

Object Oriented Programming

LESSON 02

Classes and Objects



Outline

1. Classes
2. Objects
3. Object creation
4. Accessing attributes of an object
5. Encapsulation



Overview

In this chapter, you are going to learn about

- Know in detail about Class
- Know how to use Object
- Know Object References
- Know how to use keyword "this"
- Know how to use encapsulation



Learning content

1. Classes

- Class Notation
- Attribute visibilities
- Methods

2. Objects

- Object Notation
- Static and Dynamic parts of Object
- References

3. Object creation

- Default constructor
- Explicit constructors

- Memory allocation

4. Accessing attributes of an object

- Sending Message
- Message Parameters
- Current Object "this"

5. Encapsulation

- Class members visibility (in Java)
- Code Robustness
- Extensible development

Pre-Test



Question	Possible answers	Correct Answer	Feedback of the question
What can you do with Banana?	1. Eat 2. Make dessert 3. Make popcorn	1. Eat 2. Make dessert	Popcorn is made of corn not Banana

Pre-Test



Question	Possible answers	Correct Answer	Feedback of the question
Look at a Jackfruit from outside, Which one is not visible in Jackfruit?	1. Worm (s) 2. Green color 3. Thorn size	1. Worm (s)	We can see color from outside is green. We can see its thorns so we can manage to measure it.

Pre-Test



Question	Possible answers	Correct Answer	Feedback of the question
Which one that computer can do?	1. Eat papaya 2. Replace CPU 3. Shutdown	3. Shutdown	Computer is electronic machine, it eats electricity not Papaya. Computer itself can not live without CPU, so can not change CPU itself.

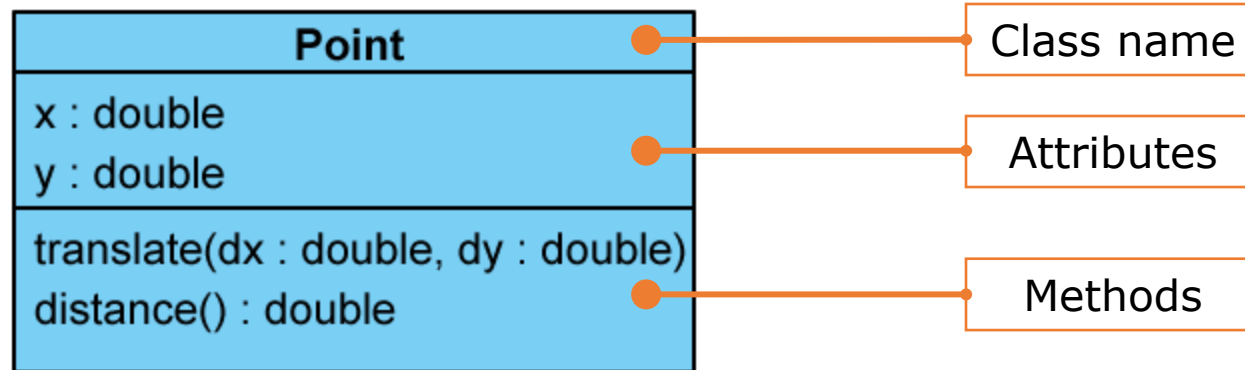


1. Classes

- A class is a **model** definition for objects
 - having the same structure (the same set of attributes)
 - having same behavior (same operations, methods)
 - having common semantics.
- The objects are dynamic representations (**instantiation**), "Living" of model defined for them through the class.
 - A class can instantiate (create) many objects
 - Each object is an instance of a (single) class

1.1. Class Notation

- This notation is drawn in Visual Paradigm





1.1. Class Notation

- Class and members of the class

```
public class Point {
```

Class name

```
double x;  
double y;
```

Attributes

```
/**
```

```
 * Translate coordinates  
 * @param dx delta x  
 * @param dy delta y  
 * @return void  
 */
```

Method

```
void translate(double dx, double dy) {  
    x += dx;  
    y += dy;  
}
```

1.1. Class Notation

- Class and members of the class

```
/**  
 * Calculate distance from 0, 0  
 * @return distance calculated  
 */  
double distance() {  
    double dist;  
    dist = Math.sqrt(x*x + y*y);  
    return dist;  
}  
}
```

Method

1.2. Attribute visibilities

- Attributes are “global” variables in modules that are in the class: they are accessible in all methods of the class.

```
public class Point {  
    double x;  
    double y;  
  
    void translate(double dx, double dy) {  
        x += dx;  
        y += dy;  
    }  
  
    double distance() {  
        double dist;  
        dist = Math.sqrt(x*x + y*y);  
        return dist;  
    }  
}
```



1.2. Attribute visibilities

- Declaring attributes

```
type attributeName;  
Or  
type attributeName = initializationExpression;
```

**Simple type
(not object):**

char
int
byte
short
long
double
float
boolean

**Structured type
(Object):**

type is the name
of the class known
in context of
compilation and
execution



1.2. Attribute visibilities

- Declaring attributes

It is required to put it when your class use another class in another package beside java.lang

```
import java.awt.Color;
```

```
public class Point {  
    double x;  
    double y;  
    Color c;  
    ...  
}
```

A point has color; that color is defined by an object of type Color



1.3. Methods

- “A method declaration defines the executable code that can be invoked, possibly passing a fixed number of values as arguments” The Java Language Specification J. Gosling, Joy B, G. Steel, G. Bracha
- Declaring methods

```
<returnType> MethodName( <parameter list> ) {  
    <method body>  
}
```

Method signature

```
double min(double a, double b){  
    if(a < b)  
        return a;  
    else  
        return b;  
}
```



1.3. Methods

- Declaring methods

```
<returnType> MethodName( <parameter list> ) {  
    <method body>  
}
```

- <returnType>

- When the method returns a value (function) indicates the type of value returned (simple type or name of a class)

```
double min(double a, double b)  
int[] premiers(int n)  
Color getColor()
```

- void if method does not return a value (procedure)

```
void afficher(double[][] m)
```




1.3. Methods

- Declaring methods

```
<returnType> MethodName( <parameter list> ) {  
    <method body>  
}
```

- <parameter list>
 - empty if method does not have parameter

```
int lireEntier()  
void afficher()
```

- a suite type identifier pairs separated by commas

```
double min(double a, double b)  
int[] premiers(int n)  
void afficher(double[][] m)  
int min(int[] tab)  
void setColor(Color c)
```

1.3. Methods

▪ Declaring methods

```
<returnType> MethodName( <parameter list> ) {  
    <method body>  
}
```

▪ <method body>

- Sequence of local variable declarations and instructions
- if the method has a return type, the method body must contain at least one expression where return statement delivers a value compatible with the declared return type

```
double min(double a, double b){  
    double vMin;  
    if(a < b)  
        vMin = a;  
    else  
        vMin = b;  
    return vMin;  
}
```

Local variable

Instruction return

1.3. Methods

- Local variables are variables declared within a method
 - they retain the data that is manipulated by the method
 - they are accessible only in the block in which they were declared and their value is lost when the method finishes its execution

```
void method1(...) {  
    int i;  
    double y;  
    int[] tab;  
    ...  
}
```

Possibility to use the same identifier in two different methods no conflict, it is the local declaration which is used in the method body

```
double method2(...) {  
    double x;  
    double y;  
    double[] tab;  
    ...  
}
```

1.3. Methods

```
public class Point {  
    double x;  
    double y;
```

Two different classes can have the same name members

```
    public void translate(double dx, double dy) {  
        x += dx;  
        y += dy;  
    }  
    ...  
}
```

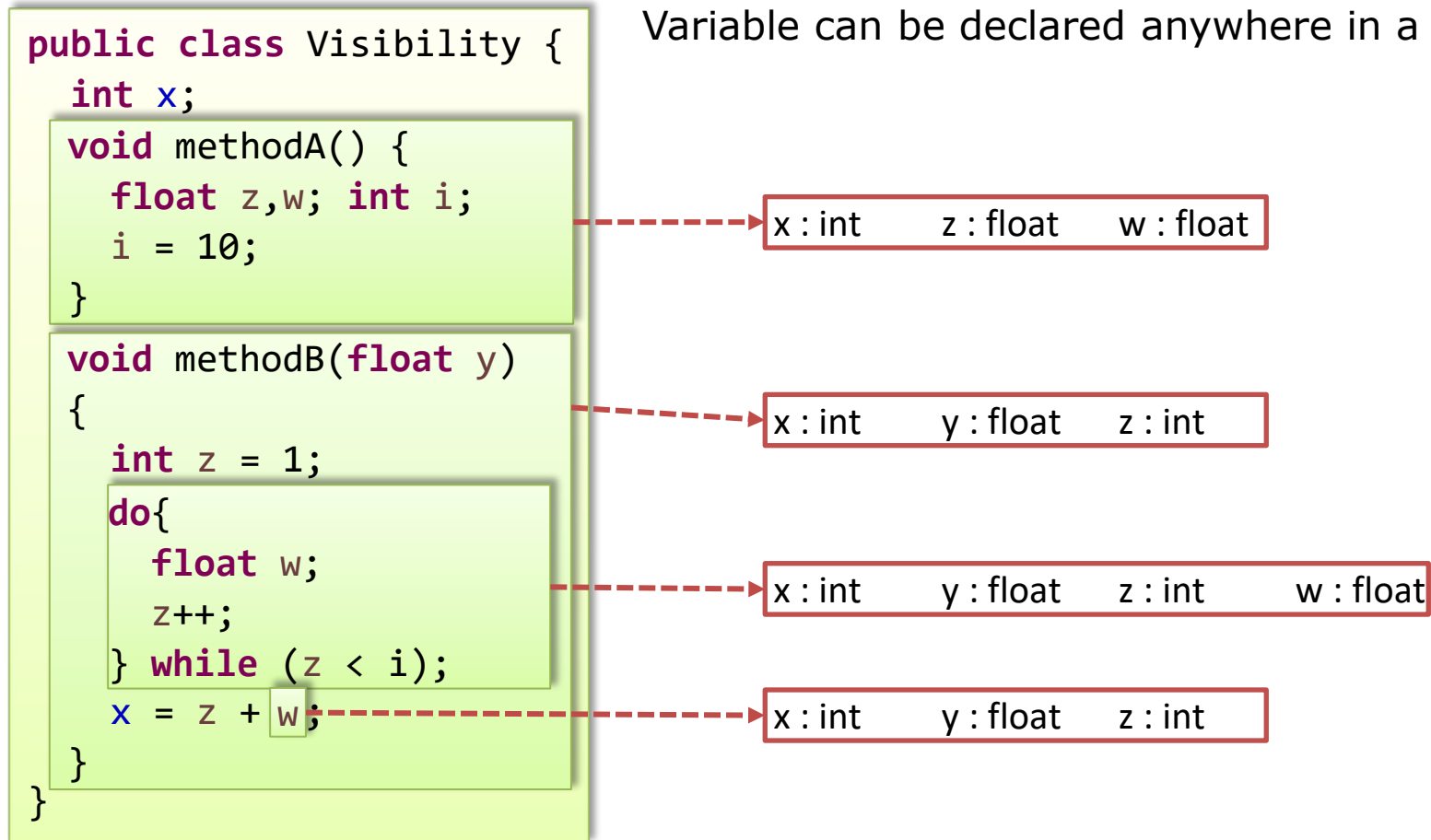
The members of a class will be accessed via objects. Depending on the type of the object (Point or Circle), Java will distinguish what attributes or methods reference is made

```
public class Circle {  
    double x;  
    double y;  
    double r;
```

```
    void translate(double dx, double dy) {  
        x += dx;  
        y += dy;  
    }  
}
```

1.3. Methods

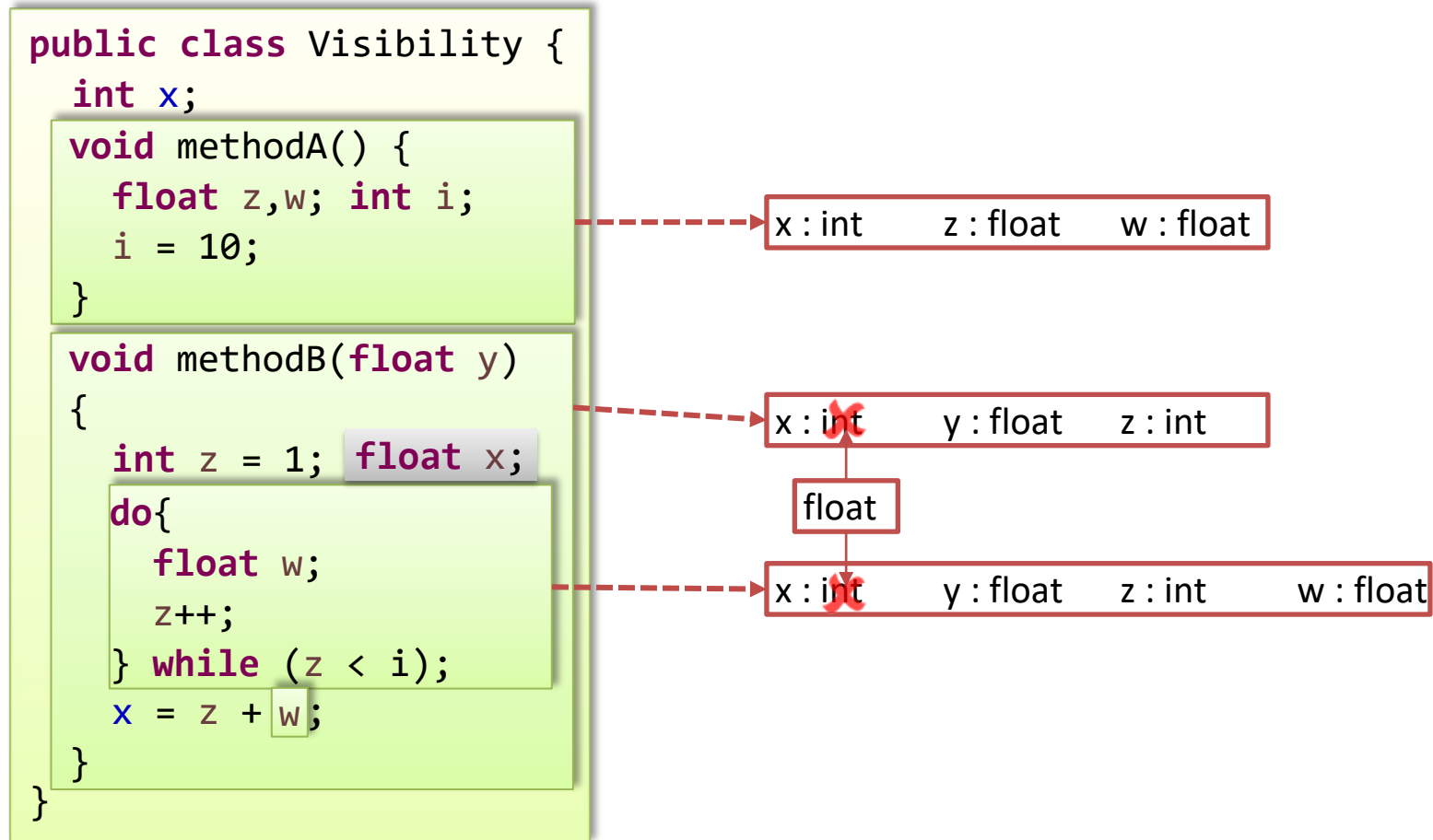
- In general variables are visible within the block (sets of statements between {...}) where they are defined.



1.3. Methods

▪Attention!!!

- Redefining a variable, mask the definition at global block level.



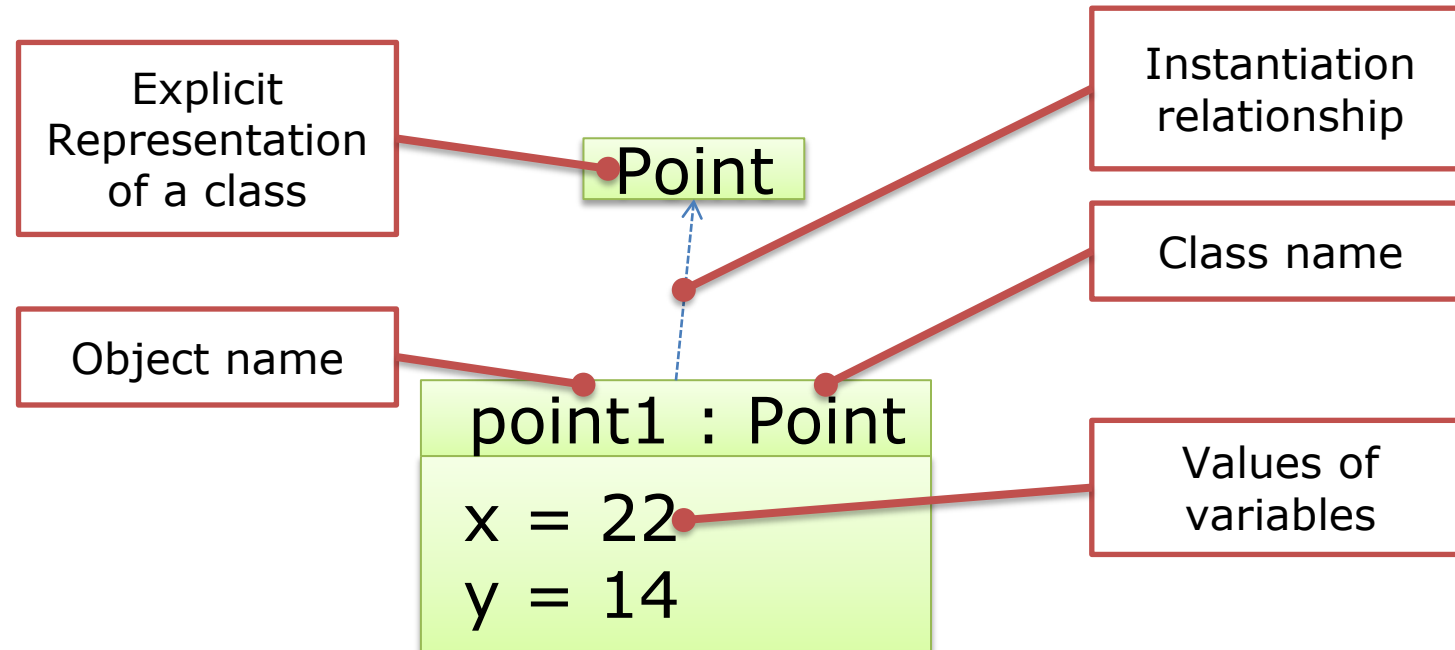


2. Objects

- An object is an instance of (single) class:
 - it conforms to the description it provides,
 - it recognizes a value (its own) for each attribute declared in the class,
 - these values characterize the state of the object
 - it is possible to apply to all operations (method) defined in the class
- Any object has an identity that distinguishes it from other objects:
 - it can be named and be referenced by a name

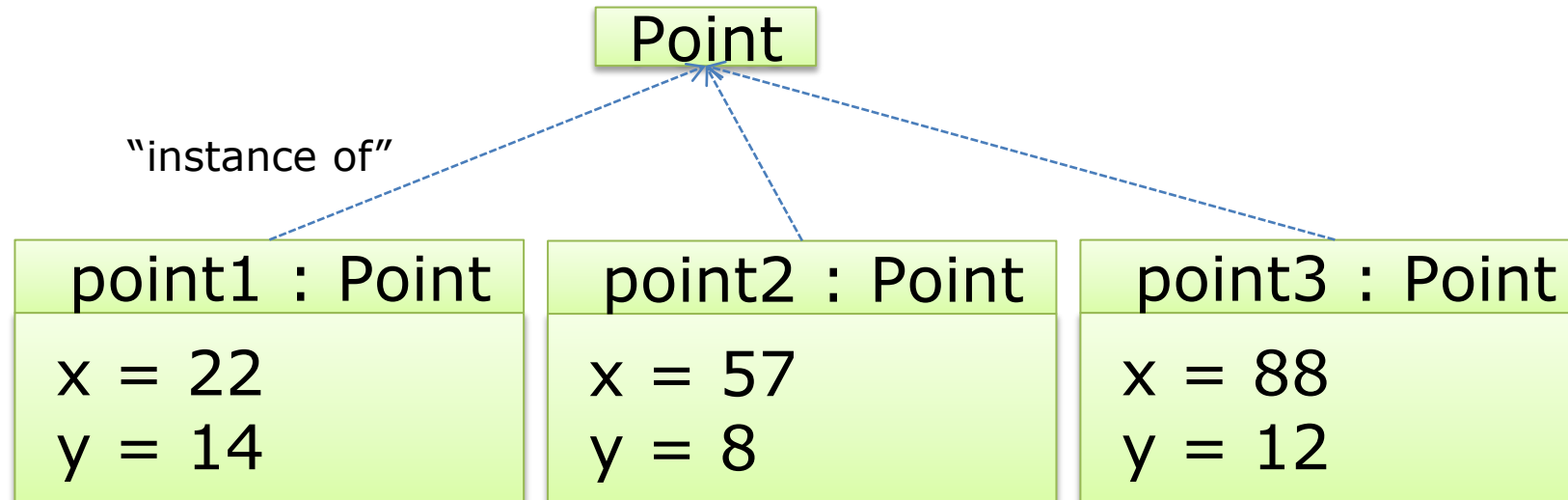
2.1. Object Notation

- Notation of object *point1*, instance of class *Point*



2.1. Object Notation

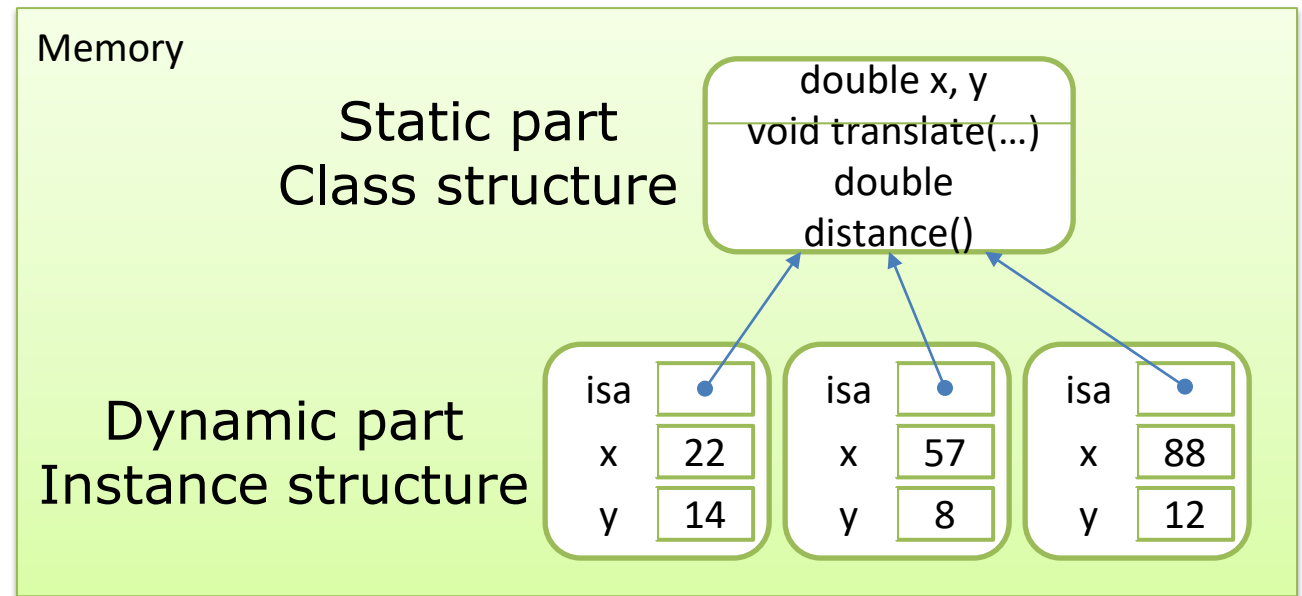
- Each point object that instance of class Point, will contains its own x and y.



2.2. Static and Dynamic parts of Object

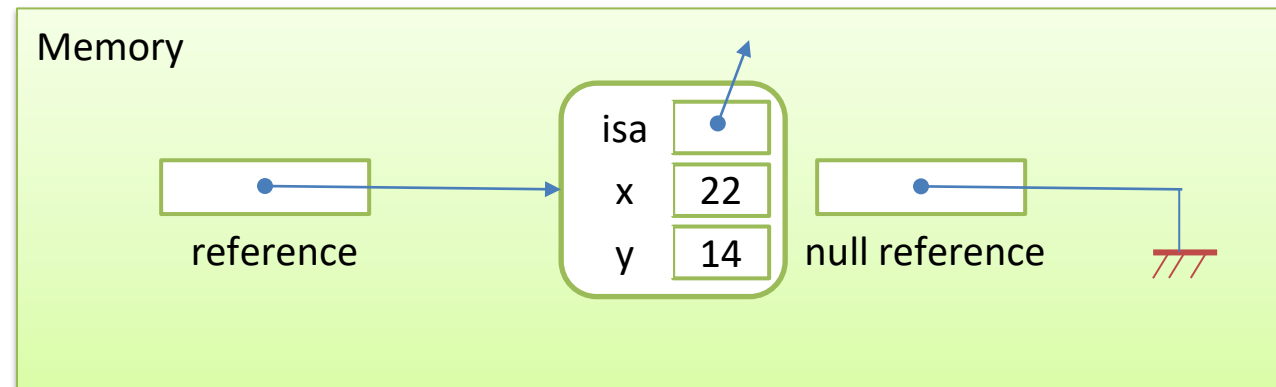


- Object consisting of "Static" part and "Dynamic" part
 - Static part:
 - does not vary from a class instance to another
 - one copy for all instances of a class
 - enable object activation
 - consists of class methods
 - Dynamic part:
 - varies from a class instance to another
 - varies during the lifetime of an object
 - consists of one of each attribute of the class.



2.3. References

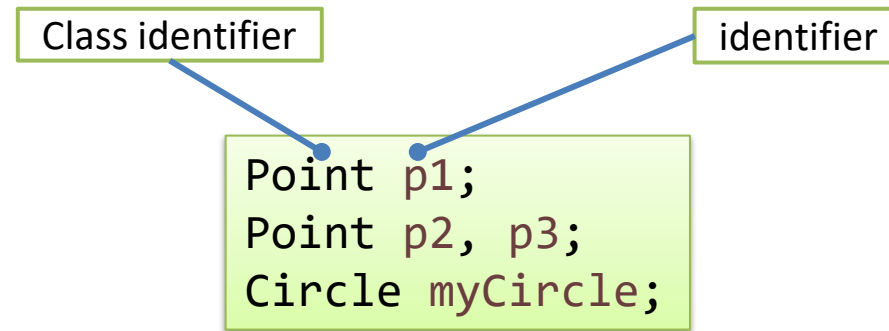
- To describe objects in a class (attributes or variables in body of a method), we use variables of a particular type: the references
- A reference contains address of an object
 - pointer to the data structure corresponding to the attributes (instance variables) specific to the object.



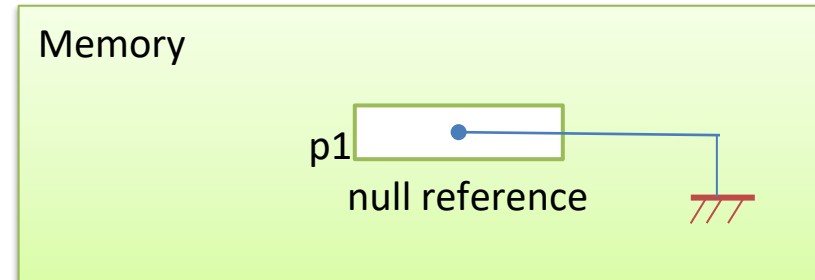
- A reference can have null value
 - no object is accessible by this reference
- Declaring a reference does not create object
 - a reference is not an object, it's name to access an object

2.3. References

- Reference declarations



- By default at the declaration, a reference will be null
- It is not "point" to any object



2.3. References

- Java references: pointers
- Same as a pointer, a reference contains address of a structure
- But unlike pointers, reference has only one operation permitted is the assignment of the same type of reference

```
Point p1;
```

```
...
```

```
Point p2;
```

```
p2 = p1;
```

```
p1++;
```

```
...
```

```
p2 += *p1 + 3;
```

Segmentation fault
Core dump



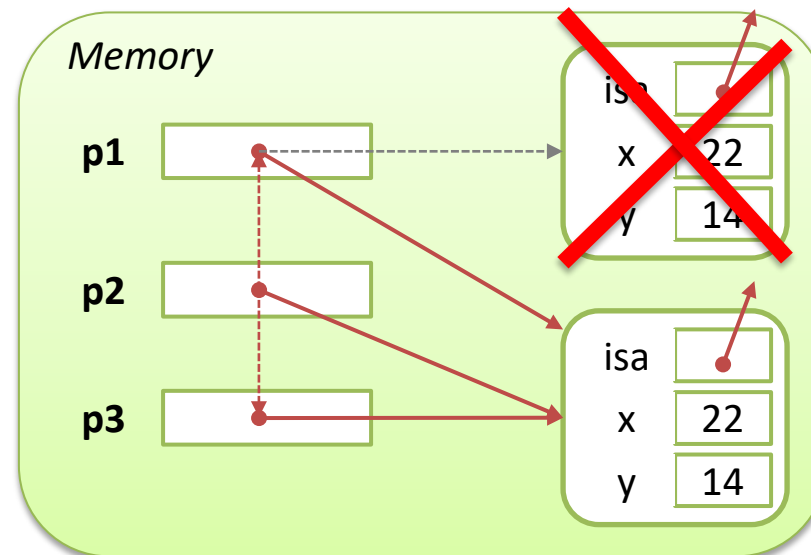
3. Object Creation

- The creation of an object from a class is called instantiation. The created object is an instance of the class.
- Instantiation consists of three phases:
 - 1: obtaining the necessary memory space in dynamic part of the object and initialize the attributes in memory (like a structure)
 - 2: Call for particular methods, constructors, defined in the class. We'll talk later!
 - 3: return a reference to the object (identity) now created and initialized

3.3. Memory Allocation

- The instantiation causes dynamic memory allocation
- In Java, programmer did not have to worry about memory management
 - If an object is no longer referenced (not accessible through a reference), the memory that was allocated is automatically "released" (the "Garbage collector" will retake memory back when desired).
 - Caution: Asynchronous destruction (as managed by a thread), No warranty of destruction (except at the end of program! Or call garbage collector explicitly)

```
Point p1;  
p1 = new Point();  
Point p2 = new Point();  
Point p3 = p2;  
p1 = p2;
```



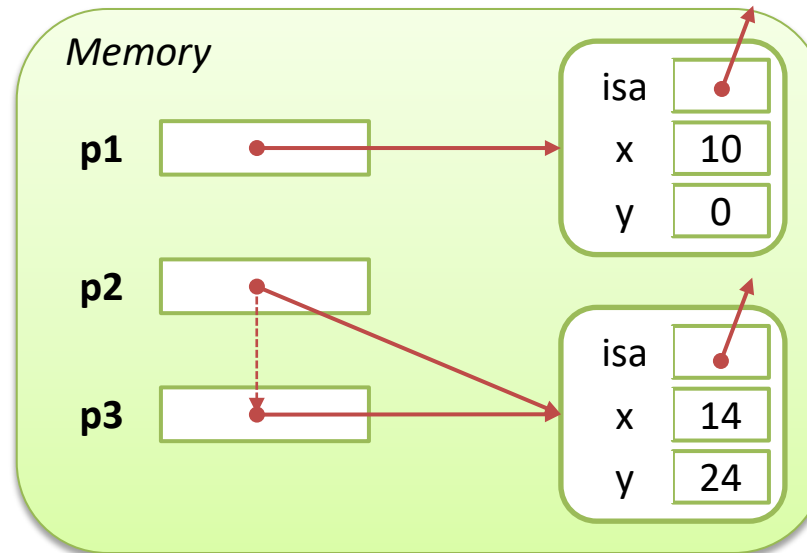
4. Access to Object Attributes

- To access to attributes of an object, we use pointing notation:

objectName.instanceVariableName

Similar the use in C language for accessing to an item of a structure.

```
Point p1;  
p1 = new Point();  
Point p2 = new Point();  
Point p3 = p2;  
p1.x = 10;  
p2.x = 14;  
p3.y = p1.x + p2.x;
```





4.1. Sending Messages

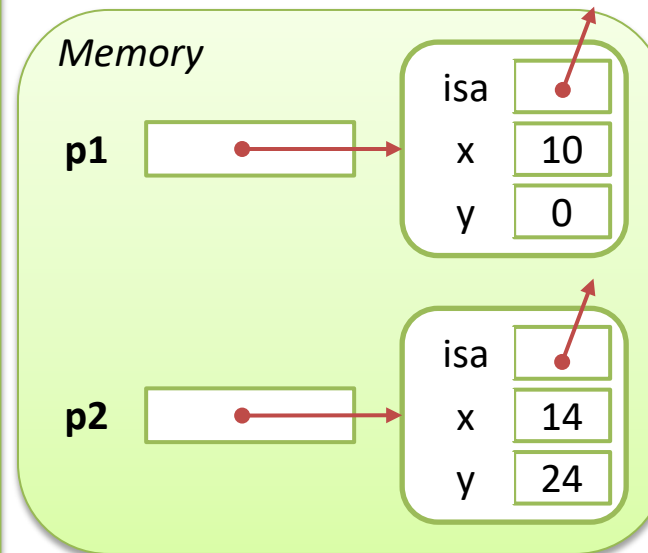
- For "request" to an object to perform an operation (execute one of its methods), must send a message
- A message is composed of 3 parts:
 - a reference that point out the object that the message is sent
 - the method name for execution (this method must be defined in the class of the object)
 - any method parameters
- message sending is similar to calling a function
 - the defined instructions in the method, are executed (it applies to attributes of object that received that message)
 - then, the control is returned to message caller

4.1. Sending Messages

▪ Example in Java: the syntax:

▪ `objectName.methodName(<effective parameters>)`

```
public class Point {  
    double x;  
    double y;  
  
    void translate(double dx, double dy) {  
        x += dx;  
        y += dy;  
    }  
  
    double distance() {  
        return Math.sqrt(x * x + y * y);  
    }  
} // Point
```



```
Point p1 = new Point();  
Point p2 = new Point();  
p1.translate(10.0,10.0);  
p2.translate(p1.x, 3.0 * p1.y);  
System.out.println("distance from p1 to origin " + p1.distance());
```

If the method has no parameter, list is empty, but like in C language, brackets are required.

4.2. Message Parameters

- A parameter of a method can be:
 - a variable of simple type
 - a typed reference by any class (known in the context of compilation)
 - example: to know if a Point is closer to origin than another Point

Add a method to class Point

```
/**
 * Test if the Point (that received message) is closer to origin than another Point.
 *
 * @param p the Point that is used to compare with the one that received message
 * @return true if the point that received message is closer to origin than p,
 * otherwise false.
 */
boolean closerOriginThan(Point p) {
    //...
}
```

Usage

```
Point p1 = new Point();
Point p2 = new Point();
// ...
if (p1.closerOriginThan(p2))
    System.out.println("p1 is further origin p2");
else
    System.out.println("p1 is closer origin than p2");
```

4.2. Message Parameters

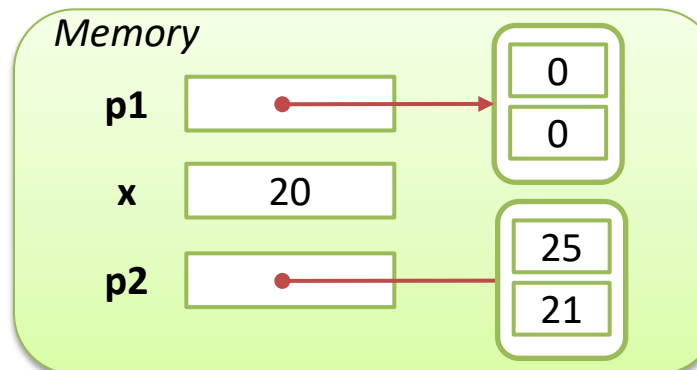
- Passing parameters:

- Passing parameters when sending message is a pass by value.
- At runtime the formal parameter in the method signature corresponds to a local variable to the method block
- It is initialized with the value of the expression defined by the effective parameter.

```
public class Point {  
    ...  
    void foo(int x, Point p) {  
        ...  
        p.translate(10, 10);  
        x = x + 10;  
        p = new Point();  
        p.translate(10, 10);  
        ...  
    }  
}
```

```
p1 = new Point();  
p2 = new Point();  
p2.x = 15;  
p2.y = 11;  
int x = 20;  
p1.foo(x, p2);  
System.out.println("x " + x);  
System.out.println("p2.x " + p2.x);  
System.out.println("p2.y " + p2.y);
```

```
x: 20  
p2.x: 25  
p2.y: 21
```





4.3. Current Object “this”

- Targeted Message is placed on the object (and name, not by calling function)
 - In JAVA (and more generally in OOP) is written:

```
d1 = p1.distance (); d2 = p2.distance ();
```

- In C it probably would have written:

```
d1 = distance (p1); d2 = distance (p2);
```

- the object that receives a message is implicitly passed as an argument to invoked method
- This argument implicitly defined by the keyword **this** (*self*, *current* in other languages)
 - a particular reference
 - refers to the current object (object receiving the message, to which apply instructions in the method body where **this** is used)
 - can be used to make explicit access to specific attributes and methods defined in the class

4.3. Current Object “this”

- In the code of a class to invoke one of the methods it defines (recursion possible)

```
public class Point {  
    double x, y;  
    // constructors  
    Point(double x_, double y_) {  
        x = x_;  
        y = y_;  
    }  
  
    // methods  
    boolean isCloserOriginThan(Point p) {  
        return p.distance() > this.distance();  
    }  
  
    double distance() {  
        return Math.sqrt(x * x + y * y);  
    }  
}
```

- Object that received the message, has sent another message to **itself**
- this is not necessary
- Order of method definitions are not important

4.3. Current Object “this”

- In the code of a class to invoke one of the methods it defines (recursion possible)

```
class Point {  
    double x;  
    double y;  
  
    void translate(int dx, int dy) {  
        x += dx; y += dy;  
        <==> this.x += dx; this.y += dy;  
    }  
  
    double distance() {  
        return Math.sqrt(x * x + y * y);  
    }  
  
    void placeAtPoint(double x1, double y1) {  
        this.x = x1; y = y1;  
    }  
}
```

Implicitly when an attribute is used in the body of a method, it is this attribute of the current object


this is necessarily used to remove ambiguities

4.3. Current Object “this”

- When the receiver of the message must be passed in parameter of a method or its reference returned by the method

```
public class AppSmiley {  
    public static void main(String[] args) {  
        Design d = new Design();  
        SmileyFace v1 = new SmileyFace();  
        // ...  
        d.add(v1);  
    }  
}
```

```
public class Design {  
    // ...  
    void add(SmileyFace v) {  
        // ...  
        v.setDesign(this);  
        // ...  
    }  
  
    int width() {  
        return 500;  
    }  
    // ...  
}
```



```
public class SmileyFace {  
    Design d;  
    int x;  
    int y;  
  
    // ...  
    boolean outOfBorder() {  
        if (x > d.width() /* ... */)   
            return false;  
        else {  
            return true;  
        }  
    }  
  
    // ...  
    void setDesign(Design d) {  
        this.d = d;  
    }  
}
```



5. Encapsulation

- Direct access to the variables of an object is possible in JAVA
- but... is not recommended because contrary to the principle of encapsulation
 - the data of an object must be private (it means that protected and accessible (and all modifiable) through methods provided for this purpose).
- In JAVA, possible in their definition of acting on the visibility (access) of the members (attributes and methods) of a face to face to other classes
- Multiple levels of visibility can be defined before a modifier (private, public, protected, -) the declaration of each attribute, method or constructor

5.1. Class members visibility

	public	Private
class	The class can be used by any other class	
attribute	Attribute is accessible directly from code of any other class	Attribute is accessible only from code of the class that defined it
method	Method can be called from code of any other class	Method is usable only in code of the class that defined it

- *We will see the other levels of visibility (-, **protected**) in detail when the notions of inheritance and packages have been addressed*

5.1. Class members visibility

- The attributes declared as private are totally protected
 - Can not access direct from code of another class

```
Point p1 = new Point();
p1.x = 10;
p1.y = 10;
Point p2 = new Point();
p2.x = p1.x;
p2.y = p1.x + p1.y;
```

```
p1.setX(10);
p1.setY(10);
p2.setX(p1.getX());
p2.setY(p1.getX() + p1.getY());
```

- To modify it, must pass through a method of type procedure
- To access to its value, must pass through a method of type function

```
public class Point {
    private double x;
    private double y;
    public void translate(int dx, int dy) {
        x += dx; y += dy;
    }
    public double distance() {
        return Math.sqrt(x*x+y*y);
    }

    public void setX(double x1){
        x = x1;
    }

    public double getX(){
        return x;
    }

    public void setY(double y1){
        y = y1;
    }

    public double getY(){
        return y;
    }
}
```

5.1. Class members visibility

- The attributes declared as private are totally protected
 - Can not access direct from code of another class

```
public class Point {  
    private double x;  
    private double y;  
    // constructors  
    public Point(double dx, double dy){  
        ...  
    }  
    // methods  
    private double distance() {  
        return Math.sqrt(x*x+y*y);  
    }  
    public boolean isCloserOriginThan(Point p){  
        return p.distance() < distance();  
    }  
    ...  
}
```

- Private method can not be called outside the code of the class that defined it

```
public class X {  
    Point p=new Point(...);  
  
    ... p.distance() ...
```



5.2. Code Robustness

▪Data access is done only through methods

An object can only be used as intended in the design of its class, with no risk of inconsistent use → **Robust code.**

```
/**
 * représentation d'un point de l'écran
 */
public class Pixel {
    // representation in cartesian coordinates
    private int x; // 0 <= x < 1024
    private int y; // 0 <= y < 780

    public int getX() {
        return x;
    }

    public void translate(int dx, int dy) {
        if (((x + dx) < 1024) && ((x + dx) >= 0))
            x = x + dx;
        if (((y + dy) < 780) && ((y + dy) >= 0))
            y = x + dy;
    }
    // ...
} // Pixel
```

Code that use class Pixel

```
Pixel p1 = new Pixel();
p1.translate(100, 100);
p1.translate(1000, -300);
// ...
p1.x = -100;
```

Impossible to have a pixel in incoherent state

(x < 0 or x >= 1024 or y < 0 or y >= 780)

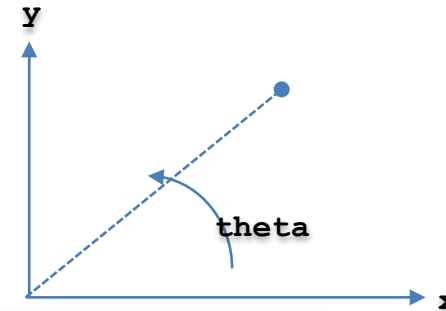
5.3. Extensible development

- **Hide implementation → facilitates development of software**

```
/**
 * representation a point in plan
 */
public class Point {
    // representation of Cartesian coordinates
    private double x;
    private double y;

    public double distance() {
        return Math.sqrt(x * x + y * y);
    }

    public void translate(double dx, double dy) {
        // ...
    }
    // ...
} // Point
```



```
// representation in
// polar coordinates
private double ro;
private double tetha;
```

```
return ro;
```

Update implementation no impact on code that use this class if static part (class's interface) remain unchanged

Code that use class Point:

```
Point p1 = new Point();
p1.translater(x1,y1);
double d = p1.distance();
...
```

Test



Question	Possible answers	Correct Answer
1. Object is:	a) Any visible thing b) An Identifiable entity model handled by software c) Any movable thing on earth	
2. Completing blank field:	The creation of an object from a class is called	
3. Is it possible to have object without class?:	a) possible b) impossible	
4. Encapsulation is:	a) Class hiding b) Data hiding c) Object hiding	
5. Keyword this is refer to:	a) Current method b) Current attribute c) Current object	

Practice



No.	Exercise	Solution
1.	Create Point class	
2,	Create Car class	
3,	Create Pen class	



Summarize

- Objects are all identifiable entity models, concrete or abstract, handled by the software application.
- Class is schema/mold/object model.
- We can Instantiate many objects from one class.
- Inheritance inherits all attributes and methods of super class.
- Java provides implements OOP.



Reference

- “Object Oriented Programming” by Ph. Genoud – Université Joseph Fourier, 2006
- “Building Skills in Object-Oriented Design” by Steven F. Lott, 2009
- “Exercices en Java” 2nd edition by Groupe Eyrolles, 2006
- “Java Programming – Introductory” by Joyce Farrell, 1999
- “Java Examples in a Nutshell” 3rd Edition by David Flanagan, 2004
- “Programmer en Java” 5th edition by Groupe Eyrolles, 2008
- <https://docs.oracle.com/javase/tutorial/java/javaOO/>