number of Circle objects is 0

After creating c1

c1: radius (1.0) and number of Circle objects (1)

After creating c2 and modifying c1

c1: radius (9.0) and number of Circle objects (2)

c2: radius (5.0) and number of Circle objects (2)

CircleWithStaticMembers.java.
TestCircleWithStaticMembers.java



Demonstration Program

CIRCLEWITHSTATICMEMBERS.JAVA
TESTCIRCLEWITHSTATICMEMBERS.J
AVA



Assignment

FUEL 1 ASSIGNMENT

SUBMIT YOUR PROGRAM TO
MOODLE COURSE UPLOAD LINK



Classes and Objects (2) :

Data Encapsulation I
Data Abstraction

Lecture 2

What is Encapsulation

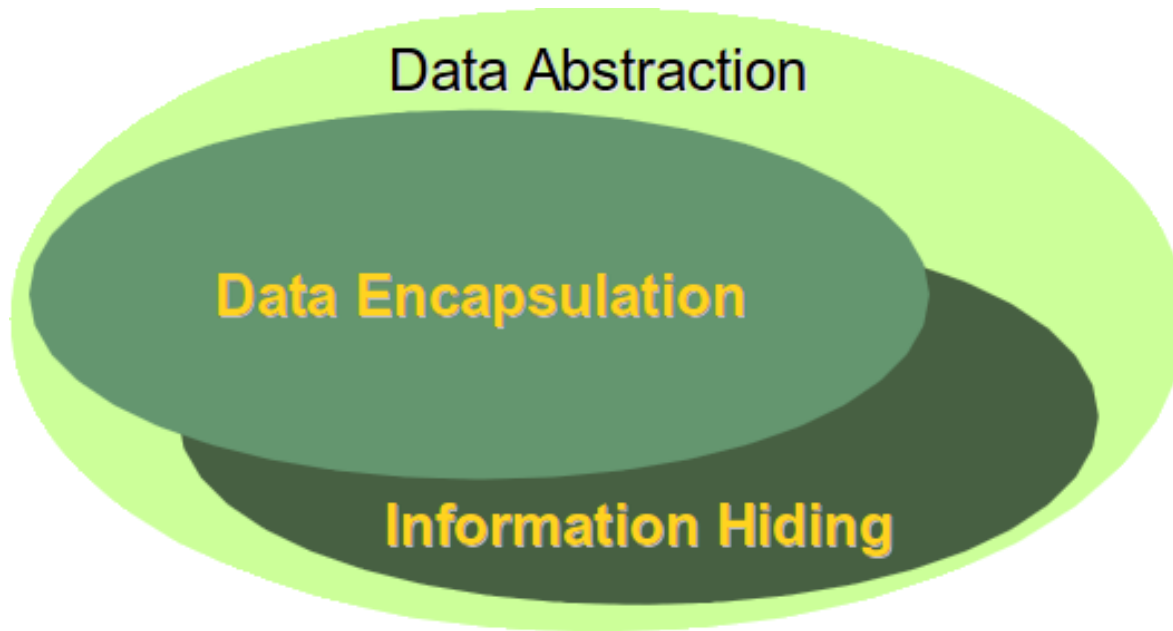
Private Data Fields and Public Accessors and Mutators

- The whole idea behind encapsulation is to hide the implementation details from users. If a data member is private it means it can only be accessed within the same class. No outside class can access private data member (variable) of other class. However if we setup public getter and setter methods to update (for e.g. `void setSSN(int ssn)`) and *read* (for e.g. `int getSSN()`), the private data fields. Then, the outside class can access those private data fields via public methods.
- This way data can only be accessed by public methods thus making the private fields and their implementation hidden for outside classes.** That's why encapsulation is known as **data hiding**. We will see an example to understand this concept later.

Data Field Encapsulation

Why Data Fields Should Be private?

- To protect data.
- To make class easy to maintain.

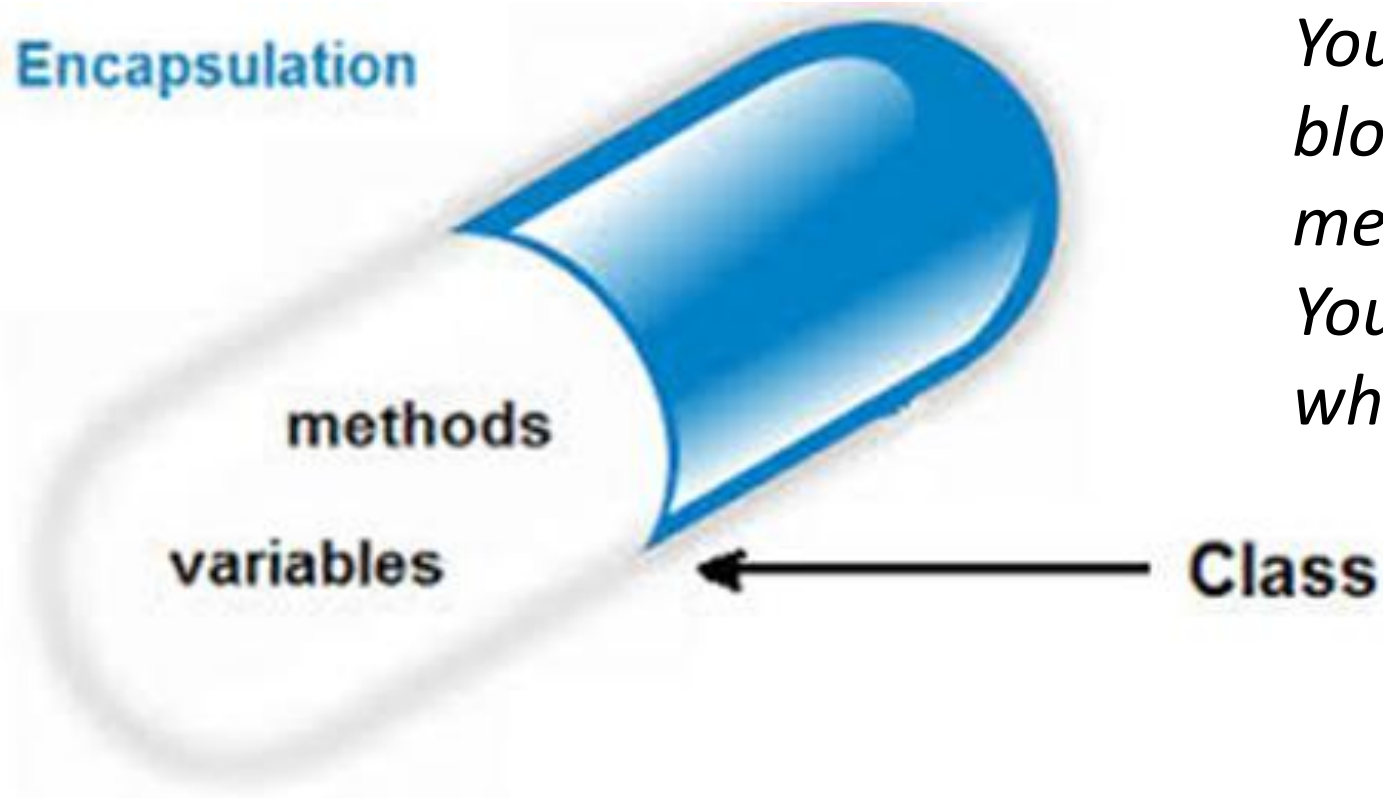


Class/Methods are also considered abstraction.

*Constants, Static Variables, ...
Overloading are also considered as
Information hiding.*

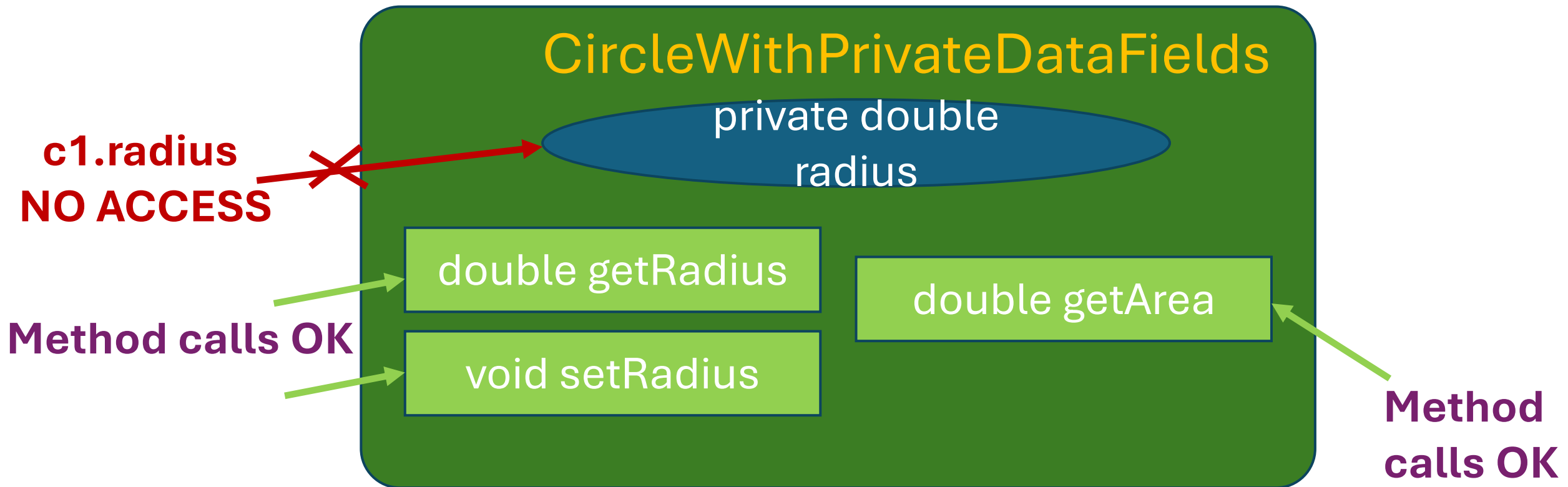
Data Field Encapsulation

Why Data Fields Should Be private?



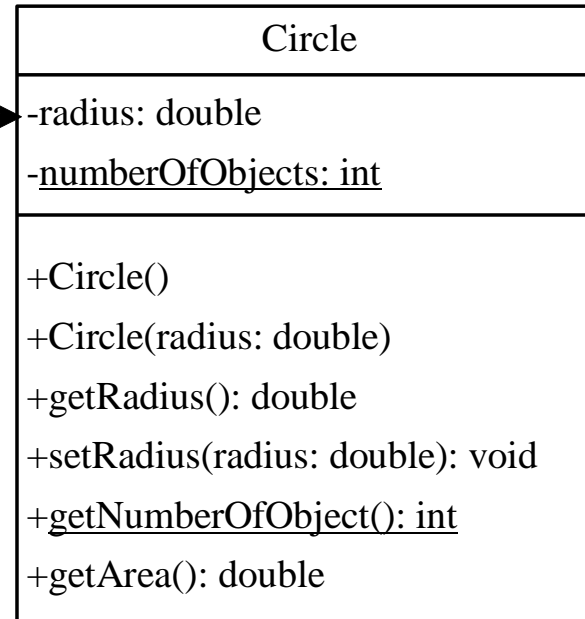
*You just need to know it is
blood pressure control
medicine.
You do not need to know
what chemical formula it is.*

Encapsulation Using Private Data Fields



Example of Data Field Encapsulation

The - sign indicates
private modifier



CircleWithPrivateDataFields

Class Header (Body Omitted)

```
public class CircleWithPrivateDataFields {  
    private double radius;  
    private static int numberOfObjects;  
    public CircleWithPrivateDataFields() {..}  
    public CircleWithPrivateDataFields(double newRadius) {..}  
    public double getRadius() {..}  
    public void setRadius(double newRadius) {..}  
    public static int getNumberOfObjects() {..}  
    public double getArea() {..}  
}
```

Static Members (Class Members)

Properties

- static int numberOfObjects

Methods

+ static int

Non-static Members (Instance Members)

Properties

- double radius

Constructors

+ CircleWithStaticMembers()

+
CircleWithStaticMemembrs(double
newRadius)

Methods

+ double getRadius()

+ void setRadius()

+ double getArea()



Demonstration Program

CIRCLEWITHPRIVATEDATAFIELDS.JAVA
TESTCIRCLEWITHPRIVATEDATAFIELDS.J
AVA



Assignment

FUEL 2 ASSIGNMENT

SUBMIT YOUR PROGRAM TO
MOODLE COURSE UPLOAD LINK



Classes and Objects (3):

Data Encapsulation II Passing
Objects to Methods

Lecture 2

JVM

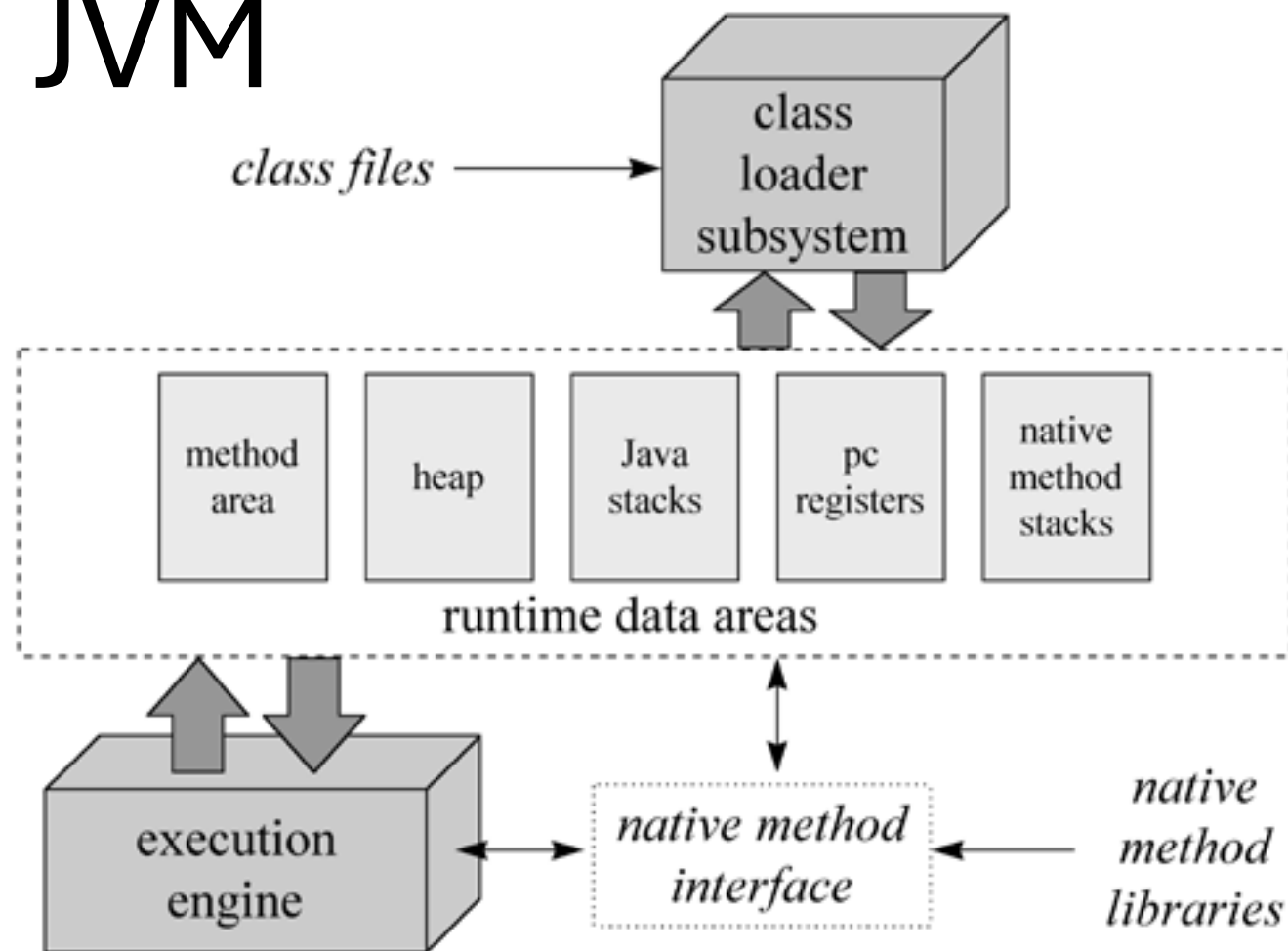


Figure 5-1. The internal architecture of the Java Virtual Machine.

Runtime data areas

Thread 1

pc register

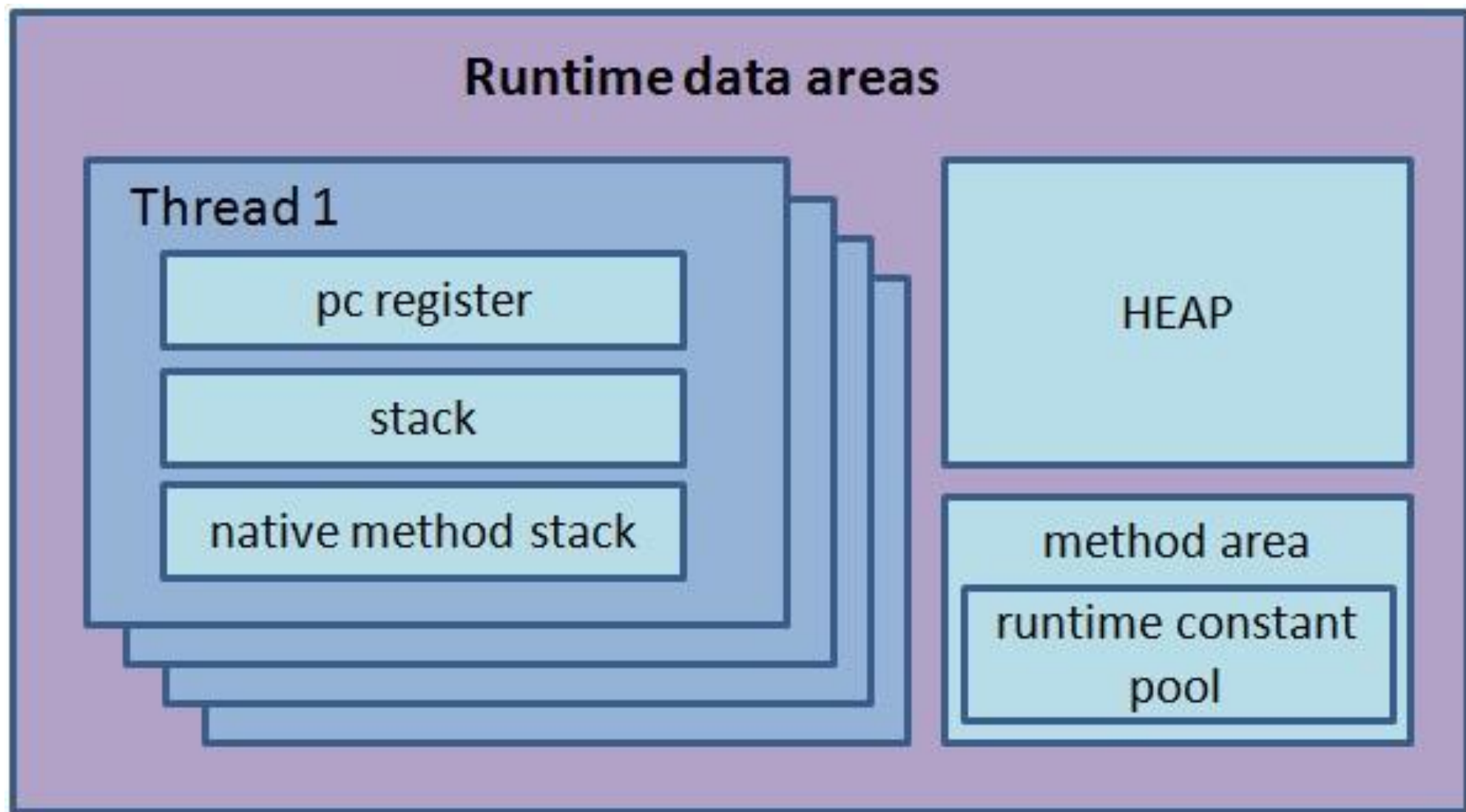
stack

native method stack

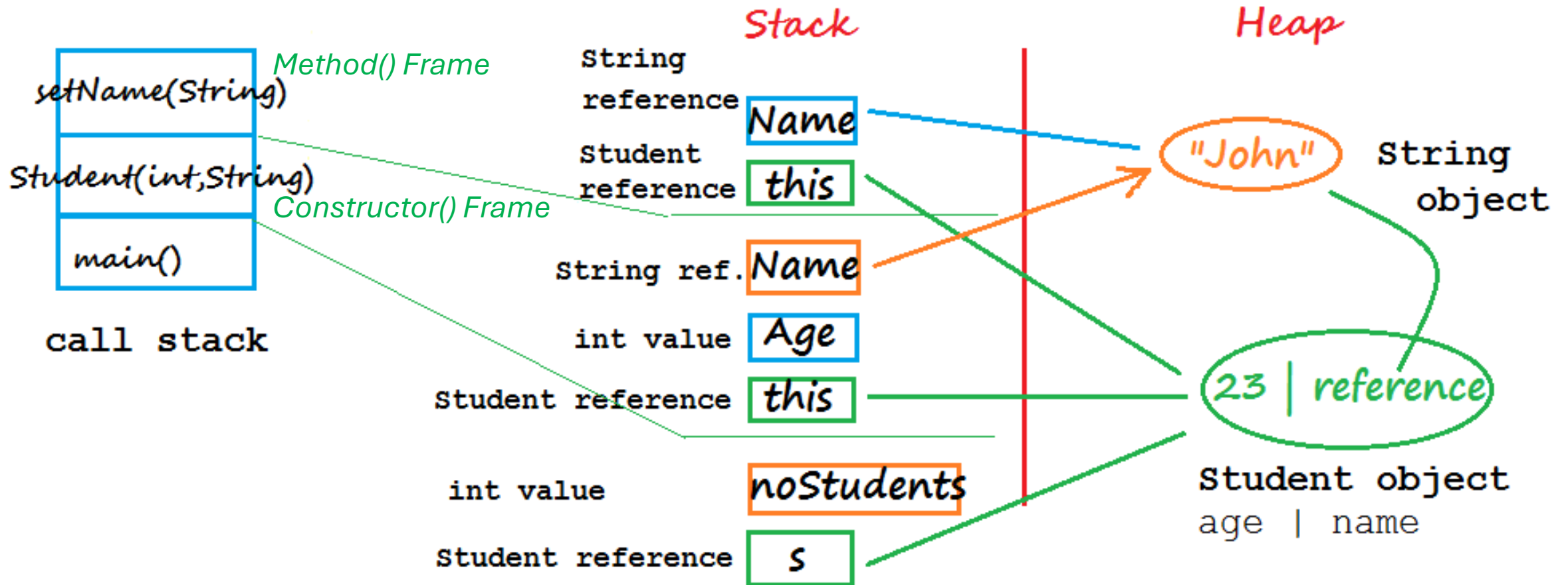
HEAP

method area

runtime constant
pool



Memory Allocation

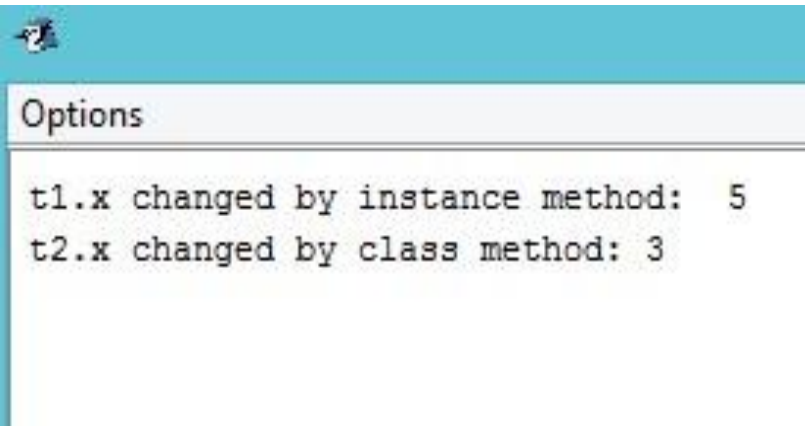


Two Ways to Update a Data Field from Other Class

INT2.java TestINT2.java

```
class TestINT2{
    public static void main(){
        INT2 t1 = new INT2();
        INT2 t2 = new INT2();
        System.out.println
            ("t1.x changed by instance method: " + t1.change());
        System.out.println
            ("t2.x changed by class method: " + INT2.change(t2));
    }
}
```

```
class INT2{
    private int x=0;
    public int change(){
        x = 5;
        return x;
    }
    public static int change(INT2 a){
        a.x = 3; // can not be x (nonstatic)
        return a.x;
    }
    // static can work on nonstatic instance data
    // if you pass object.
}
```



Options

```
t1.x changed by instance method: 5
t2.x changed by class method: 3
```

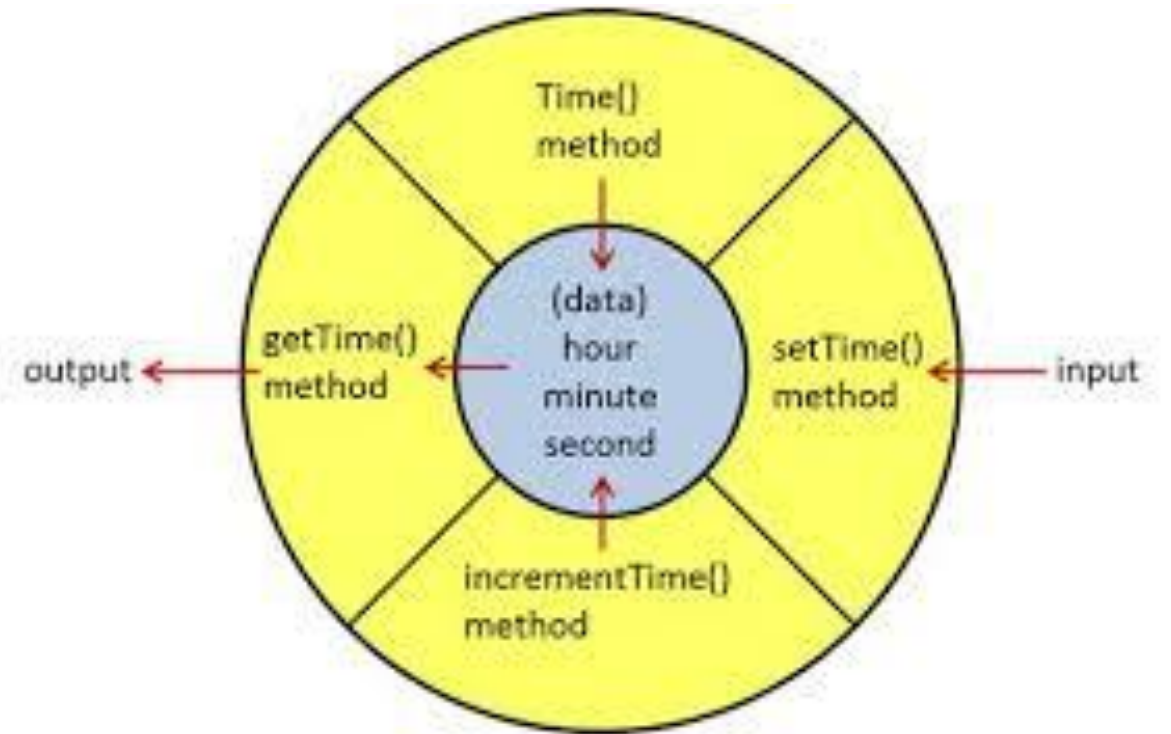
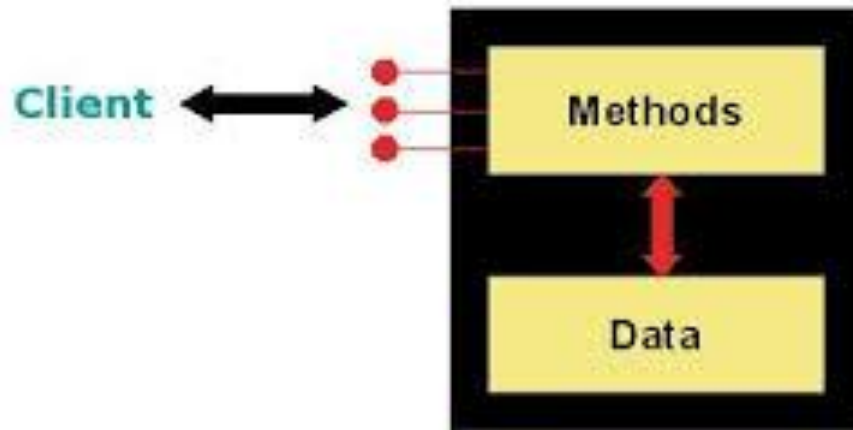


Demonstration Program

INT2.JAVA + TESTINT2.JAVA

First Way: Accessor/Mutator Methods

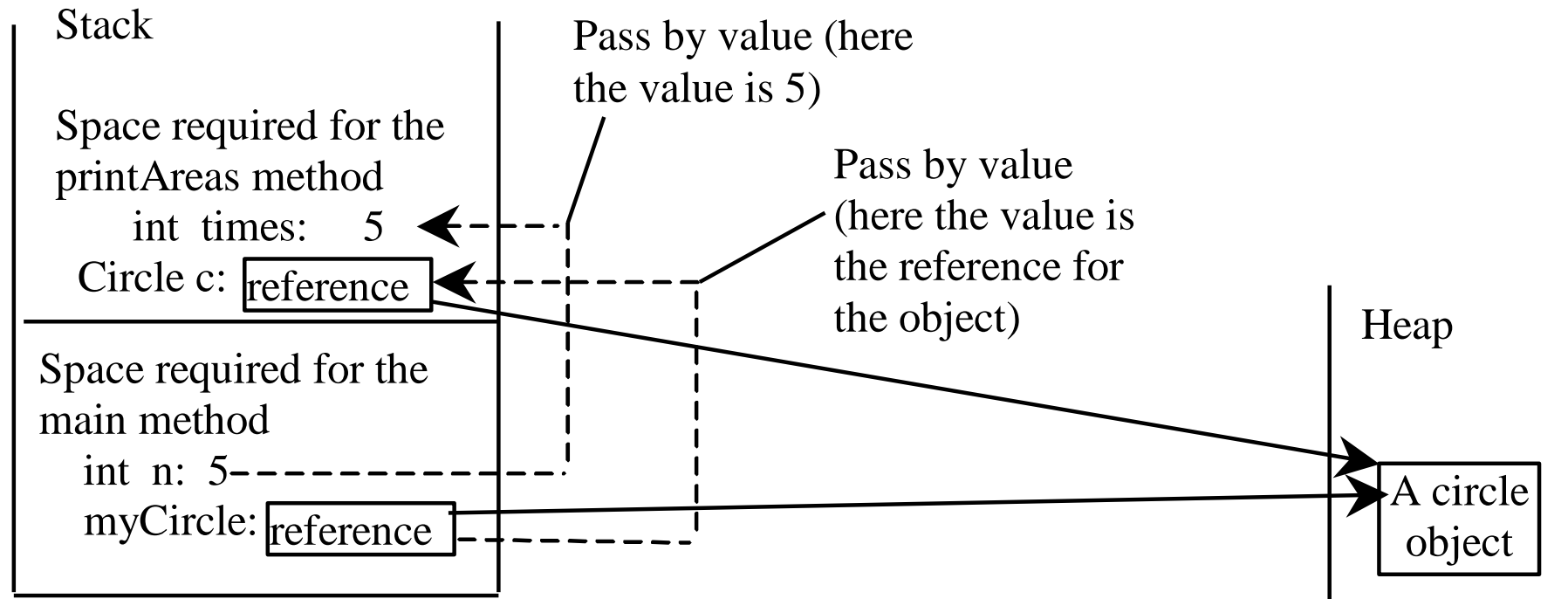
For Encapsulated Objects



Passing Objects to Methods

- Passing by value for primitive type value (the value is passed to the parameter)
- Passing by value for reference type value (the value is the reference to the object)

Passing Objects to Methods, cont.



Passing Objects is Another Way to Enter into **Data Capsule (Object)**

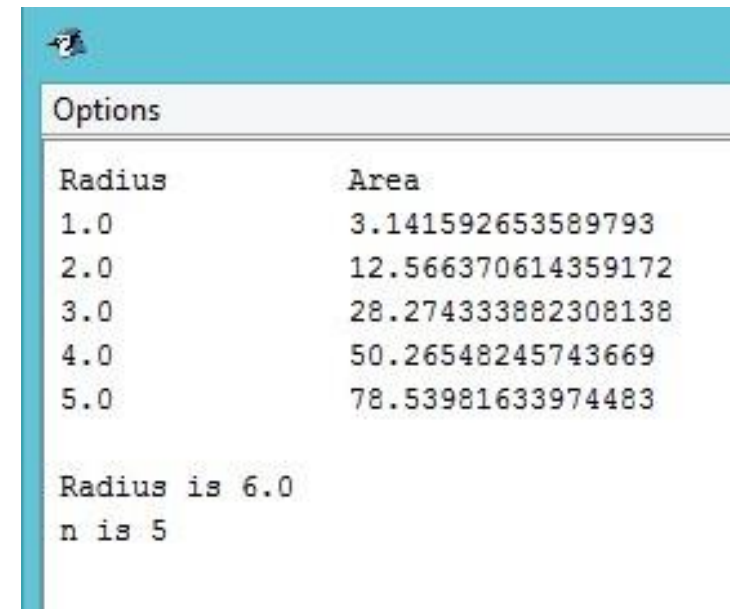
- Reference data type pointing to the body of the object. So, when the method get a argument of reference type, the method only get the pointer.
- The pointer's accessing power enables the method's other code to access and modify the data in the **Data Capsule (Encapsulated Objects)**
- **Passing Objects** is an action of opening data capsule.

Pointer View for Passing Objects to Methods

PassingObjects.java

```
public static void main(String[] args) {  
    // Create a Circle object with radius 1  
    CircleWithPrivateDataFields myCircle =  
        new CircleWithPrivateDataFields(1);  
    // Create a circle with radius 1.0  
  
    // Print areas for radius 1, 2, 3, 4, and 5.  
    int n = 5;  
    printAreas(myCircle, n);  
  
    // See myCircle.radius and times  
    System.out.println("\n" + "Radius is "  
        + myCircle.getRadius());  
    System.out.println("n is " + n);  
}
```

```
public static void printAreas(CircleWithPrivateDataFields c,  
    int times) {  
    System.out.println("Radius \t\tArea");  
    while (times >= 1) {  
        System.out.println(c.getRadius() + "\t\t" + c.getArea());  
        c.setRadius(c.getRadius() + 1);  
        times--;  
    }  
}
```



Radius	Area
1.0	3.141592653589793
2.0	12.566370614359172
3.0	28.274333882308138
4.0	50.26548245743669
5.0	78.53981633974483

Radius is 6.0
n is 5



Demonstration Program

PASSINGOBJECTS.JAVA



Assignment

FUEL 3 ASSIGNMENT

SUBMIT YOUR PROGRAM TO
MOODLE COURSE UPLOAD LINK



Classes and Objects (4):

Immutable Objects and Classes

Lecture 4

Immutable Objects and Classes

You can define immutable classes to create immutable objects. The contents of immutable objects cannot be changed.

- The String class is immutable.
- If a class is immutable, then all its data fields must be private and it cannot contain public setter methods for any data fields. A Class with **all private data fields** and **no mutators** is not necessarily immutable. For example, the following Student class has all private data fields and no setter methods,

Immutable Objects and Classes

```
public class Student {  
    private int id;  
    private String name;  
    private java.util.Date dateCreated;  
    public Student(int ssn, String newName) {  
        id = ssn;  
        name = newName;  
        dateCreated = new java.util.Date();  
    }  
    public int getId() { return id; }  
    public String getName() {return name; }  
    public java.util.Date getDateCreated() {  
        return dateCreated;  
    }  
}
```

Immutable Objects and Classes

- As shown in the following code, the data field `dateCreated` is returned using the `getDateCreated()` method. This is a reference to a `Date` object. Through this reference, the content for `dateCreated` can be changed.

```
• public class Test{  
•     public static void main(String[] args) {  
•         Student student = new Student(111223333,  
•         "John");  
•         java.util.Date dateCreated =  
•         student.getDateCreated();  
•         dateCreated.setTime(200000);  
•     }  
• }
```

**// Similar to shallow copy, you make the reference data type private, but not its body. So, leave a door open to change of contents.
// It is no longer immutable.**

Immutable Objects and Classes

- For a class to be immutable, it must meet the following requirements:
- All **data fields** must be **private**.
- There can't be any mutator methods for data fields. (**No Mutator**)
- No accessor methods can return a reference to data field that is mutable. (**No accessor method for reference data.**)



Assignment

FUEL 4 ASSIGNMENT

SUBMIT YOUR PROGRAM TO
MOODLE COURSE UPLOAD LINK



this Reference

Lecture 5

The **this** Reference

- The keyword `this` refers to the object itself. It can also be used inside a constructor to invoke another constructor of the same class, when the constructor is overloaded.
- The `this` keyword is the name of a reference that an object can use to refer to itself. You can use the `this` keyword to reference the object's instance members. For example, the following code in (a) uses `this` to reference the object's radius and invokes its `getArea()` method explicitly. The `this` reference is normally omitted, as shown in (b). However, the `this` reference is needed to reference hidden data fields or invoke an overloaded constructor.

The **this** Reference

```
public class Circle {  
    private double radius;  
    ...  
    public double getArea(){  
        return this.radius * this.radius * Math.PI;  
    }  
    public String toString(){  
        return "radius: " + this.radius + "area: " + this.getArea();  
    }  
}
```

(a)

(a) is equivalent to (b)

```
public class Circle {  
    private double radius;  
    ...  
    public double getArea(){  
        return radius * radius * Math.PI;  
    }  
    public String toString(){  
        return "radius: " + radius + "area: " + getArea();  
    }  
}
```

(b)

Using this to Reference Hidden Data Fields

- When there is a local variable or a parameter sharing the same name with a class variable, we can use `this.variable` to refer to the instance variable (object variable) while the `variable` is used for the local or parameter variable. The instance variable is in fact hidden data field according to the rule for variable scope.
- A hidden static variable can be accessed simply by using the `ClassName.staticVariable`. A hidden instance variable can be accessed by using the keyword `this`.

Using this to Reference Hidden Data Fields

class variable(Class.var), instance variable (this.var)

```
public class F {  
    private int i = 5;  
    private static double k = 0;  
    public void setI(int i) { this.i = i; }  
    public static void setK(double k) {  
        F.k = k;  
    }  
}
```

(a)

Suppose that f1 and f2 are two objects of F.
Invoking f1.setI(**10**) is to execute **this.i = 10**,
where this refers f1

Invoking f2.setI(**45**) is to execute **this.i = 45**,
where this refers f2

Invoking F.setK(**33**) is to execute F.k = **33**. setK is
a static method

The this Reference

- The **this** keyword gives us a way to reference the object that invokes an instance method. To invoke **f1.setI(10)**, **this.i = i** is executed, which assigns the value of parameter **i** to the data field **i** of this calling object **f1**.
- The keyword **this** refers to the object that invokes the instance method **setI**. The line **F.k = k** means that the value in parameter **k** is assigned to the static data field **k** of the class, which is shared by all objects of the class.

Using this to Invoke a Constructor

- The `this` keyword can be used to invoke another constructor of the same class. For example, you can rewrite the `Circle` class as follows:

```
public class Circle {  
    private double radius;  
    public Circle(double radius) {  
        // The this keyword is used to reference  
        // the hidden data field radius  
        this.radius = radius;  
    }  
    // of the object being constructed.  
    public Circle() {  
        // The this keyword is used to invoke another constructor.  
        this(1.0);  
    }  
    ...  
}
```

Java requires that the `this(arg-list)` statement appear first in the constructor before any other executable statements. (build a default object first) Use `this(arg-list)` as much as possible if there is multiple constructor.

Fast Encapsulation Using **this** Reference

- Converting a non-object-oriented programming to object-oriented programming efficiently. Direct copying the code and hook up with the instance variable using this reference.



Assignment

FUEL 5 ASSIGNMENT

SUBMIT YOUR PROGRAM TO
MOODLE COURSE UPLOAD LINK

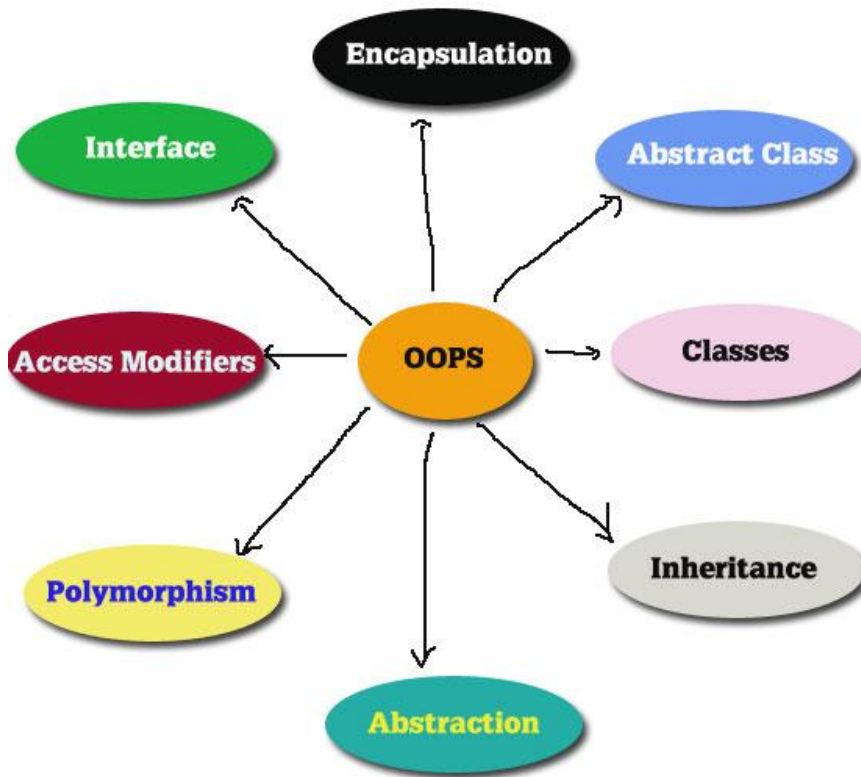


Loading of a Class and Its Objects

Lecture 6

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)

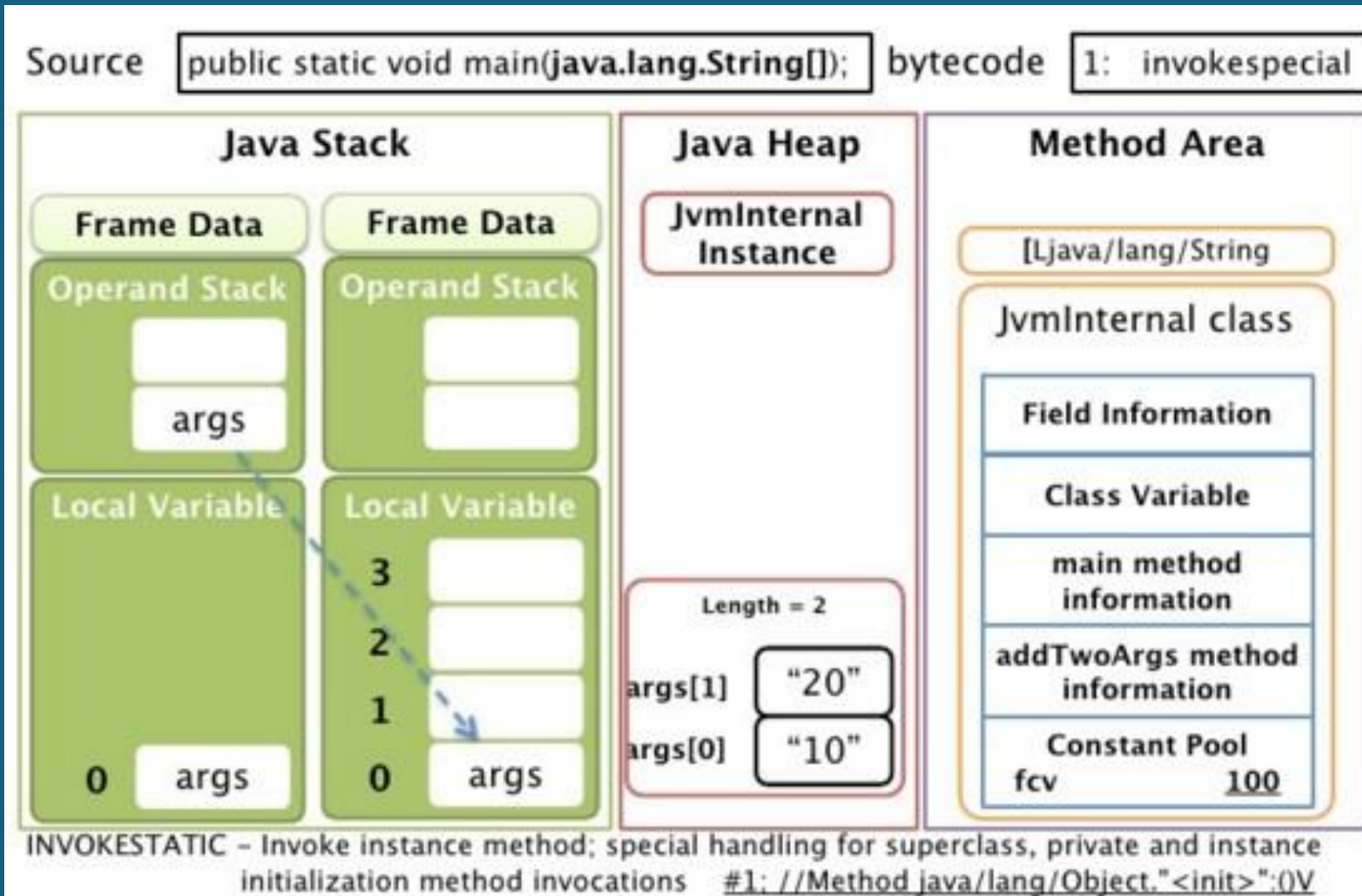
accessor/mutator methods: (public method)

client methods: (private method)

Encapsulated Data Class: private data/public method

Immutable Data Class: private data/no mutator method
no returned pointer

Memory Allocation



Loading a Class to JVM

Static Variable/Static Method:

Constants: (final static variables)

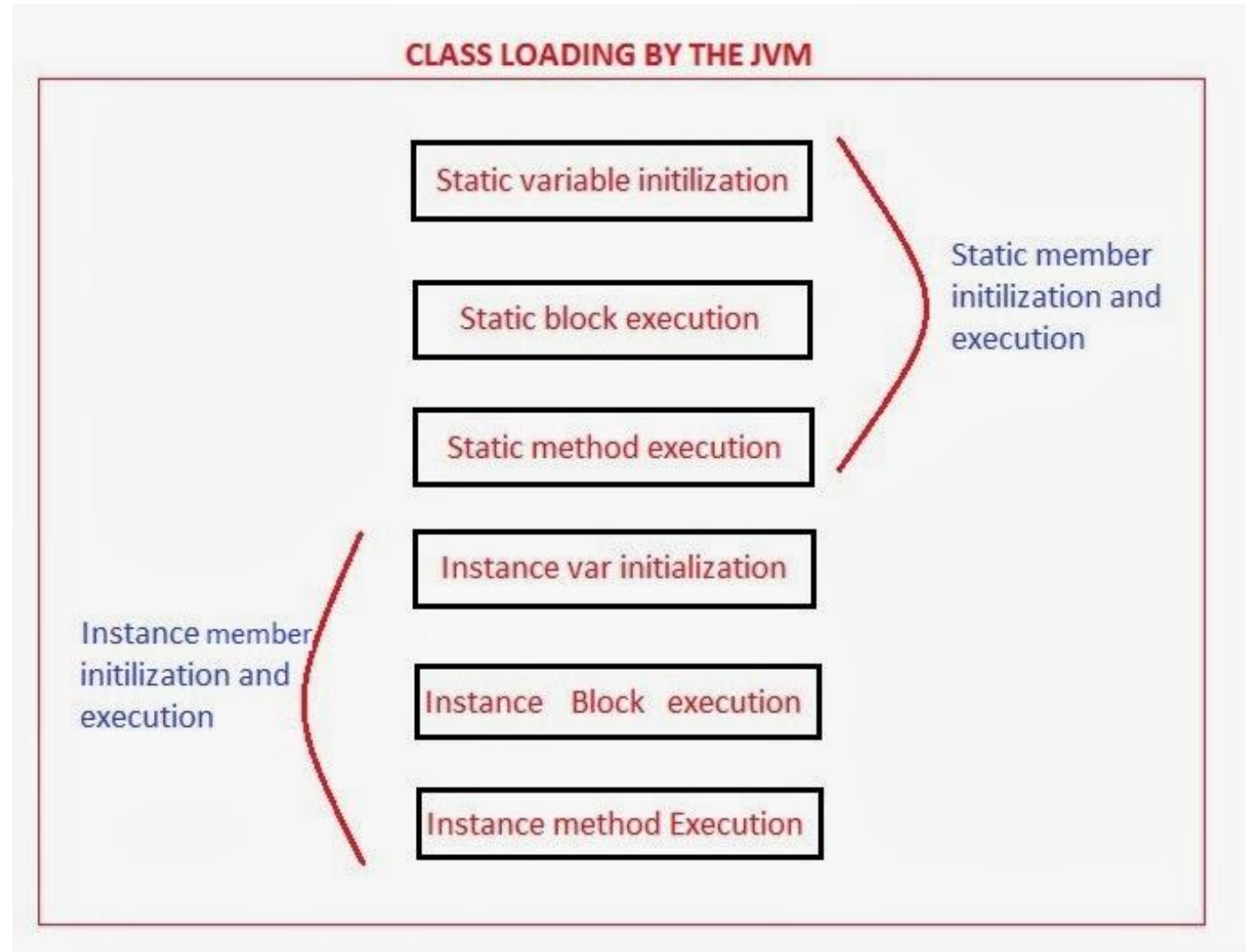
Math.PI, Integer.MIN_VALUE

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)

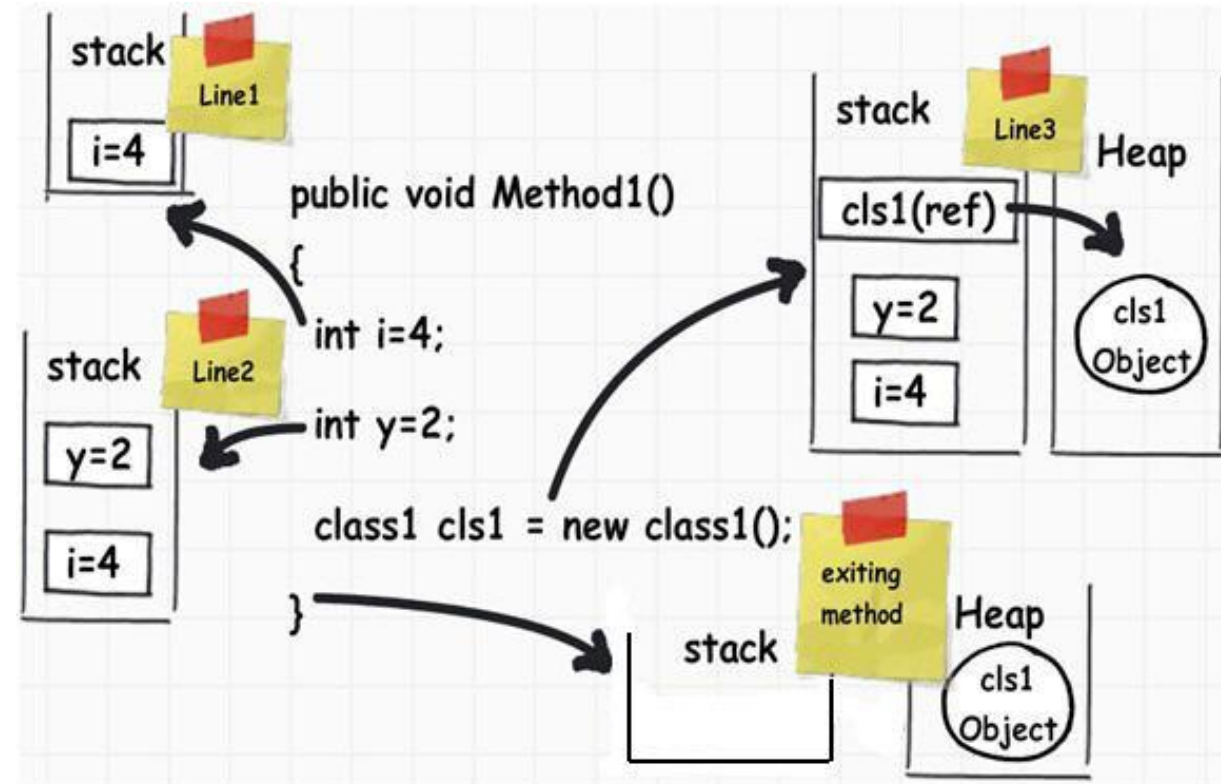
Instance Variable/Instance Method:

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 7

Object-Oriented Thinking

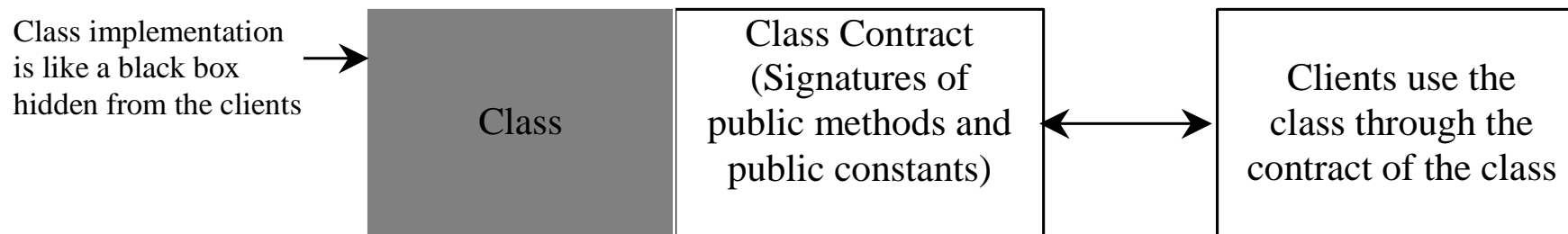
- Part 1-Chapters 1-8 introduced fundamental programming techniques for problem solving using loops, methods, and arrays.
- The studies of these techniques lay a solid foundation for object-oriented programming. Classes provide more flexibility and modularity for building reusable software.

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 8

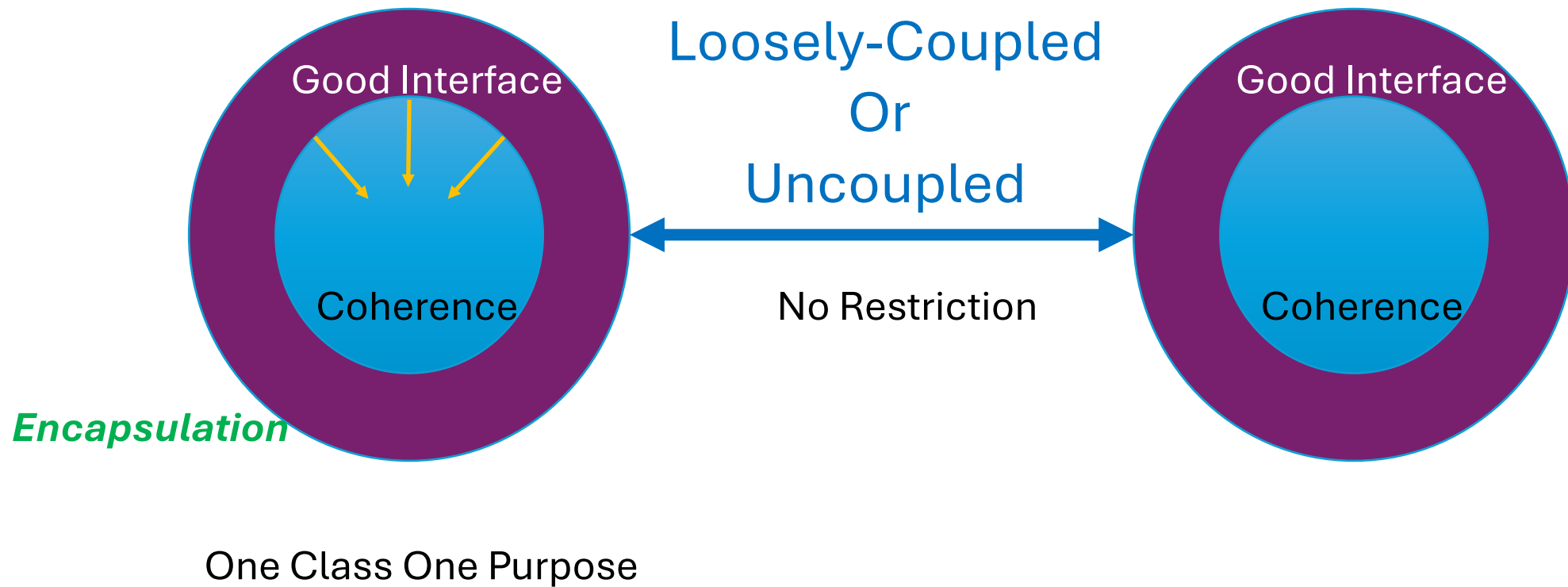
Guidelines for Class Design

- 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



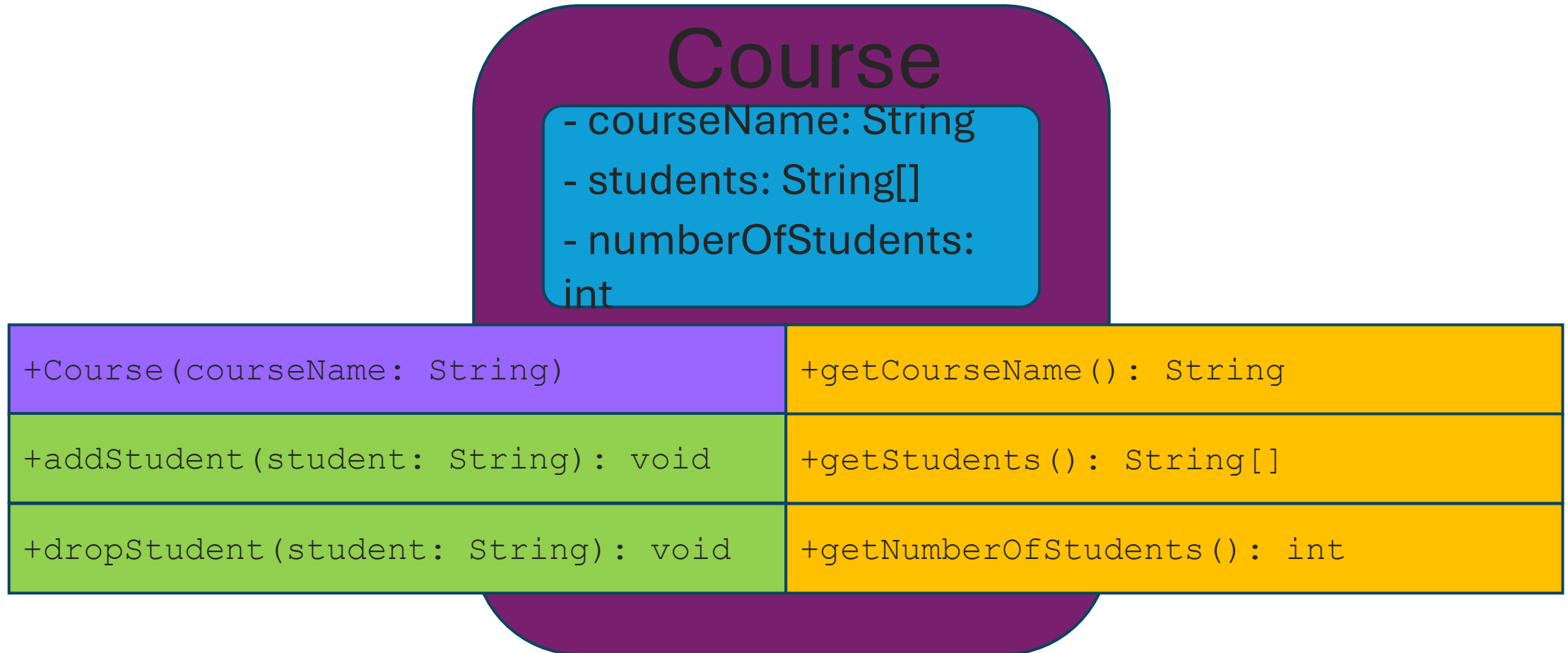
Designing a good class interface

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 9

Class Design

- **Class Data Fields:**
 - Constants
 - Class Variables
- **Member Data Fields:**
 - Instance Variables
- **Constructors:**
 - no-arg
 - Long-form
- **Getters/Setters Method:**
 - no-arg
 - Long-form

Inherited Methods:

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

Derived Data Fields:

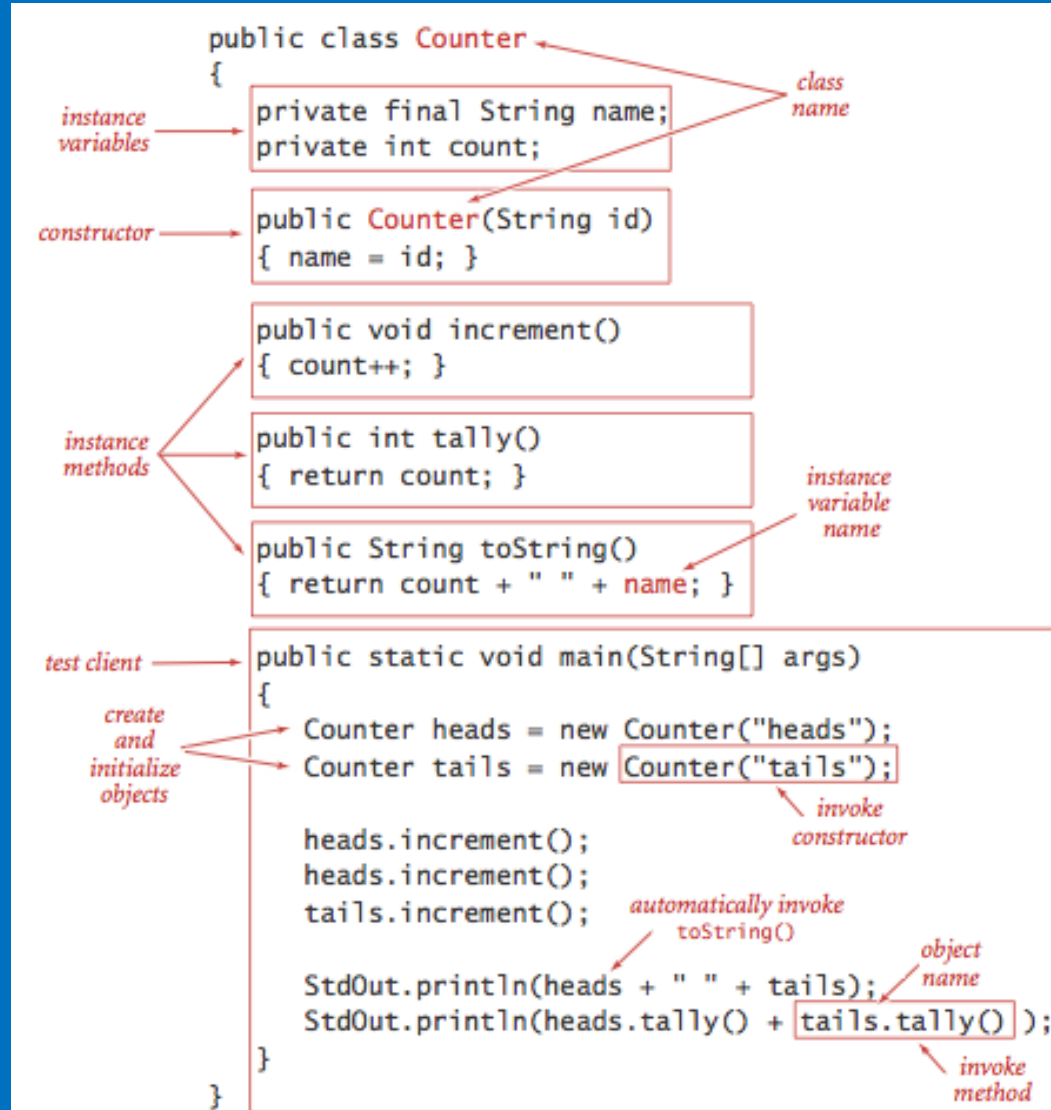
- getDerivedData()

Utility Methods:

- Static Methods

Class Writing Styles

(Not Syntax, Design Conventions)



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)

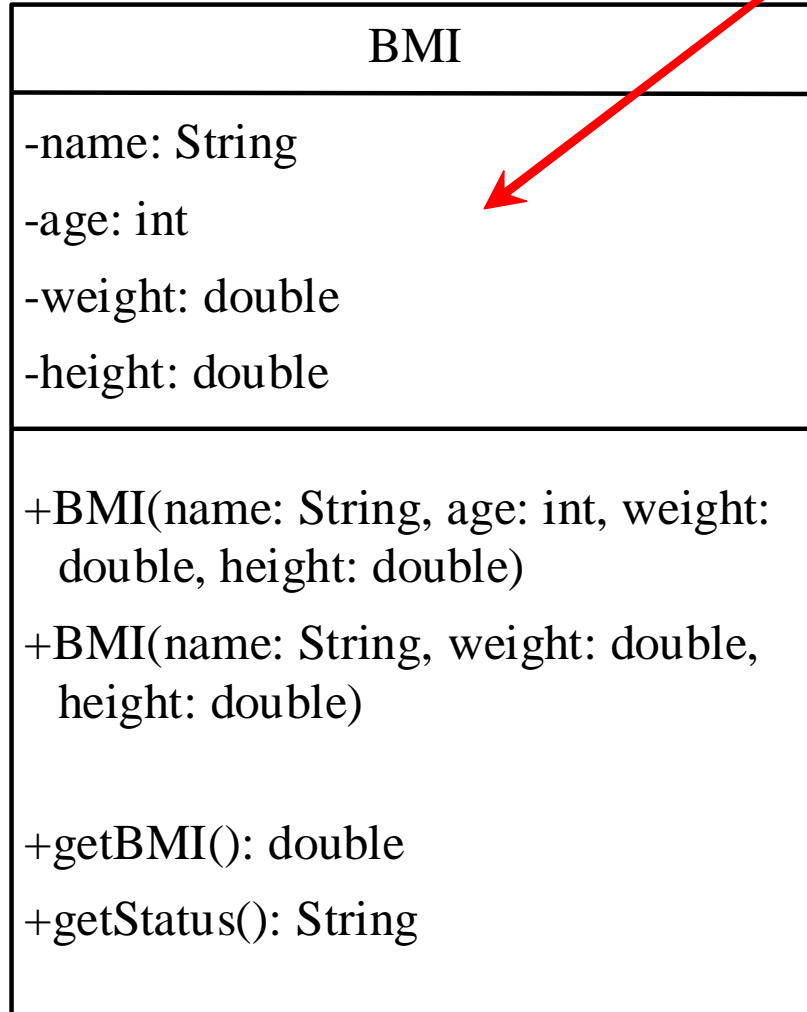
Body mass index (BMI) is a measure of body fat based on height and weight that applies to adult m

- http://www.nhlbi.nih.gov/health/educational/lose_wt/BMI/bmi_calc.htm

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"> - name: String - age: int - weight: double - height: double + final KILOGRAMS_PER_POUND: static double + final METERS_PER_INCH: static double
<pre> << Constructors >> + BMI(String name, int age, double weight, double height) + BMI(String name, double weight, double height) << Methods >> + 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

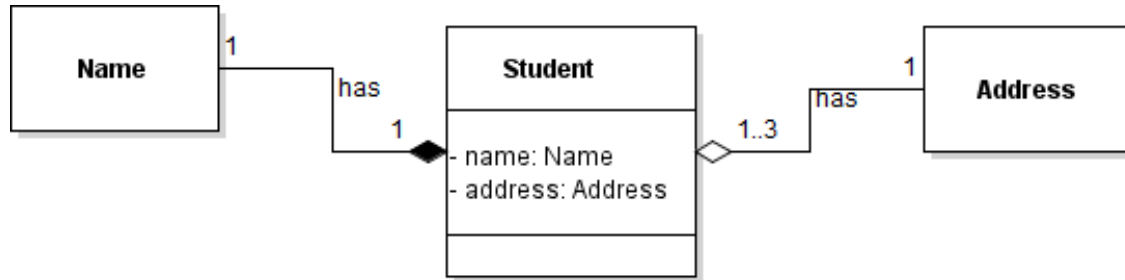


Class Use Relationship

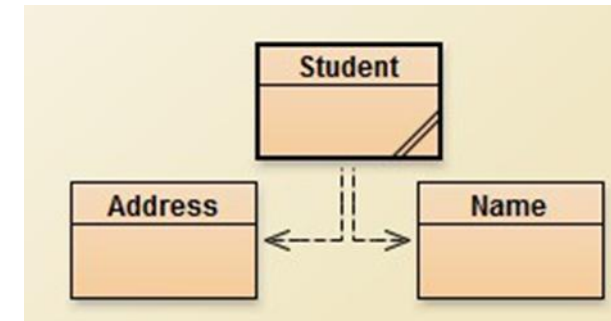
Lecture 10

has A Relationship

Violet



BlueJ

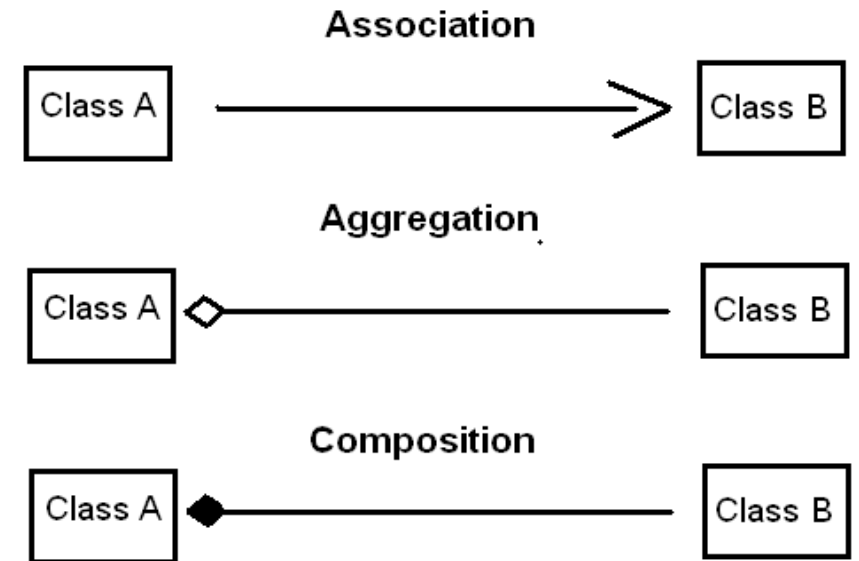


Class Relationships (use and inherit)

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

These three relationships are use-relationships 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 relationship. (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.

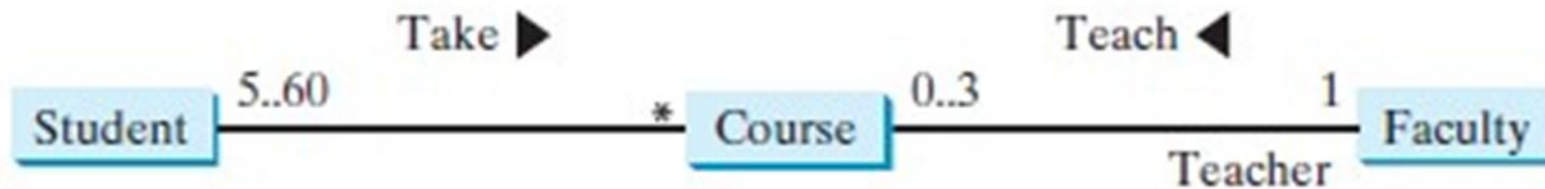
- A multiplicity could be a number or an interval that specifies how many of the class's objects are involved in the relationship.
- The character ***** means an unlimited number of objects, and
- the interval **m..n** indicates that the number of objects is between **m** and **n**, inclusively.
- In 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) { ... }  
}
```

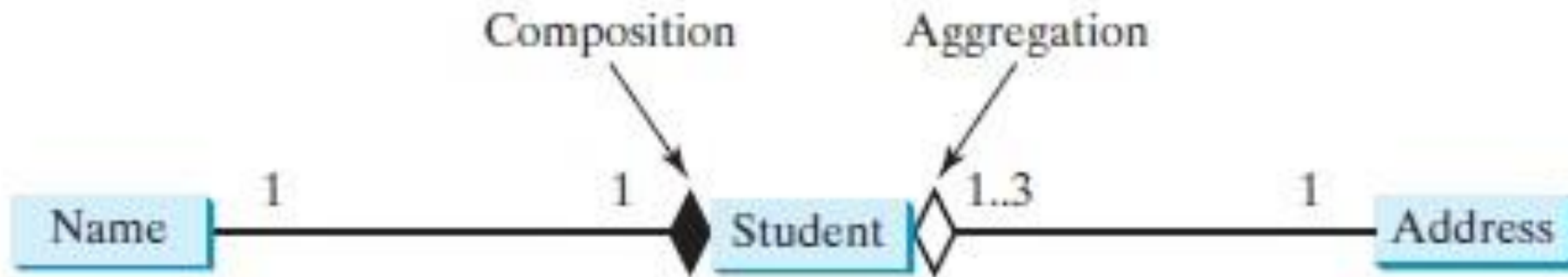
```
public class Course {  
    private Student[]  
        classList;  
    private Faculty faculty;  
  
    public void addStudent(  
        Student s) { ... }  
  
    public void setFaculty(  
        Faculty faculty) { ... }  
}
```

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



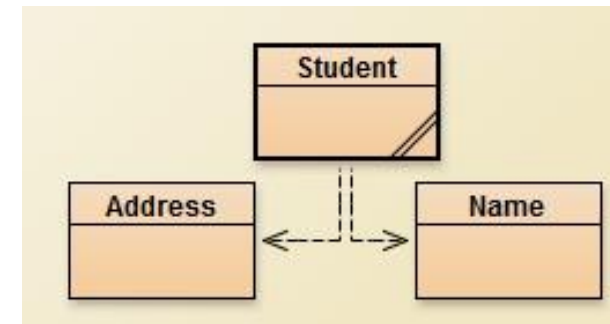
Aggregation or Composition

- 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. Student has an address non-exclusively
(Composition) (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 {  
    ...  
}
```

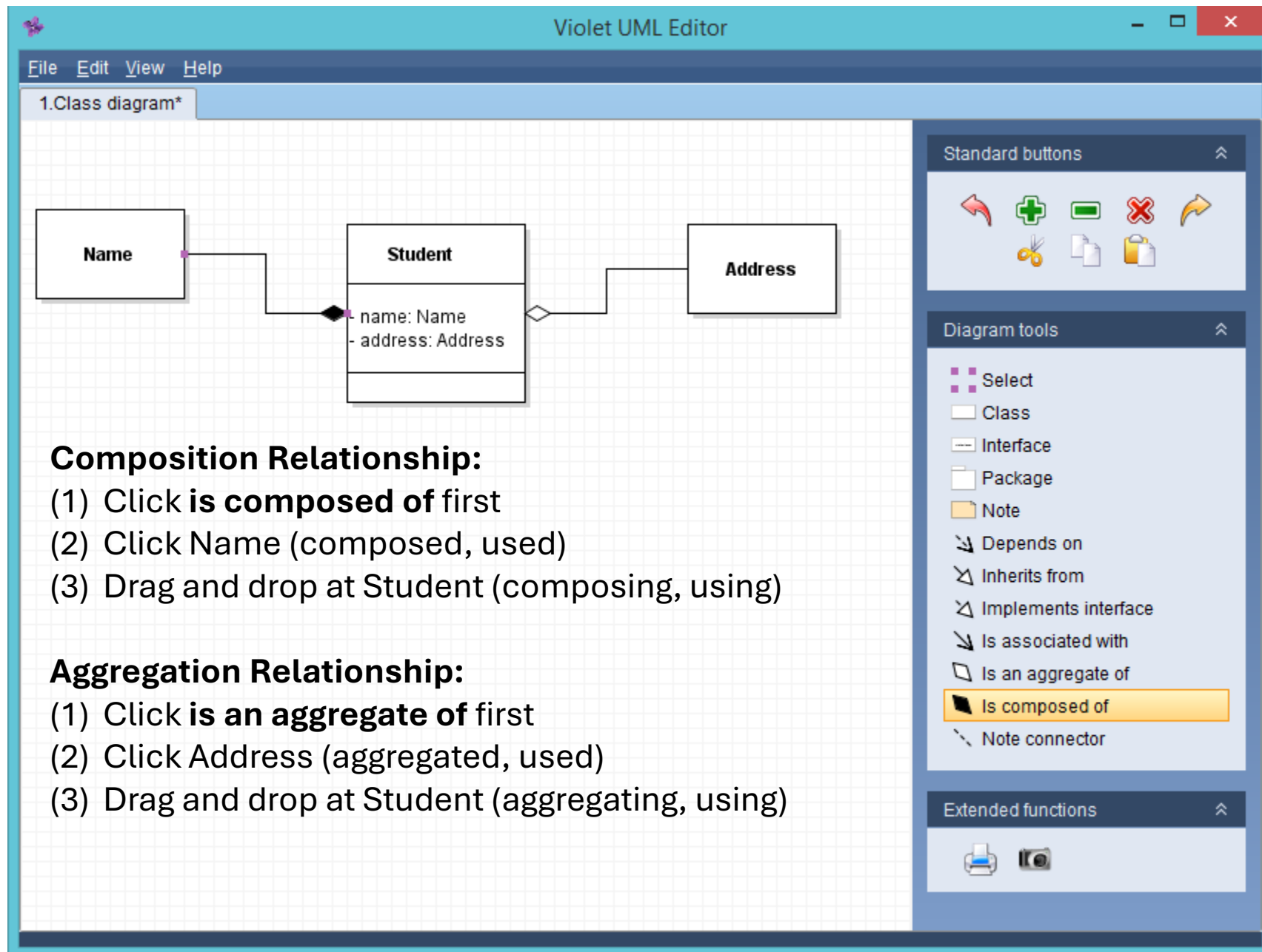
Aggregated class

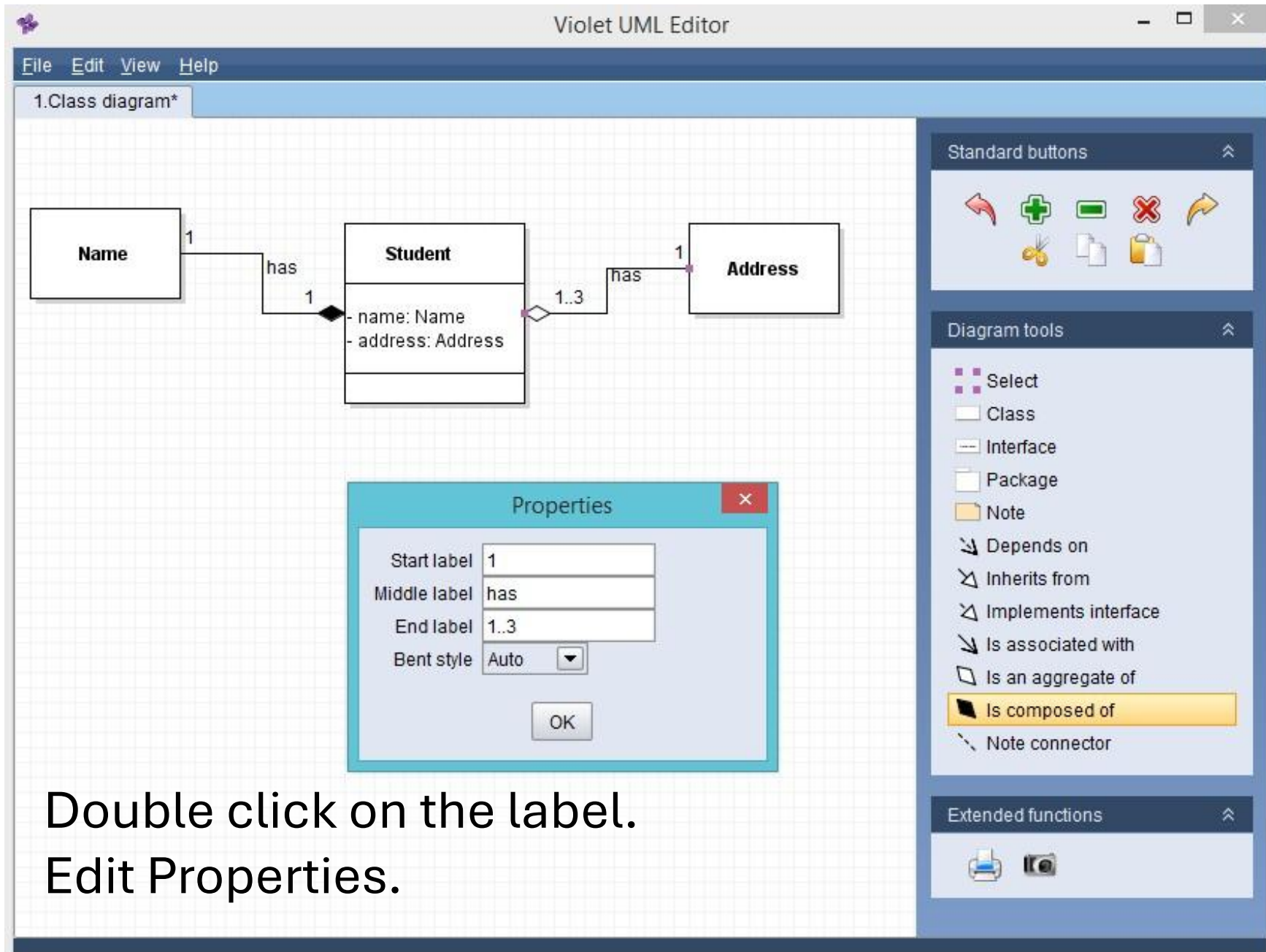
Aggregating: Using

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**.

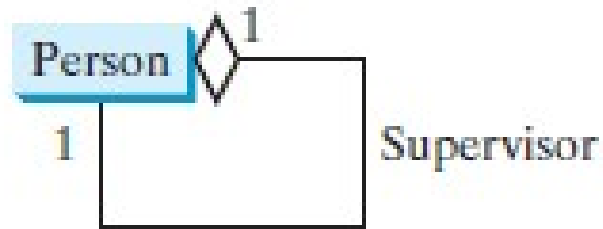




Double click on the label.
Edit Properties.

Aggregation Between Same Class

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

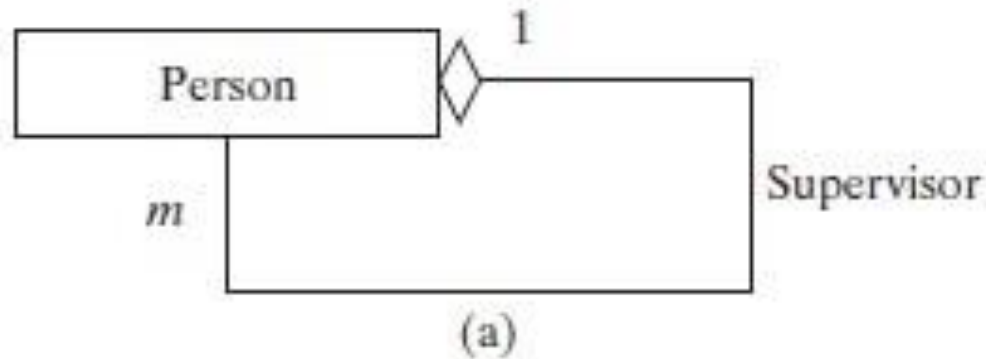


```
public class Person {  
    // The type for the data is the class itself  
    private Person supervisor;  
    ...  
}
```

Aggregation Between Same Class

What happens if a person has several supervisors?

(One class using itself many times)



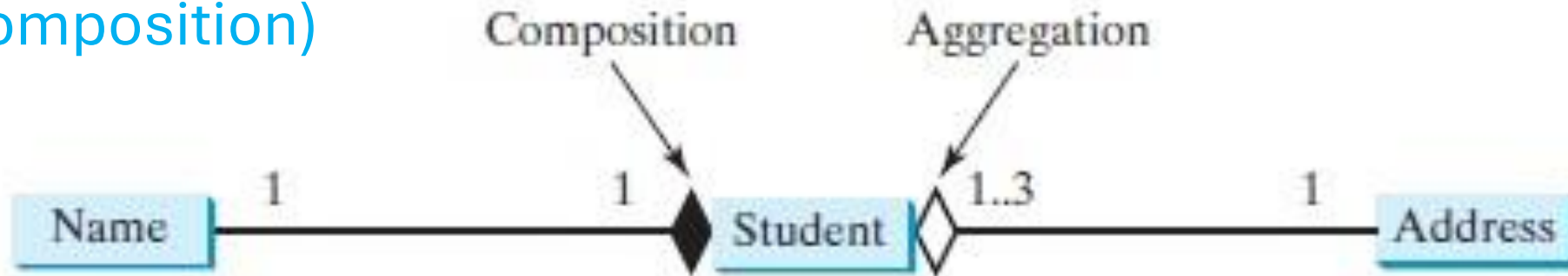
```
public class Person {  
    ...  
    private Person[] supervisors;  
}
```

Below the code block is the label "(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)

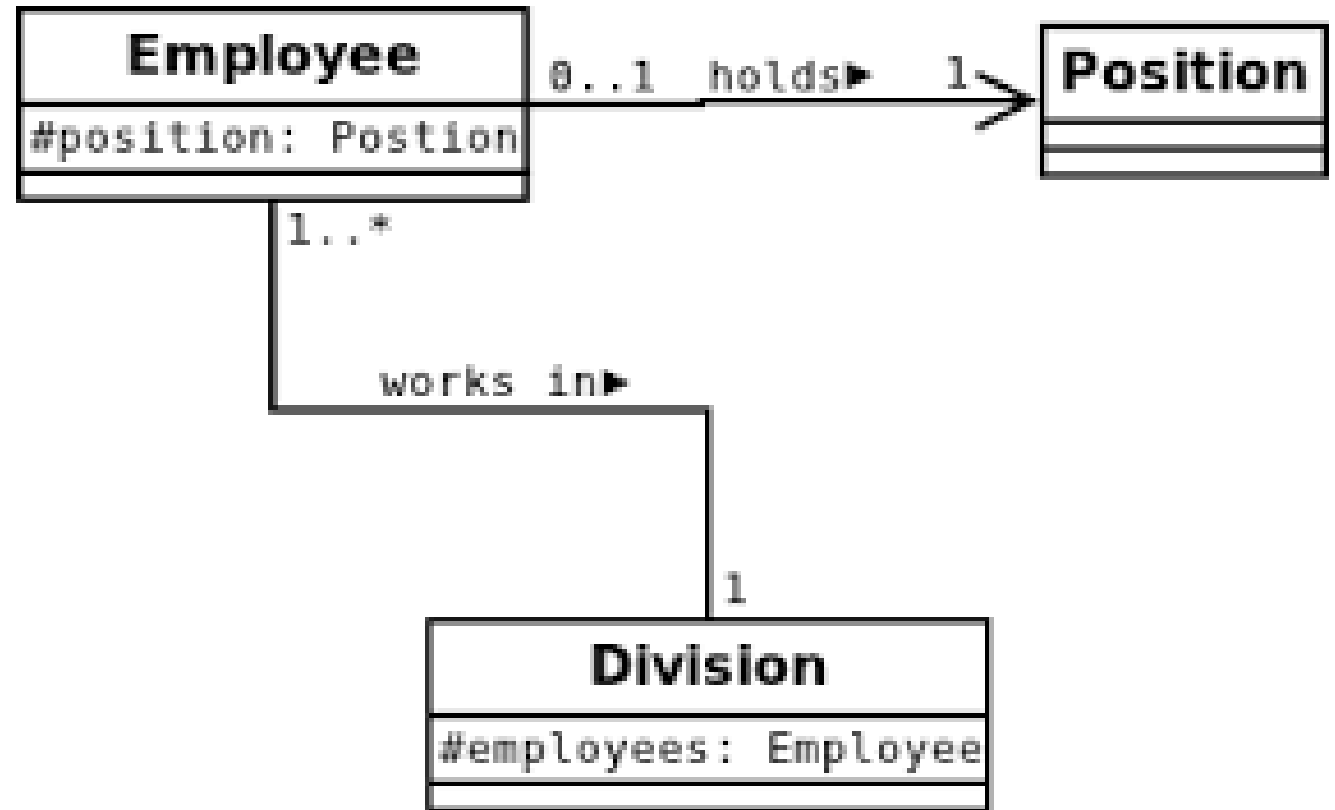


Why analyze the multiplicity of relationship?

<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

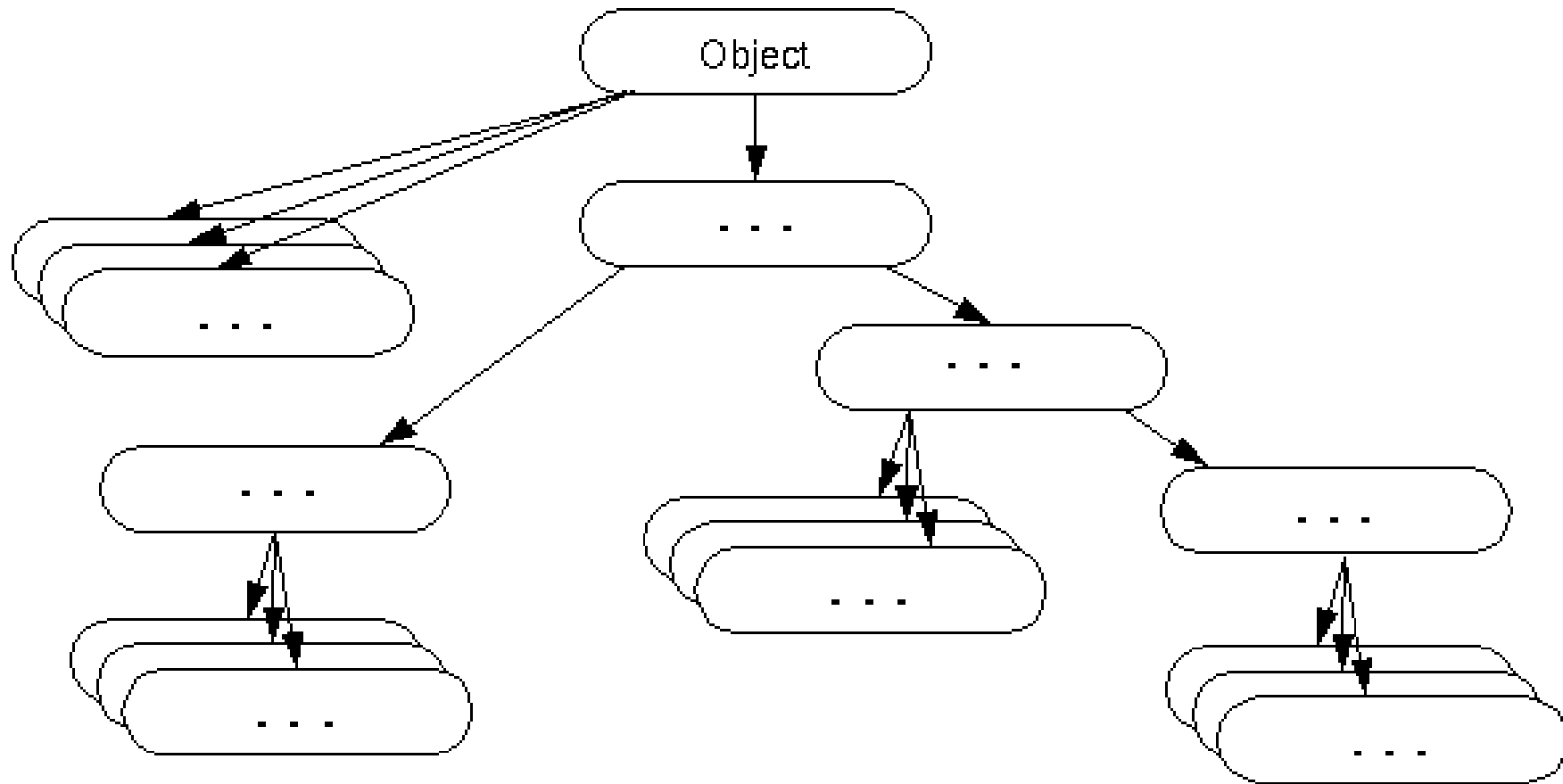
SUBMIT YOUR PROGRAM TO
MOODLE COURSE UPLOAD LINK



Object Class

Lecture 11

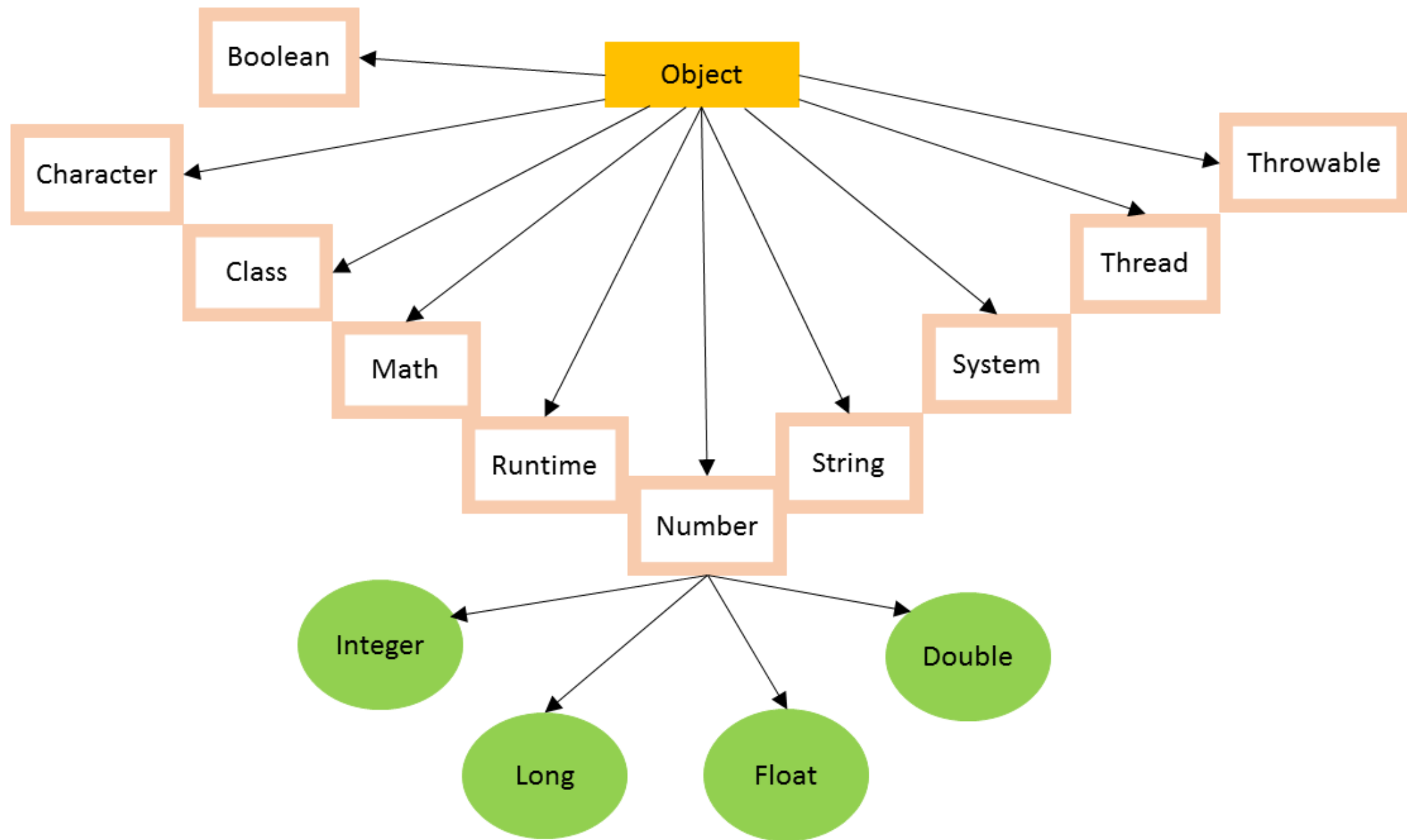
Every Class Inherits from Object Class



Object as a Superclass

Object is superclass for all objects in java.lang language such as Integer, Double, String, Arrays, and etc.

- The Object class, in the **java.lang** package, sits at the top of the class hierarchy tree. Every class is a descendant, direct or indirect, of the Object class. **Every class you use or write inherits the instance methods of Object.** You need not use any of these methods, but, if you choose to do so, you may need to override them with code that is specific to your class. The methods inherited from Object that are discussed in this section are:



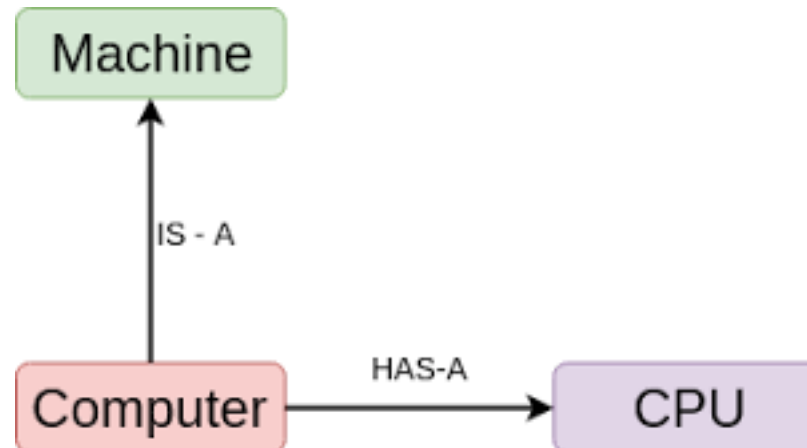
Object as a Superclass

Object is superclass for all objects in java.lang language such as Integer, Double, String, Arrays, and etc.

- **protected Object clone()** throws CloneNotSupportedException
Creates and returns a copy of this object.
- **public boolean equals(Object obj)**
Indicates whether some other object is "equal to" this one.
- **protected void finalize() throws Throwable**
Called by the garbage collector on an object when garbage collection determines that there are no more references to the object
- **public final Class getClass()**
Returns the runtime class of an object.
- **public int hashCode()**
Returns a hash code value for the object.
- **public String toString()**
Returns a string representation of the object.

Subclass Polymorphism

- **Generalization (Grouping):** Java polymorphism creating a subclass object using its superclass variable.
- **Overloading:** Inheritance of Member Methods (Object Class)
- **Is_A Relationship:** Subclass object is also a superclass object.





Standard Methods for Object Class

Lecture 12

The clone() Method

One way to copy

- If a class, or one of its superclasses, implements the Cloneable interface, you can use the clone() method to create a **copy** from an existing object. To create a clone, you write:

```
aCloneableObject.clone();
```

The clone() Method

One way to copy

- Object's implementation of this method checks to see whether the object on which clone() was invoked implements the Cloneable interface. If the object does not, the method throws a CloneNotSupportedException exception. Exception handling will be covered in a later lesson. For the moment, you need to know that clone() must be declared as

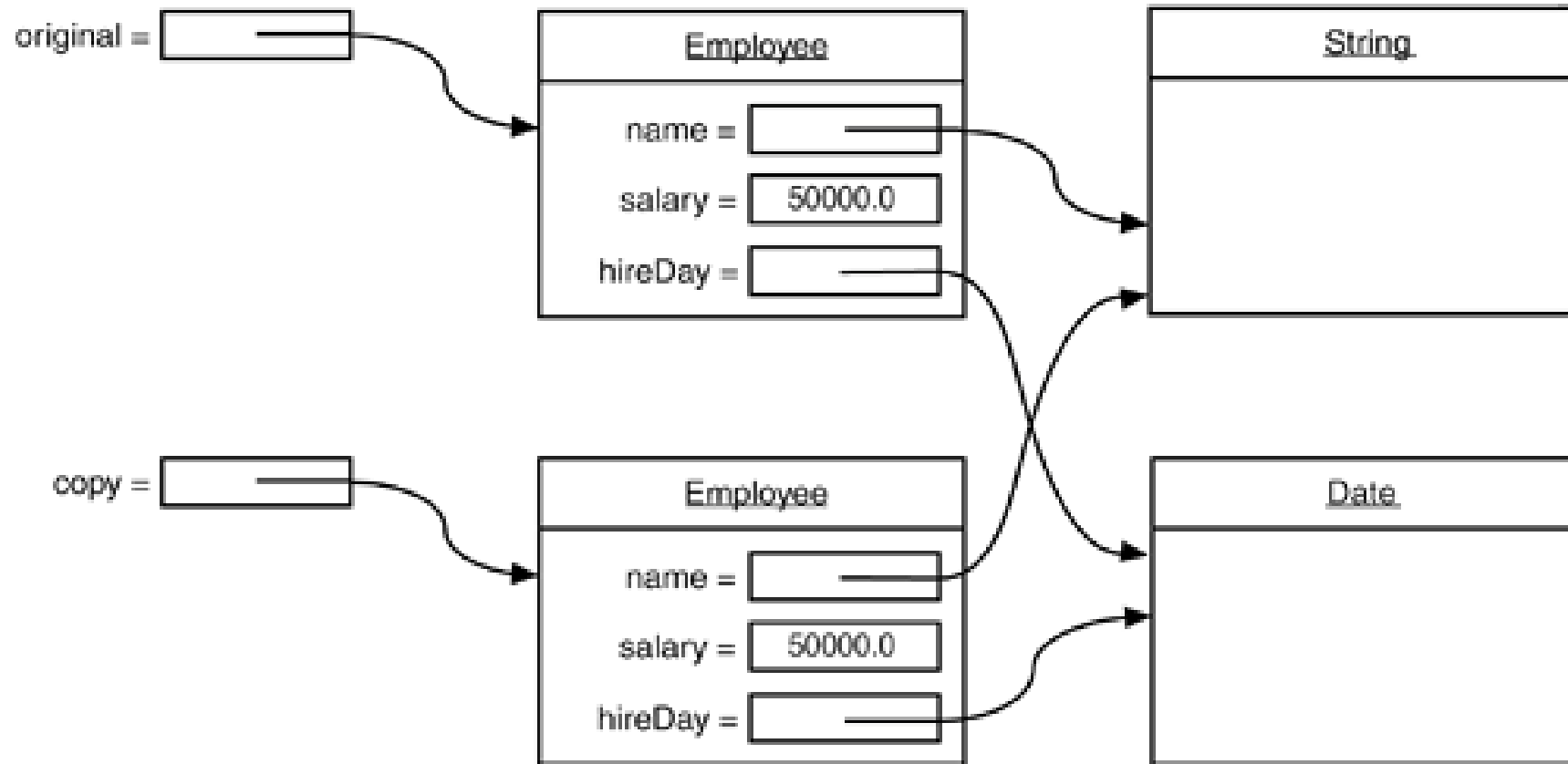
protected Object clone() throws CloneNotSupportedException

or:

public Object clone() throws CloneNotSupportedException

clone() method

(Sometimes shallow copy, sometimes deep copy, need to check)

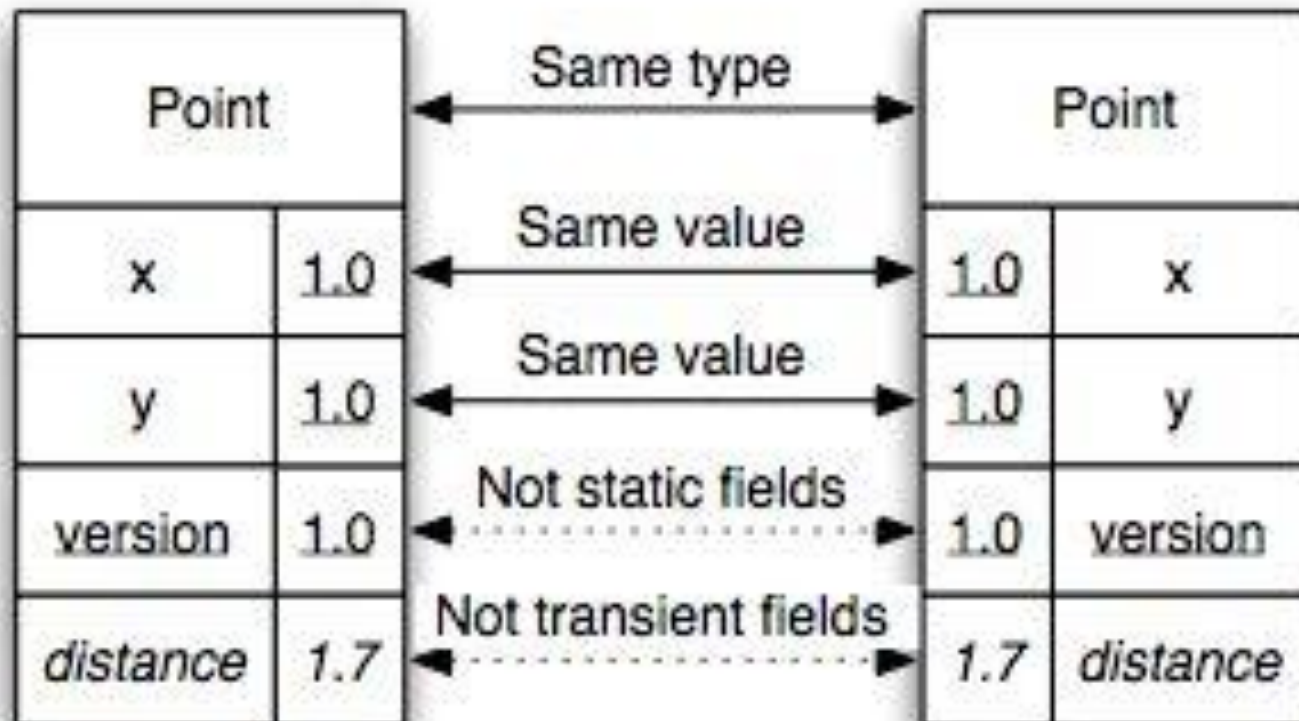


The equals() Method

- The equals() method compares two objects for equality and returns true if they are equal. The **equals()** method provided in the Object class uses the identity operator (==) to determine whether two objects are equal. For primitive data types, this gives the correct result. For objects, however, it does not. The **equals()** method provided by Object tests whether the object references are equal—that is, if the objects compared are the exact same object.
- To test whether two objects are equal in the sense of equivalency (containing the same information), you must override the **equals()** method. Here is an example of a Book class that overrides **equals()**:

```
public class Book {  
    ...  
    public boolean equals(Object obj) {  
        if (obj instanceof Book)  
            return ISBN.equals( (Book) obj.getISBN() );  
        else  
            return false;  
    }  
}
```

Equality Check



Static data field is shared.
Nothing to compare.

Overriding equals() gives us a chance to redefine equality

- **Different definition for equality of Book class:**
 - A Book is equal if the book's title is the same.
 - A Book is equal if the book's ISBN is the same. (same print, or same edition)
 - A Book is equal if the book is the same copy. (For school library management)

The hashCode() Method

(another equality compare method)

- The value returned by hashCode() is the object's hash code, which is the object's **memory address** in hexadecimal.
- By definition, if two objects are equal, their hash code must also be equal. If you override the equals() method, you change the way two objects are equated and Object's implementation of hashCode() is no longer valid. Therefore, if you override the equals() method, you must also override the hashCode() method as well.

The toString() Method

- You should always consider overriding the **toString()** method in your classes.
- The Object's **toString()** method returns a String representation of the object, which is very useful for debugging. The String representation for an object depends entirely on the object, which is why you need to override **toString()** in your classes.
- You can use **toString()** along with `System.out.println()` to display a text representation of an object, such as an instance of Book:
- **`System.out.println(firstBook.toString());`**
- which would, for a properly overridden **toString()** method, print something useful, like this:
- ISBN: 0201914670; The Swing Tutorial; A Guide to Constructing GUIs, 2nd Edition

What you need to remember when overriding `toString()` manually?

- Return as much information as needed (that may be interesting)
- It is obligatory in data classes
- if you decide that your **`toString()`** provide result in format presentable to the user, then you have to clearly document output print format and remain it unchanged for life. In that case you need to be aware that **`toString()`** output may be printed in UI somewhere
- beside **`toString()`** you still need to provide accessor methods for class fields, if needed

The getClass() Method

(Class object is a information object of another object. It is like Color/Font Class)

- You cannot override getClass.
- The **getClass()** method returns a **Class** object, which has methods you can use to get information about the class, such as its name (**getSimpleName()**), its superclass (**getSuperclass()**), and the interfaces it implements (**getInterfaces()**). For example, the following method gets and displays the class name of an object:

```
void printClassName(Object obj) {  
    System.out.println("The object's" + " class is " +  
        obj.getClass().getSimpleName());  
}
```

- The **Class** class, in the **java.lang** package, has a large number of methods (more than 50). For example, you can test to see if the class is an annotation (**isAnnotation()**), an interface (**isInterface()**), or an enumeration (**isEnum()**). You can see what the object's fields are (**getFields()**) or what its methods are (**getMethods()**), and so on.

getClass() and instanceof

- **object.getClass()** return a **Class** object which contains the **Class** information of the object.
- **object instanceof class** will return a boolean value whether the **object** is of the **class**.
- **instanceof** is an operator (keyword) while **getClass()** is an method.
- **getClass()** has more information than **instanceof**.

Example of instanceof Operator

(if a pointer is null, it will return false)

The instanceof keyword can be used to test if an object is of a specified type.

```
if (objectReference instanceof type)
```

The following if statement returns true.

```
public class MainClass {  
    public static void main(String[] a) {  
  
        String s = "Hello";  
        if (s instanceof java.lang.String) {  
            System.out.println("is a String");  
        }  
    }  
}
```

```
is a String
```

The finalize() Method

- The Object class provides a callback method, **finalize()**, that may be invoked on an object when it becomes garbage. Object's implementation of **finalize()** does nothing—you can override **finalize()** to do cleanup, such as freeing resources.
- The **finalize()** method may be called automatically by the system, but when it is called, or even if it is called, is uncertain. Therefore, you should not rely on this method to do your cleanup for you. For example, if you don't close file descriptors in your code after performing I/O and you expect **finalize()** to close them for you, you may run out of file descriptors.

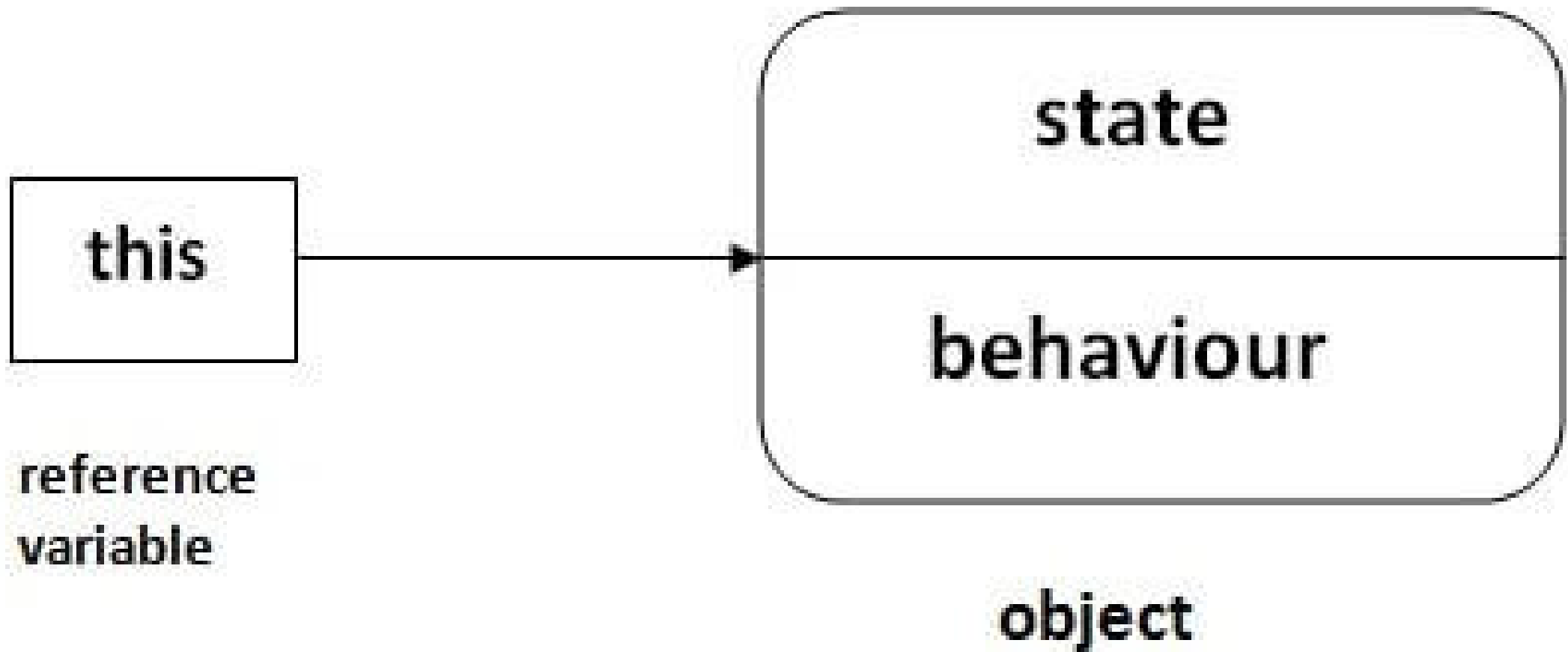


Use of this Reference

Lecture 13

Use of **this** reference

- The pointer to the object in the heap memory for the current object.
- Reference to current object: `this.x`;
- Reference to constructor `this(1.0)`;
- Calling a method: `this.getArea()`;




Calling overloading constructor


shorter constructor calling longer constructor method


- (1) eliminate the need to re-write different format of constructors.
- (2) Write the longer constructor first. Then, write constructors of all possible lengths.
- (3) Higher maintainability

Calling Overloaded Constructor

```
public class Circle {  
    private double radius;  
  
    public Circle(double radius) {  
        this.radius = radius;  
    }  
  
    public Circle() {  
        this(1.0);  
    }  
  
    public double getArea() {  
        return this.radius * this.radius * Math.PI;  
    }  
}
```

 this must be explicitly used to reference the data field radius of the object being constructed

 this is used to invoke another constructor

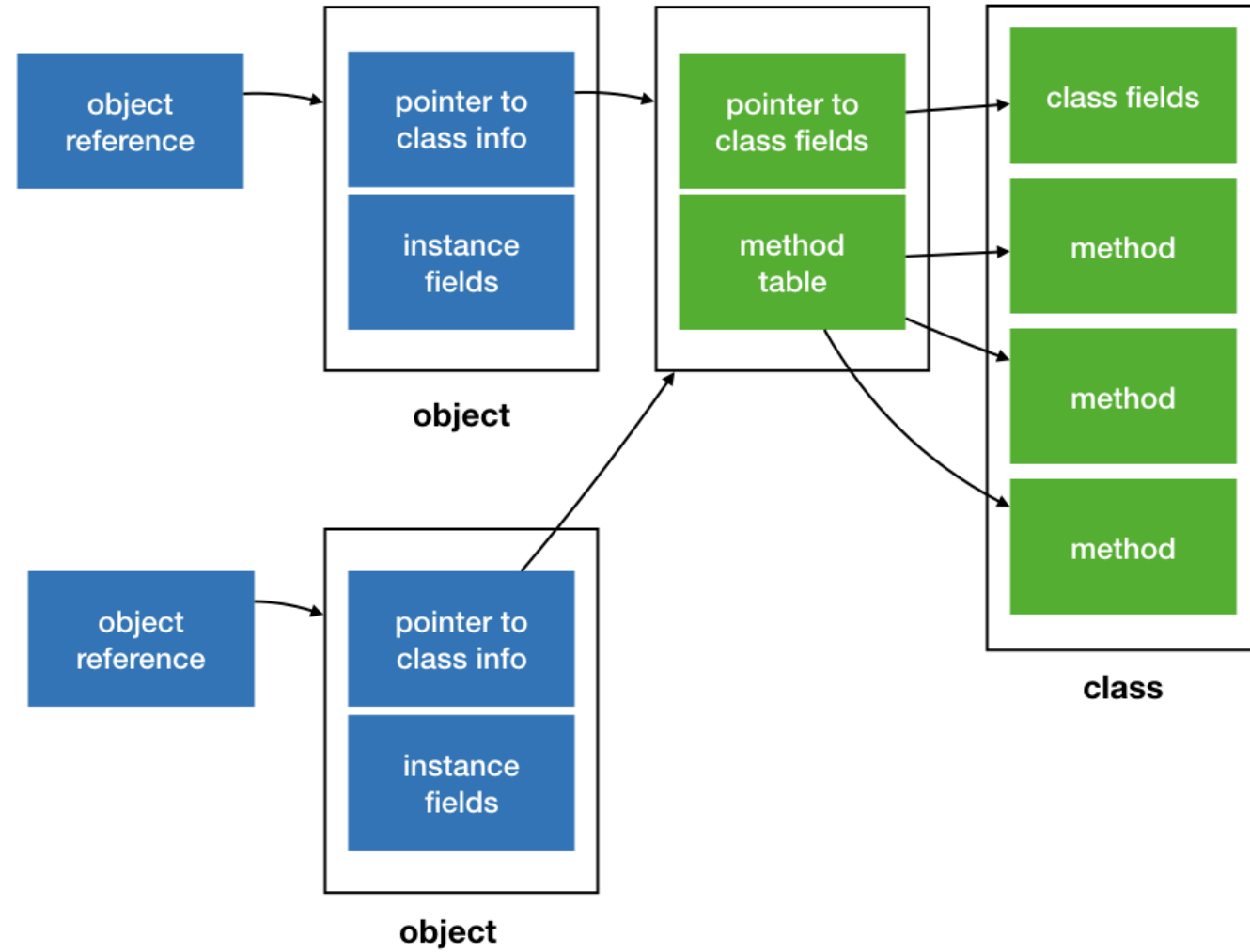
 Every instance variable belongs to an instance represented by this, which is normally omitted

new Temp(8, 10); // invokes parameterized constructor 3

```
Temp(int x, int y)
{
    //invokes parameterized constructor 2
    this(5);
    System.out.println(x * y);
}
```

```
Temp(int x)
{
    //invokes default constructor
    this();
    System.out.println(x);
}
```

```
Temp()
{
    System.out.println("default");
}
```



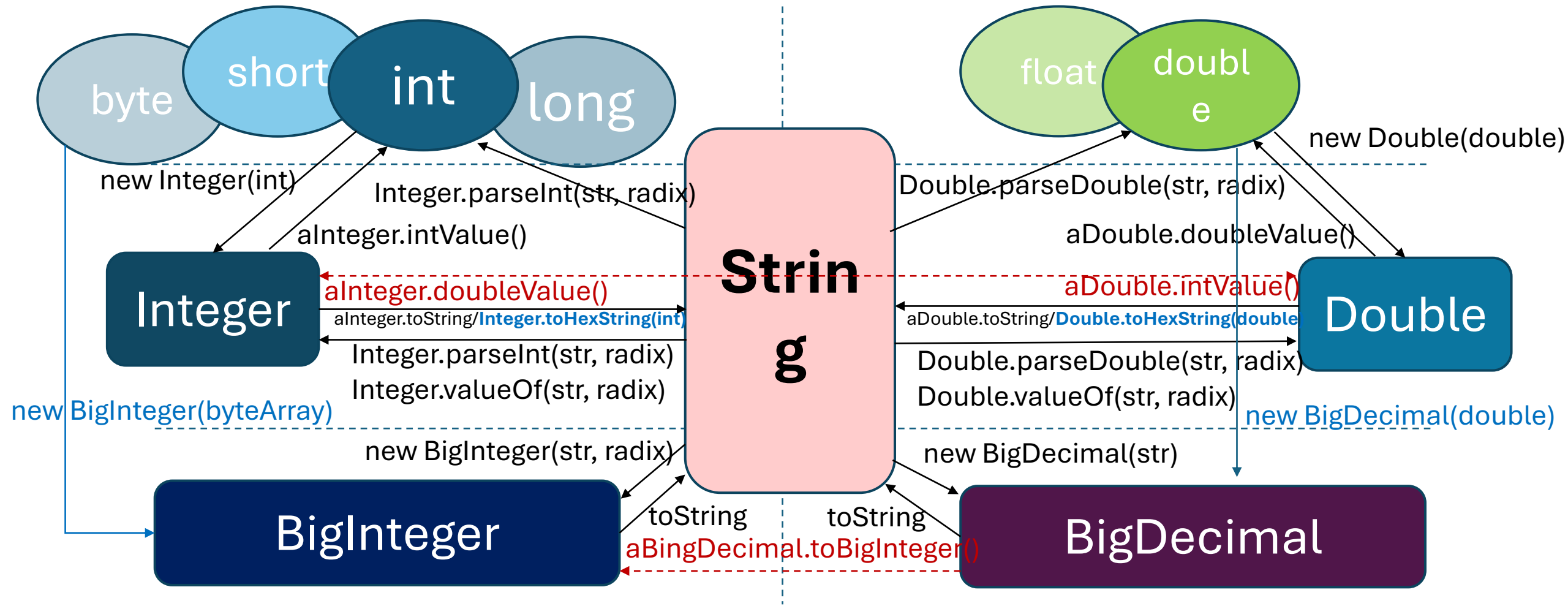


Math Processing I:

Data/Object Type Conversion

Lecture 14

Map for Java Number Space



The Integer and Double Classes

java.lang.Integer
<code>-value: int</code> <code>+<u>MAX VALUE: int</u></code> <code>+<u>MIN VALUE: int</u></code>
<code>+Integer(value: int)</code> <code>+Integer(s: String)</code> <code>+byteValue(): byte</code> <code>+shortValue(): short</code> <code>+intValue(): int</code> <code>+longVlaue(): long</code> <code>+floatValue(): float</code> <code>+doubleValue():double</code> <code>+compareTo(o: Integer): int</code> <code>+toString(): String</code> <code>+<u>valueOf(s: String): Integer</u></code> <code>+<u>valueOf(s: String, radix: int): Integer</u></code> <code>+<u>parseInt(s: String): int</u></code> <code>+<u>parseInt(s: String, radix: int): int</u></code>

java.lang.Double
<code>-value: double</code> <code>+<u>MAX VALUE: double</u></code> <code>+<u>MIN VALUE: double</u></code>
<code>+Double(value: double)</code> <code>+Double(s: String)</code> <code>+byteValue(): byte</code> <code>+shortValue(): short</code> <code>+intValue(): int</code> <code>+longVlaue(): long</code> <code>+floatValue(): float</code> <code>+doubleValue():double</code> <code>+compareTo(o: Double): int</code> <code>+toString(): String</code> <code>+<u>valueOf(s: String): Double</u></code> <code>+<u>valueOf(s: String, radix: int): Double</u></code> <code>+<u>parseDouble(s: String): double</u></code> <code>+<u>parseDouble(s: String, radix: int): double</u></code>

Numeric Wrapper Class Constants

- Each numerical wrapper class has the constants `MAX_VALUE` and `MIN_VALUE`. `MAX_VALUE` represents the maximum value of the corresponding primitive data type.
- For `Byte`, `Short`, `Integer`, and `Long`, `MIN_VALUE` represents the minimum byte, short, int, and long values. For `Float` and `Double`, `MIN_VALUE` represents the minimum *positive* float and double values.
- The following statements display the maximum integer (2,147,483,647), the minimum positive float (1.4E-45), and the maximum double floating-point number (1.79769313486231570e+308d).

Conversion Methods

- Each numeric wrapper class implements the abstract methods `doubleValue`, `floatValue`, `intValue`, `longValue`, and `shortValue`, which are defined in the `Number` class.
- These methods “convert” objects into primitive type values.

The Static valueOf Methods

- The numeric wrapper classes have a useful class method, `valueOf(String s)`. This method creates a new object initialized to the value represented by the specified string. For example:

```
Double doubleObject = Double.valueOf("12.4");
```

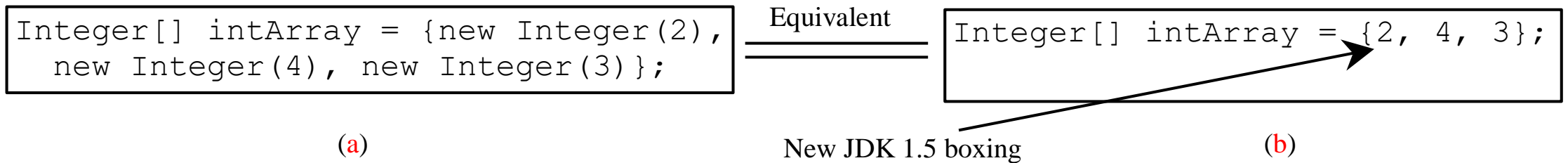
```
Integer integerObject = Integer.valueOf("12");
```

The Methods for Parsing Strings into Numbers

- You have used the `parseInt` method in the `Integer` class to parse a numeric string into an `int` value and the `parseDouble` method in the `Double` class to parse a numeric string into a `double` value.
- Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value.

Automatic Conversion Between Primitive Types and Wrapper Class Types

JDK 1.5 allows primitive type and wrapper classes to be converted automatically. For example, the following statement in (a) can be simplified as in (b):



Integer[] intArray = {1, 2, 3};
System.out.println(intArray[0] + intArray[1] + intArray[2]);

Unboxing

8 java.math.BigInteger

BigInteger (byte[] val)
BigInteger (String val)
BigInteger (int signum, byte[] magnitude)
BigInteger (String val, int radix)
BigInteger (int numBits, Random rnd)
BigInteger (int bitLength, int certainty, Random rnd)

Static Methods

BigInteger **probablePrime** (int bitLength, Random rnd)
BigInteger **valueOf** (long val)

Accessors + Collectors

int **getLowestSetBit** ()
boolean **isProbablePrime** (int certainty)
BigInteger **setBit** (int n)
BigInteger **add** (BigInteger val)

Object

boolean **equals** (Object x)
int **hashCode** ()
String **toString** ()

Other Public Methods

BigInteger **abs** ()
BigInteger **and** (BigInteger val)
BigInteger **andNot** (BigInteger val)
int **bitCount** ()
int **bitLength** ()
BigInteger **clearBit** (int n)
int **compareTo** (BigInteger val)
int **compareTo** (Object o)
BigInteger **divide** (BigInteger val)
BigInteger[] **divideAndRemainder** (BigInteger val)
BigInteger **flipBit** (int n)
BigInteger **gcd** (BigInteger val)
BigInteger **max** (BigInteger val)
BigInteger **min** (BigInteger val)
BigInteger **mod** (BigInteger m)
BigInteger **modInverse** (BigInteger m)
BigInteger **modPow** (BigInteger exponent, BigInteger m)
BigInteger **multiply** (BigInteger val)
BigInteger **negate** ()
BigInteger **not** ()
BigInteger **or** (BigInteger val)
BigInteger **pow** (int exponent)
BigInteger **remainder** (BigInteger val)
BigInteger **shiftLeft** (int n)
BigInteger **shiftRight** (int n)
int **signum** ()
BigInteger **subtract** (BigInteger val)
boolean **testBit** (int n)
byte[] **toByteArray** ()
String **toString** (int radix)
BigInteger **xor** (BigInteger val)

BigInteger **ZERO**, **ONE**

8 java.math.BigDecimal

BigDecimal (String val)
BigDecimal (double val)
BigDecimal (BigInteger val)
BigDecimal (BigInteger unscaledVal, int scale)

Static Methods

BigDecimal **valueOf** (long val)
BigDecimal **valueOf** (long unscaledVal, int scale)

Accessors + Collectors

BigDecimal **setScale** (int scale)
BigDecimal **setScale** (int scale, int roundingMode)
BigDecimal **add** (BigDecimal val)

Object

boolean **equals** (Object x)
int **hashCode** ()
String **toString** ()

Other Public Methods

BigDecimal **abs** ()
int **compareTo** (BigDecimal val)
int **compareTo** (Object o)
BigDecimal **divide** (BigDecimal val, int roundingMode)
BigDecimal **divide** (BigDecimal val, int scale, int roundingMode)
BigDecimal **max** (BigDecimal val)
BigDecimal **min** (BigDecimal val)
BigDecimal **movePointLeft** (int n)
BigDecimal **movePointRight** (int n)
BigDecimal **multiply** (BigDecimal val)
BigDecimal **negate** ()
int **scale** ()
int **signum** ()
BigDecimal **subtract** (BigDecimal val)
BigInteger **toBigInteger** ()
BigInteger **unscaledValue** ()

int **ROUND_UP**, **ROUND_DOWN**, **ROUND_CEILING**,
ROUND_FLOOR, **ROUND_HALF_UP**,
ROUND_HALF_DOWN, **ROUND_HALF_EVEN**,
ROUND_UNNECESSARY

BigInteger and BigDecimal

- If you need to compute with very large integers or high precision floating-point values, you can use the BigInteger and BigDecimal classes in the java.math package.
- Both are *immutable*.
- Both extend the Number class and implement the Comparable interface.

BigInteger and BigDecimal

(Data Class with Operations)

```
BigInteger a = new  
BigInteger("9223372036854775807");  
BigInteger b = new BigInteger("2");  
BigInteger c = a.multiply(b); // 9223372036854775807 *  
2  
BigDecimal a = new BigDecimal(1.0);  
BigDecimal b = new BigDecimal(3);  
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);  
System.out.println(c);
```