AP Computer Science B
Java Object-Oriented Programming [Ver. 2.0]

Unit 4: Object-Oriented Design

WEEK 3: CHAPTER 10 CLASSES AND OBJECTS (PART 3: DATA ENCAPSULATION)

DR. ERIC CHOU IEEE SENIOR MEMBER

Objectives

- Class Design Style
- Class Data Encapsulation
- Immutable Class
- **this** Reference
- Fuel Efficiency Series of Assignments

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.

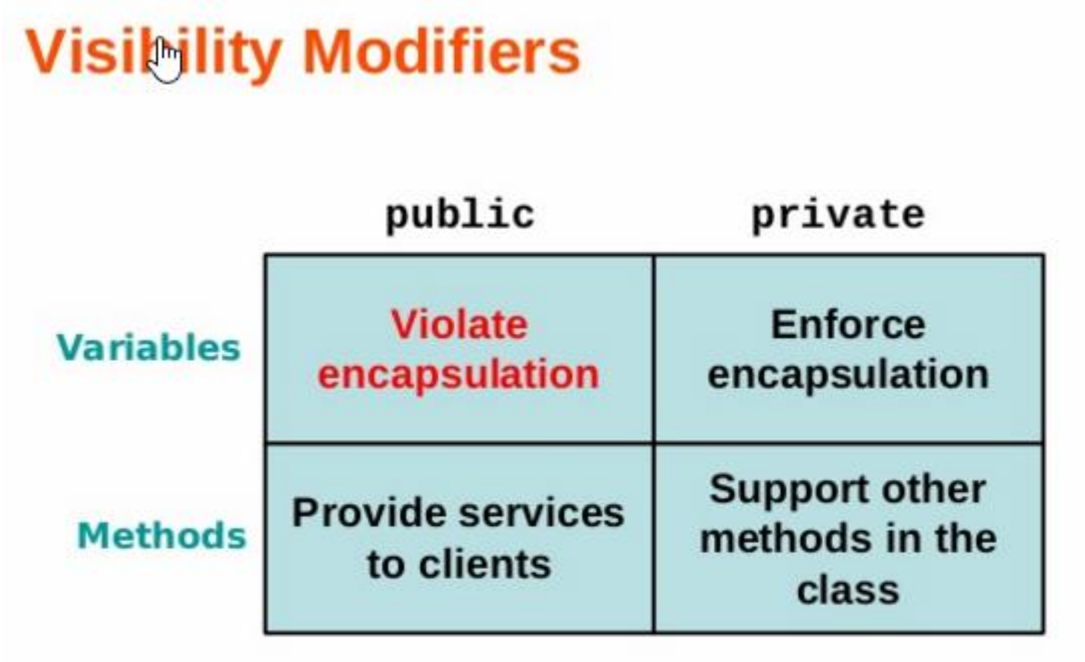
Classes and Objects (3)

Classes and Objects Design Styles

Classification by Styles

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

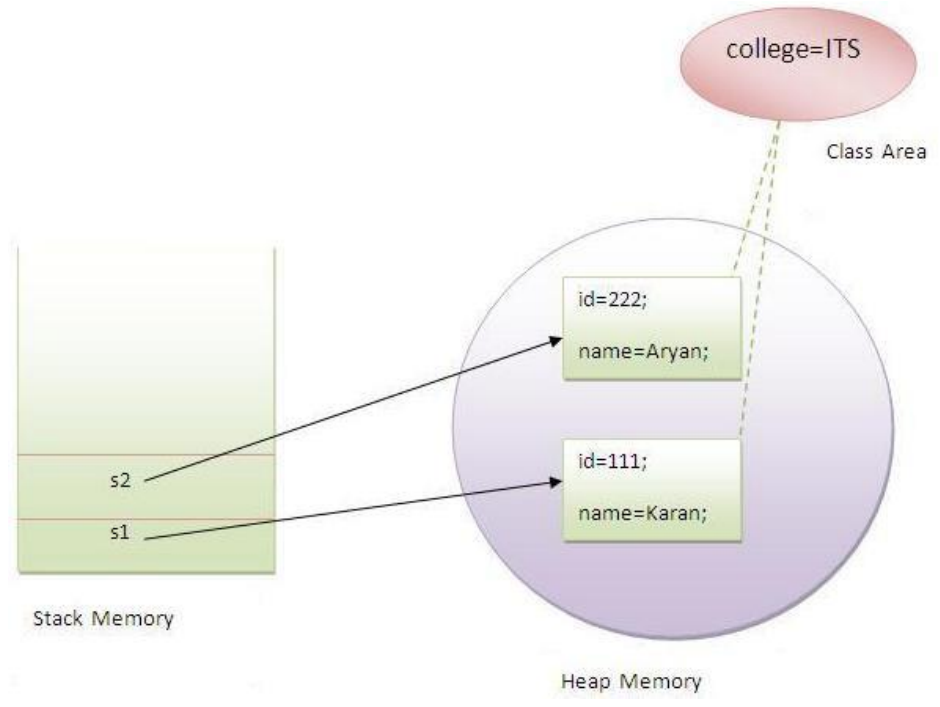
Classes and Objects (4)

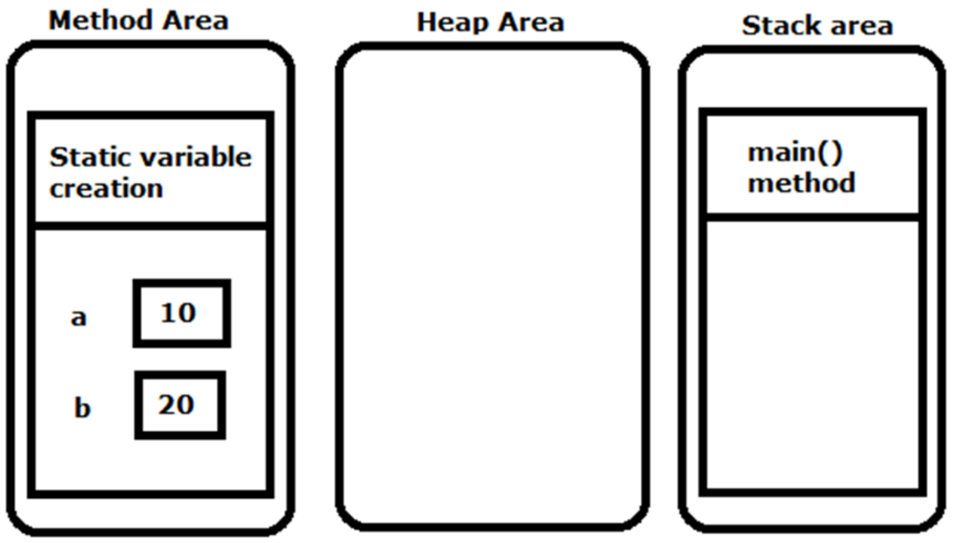


Visibility Modifiers

	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





JVM Architecture

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

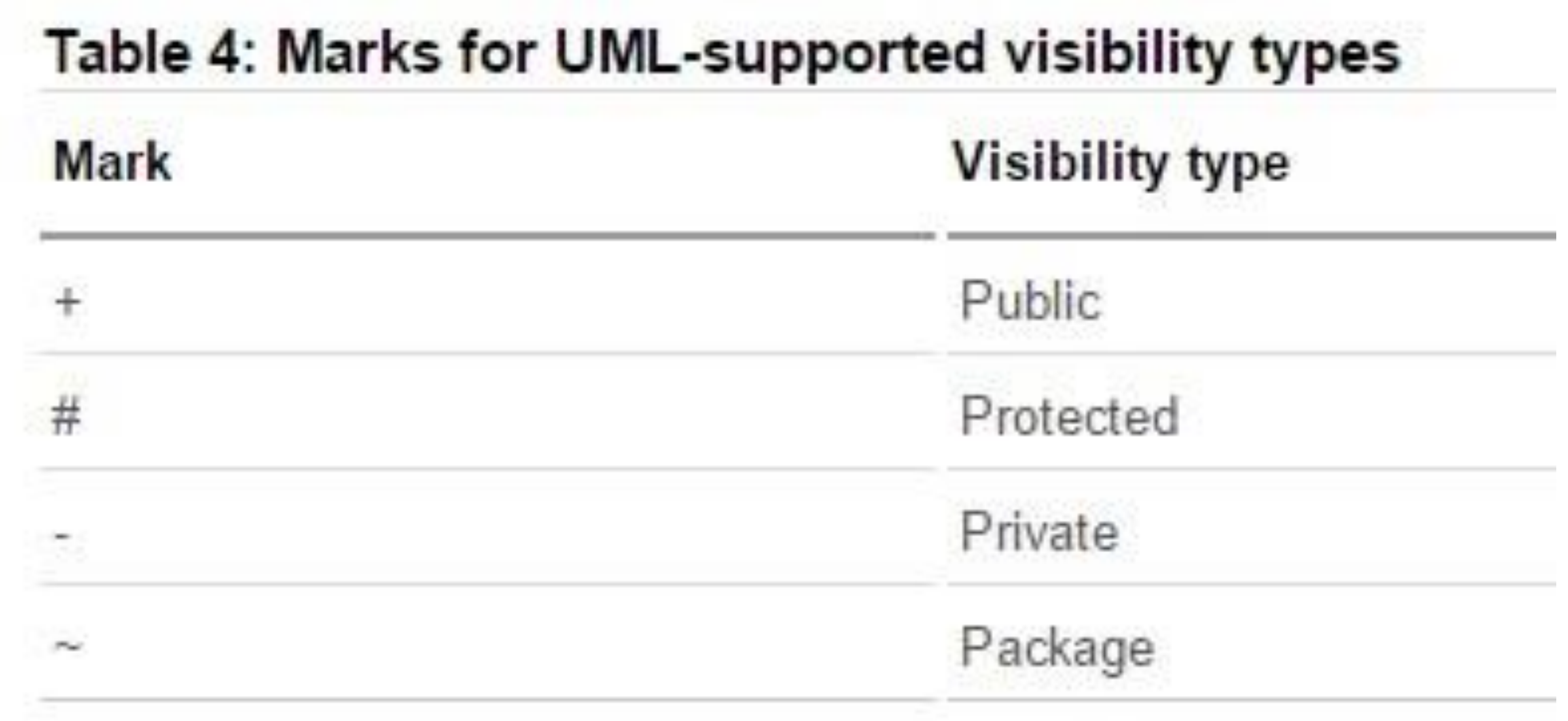


Table 4: Marks for UML-supported visibility types

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()	~CircleWithStaticMembers()
~CircleWithStaticMemebrs(double newRadius)	~CircleWithStaticMemebrs(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()
~CircleWithStaticMemebrs(double 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;
}
```

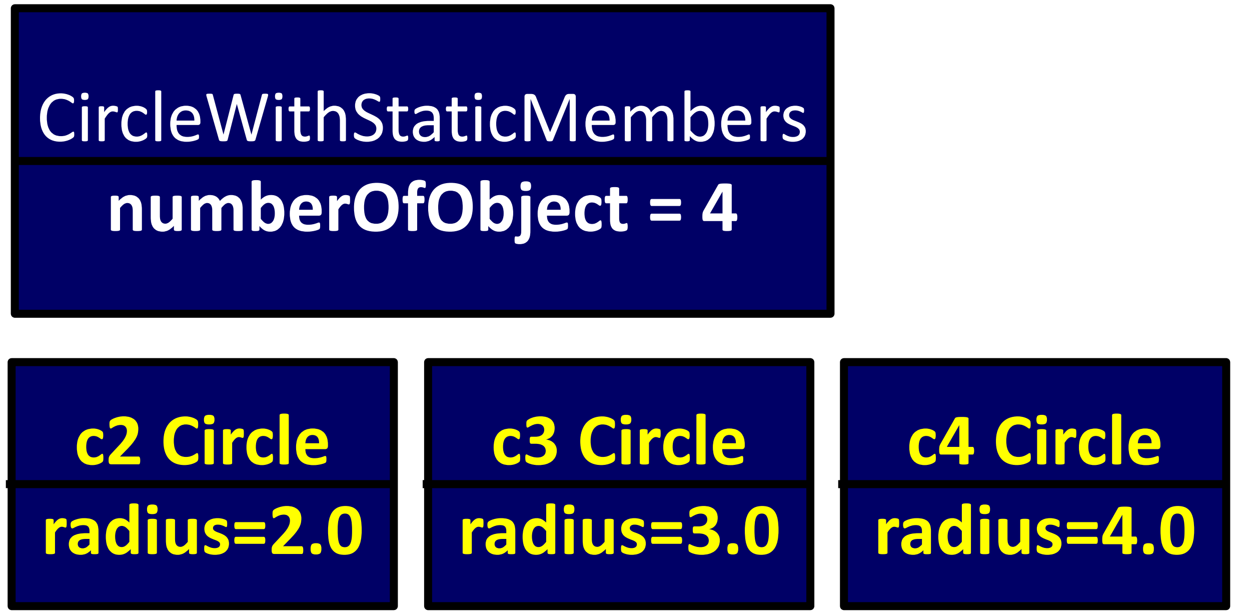
Class Variable Update

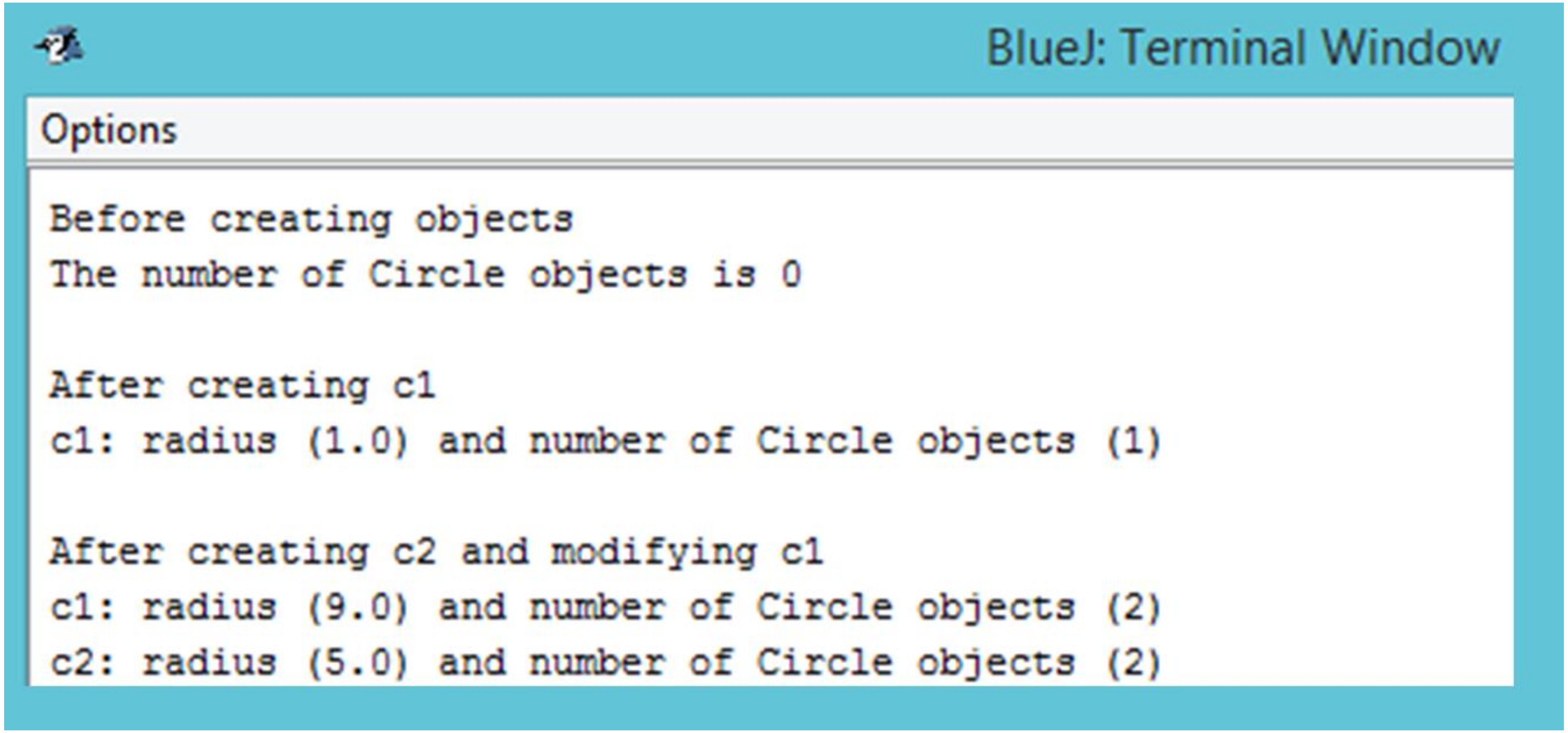
ClassName.variable, ClassName.update()

Static Variables/Methods Stay With Class:

Available when JVM load the class.

(Instance variable stays in heap temporarily)





```
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)
```

Demo Program: **CircleWithStaticMembers.java. TestCircleWithStaticMembers.java**

Demonstration Program

CIRCLEWITHSTATICMEMBERS.JAVA
TESTCIRCLEWITHSTATICMEMBERS.JAVA

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

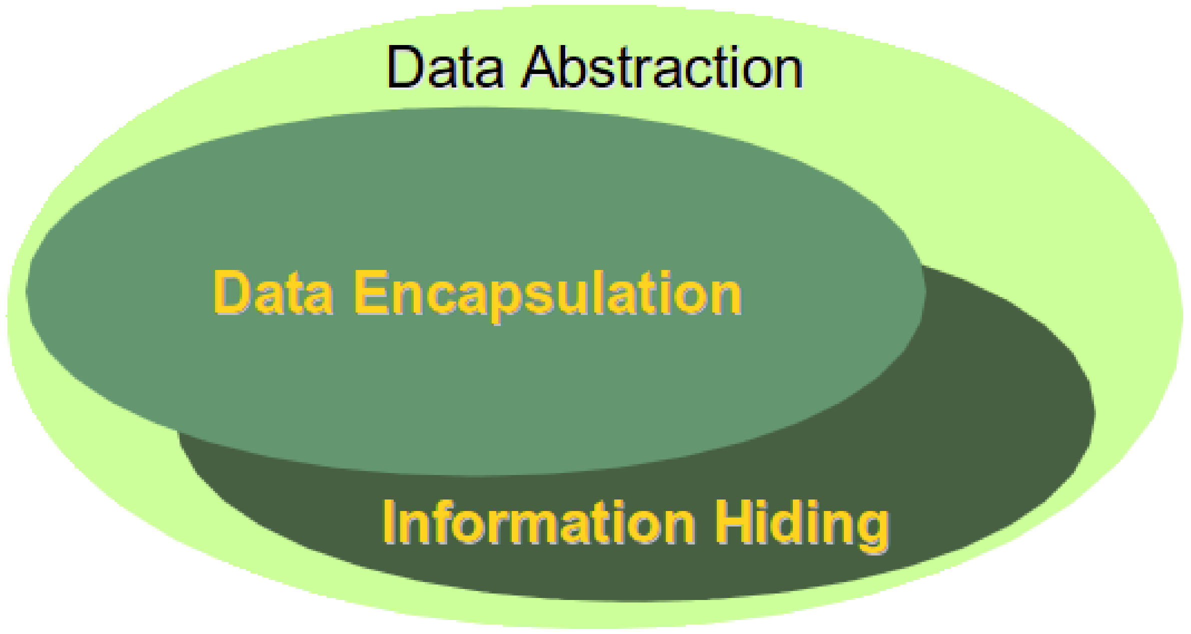
Private Data Fields and Public Accessors and Mutators

- 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?

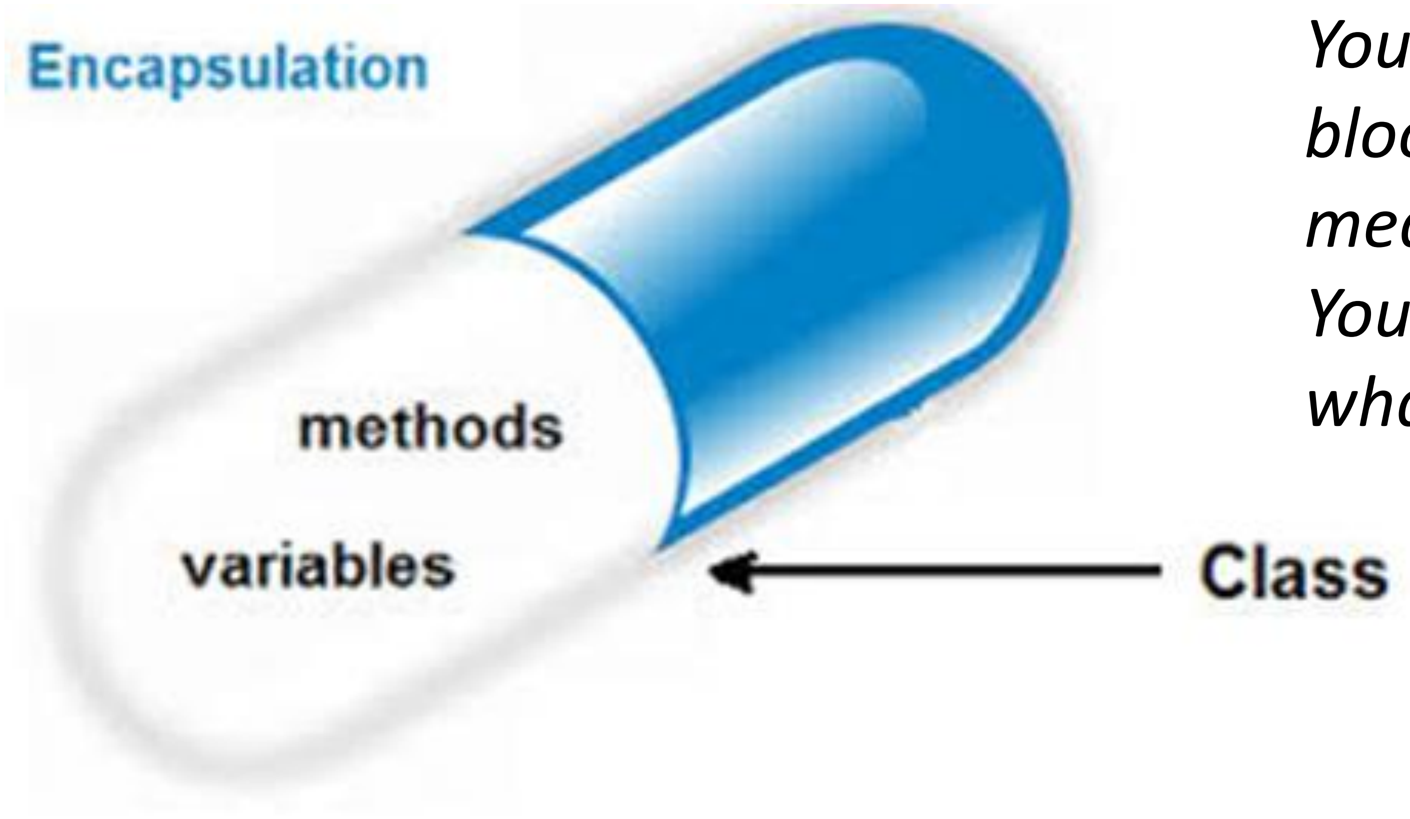
- To protect data.
- To make class easy to maintain.



Class/Methods are also considered abstraction.

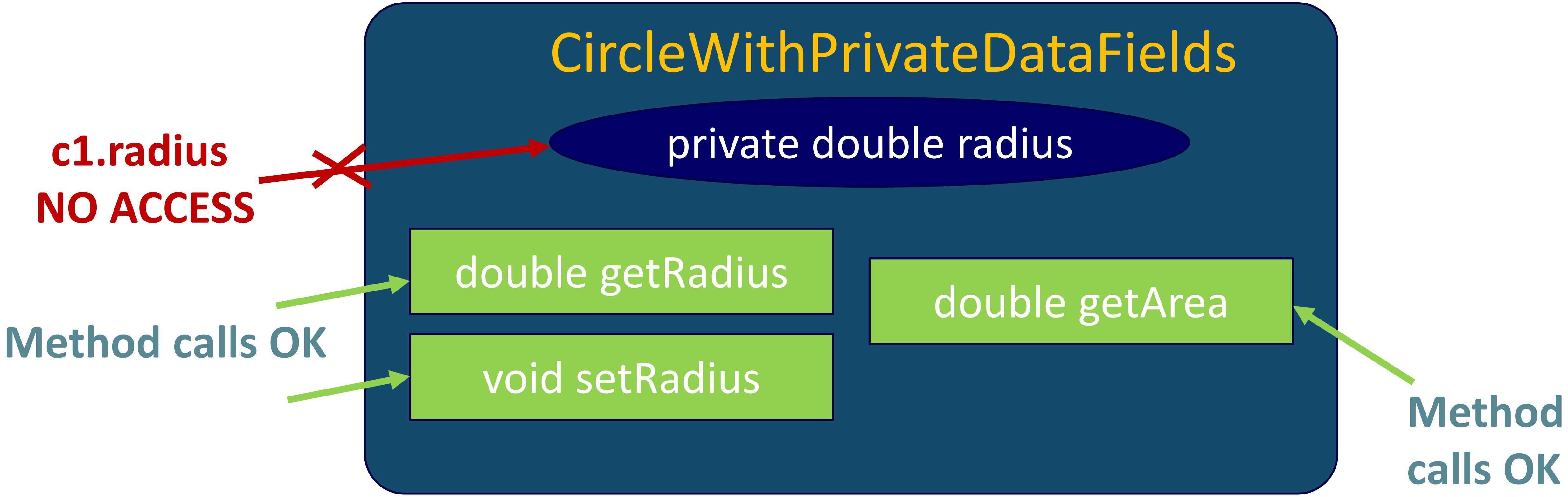
Constants, Static Variables, … Overloading are also considered as Information hiding.

Data Field Encapsulation
Why Data Fields Should Be private?

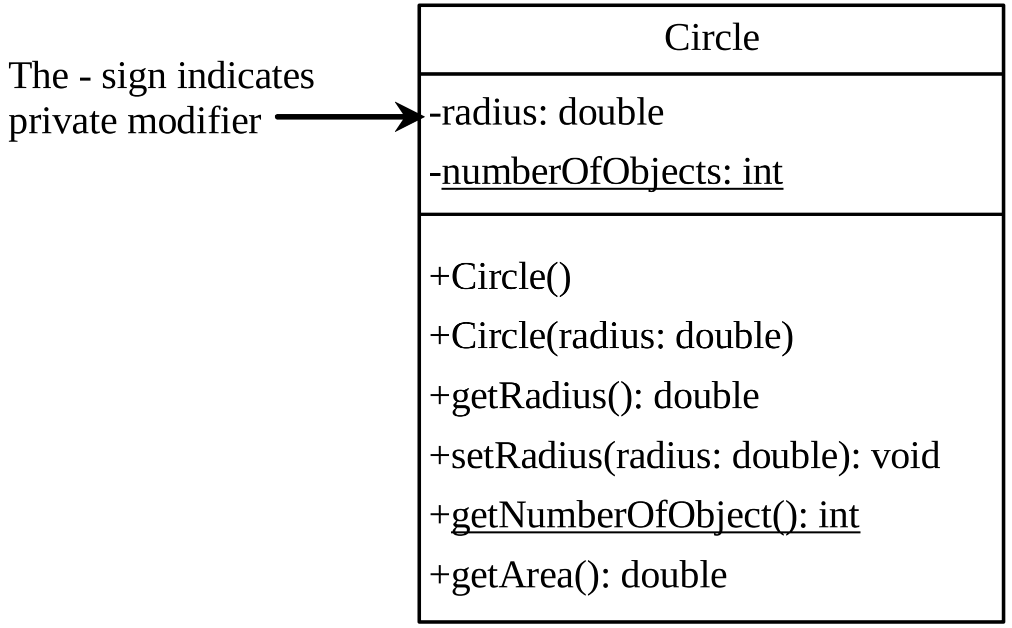


You just need to know it is blood pressure control medicine.
You do not need to know what chemical formula it is.

Encapsulation Using Private Data Fields



Example of Data Field Encapsulation



Circle	
-radius: double	The radius of this circle (default: 1.0).
-numberOfObjects: int	The number of circle objects created.
+Circle()	Constructs a default circle object.
+Circle(radius: double)	Constructs a circle object with the specified radius.
+getRadius(): double	Returns the radius of this circle.
+setRadius(radius: double): void	Sets a new radius for this circle.
+getNumberOfObject(): int	Returns the number of circle objects created.
+getArea(): double	Returns the area of this circle.

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 getNumberOfObjects()
Non-static Members (Instance Members)
Properties
- double radius
Constructors
+ CircleWithStaticMembers()
+ CircleWithStaticMemebrs(double newRadius)
Methods
+ double getRadius()
+ void setRadius()
+ double getArea()

Demonstration Program

CIRCLEWITHPRIVATEDATAFIELDS.JAVA

TESTCIRCLEWITHPRIVATEDATAFIELDS.JAVA

Assignment

FUEL 2 ASSIGNMENT

SUBMIT YOUR PROGRAM TO MOODLE COURSE UPLOAD LINK

Classes and Objects (3): Data Encapsulation II

Passing Objects to Methods

LECTURE 3

JVM

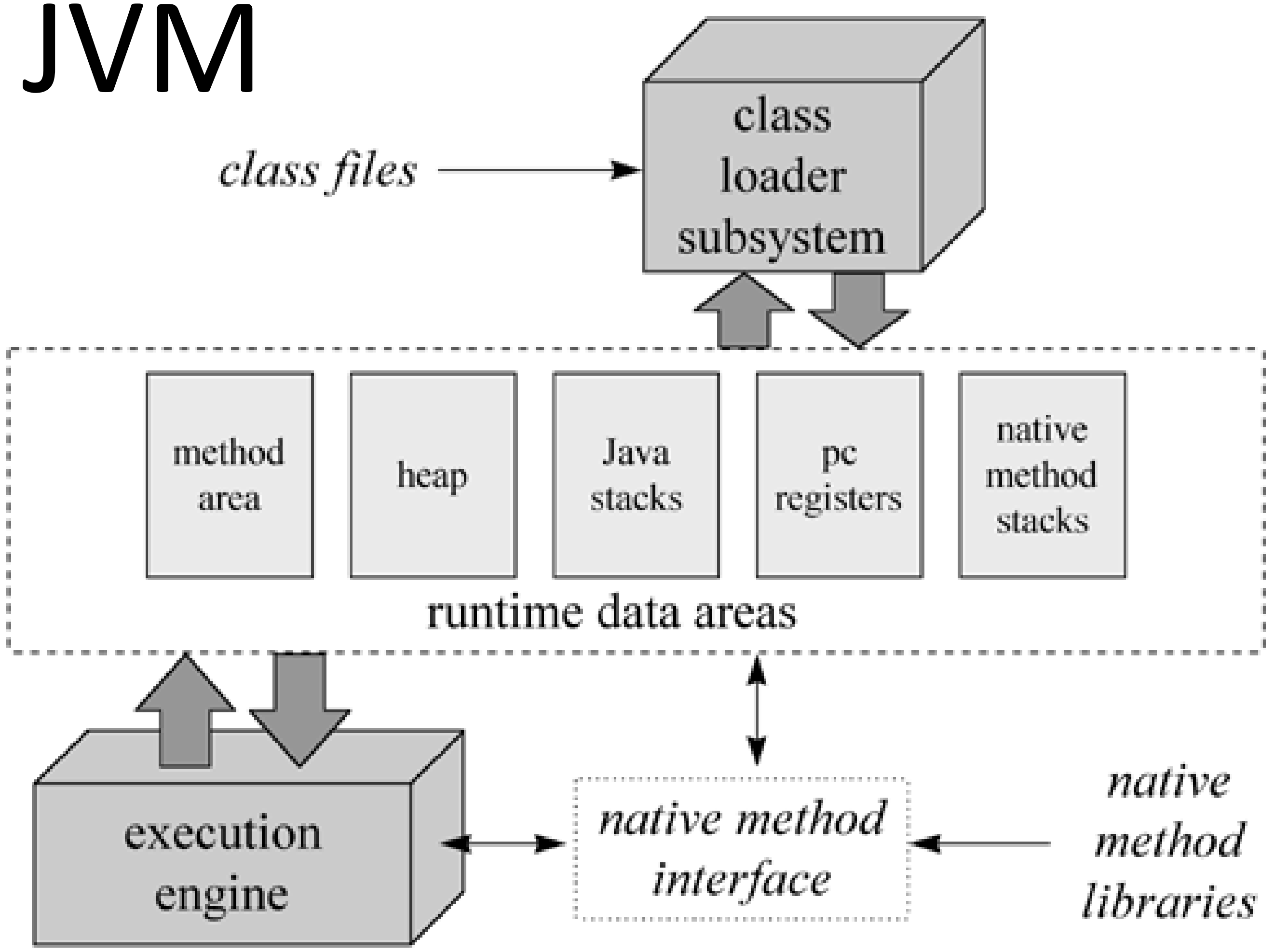
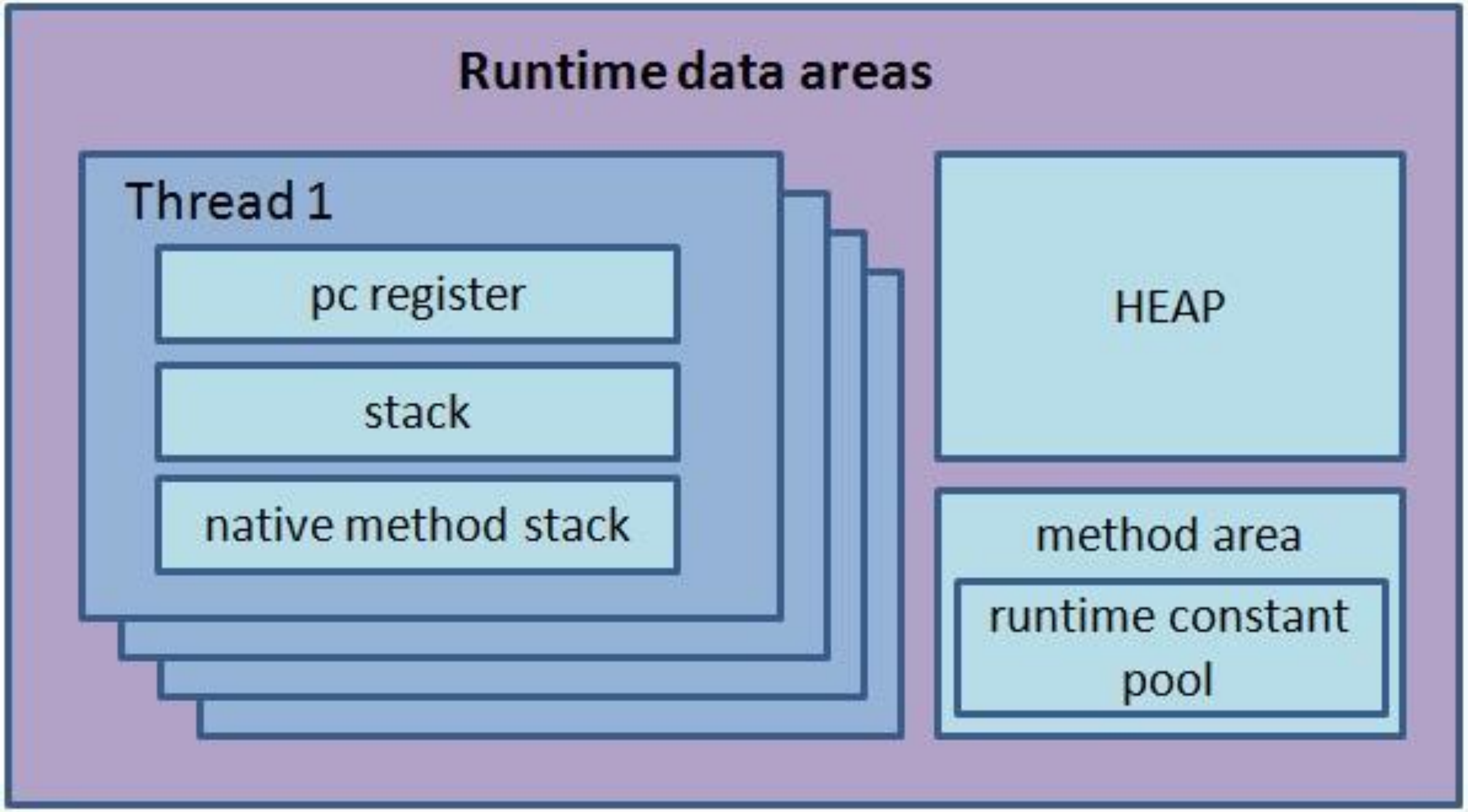
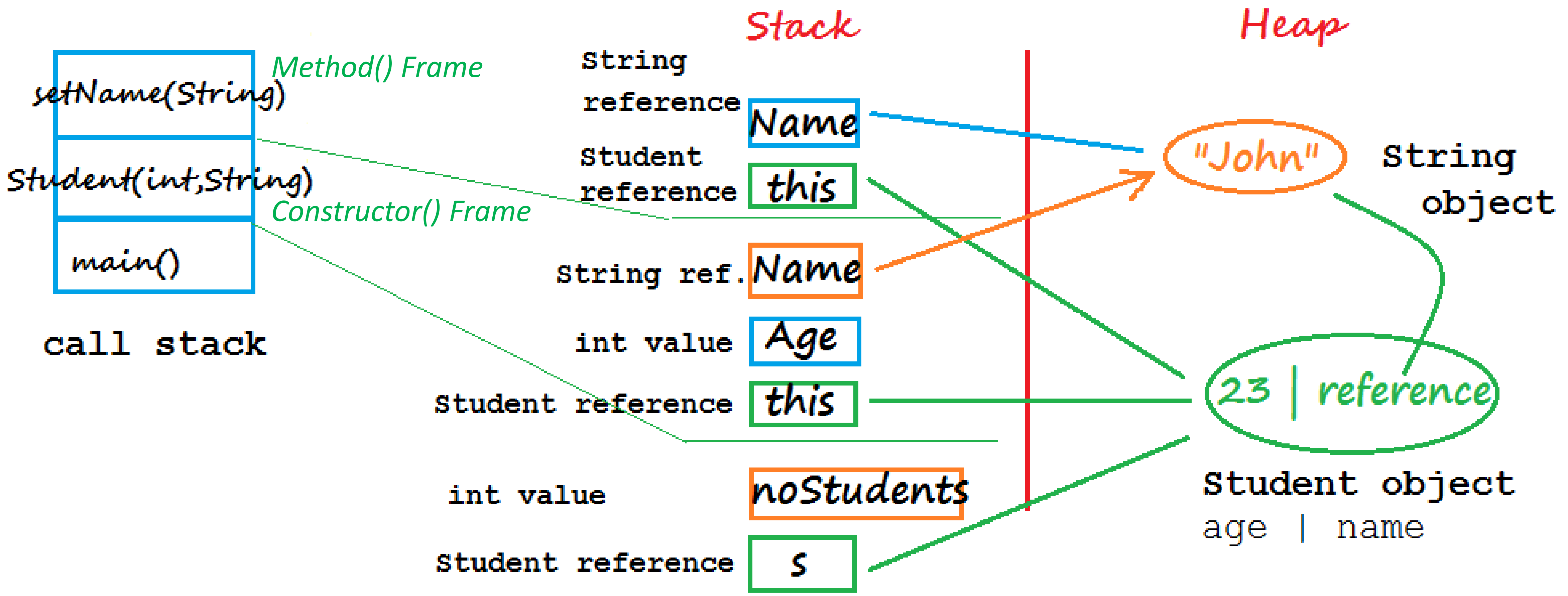


Figure 5-1. The internal architecture of the Java Virtual Machine.


Runtime data areas
Thread 1
pc register
stack
native method stack
HEAP
method area
runtime constant pool

Memory Allocation

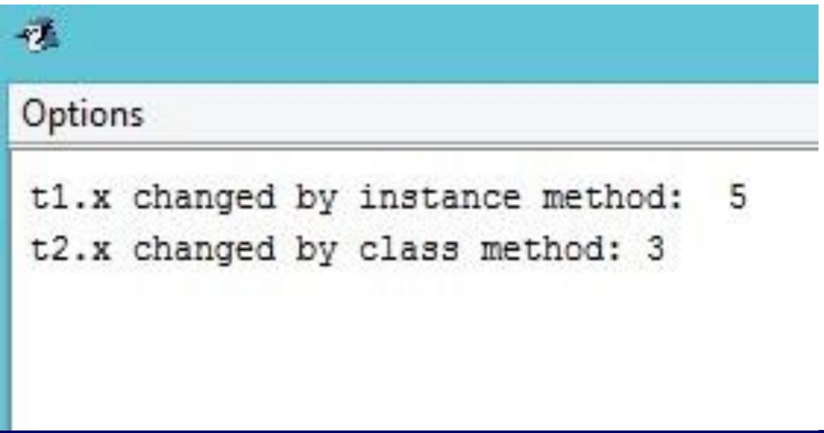


Two Ways to Update a Data Field from Other Class

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

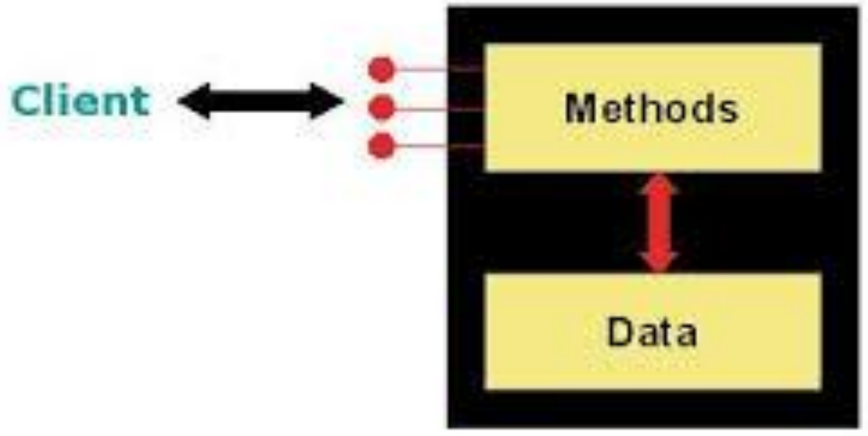
```
t1.x changed by instance method:  5
t2.x changed by class method: 3
```

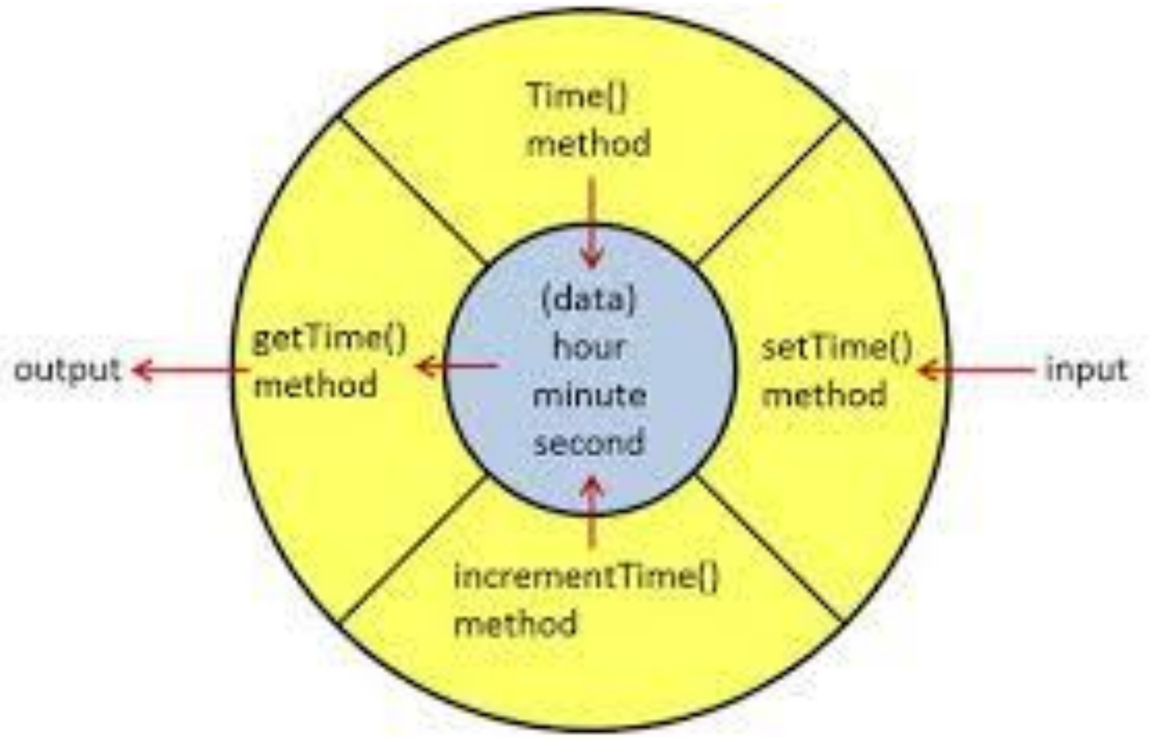
Demonstration Program

INT2.JAVA + TESTINT2.JAVA

First Way: Accessor/Mutator Methods

For Encapsulated Objects

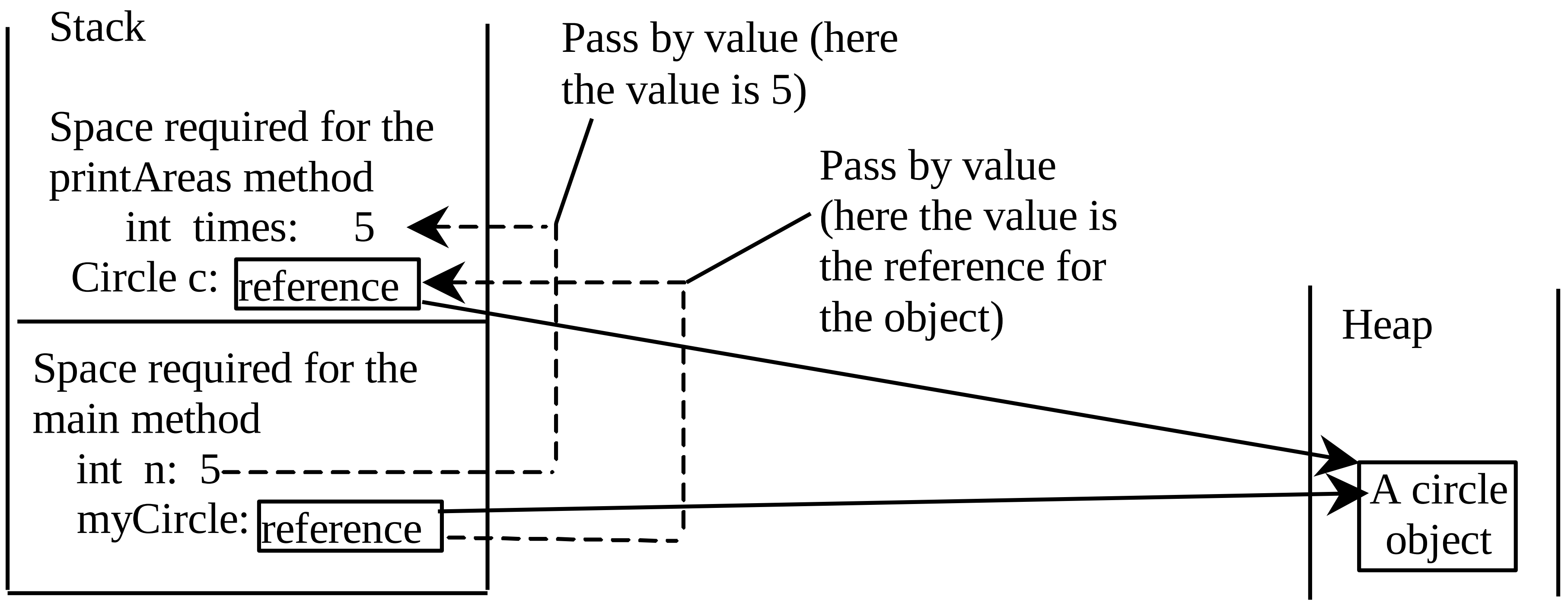




Passing Objects to Methods

- 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.


Stack
Space required for the printAreas method
int times: 5
Circle c: reference
Space required for the main method
int n: 5
myCircle: reference
Pass by value (here the value is 5)
Pass by value (here the value is the reference for the object)
Heap
A circle object

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

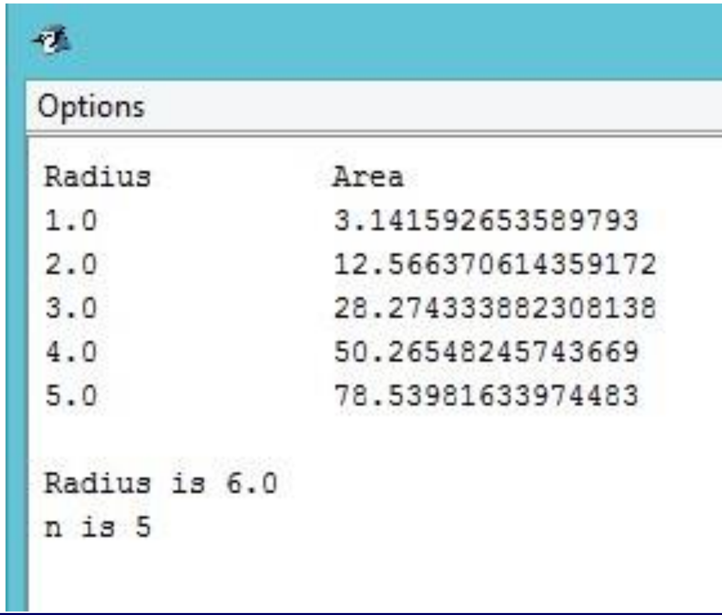
PassingObjects.java

```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);
}
```

```java
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--;
  }
}
```


Options

```
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

(a) is equivalent to (b)

```
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)

```
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

Chapter Project:

LECTURE 6

Catapult

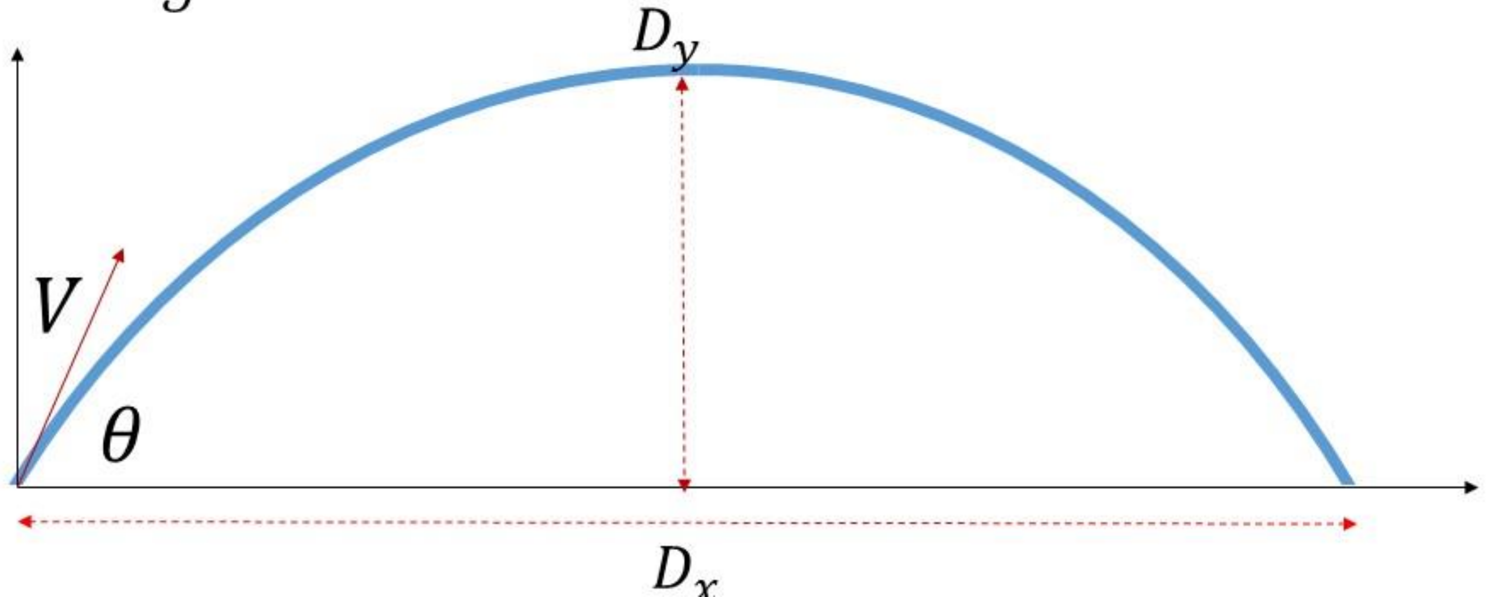
Instructions: Write a program to calculate the trajectory of a projectile based on launch angles and launch velocities.

1. Create a **CatapultTester** and a **Catapult** class in the project folder.
2. Review the information below about calculating projectile trajectories.
3. Work out several answers with pencil, paper, and calculator first, before attempting to write the program.
4. You will need to study the toRadians()and the sin()methods in the Java API for the Math class.
5. Calculate the distance an object can be catapulted for at least six launch angles and seven launch speeds (see expected output). Store your data in a two dimensional array. Remember to covert the distance from meter to feet.

Projectile:

$$V = [V_x, V_y];$$
$$V_x = V\cos(\theta)$$
$$V_y = V\sin(\theta)$$
$$D_x = V_x \times 2t$$
$$t = \frac{V_y}{g}$$

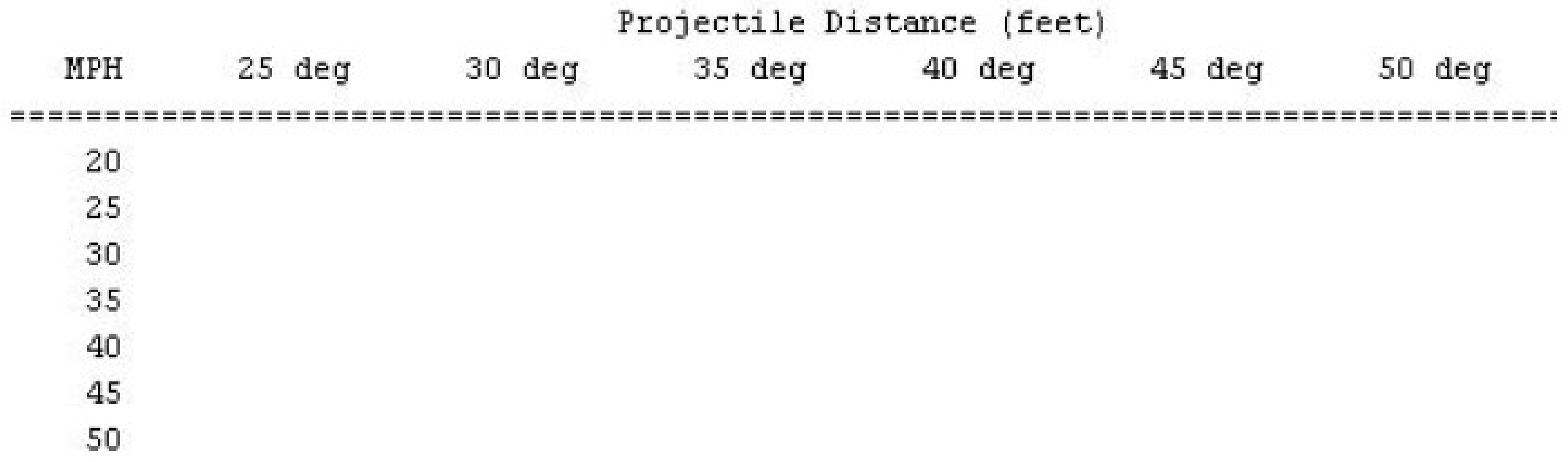
$$D_x = V_x \times 2t = \frac{2\,V^2\cos(\theta)\,\sin(\theta)}{g} = \frac{V^2\sin(2\theta)}{g}$$



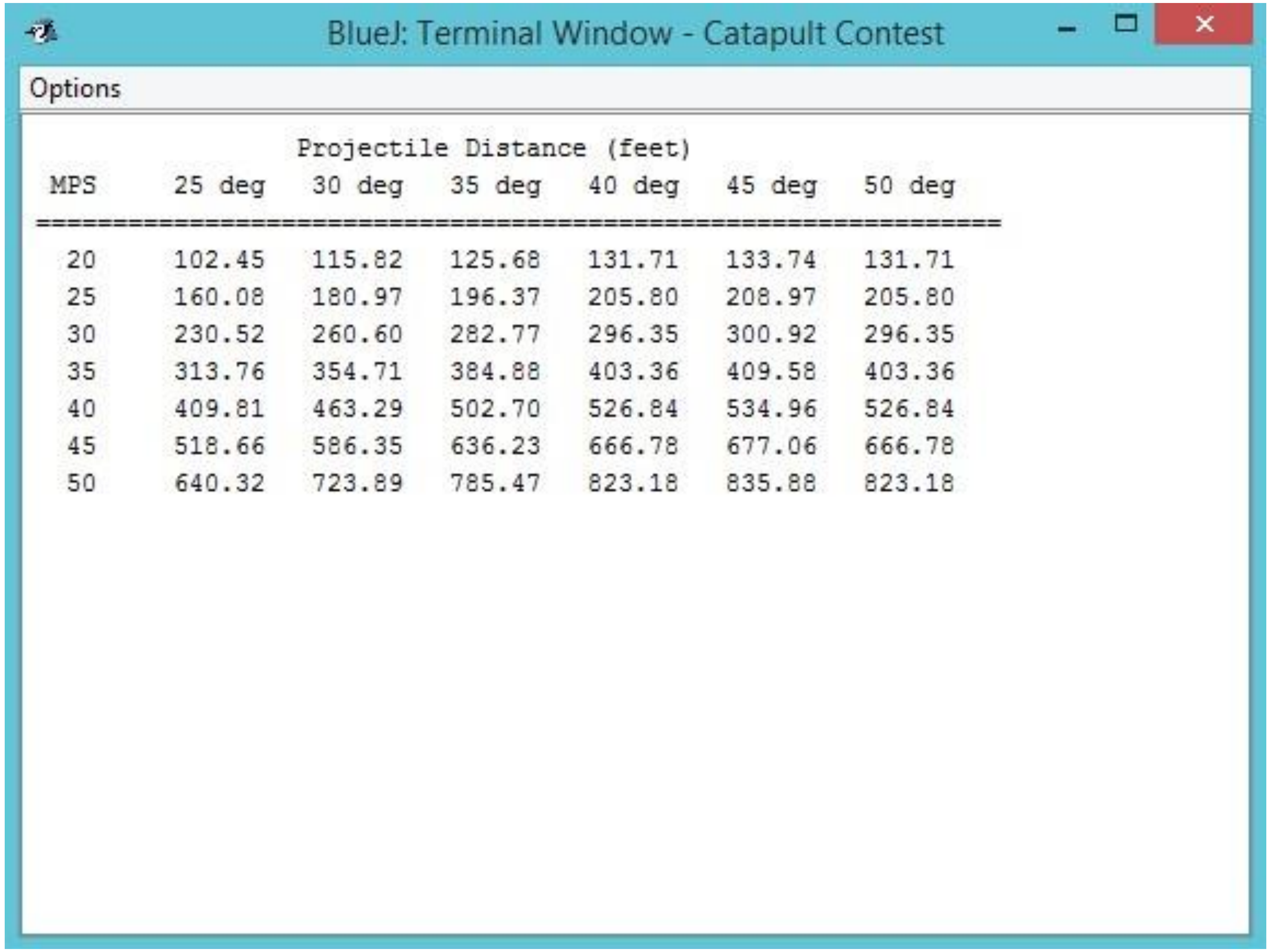
Expected Output

Expected Output:

- When your program runs correctly, the format of the output table should resemble the following, but with the appropriate data for each row and column.
- You may use appropriate angles and velocities of your choice.



```
                                Projectile Distance (feet)
   MPH      25 deg      30 deg      35 deg      40 deg      45 deg      50 deg
=================================================================================
    20
    25
    30
    35
    40
    45
    50
```



```
                 Projectile Distance (feet)
 MPS      25 deg   30 deg   35 deg   40 deg   45 deg   50 deg
===============================================================
  20      102.45   115.82   125.68   131.71   133.74   131.71
  25      160.08   180.97   196.37   205.80   208.97   205.80
  30      230.52   260.60   282.77   296.35   300.92   296.35
  35      313.76   354.71   384.88   403.36   409.58   403.36
  40      409.81   463.29   502.70   526.84   534.96   526.84
  45      518.66   586.35   636.23   666.78   677.06   666.78
  50      640.32   723.89   785.47   823.18   835.88   823.18
```

Summary

LECTURE 7

Object-Oriented Programming

Package

Module

Classes

Interfaces

Abstract Classes

enum

Statics

Objects

Functions

Container

Constants

Access Modifiers

Visibility

public

protected

default

private

Encapsulation

Information Hiding

Wrapper Classes

Immutable

Relations

has_A Composition

Many to 1 Aggregation

Many to Many Association

Coherence

Inheritance

Is_A Inheritance

this

super

Multiple Inheritance

Polymorphism

Overloading

Generics

Overriding

Generic Container

Dynamic Binding

Generic Method

Polymorphic Methods

Object Generic