

# AP Computer Science B

Java Object-Oriented Programming [Ver. 3.0]

## Unit 4: Object-Oriented Programming



CHAPTER 10A: CLASSES AND OBJECTS

DR. ERIC CHOU

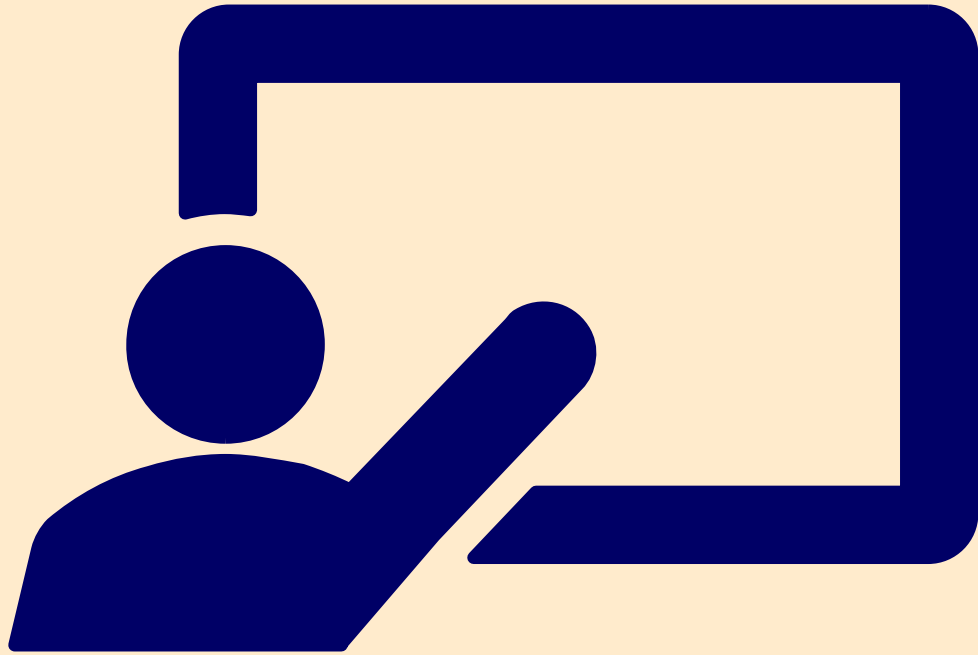
IEEE SENIOR MEMBER



# Objectives

---

- Class Definition – Data Fields, Constructor, and Member Methods
- Object Declaration – Instantiation and Initialization
- Data Fields
- Derived Data Fields
- Inherited Standard Functions
- Static Members
- Class, Reference, and Objects



# Motivation

---

## LECTURE 1



# Introduction to Object-Oriented Programming

---

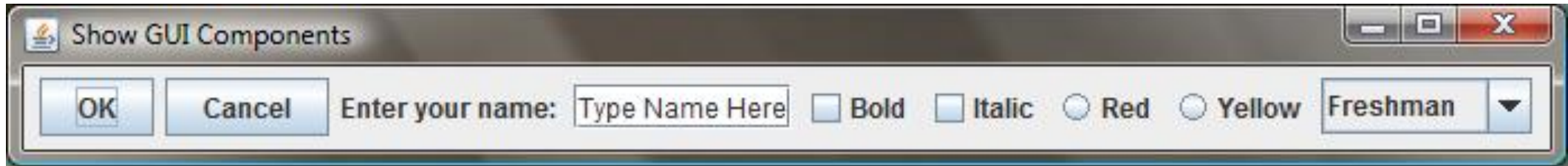
- After learning the preceding chapters, you are capable of solving many programming problems using selections, loops, methods, and arrays. However, these Java features are not sufficient for developing graphical user interfaces and large-scale software systems.
- Suppose you want to develop a graphical user interface as shown below. How do you program it?



# Introduction to Object-Oriented Programming

---

- Without Object-Oriented Programming, things shall still work. Why we need Object-Oriented Programming?





# Motivation for Object-Oriented Programming

---

- More Compatible for Event-Driven Programming
- More Manageable for GUI Components
- More Organized Data and Methods related to a Certain Objects

## **Replacing:**

- (1) Library
- (2) Data Records
- (3) Event-Loop (Execution Flow)
- (4) Thread Execution Control



# Object-Oriented Programming Concepts

---

- Object-oriented programming (OOP) involves programming using objects. An **object** represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects.
- An object has a unique identity, state, and behaviors. The *state* of an object consists of a set of **data fields** (also known as **properties**) with their current values. The *behavior* of an object is defined by a set of methods.

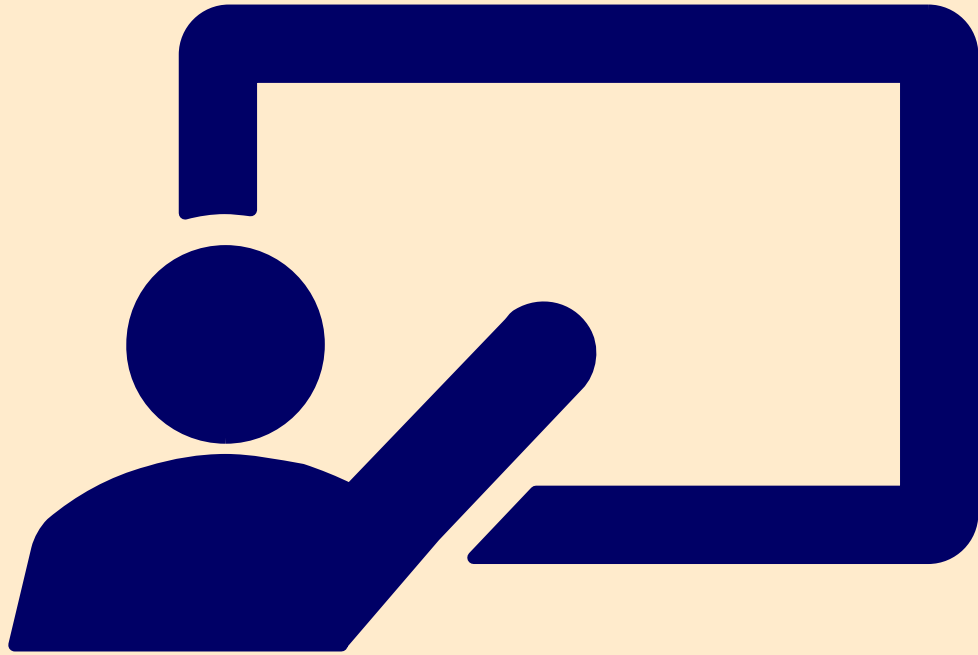


# Why object is different from data?

---

- An object has both a **state** and **behavior**. The state defines the object, and the behavior defines what the object does.





# Class Definition and Object Creation

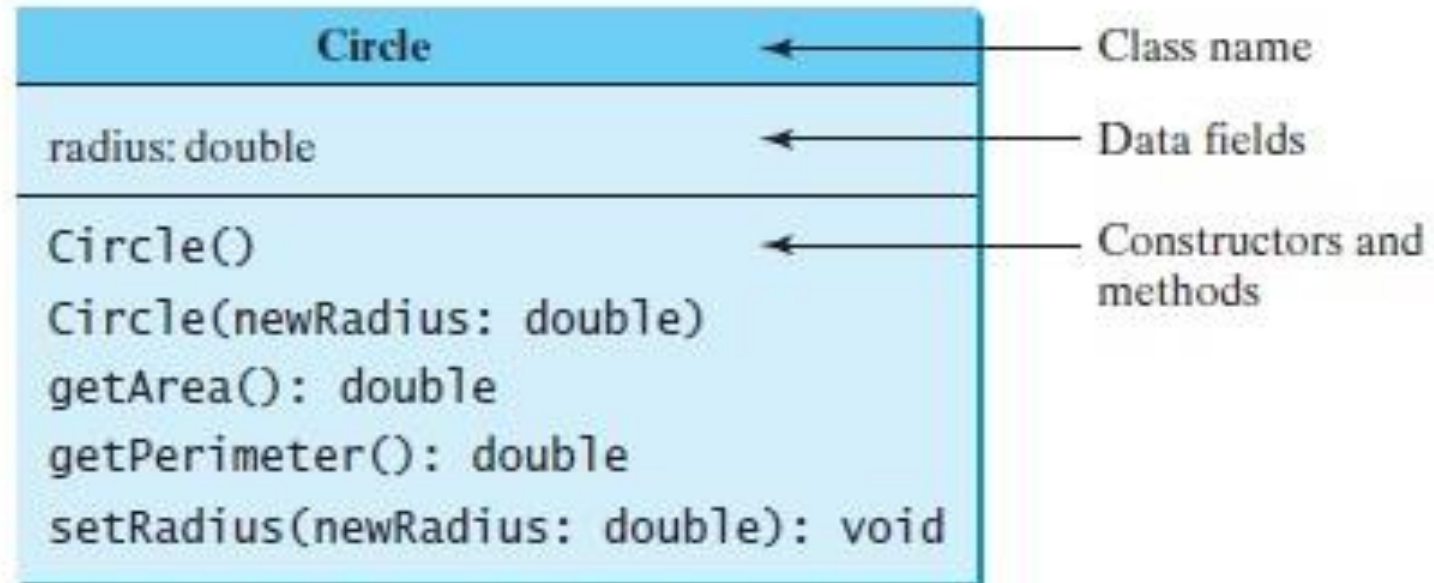
---

LECTURE 2



# Objects: UML Class/Object Diagram

UML Class Diagram





# Classes

---

- **Classes** are **constructs** that define objects of the same type. A Java class uses variables to define **data fields** and **methods** to define behaviors.
- Additionally, a class provides a special type of methods, known as constructors, which are invoked to construct objects from the class.

# Classes

```
class Circle {  
    /** The radius of this circle */  
    double radius = 1.0;  
  
    /** Construct a circle object */  
    Circle() {  
    }  
  
    /** Construct a circle object */  
    Circle(double newRadius) {  
        radius = newRadius;  
    }  
  
    /** Return the area of this circle */  
    double getArea() {  
        return radius * radius * 3.14159;  
    }  
}
```

← Data field

← Constructors

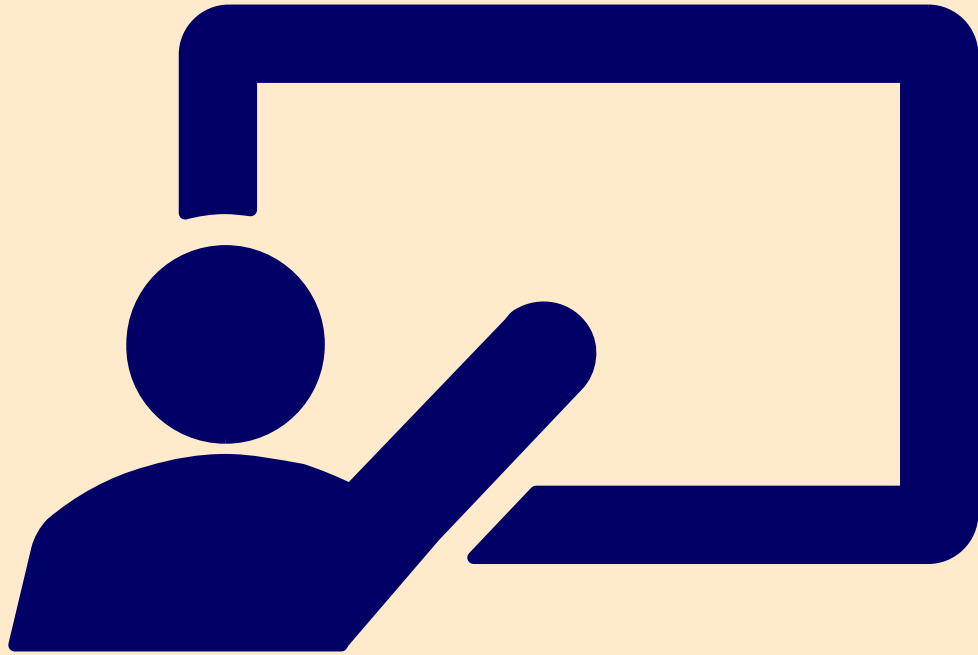
← Method



# Demonstration Program

---

TESTSIMPLECIRCLE.JAVA



# Constructors

---

## LECTURE 3



# Constructors

---

- Constructors are a special kind of methods that are invoked to construct objects.

```
Circle() {  
}
```

```
Circle(double newRadius) {  
    radius = newRadius;  
}
```



# Constructors, cont.

---

A **constructor** with no parameters is referred to as a *no-arg constructor*. If no constructor is given, default one will be used.

- Constructors must have the same name as the class itself.
- Constructors do not have a return type—not even void.
- Constructors are invoked using the new operator when an object is created. Constructors play the role of initializing objects.





# Creating Objects Using Constructors

---

```
new ClassName();
```

## **Example:**

```
new Circle();  
new Circle(5.0);
```



# Declaring Object Reference Variables

---

- To reference an object, assign the object to a reference variable. For an object, you may have as many reference variable (pointer) as you wish.
- To declare a reference variable, use the syntax:

```
ClassName objectRefVar;
```

## **Example:**

```
Circle myCircle;
```



# Declaring/Creating Objects in a Single Step

---

```
ClassName objectRefVar = new ClassName();
```

## **Example:**

```
Circle myCircle = new Circle();
```



# Accessing Object's Members

## (Properties or Methods)

---

### **Referencing the object's data:**

`objectRefVar.data`

*e.g., myCircle.radius*

### **Invoking the object's method:**

`objectRefVar.methodName (arguments)`

*e.g., myCircle.getArea ()*



# Trace Code

---

```
Circle myCircle = new Circle(5.0);
```

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

Declare myCircle

myCircle

no value



# Trace Code

```
Circle myCircle = new Circle(5.0);
```

myCircle

no value

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

: Circle

radius: 5.0

Create a circle



# Trace Code

```
Circle myCircle = new Circle(5.0);
```

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

Assign object reference  
to myCircle

myCircle

reference value

: Circle

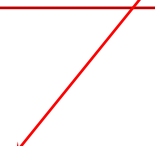
radius: 5.0



# Trace Code

```
Circle myCircle = new Circle(5.0);  
Circle yourCircle = new Circle();  
yourCircle.radius = 100;
```

myCircle reference value



<u>: Circle</u>
radius: 5.0

yourCircle no value



Declare yourCircle





# Trace Code

```
Circle myCircle = new Circle(5.0);  
Circle yourCircle = new Circle();  
yourCircle.radius = 100;
```

myCircle reference value

<u>: Circle</u>
radius: 5.0

yourCircle no value

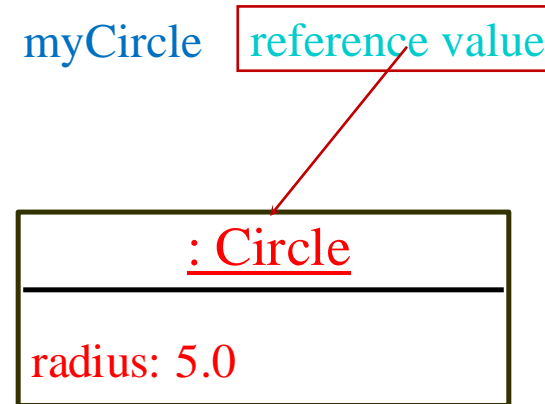
<u>: Circle</u>
radius: 1.0

Create a new  
Circle object

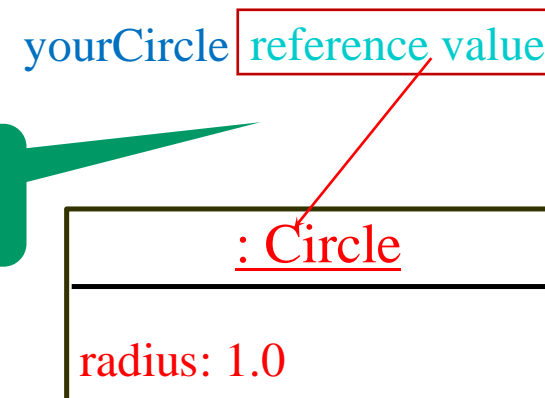


# Trace Code

```
Circle myCircle = new Circle(5.0);  
Circle yourCircle = new Circle();  
yourCircle.radius = 100;
```



Assign object reference  
to yourCircle





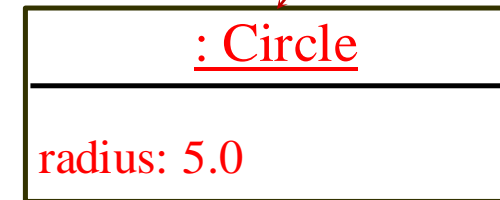
# Trace Code

```
Circle myCircle = new Circle(5.0);
```

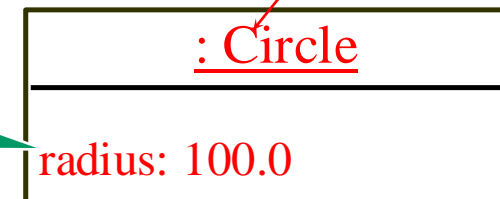
```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

myCircle reference value



yourCircle reference value



Change radius in  
yourCircle



# Caution

(static members belong to a class, non-static members belongs to objects)

---

- Recall that you use

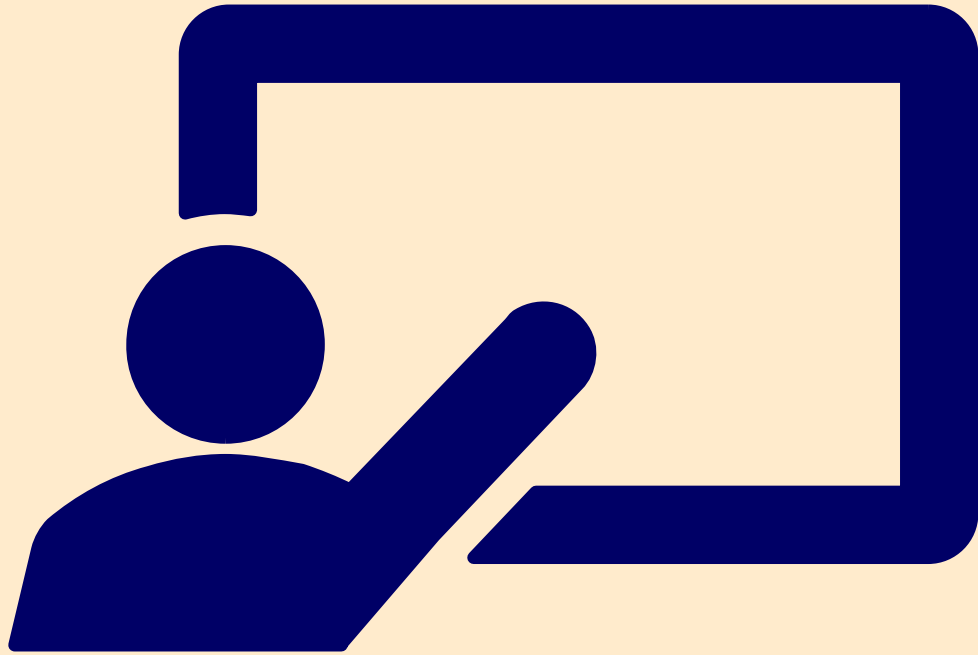
`Math.methodName (arguments) (e.g., Math.pow(3, 2.5))`

- to invoke a method in the Math class. Can you invoke `getArea()` using **Circle**.`getArea()`? The answer is no. All the methods used before this chapter are static methods, which are defined using the static keyword. However, `getArea()` is non-static. It must be invoked from an object using

`objectRefVar.methodName (arguments)`

`(e.g., myCircle.getArea())`.

- More explanations will be given in the section on “Static Variables, Constants, and Methods.”



# Data Fields

---

LECTURE 4



# Data Class

---

A data class has 3 major components:

- (1) Instance data fields
- (2) Instance methods
- (3) Constructors



# Data Class

---

- A data class is the template of data objects.
- An object is a group of data fields to form a higher level of data abstraction.
- Point, vector, tuple, list, and ... (These are abstract data types. Many of them can be realized by Class)



# Point Class

---

```
public class Point{  
    int x, y;           // instance data field  
    Point(int a, int b){ // object-builder  
        x = a;  y = b;  
    }  
}
```





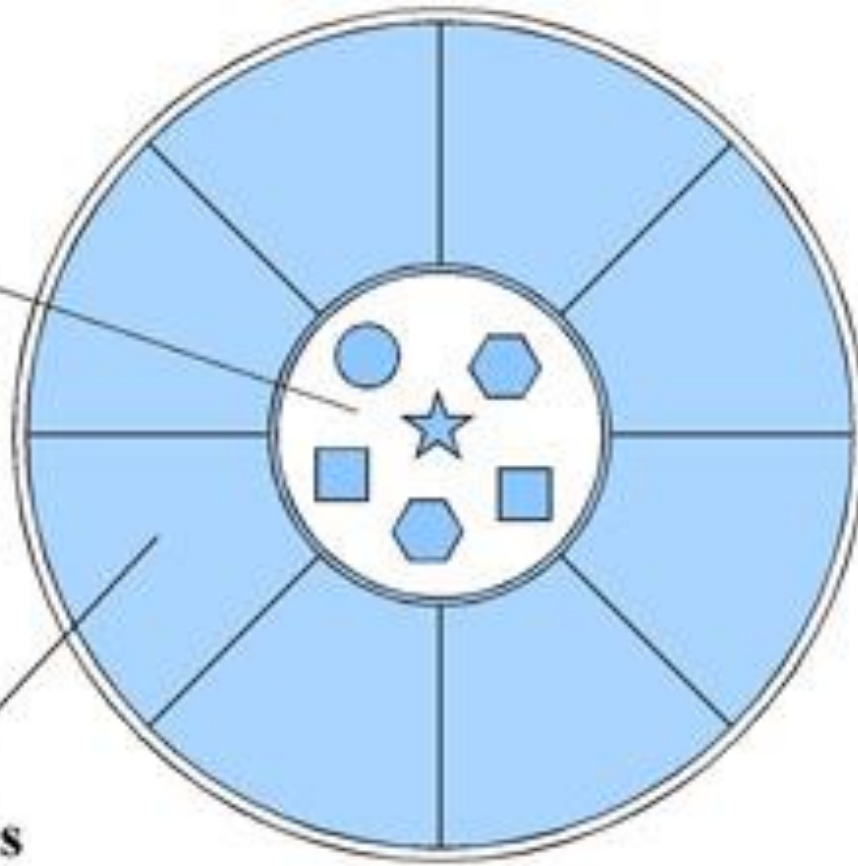
# Point Class – Getters/Setters

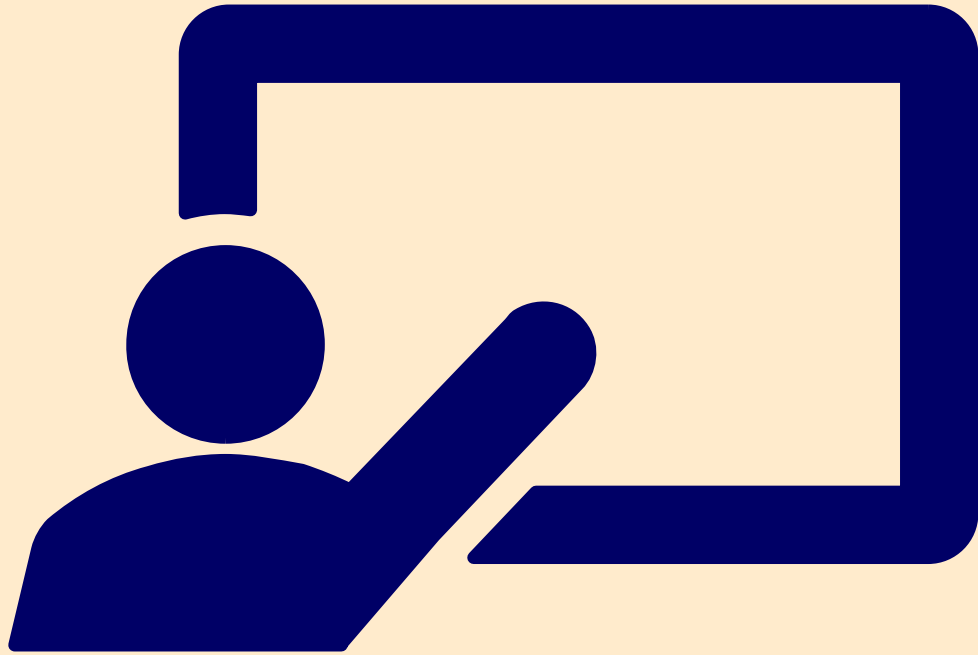
---

```
public class Point{
    int x, y;                // instance data field
    Point(int a, int b){    // object-builder
        x = a;  y = b;
    }
    public int getX(){ return x; }
    public int getY(){ return y; }
    public void setX(int a){ x = a; }
    public void setY(int b){ y = b; }
}
```

**Fields**  
Defined States

**Methods**  
Defined Behaviors  
Defined Ways of Modifying the State





# Derived Data Fields

---

LECTURE 5



# Derived Data Fields

---

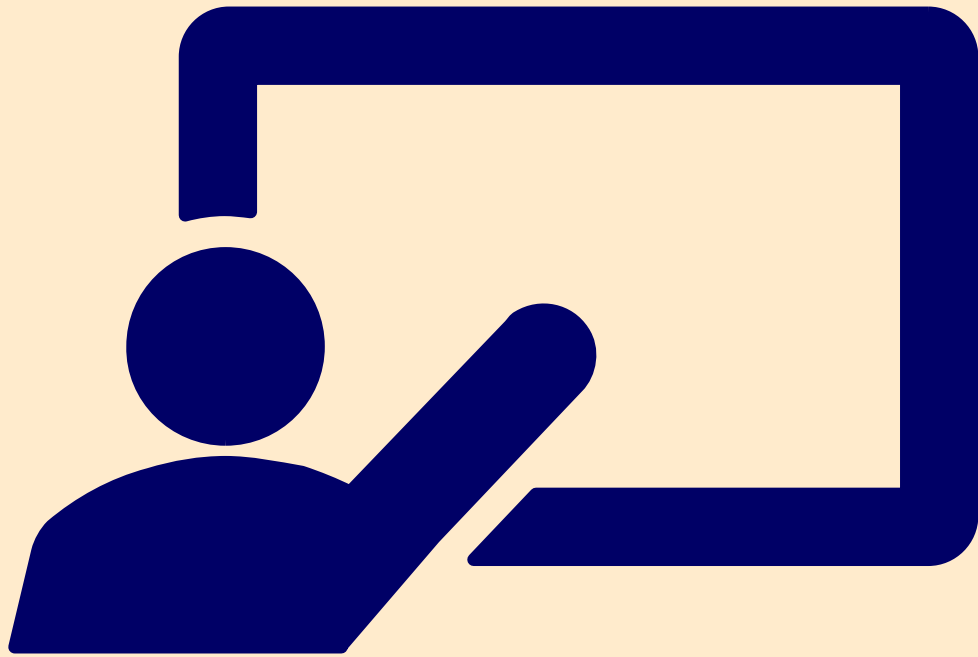
- Derived data fields are methods that will calculate some value based on the instance data fields.
- In reality, there are no such instance data fields but it looks like there are.



# Rectangle Class – Derived Data Fields

---

```
public class Rectangle{
    int width, height;    // instance data field
    Rectangle(int a, int b){ // object-builder
        width = a;   height = b;
    }
    public int getWidth(){ return width; }
    public int getHeight(){ return height; }
    public void setWidth(int a){ width = a; }
    public void setHeight(int b){ height = b; }
    public int getArea() {return width*height; }
}
```



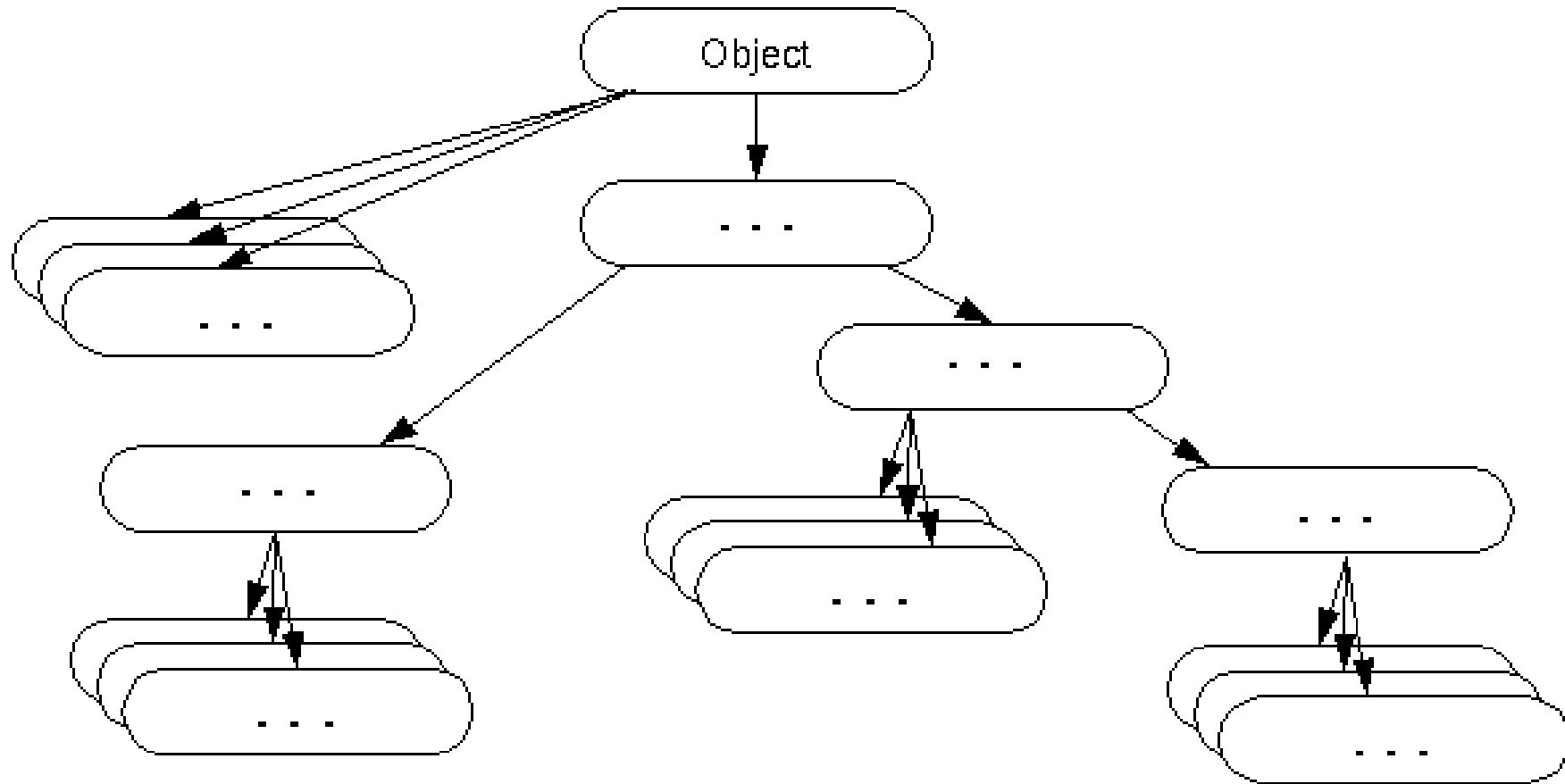
# Inherited Standard Functions

---

LECTURE 6



# Every Class Inherits from Object Class





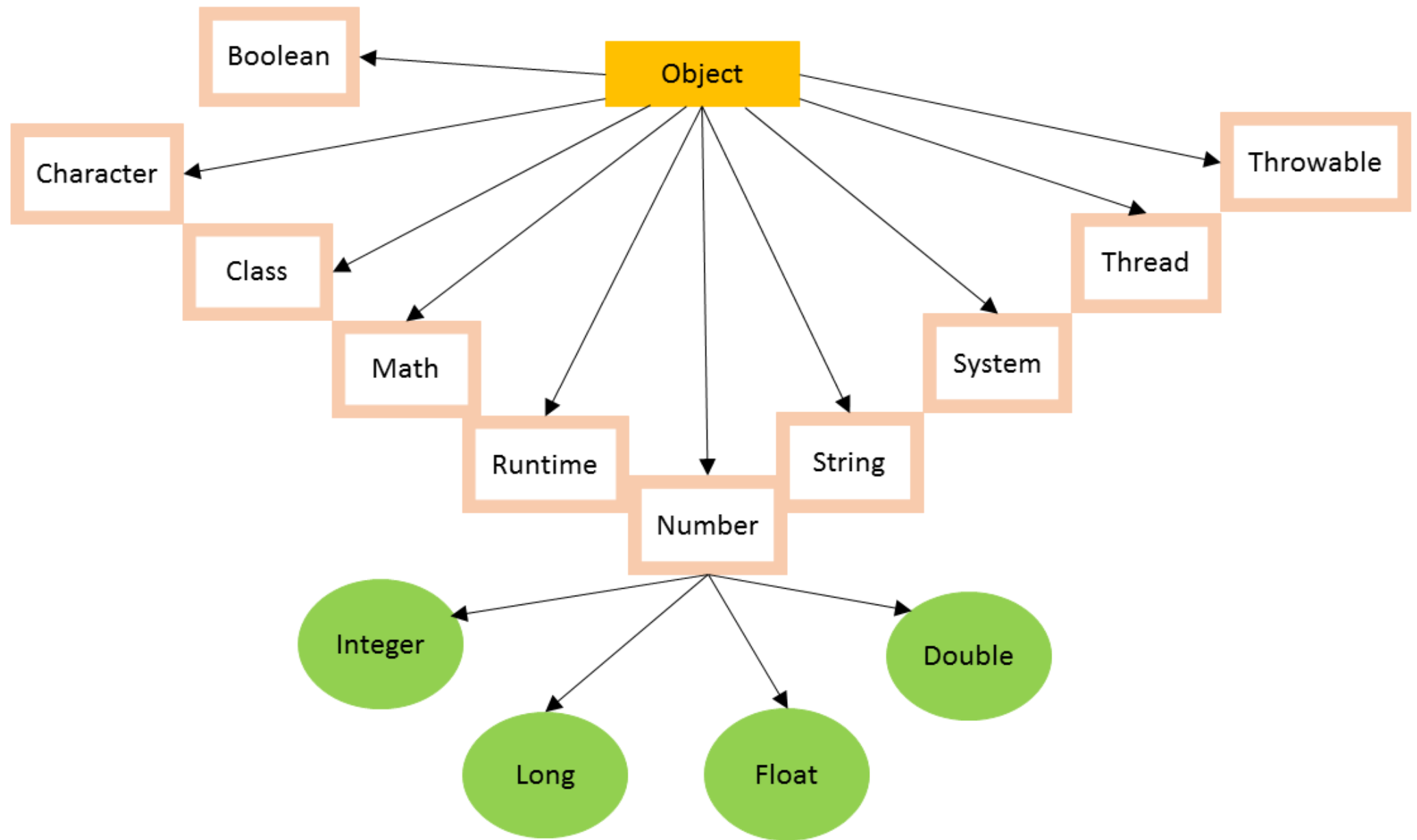
# Object as a Superclass

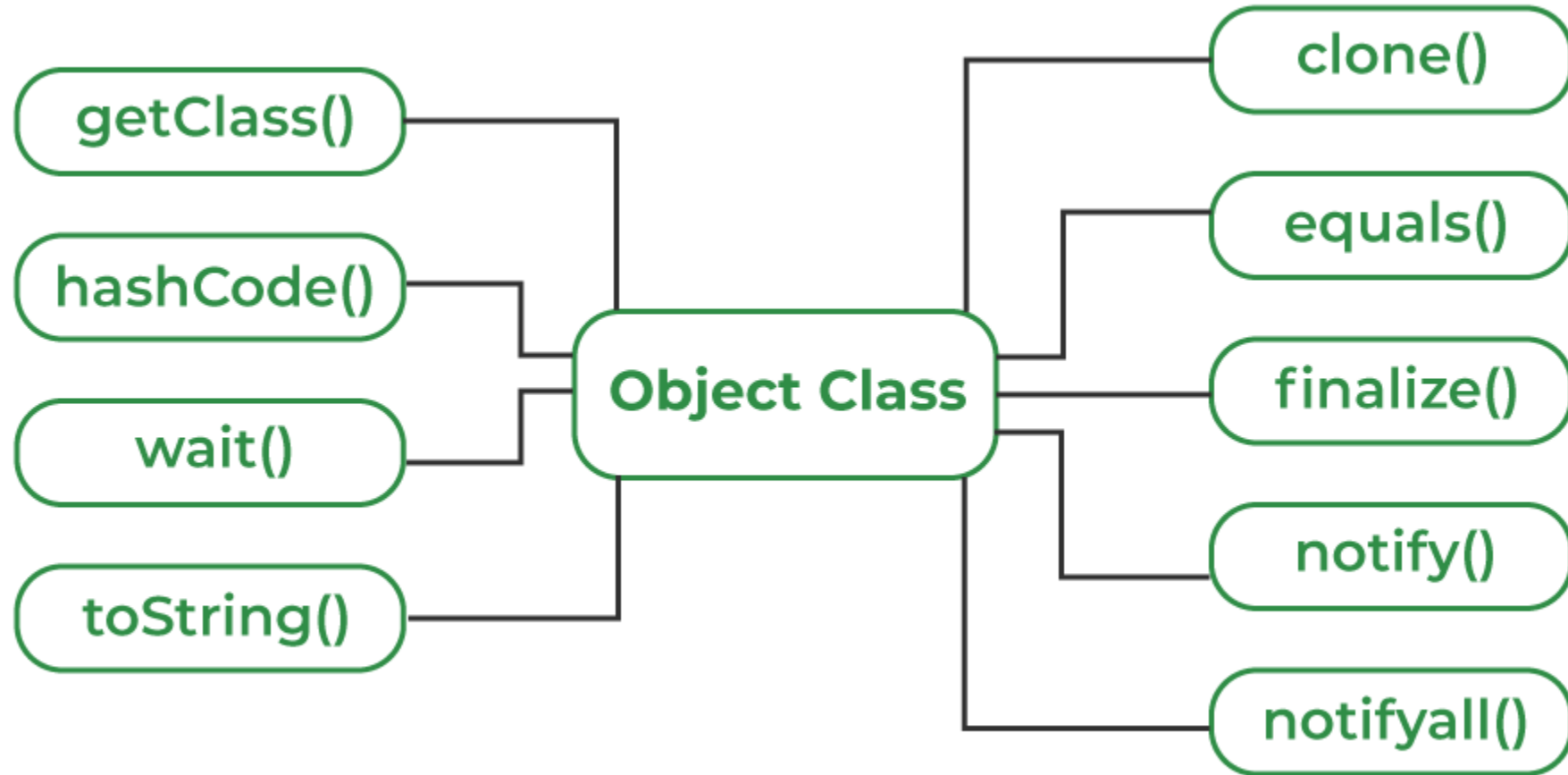
Object is superclass for all objects in java.lang language such as Integer, Double, String, Arrays, and etc.

---

- 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:









# 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()**:



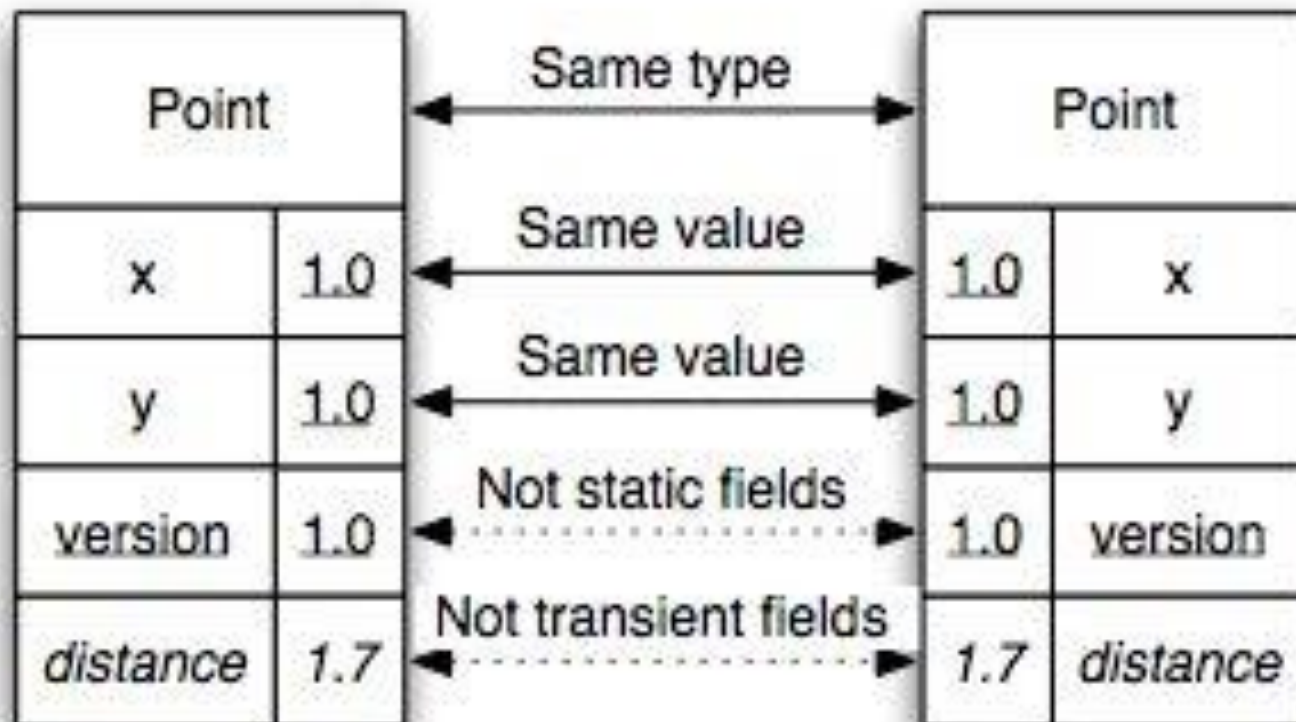
# The equals() Method

---

```
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.



# 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



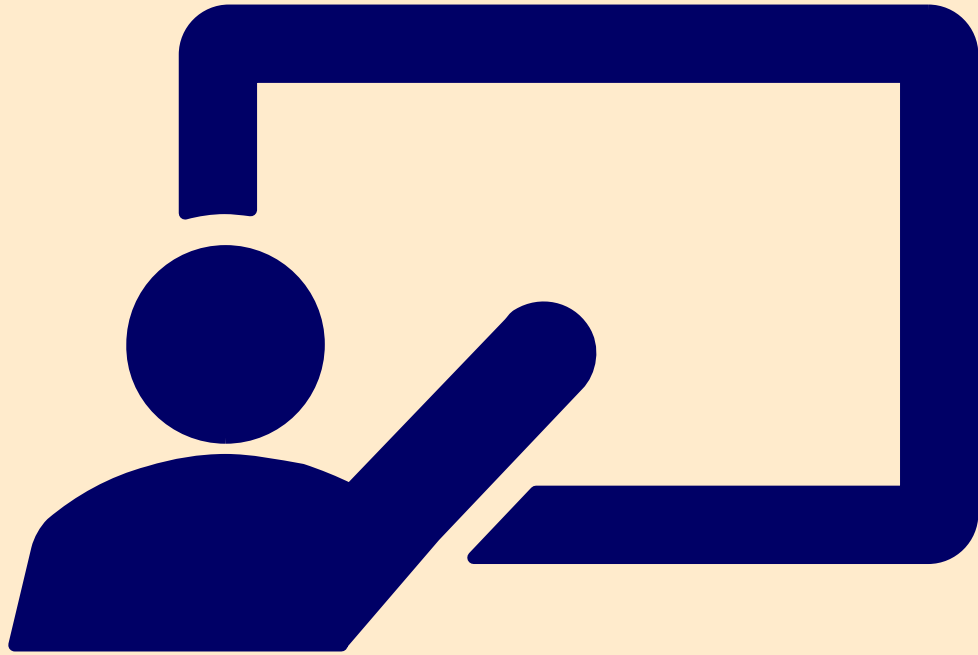
# What you need to remember when overriding toString() manually?

---

- 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

```
public class Rectangle{
    int width, height;    // instance data field
    Rectangle(int a, int b){ // object-builder
        width = a;  height = b;
    }
    public int getWidth(){ return width; }
    public int getHeight(){ return height; }
    public void setWidth(int a){ width = a; }
    public void setHeight(int b){ height = b; }
    public void getArea() {return width*height; }
    public boolean equals(Object other){
        Rectangle that = (Rectangle) other;
        return this.width == that.width && this.height == that.height;
    }
    public String toString(){
        return String.format("Rect[%d, %d]", width, height);
    }
}
```





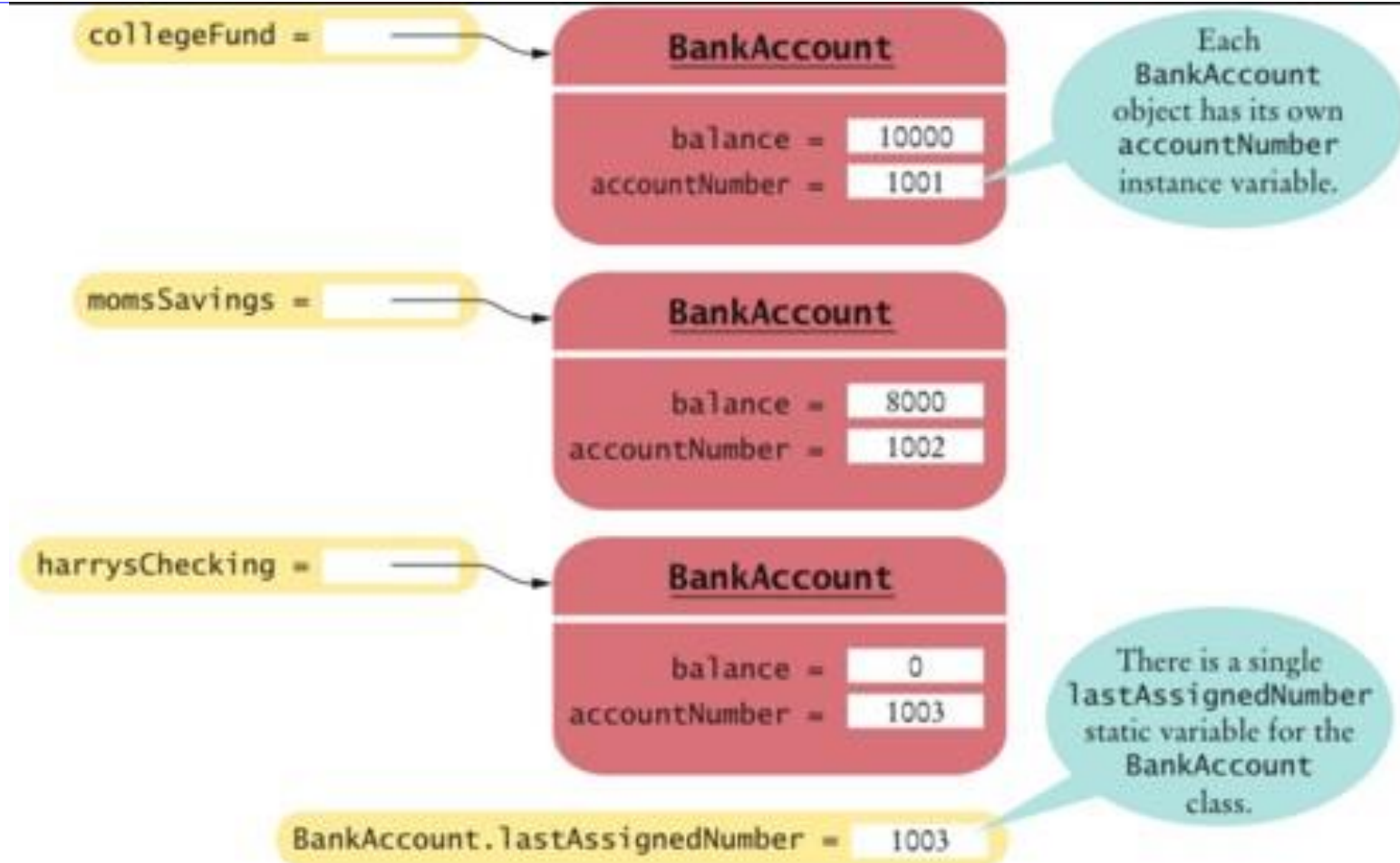
# Static Members

---

LECTURE 7

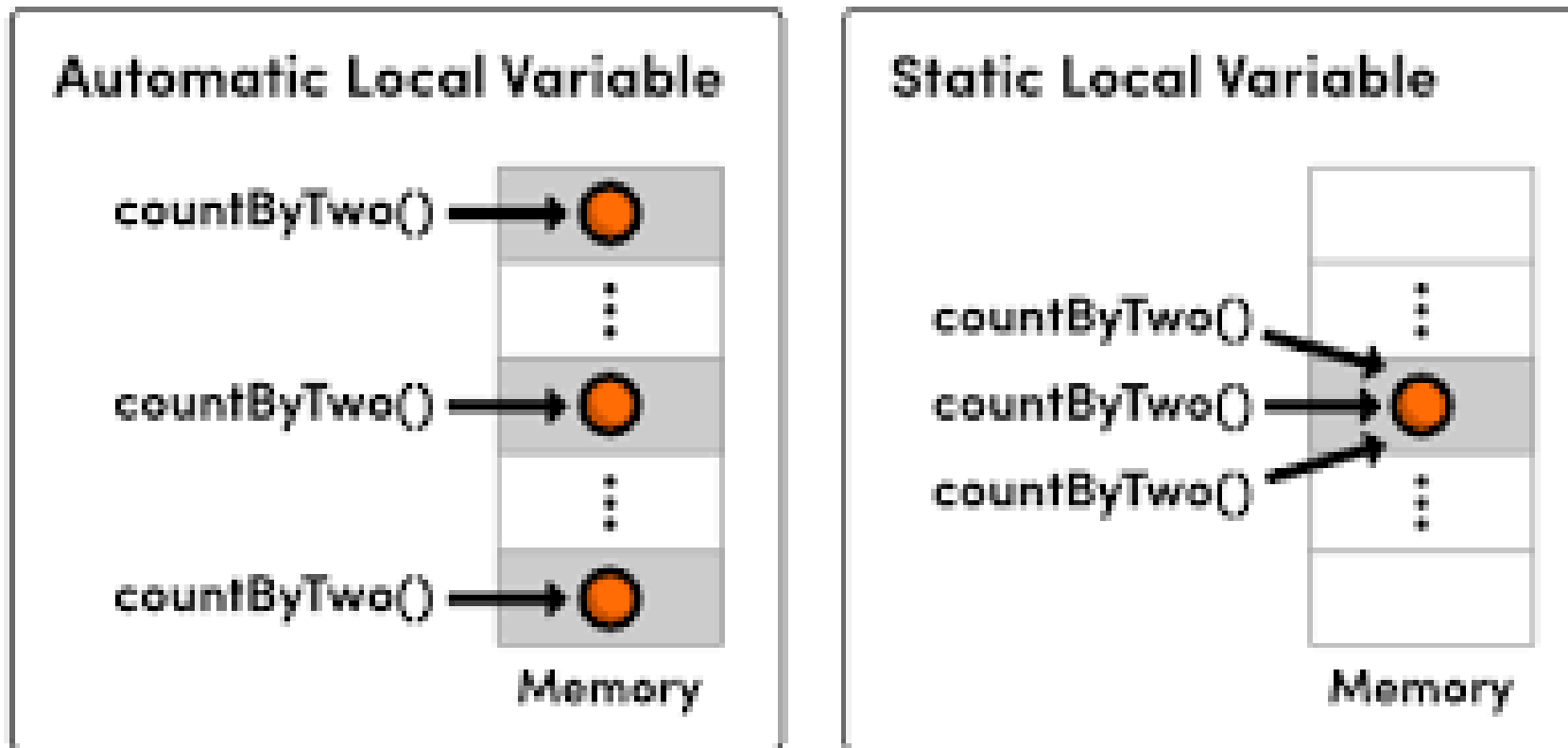


# Static Members





# Static Members



```
public class Point{
    static int numOfPointsCreated;
    int x, y;           // instance data field
    Point(int a, int b){ // object-builder
        x = a;  y = b;
        numOfPointsCreated++;
    }
    public int getX(){ return x; }
    public int getY(){ return y; }
    public void setX(int a){ x = a; }
    public void setY(int b){ y = b; }
}
```



# Static Variable is Class Variable

---

- The static variables belong to the Class. It does not belong to any instance.
- Static variable is also known as class variable. Static variable can be accessed by static method or instance methods, while instance variable (non-static) can only be accessed by instance method.



# Static Variable is Class Variable

---

- The static variables can be accessed by a client program using

Classname.variable\_name

Point.numOfPointsCreated



# Static Methods are Usually Utility Methods

---

- The static methods can be designed to be utility methods, such as `Math.cos()`, `Math.sin()`, `Math.random()`

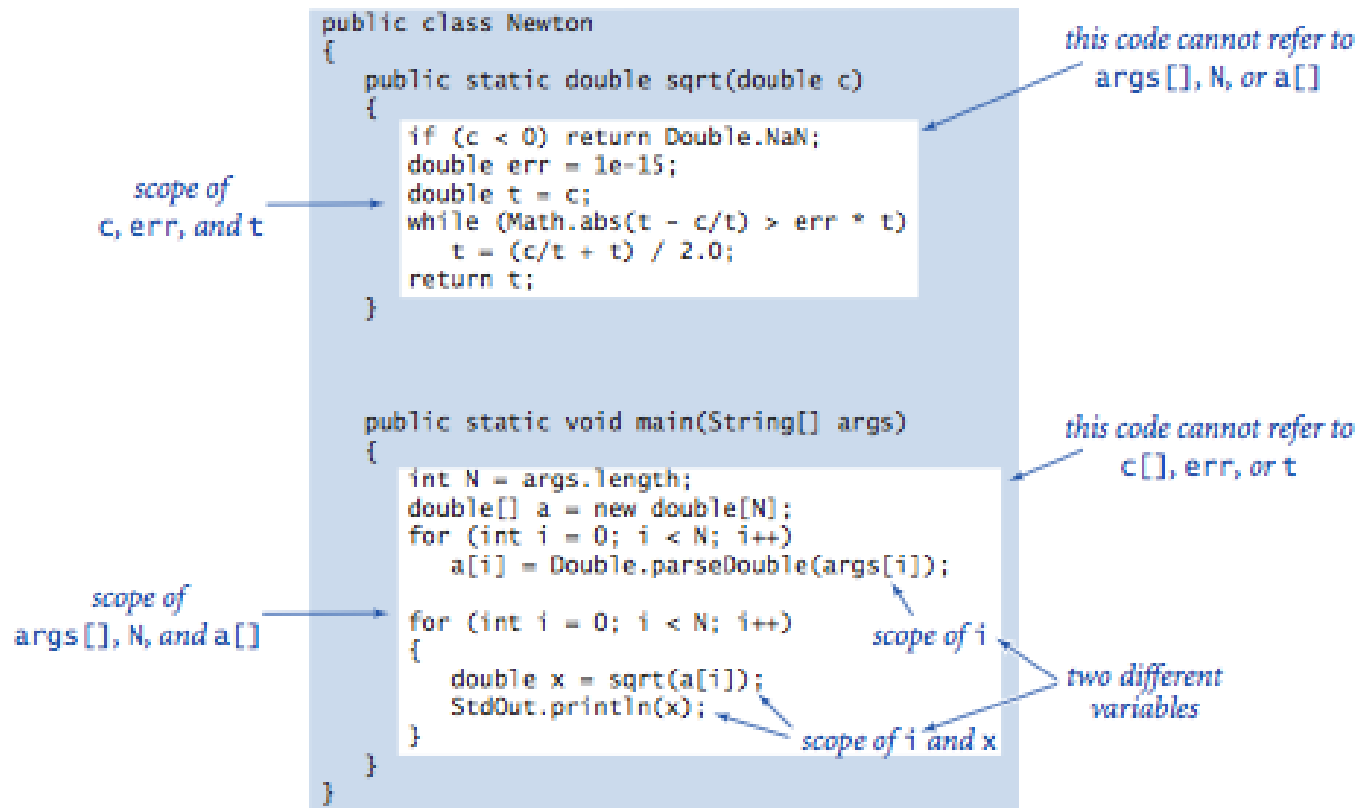
```
public class Point{
    static int numOfPointsCreated;
    int x, y;           // instance data field
    Point(int a, int b){ // object-builder
        x = a;  y = b;
        numOfPointsCreated++;
    }
    public int getX(){ return x; }
    public int getY(){ return y; }
    public void setX(int a){ x = a; }
    public void setY(int b){ y = b; }
    public static double distance(Point p1, Point p2){
        double dx = p1.x - p2.x, dy = p1.y - p2.y;
        return Math.sqrt(dx*dx + dy*dy);
    }
}
```





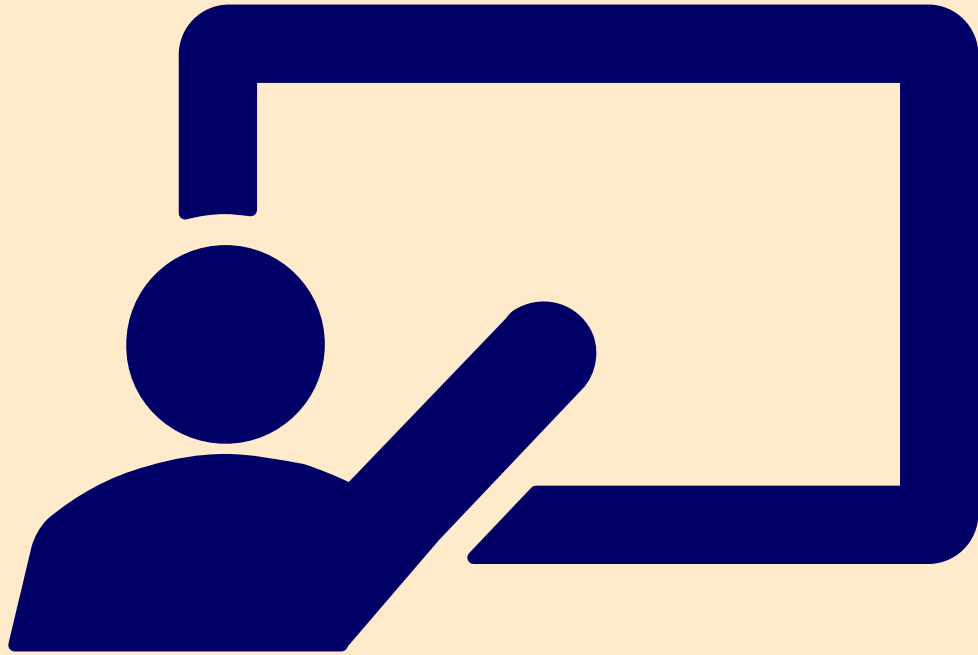
# Static Methods

(Program control structure Only, not related to data)



*Scope of local and argument variables*

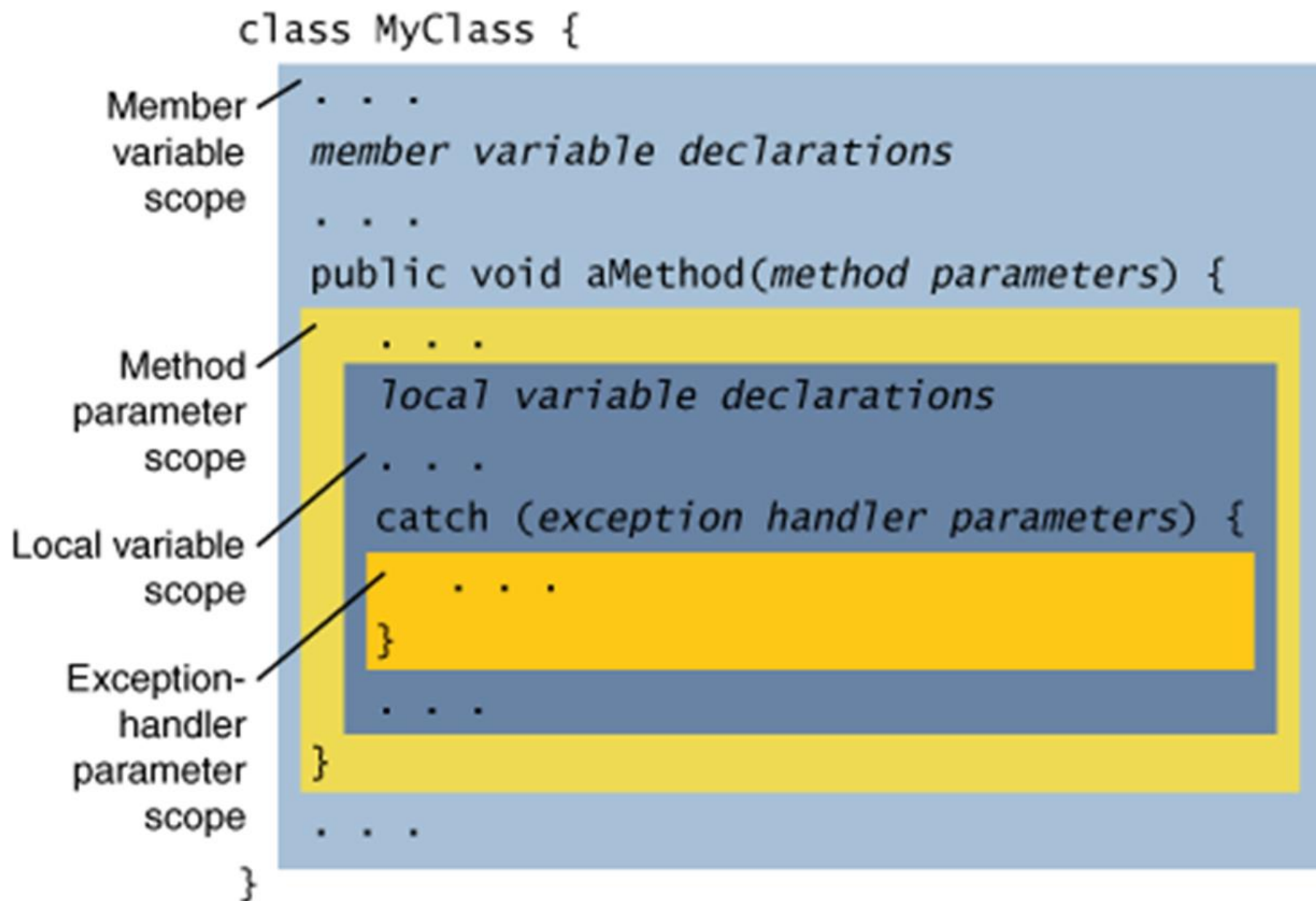
- The use of static methods is easy to understand. For example, when you write `Math.abs(a-b)` in a program, the effect is as if you were to replace that code by the value that is computed by Java's `Math.abs()` method when presented with the value `a-b`.
- If you think about what the computer has to do to create this effect, you will realize that it involves changing a program's **flow of control**.



# Scope of Members

---

LECTURE 8





# Local Variable Versus Global Variable

---

## **Global Variables:**

Member Properties:

Instance Variable - double radius;

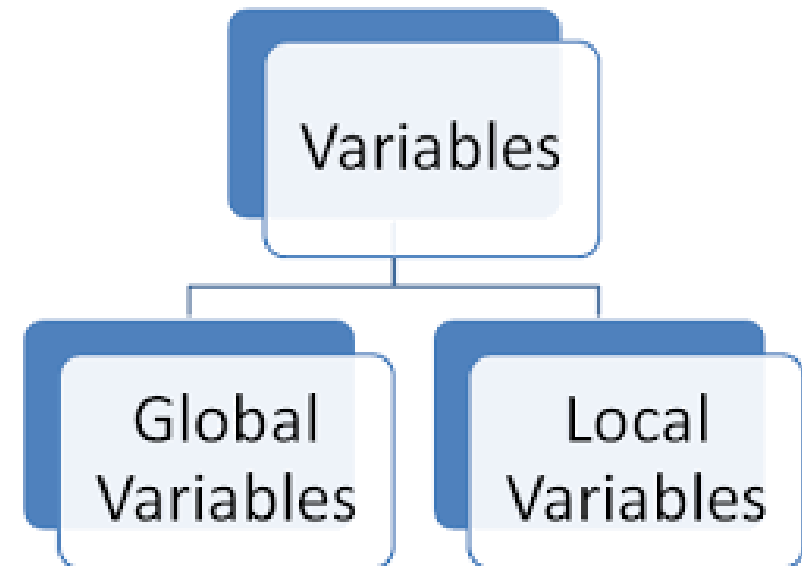
Class Variable (static) – static int num;

## **Local Variables:**

Arguments

Local Variables at Method level

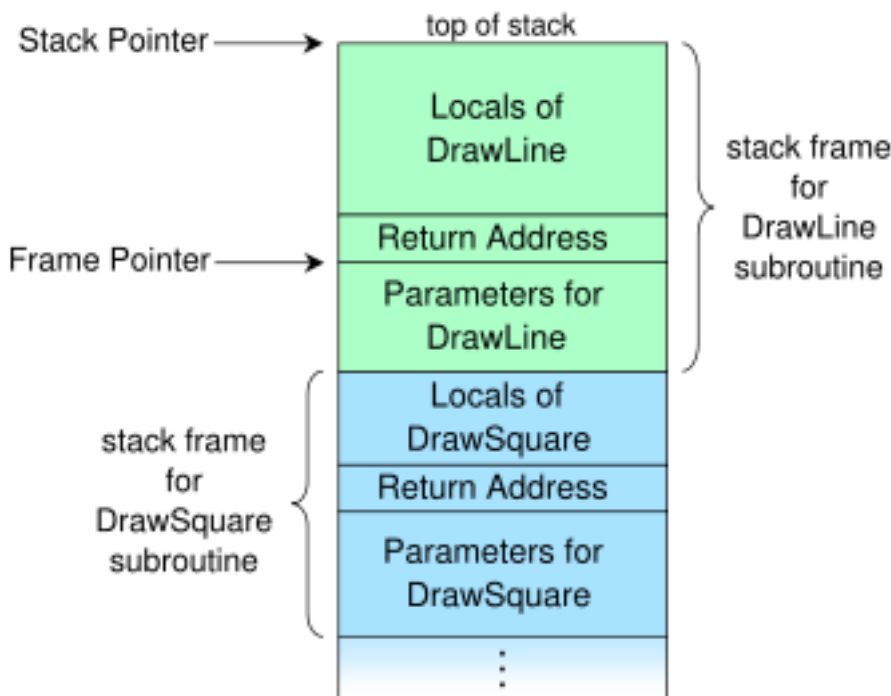
Block level local Variables





# Scope of Local Variables and Parameters

## Refer to Chapter 6: Scope of Variable (Local Variables)

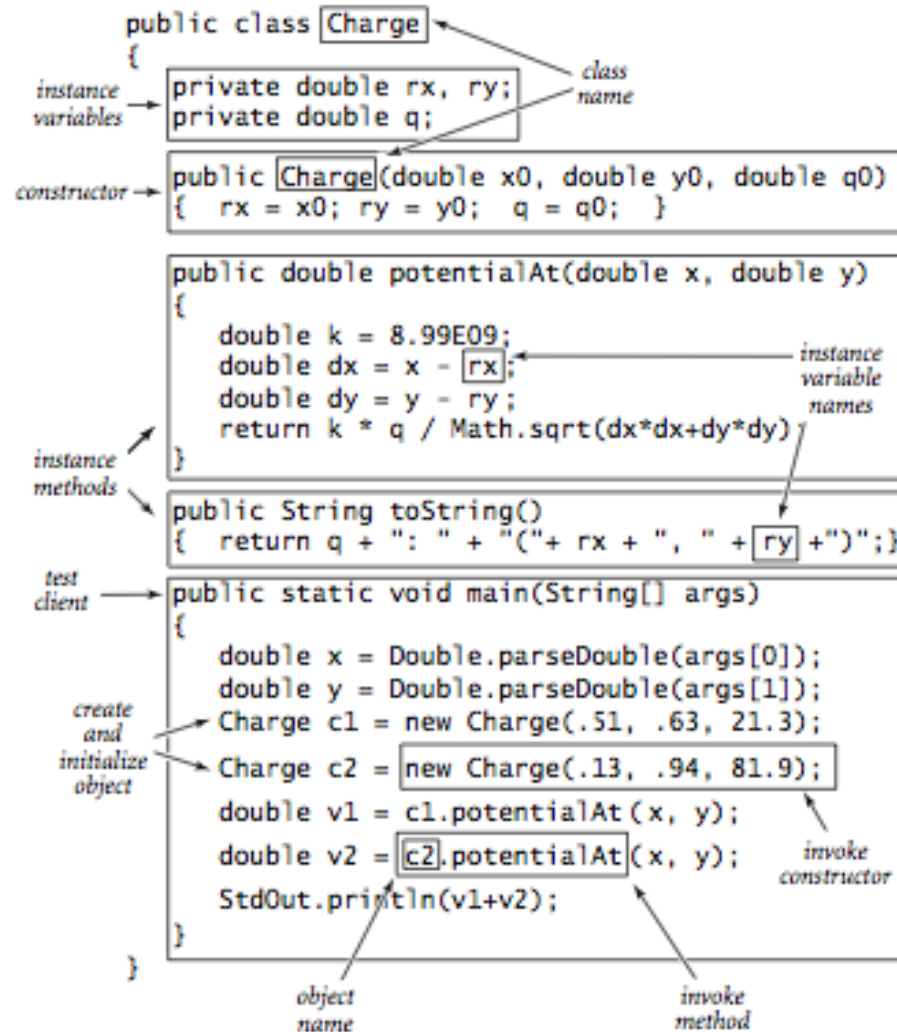


In a class definition, there are three kinds of variables:

- **instance variables** Any method in the class definition can access these variables (**is global, not local**)
- **parameter variables** Only the method where the parameter appears can access these variables. This is how information is passed to the object.
- **local variables** Only the method where the parameter appears can access these variables. These variables are used to store intermediate results.

# Instance Members

- Instance Variables
- Instance Methods



*Anatomy of a class*



```
public class Charge()
{
    private double rx, ry;
    private double q;
    .
    .
}
```

*Instance variables*

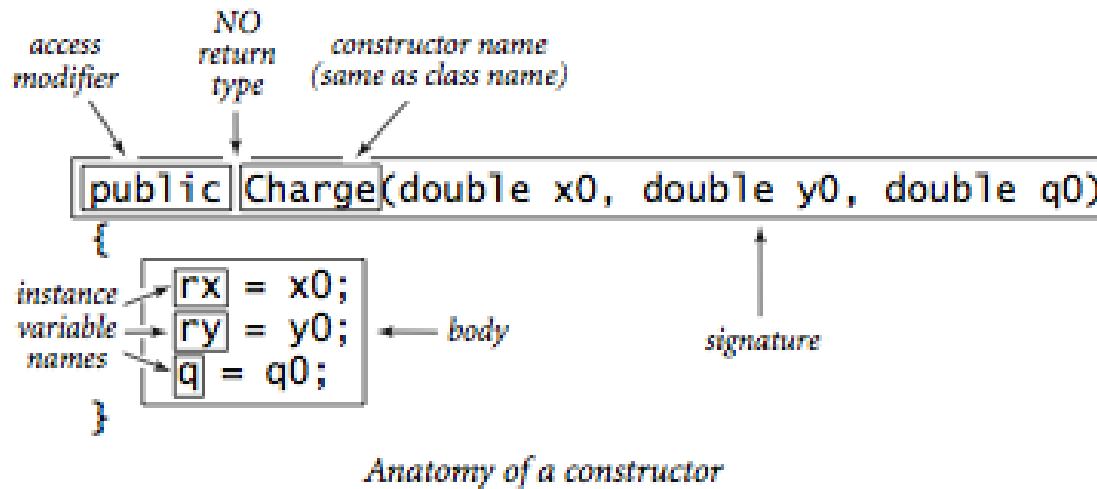
*instance variable declarations*

# Instance Variables

- To write code for the methods that manipulate data type values, the first thing that we need is to declare variables that we can use to refer to the values in code. These variables can be any type of data. We declare the types and names of these instance variables in the same way as we declare local variables.
- There is a critical distinction between **instance variables** and the local variables within a **static** method or a block that you are accustomed to using: there is just one value corresponding to each local variable name, but there are numerous values corresponding to each instance variable (one for each object that is an instance of the data type). Therefore, **static methods** cannot access **instance variables**. (instance variables may not be available)

# Constructors

## Create Instances

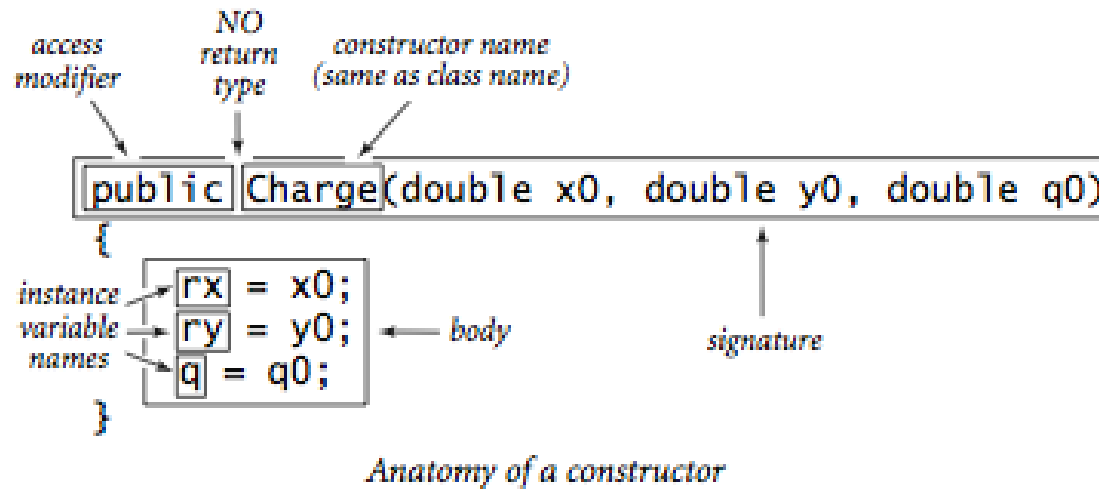


- A constructor creates an object and provides a reference to that object. Java automatically invokes a constructor when a client program uses the keyword `new`. Java does most of the work: our code only needs to initialize the instance variables to meaningful values. Constructors always share the same name as the class. To the client, the combination of `new` followed by a constructor name (with argument values enclosed within parentheses) is the same as a function call that returns a value of the corresponding type.

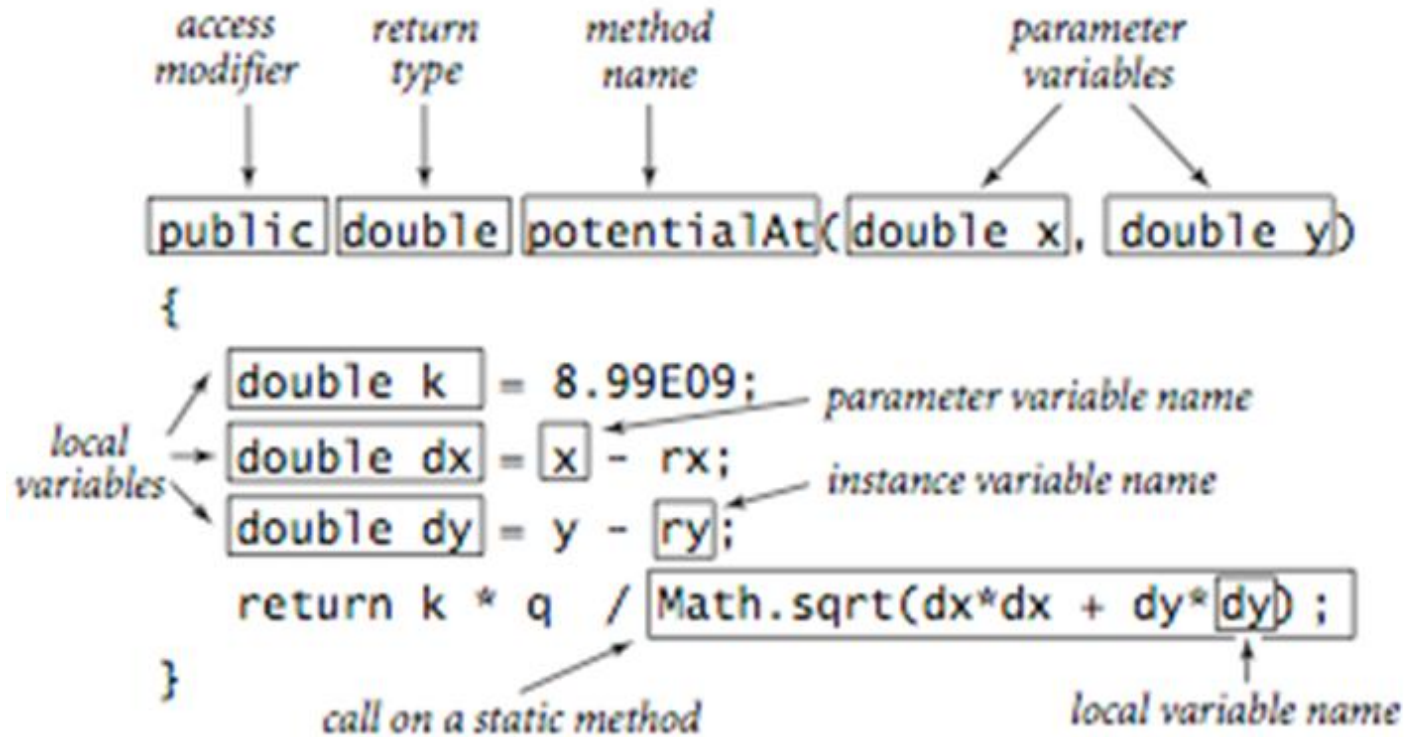


# Constructors

## Create Instances



- A **constructor signature** has **no** return type because constructors always return a reference to an object of its data type. Each time that a client invokes a constructor) Java automatically
  - allocates memory space for the object
  - invokes the constructor code to initialize the data type values
  - returns a reference to the object

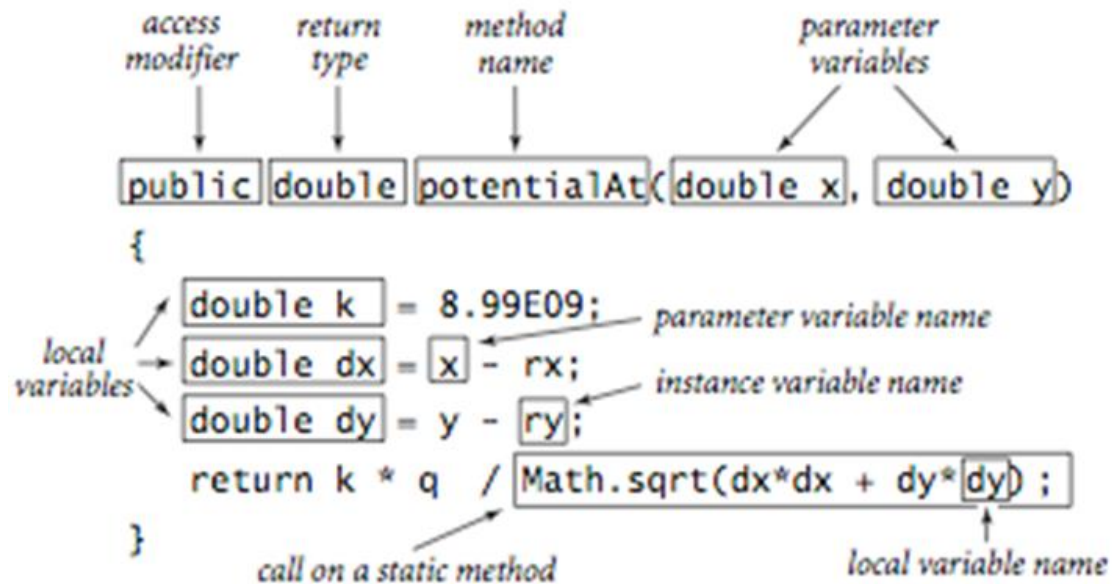


*Anatomy of a data-type method*

## Instance methods in a class with static modifier

- Each instance method has a signature (which specifies its return type and the types and names of its parameter variables) and a body (which consists of a sequence of statements, including a return statement that provides a value of the return type back to the client).

## Instance methods in a class with static modifier



*Anatomy of a data-type method*

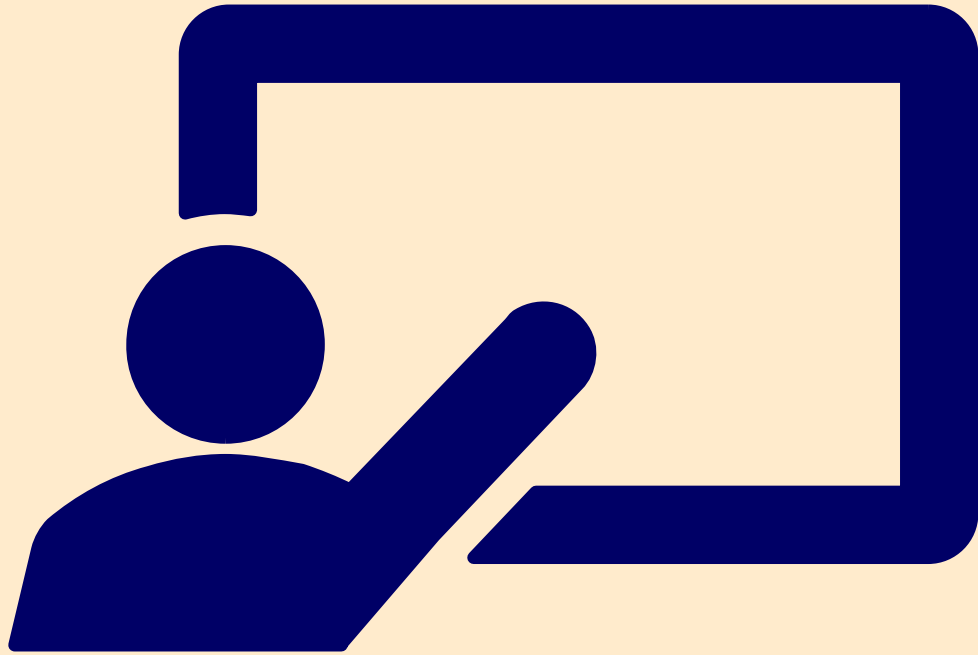
- When a client invokes a method, the parameter values are initialized with client values, the lines of code are executed until a return value is computed, and the value is returned to the client, with the same effect as if the method invocation in the client were replaced with that value. All of this action is the same as for static methods, but there is one critical distinction for instance methods: **they can perform operations on instance values.**



# Summary of Local Variables

---

<i>variable</i>	<i>purpose</i>	<i>example</i>	<i>scope</i>
instance	data-type value	rx	class
parameter	pass value from client to method	x	method
local	temporary use within method	dx	block



# Visibility Modifiers

---

LECTURE 9



# Visibility Modifiers and Accessor/Mutator Methods

---

- By default, the class, variable, or method can be accessed by any class in the same package.

## **public**

The class, data, or method is visible to any class in any package.

## **protected**

The class, data, or method is visible to any sub-class of this class in any package.

## **private**

The data or methods can be accessed only by the declaring class.

- The get and set methods are used to read and modify private properties.



# Data and Methods Visibility

Modifier on Members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass in a different package	Accessed from a different package
public	O	O	O	O
protected	O	O	O	X
default	O	O	X	X
private	O	X	X	X

package p1;

```
public class C1 {  
    public int x;  
    int y;  
    private int z;  
  
    public void m1() {  
    }  
    void m2() {  
    }  
    private void m3() {  
    }  
}
```

```
public class C2 {  
    void aMethod() {  
        C1 o = new C1();  
        can access o.x;  
        can access o.y;  
        cannot access o.z;  
  
        can invoke o.m1();  
        can invoke o.m2();  
        cannot invoke o.m3();  
    }  
}
```

package p2;

```
public class C3 {  
    void aMethod() {  
        C1 o = new C1();  
        can access o.x;  
        cannot access o.y;  
        cannot access o.z;  
  
        can invoke o.m1();  
        cannot invoke o.m2();  
        cannot invoke o.m3();  
    }  
}
```

package p1;

```
class C1 {  
    ...  
}
```

```
public class C2 {  
    can access C1  
}
```

package p2;

```
public class C3 {  
    cannot access C1;  
    can access C2;  
}
```

The private modifier restricts access to within a class, the default modifier restricts access to within a package, and the public modifier enables unrestricted access.





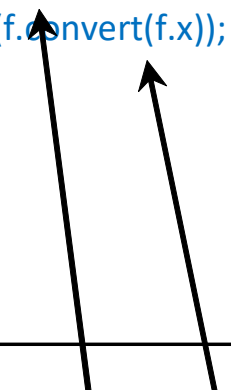
# NOTE

An object cannot access its private members, as shown in (b). It is OK, however, if the object is declared in its own class, as shown in (a).

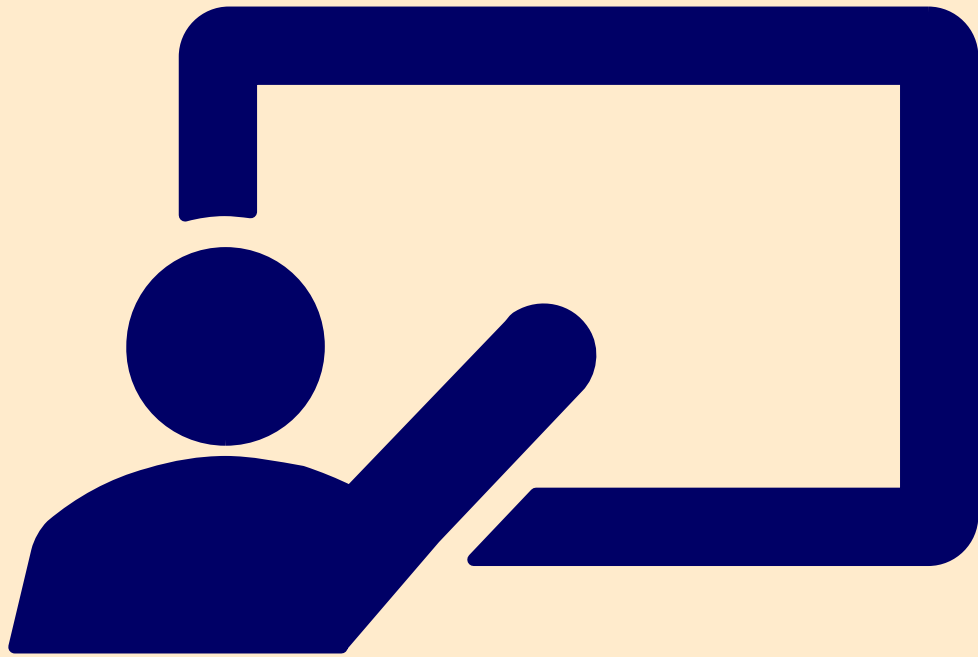
```
public class F {  
    private boolean x;  
  
    public static void main(String[] args) {  
        F f = new F ();  
        System.out.println(f.x);  
        System.out.println(f.convert());  
    }  
  
    private int convert(boolean b) {  
        return x ? 1 : -1;  
    }  
}
```

(a) This is OK because object f is used inside the F class

```
public class Test {  
    public static void main(String[] args) {  
        Foo f = new F();  
        System.out.println(f.x);  
        System.out.println(f.convert(f.x));  
    }  
}
```



(b) This is wrong because x and convert are private in F.



# Class, References and Objects

---

LECTURE 10



# Reference Data Fields

---

- The data fields can be of reference types. For example, the following **Student** class contains a data field **name** of the **String** type.

```
public class Student {  
    String name; // name has default value null  
    int age; // age has default value 0  
    boolean isScienceMajor; // isScienceMajor has default value false  
    char gender; // c has default value '\u0000'  
}
```



# The null Value

---

- If a data field of a reference type does not reference any object, the data field holds a special literal value, **null**.



# Default Value for a Data Field

---

- The default value of a data field is **null** for a reference type, 0 for a numeric type, false for a boolean type, and '\u0000' for a char type. However, Java assigns no default value to a local variable inside a method.

```
public class Test {  
    public static void main(String[] args) {  
        Student student = new Student();  
        System.out.println("name? " + student.name);  
        System.out.println("age? " + student.age);  
        System.out.println("isScienceMajor? " +  
                             student.isScienceMajor);  
        System.out.println("gender? " + student.gender);  
    }  
}
```

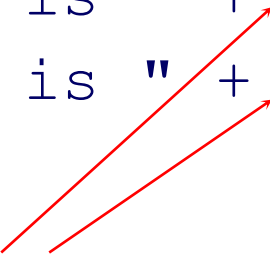


# Example

---

- Java assigns no default value to a local variable inside a method.

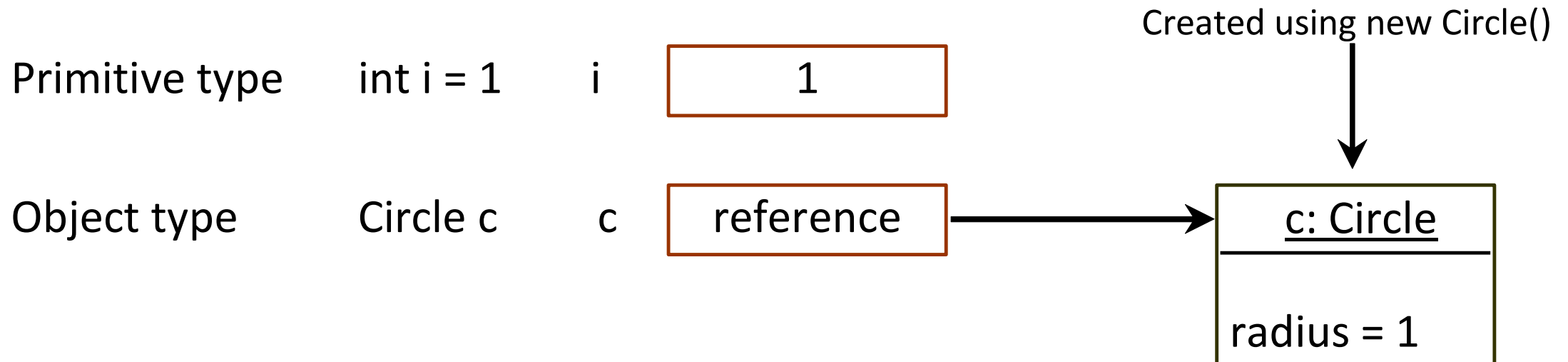
```
public class Test {  
    public static void main(String[] args) {  
        int x; // x has no default value  
        String y; // y has no default value  
        System.out.println("x is " + x);  
        System.out.println("y is " + y);  
    }  
}
```



**Compilation error: variables not initialized**



# Differences between Variables of Primitive Data Types and Object Types





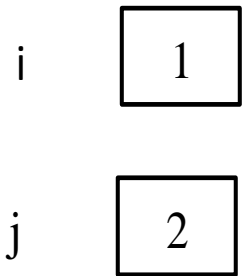


# Copying Variables of Primitive Data Types and Object Types

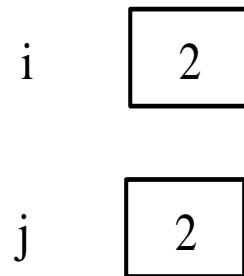
Object type assignment  $c1 = c2$

Primitive type assignment  $i = j$

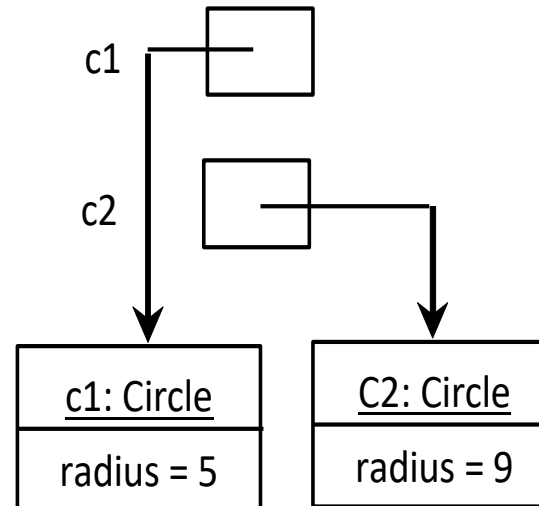
Before:



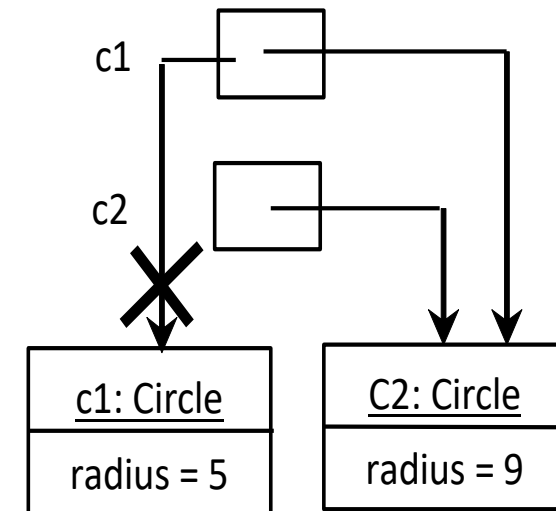
After:



Before:



After:





# Garbage Collection

---

- As shown in the previous figure, after the assignment statement  $c1 = c2$ ,  $c1$  points to the same object referenced by  $c2$ . The object previously referenced by  $c1$  is no longer referenced. This object is known as **garbage** (dangling object).
- Garbage is automatically collected by **JVM**.



## Garbage Collection, cont.

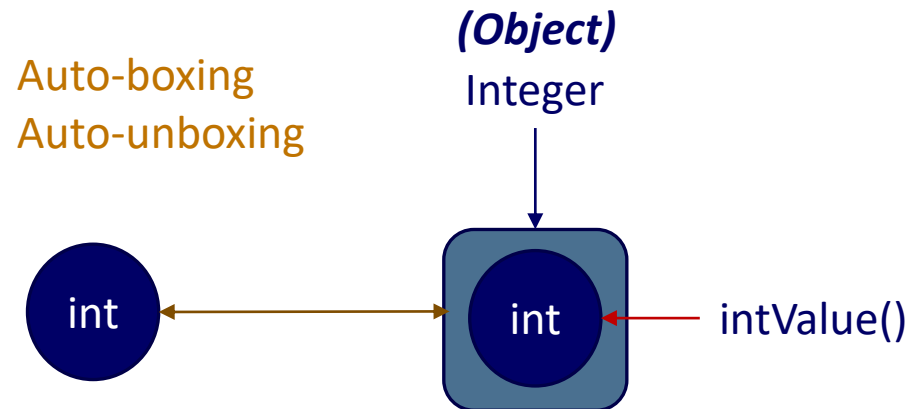
---

**TIP:** If you know that an object is no longer needed, you can explicitly assign **null** to a reference variable for the object. The **JVM** will automatically collect the space if the object is not referenced by any variable .



# Primitive Data Type V.S. Reference Data Type

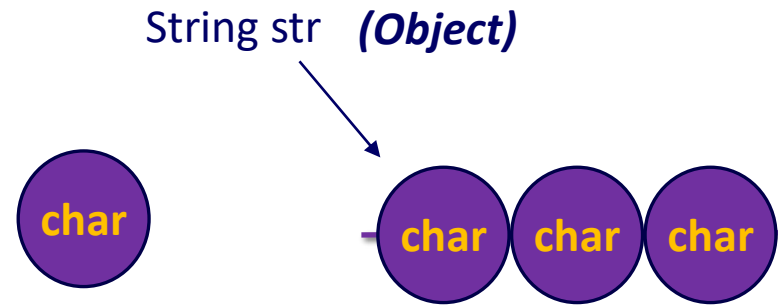
## Wrapper Class





# Primitive Data Type V.S. Reference Data Type

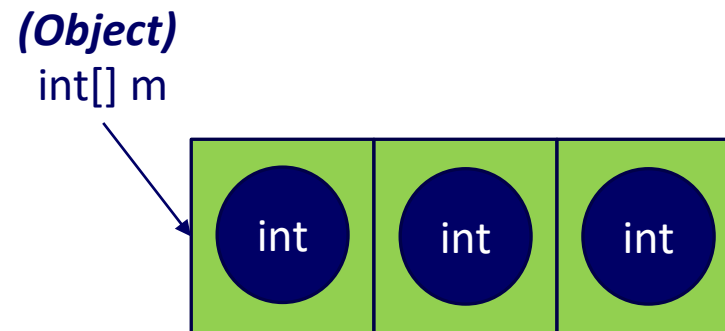
---





# Primitive Data Type V.S. Reference Data Type

---



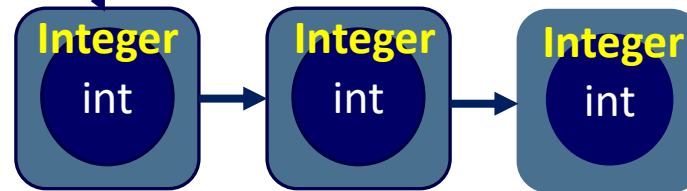


# Primitive Data Type V.S. Reference Data Type

---

*(Object)*

```
ArrayList<Integer> aList;
```





# Primitive Data Type V.S. Reference Data Type

(Object) Student a;

