Object Oriented Programming

LESSON 03

Classes and Objects (P2)

Outline

1. Constructors
2. Overloading methods
3. Class variables & Constants
4. Static initialization
5. Finalization

Overview

In this chapter, you are going to learn about

- Know in detail about constructors
- Know how to create methods with the same name
- Know how to create and use variables and constants
- Know how to initialize static members
- Know how to destroy objects

Learning content

1. Constructors
   - Explicit Constructor
   - Multiple Constructors
   - Calling a constructor from another constructor
2. Overloading methods
   - Choosing a method
   - Different return types
   - Example
3. Class variables and constants
   - Class variables
   - Class methods
   - Constants
4. Static initialization
   - Variable initialization
   - Static initializer
   - Invocation on Static initializer
5. Finalization
   - Revision
   - Finalize method
   - Example

Pre-Test

Question	Possible answers	Correct Answer	Feedback of the question
What can you do with PC?	1. Eat 2. Coding 3. Watch Movies	1. Coding 2. Watch Movies	We can not eat PC

Pre-Test

Question	Possible answers	Correct Answer	Feedback of the question
When PC is running, Which one is changeable part?	1. Monitor 2. CPU 3. HDD	1. Monitor	We can not change CPU and HDD, while PC is running

Pre-Test

Question	Possible answers	Correct Answer	Feedback of the question
What are parameters needed for constructing a Player object in game?	1. Eyes color 2. Name 3. Weapon	2. Name	If no weapon, person still can fight. Without eyes, in-game-person still can fight.

1. Constructors

- Constructor of a class:
  - Special method for creating object of this class
  - Method that has the same name as class name

- Constructor's role:
  - Do some necessary initializations for the new created object

- Every class in Java has at least one constructor
  - If a class does not define explicit constructor, a default constructor (without arguments and no t done any special initialization) is invoked

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

  public Point(double dx, double dy) {
    …
  }
  …
}
```

1.1. Explicit Constructors

Explicit declaration of a constructor

- The default constructor is masked
- Constructor name identical to class name
- No return type and not also keyword void in method signature
- Implicitly return (instance of class (this)
- No return instruction in constructor

Creation of an object

new Point()

new Point(15, 14)

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

public Point(double x, double y) {
  this.x = x;
  this.y = y;
}

public void translate(double dx, double dy) {
  x += dx;
  y += dy;
}
public double distance() {
return Math.sqrt(x * x + y * y);
}
public void setX(double x) {
this.x = x;
}
public double getX() {
return x;
}
… idem for y
```

1.2. Multiple Constructors

- It is possible to define multiple constructors in the same class
  - It is possible to initialize an object in multiple different ways

- A class can many constructors
- Each constructor has the same name as class name
- The compiler distinguish constructors by:
  - number
  - type
  - position of parameters
- We can say that constructors can be overloaded

1

```
public Point(double x, double y) {
  this.x = x;
  this.y = y;
}
```

2

```
public Point() {
  this.x = this.y = 0;
}
```

3

```
public Point(Point p) {
  this.x = p.x;
  this.y = p.y;
}
```

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

  …
}
```

1. 3. Calling a constructor from another constructor

- In classes that define many constructors, a constructor can call another constructor of this class
- Call this(…)
  - reference to constructor of class with corresponding parameters
  - can only be used at first instruction in body of constructor, it can not be invoked after other in structions
    - (we will understand more after we talk about inheritance and about automatic invocation of constructors of super class)

1. 3. Calling a constructor from another constructor

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

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

  private Point(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public Point() {
    this(0,0);
  }

  public Point(Point p) {
    this(p.x, p.y);
  }
  …
}
```

Advantages:
- Code factorization
- A generic constructor invoked by many constructors

Possibility of defining many private constructors

2. Overloading methods

- Overloading is not limited to only constructors, it is possible for any method
- It is possible to define methods which has the same name but different arguments
- Methods are distinguished by their signature:
  - name
  - number of arguments
  - type of arguments
  - position of arguments

2.1. Choosing a method

- When an overloading method is invoked:
  - the compiler automatically select a method that has number and type of parameters correspo nding to number and type of parameter passed in the call of this method

```
public class Compare {
  int max(int a, int b) {
    if (a > b)
      return a;
    else
      return b;
  }

  String max(String a, String b) {
    if (a.compareTo(b) > 0)
      return a;
    else
      return b;
  }
}
```

2.1. Choosing a method

- When an overloading method is invoked:
  - the compiler automatically select a method that has number and type of parameters correspo nding to number and type of parameter passed in the call of this method

```
public class MyMain {
  public static void main(String args[]) {
    String s1 = "Melbourne";
    String s2 = "Sydney";
    String s3 = "Adelaide";
    int a = 10;
    int b = 20;
    Compare c = new Compare();
    System.out.println(c.max(a, b));
    System.out.println(c.max(s1, s2));
    System.out.println(c.max(s1, s3));
  }
}
```

2.3. Example

- Another example of overloading method distance() of class Point

```
public class Point {
  // attributes
  private double x;
  private double y;
  // constructors
  public Point(double x, double y) {
    // ...
  }

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

  public double distance(Point p) {
    return Math.sqrt((x - p.x)*(x - p.x) + (y - p.y)*(y - p.y));
  }
  // ...
}
```

```
Point p1=new Point(10,10);
Point p2=new Point(15,14);
p1.distance();
p1.distance(p2);
```

3.1. Class variables

```java
public class Point {
  private double x;
  private double y;
  ...
  /** Compare 2 points Cartesian
   * @param p another point
   * @require ValidPoint : p!=null
   * @return true if points are nearly equal in range 1.0e-5
   */
  public boolean equal(Point p) {
    double dx = x - p.abscissa();
    double dy = y - p.ordinate();
    if (dx < 0) dx = -dx;
    if (dy < 0) dy = -dy;
    return (dx < 1.0e-5 && dy < 1.0e-5);
  }
  ...
}
```

Edit class Point in order to change the value of the constant of imprecision

3.1. Class variables

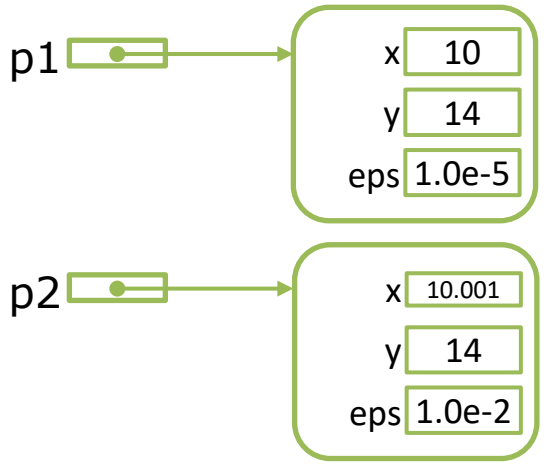
```java
public class Point {
  private double x;
  private double y;
  private double eps = 1.0e-5;
  public void setEps(double eps_in){ eps = eps_in; }
  public double getEps(){ return eps; }

  ...
 /** Compare 2 points Cartesian
  * @param p another point
  * @require ValidPoint : p!=null
  * @return true if points are nearly equal in range 1.0e-5
  */
  public boolean equal(Point p) {
    double dx = x - p.abscissa();
    double dy = y - p.ordinate();
    if (dx < 0) dx = -dx;
    if (dy < 0) dy = -dy;
    return (dx < eps && dy < eps);
  }
  ...
}
```

1st solution:
Add a variable (an attribute)
eps with getter and setter method

What are the problems linked to this solution?

3.1. Class variables

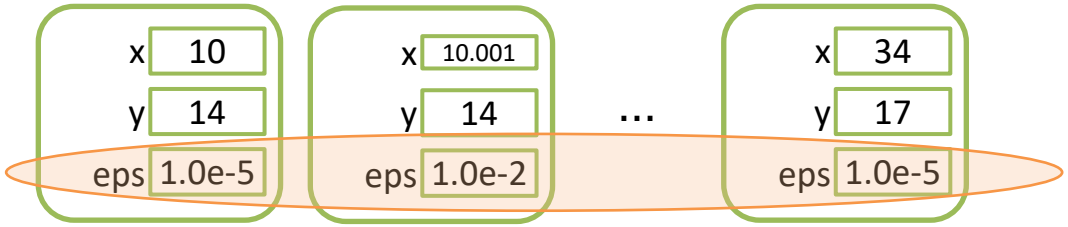
```
Point p1 = new Point(10,14);
Point p2 = new Point(10.001,14.001);
p2.setEps(10e-2);

System.out.println(p1.equal(p2));
//-> false if we use precision of p1 (0.00001)

System.out.println(p2.equal(p1));
//-> false if we use precision of p2 (0.01)
```



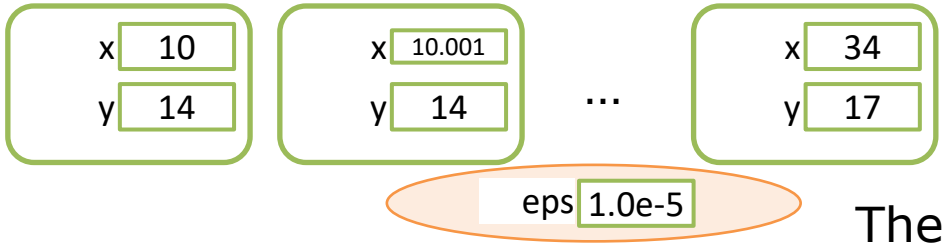
3.1. Class variables

- Each instance has its own value of precision

equal is no longer guaranteed to be symmetrical



- This information does not concern to any specific instance but the whole **Point** domain

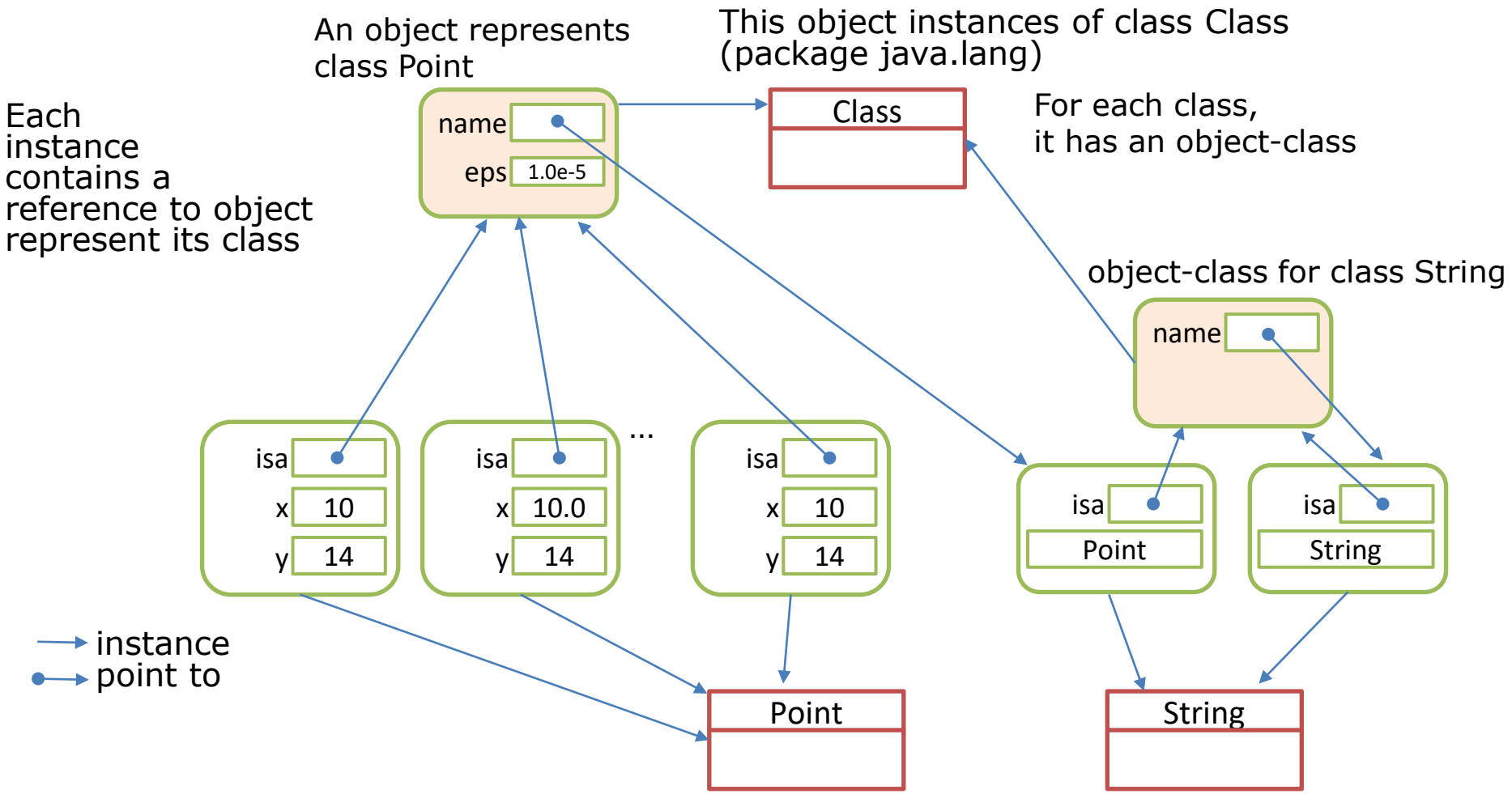


How to represent this set?

The value must be outside of objects

But in Java, nothing can be defined outside of object.
No global variables ☺

3.1. Class variables



3.1. Class variables

- `public String getName()`
  Returns the name of the entity (class, interface, array class, primitive type, or void) represente d by this Class object, as a String.

- `public Class getSuperclass()`
  Returns the Class representing the superclass of the entity (class, interface, primitive type or vo id) represented by this Class
- `public Object newInstance() throws InstantiationException, IllegalAccessException`
  Creates a new instance of the class represented by this Class object.
  …

- `public Field getField(String name) throws NoSuchFieldException, SecurityException`
  Returns a Field object that reflects the specified public member field of the class or interface rep resented by this Class object.
- `public Method[] getMethods() throws SecurityException`
  Returns an array containing Method objects reflecting all the public member methods of the cla ss or interface represented by this Class object, including those declared by the class or interfac e and those inherited from superclasses and superinterfaces.
  …

3.1. Class variables

Members of a class are themselves represented by objects whose classes are defined in java.lang.reflect. (Introspection objects)

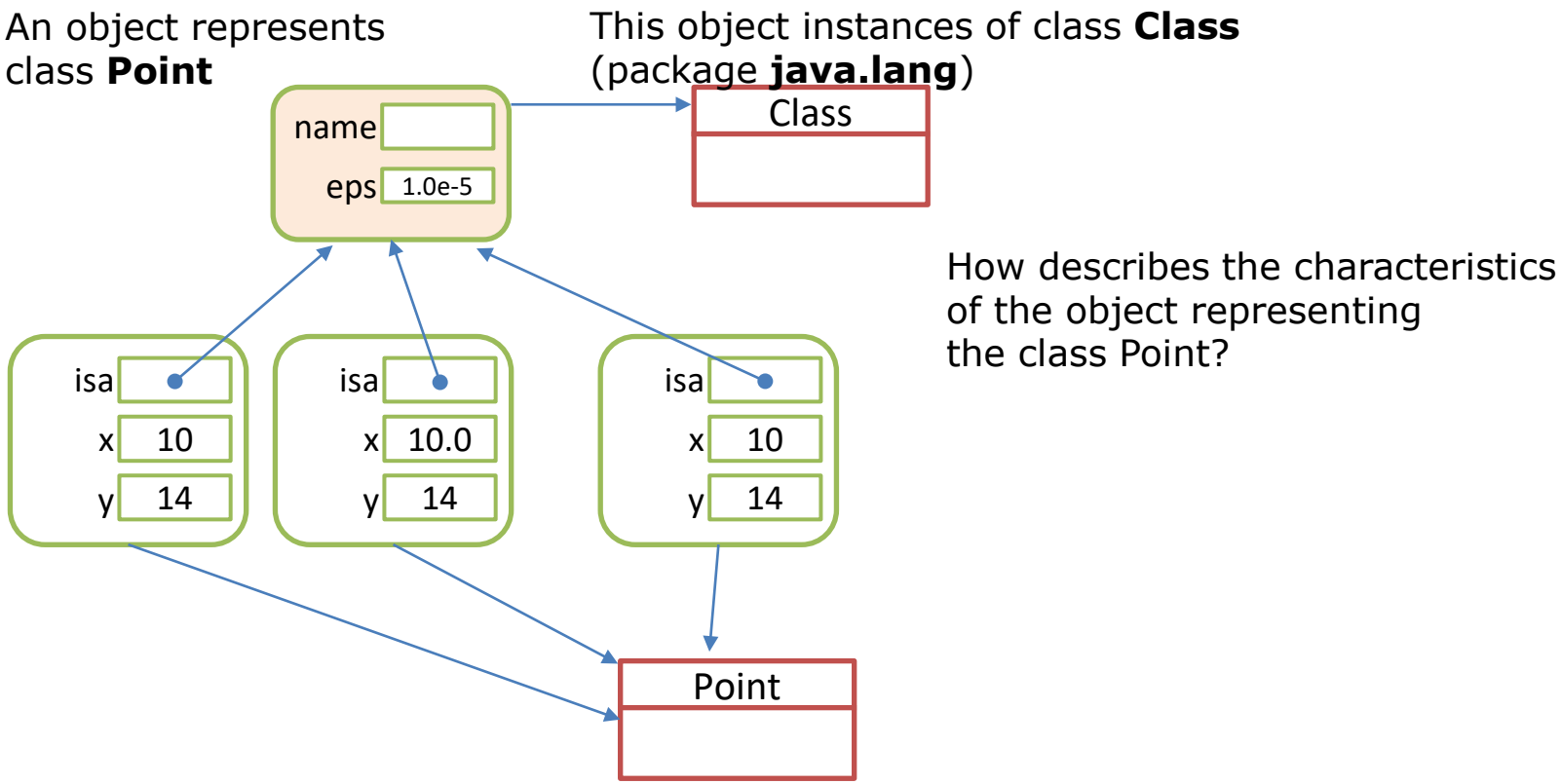
- `public Field getField(String name) throws NoSuchFieldException, SecurityException`

  Returns a Field object that reflects the specified public member field of the class or interface rep resented by this Class object.
- `public Method[] getMethods() throws SecurityException`

  Returns an array containing Method objects reflecting all the public member methods of the cla ss or interface represented by this Class object, including those declared by the class or interfac e and those inherited from superclasses and superinterfaces.

  …

3.1. Class variables



How describes the characteristics of the object representing the class Point?

In java this is done within the code of the **Point** class.
In the code of the Point class we must distinguish the level at which the descriptions apply (class or instances)
The descriptions concerning the object-class are preceded by the keyword **static**

3.1. Class variables

```
public class Point {
  private double x;
  private double y;
  private static double eps = 1.0e-5;
  public void setEps(double eps_in){ eps = eps_in;
  public double getEps(){ return eps; }
  ...
 /** Compare 2 points Cartesian
  * @param p another point
  * @require ValidPoint : p!=null
  * @return true if points are nearly
  * equal in range 1.0e-5
  */
  public boolean equal(Point p) {
    double dx = x - p.abscissa();
    double dy = y - p.ordinate();
    if (dx < 0) dx = -dx;
    if (dy < 0) dy = -dy;
    return (dx < eps && dy < eps);
  }
  ...
}
```

Instance variables

Class variable
declared with keyword static

In class body,
the class variables are used like other variables…
or nearly the same use

3.1. Class variables

```
public class Point {
  private double x;
  private double y;
  private static double eps = 1.0e-5;
  public void setEps(double eps_in){ eps = eps_in; }
  public double getEps(){ return eps; }
  ...
 /** Compare 2 points Cartesian
  * @param p another point
  * @require ValidPoint : p!=null
  * @return true if points are nearly
  * equal in range 1.0e-5
  */
  public boolean equal(Point p) {
    double dx = x - p.abscissa();
    double dy = y - p.ordinate();
    if (dx < 0) dx = -dx;
    if (dy < 0) dy = -dy;
    return (dx < eps && dy < eps);
  }
  ...
}
```

It is necessary to have a reference to an instance of Point for sending message getEps and setEps

```
Point p = new Point(10, 14);
…
p.setEps(1.e-8);
```

eps is not attribute of object Point, but an attribute of object represents class Point

How to design object-class Point?

3.1. Class variables

```java
public class Point {
  private double x;
  private double y;
  private static double eps = 1.0e-5;
  public void setEps(double eps_in){ eps = eps_in; }
  public double getEps(){ return eps; }
  ...
 /** Compare 2 points Cartesian
  * @param p another point
  * @require ValidPoint : p!=null
  * @return true if points are nearly
  * equal in range 1.0e-5
  */
  public boolean equal(Point p) {
    double dx = x - p.abscissa();
    double dy = y - p.ordinate();
    if (dx < 0) dx = -dx;
    if (dy < 0) dy = -dy;
    return (dx < eps && dy < eps);
  }
  ...
}
```

The same way as instance variables that access via references of class (instance name), class variables are accessed through class name.

```
ClassName.variableName

Ex: Point.eps
```

3.2. Class methods

```
public class Point {
  private double x;
  private double y;
  private static double eps = 1.0e-5;
  public static void setEps(double eps_in){ eps = eps_in; }
  public static double getEps(){ return eps; }
  ...
 /** Compare 2 points Cartesian
  * @param p another point
  * @require ValidPoint : p!=null
  * @return true if points are nearly
  * equal in range 1.0e-5
  */
  public boolean equal(Point p) {
    double dx = x - p.abscissa();
    double dy = y - p.ordinate();
    if (dx < 0) dx = -dx;
    if (dy < 0) dy = -dy;
    return (dx < eps && dy < eps);
  }
  ...
}
```

The method getEps and setEps are no more needed to associate with instance of class Point but with object-class Point.

Declaring class methods (static methods)

Warning: in body of static method, it is not possible to access non-static methods of this class (how about keyword *this*?)

Calling class method:

```
Point.setEps(1.0e-8)
```

3.3. Constants

- Named constants can be defined by class variables that their values can not be modified

```
public class Pixel {
public static final int MAX_X = 1024;
public static final int MAX_Y = 768;
// variables d'instance
private int x;
private int y;

// ...
// constructors
// create a Pixel at x, y
public Pixel() {
// ...
}
// ...
}
```

Declaring class variables

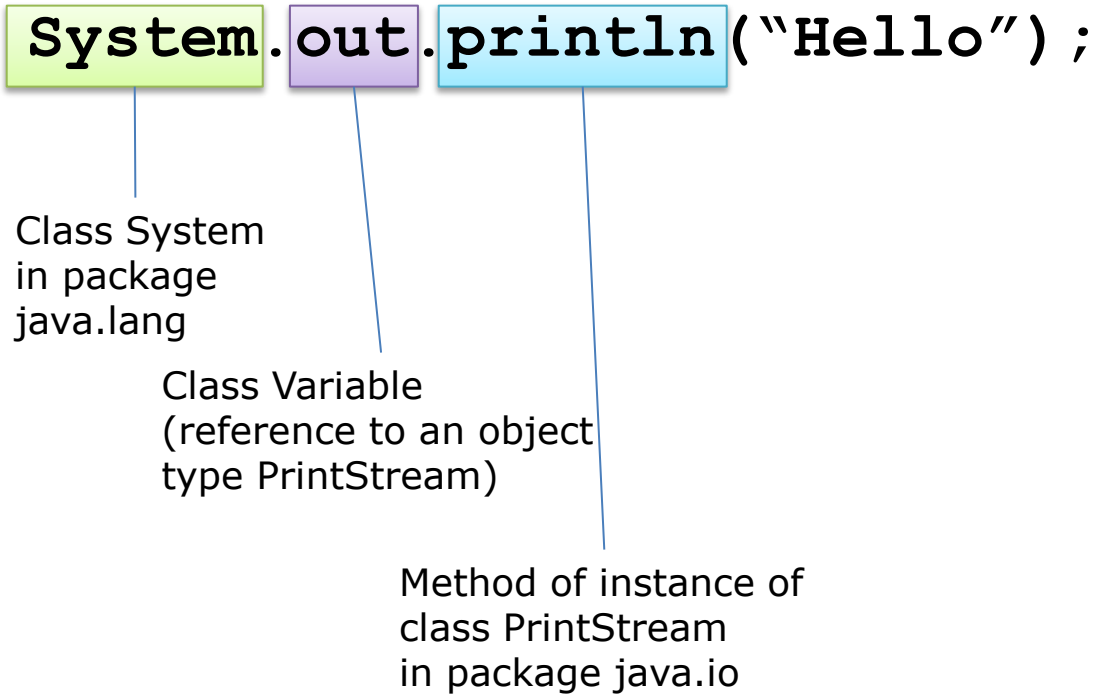
The modifier final is used for indicating the value of a variable (this class) can never be changed

4. Static initialization

- Static members are members whose statement is preceded by the static modifier.
  - Class variables: defined and exists independently from instances
  - Class method: method that its invocation is done without sending message to any instance of the class.
- Accessing to static members:
  - directly access by its name in the code of the class where it is defined,
  - with prefix by its class name outside the code of the class.
    - ClassName.VariableName
    - ClassName.MethodName(parameters list)
    - is not conditioned by the existence of the class instances

    Math.PI Math.cos(x) Math.toRadians(90)…

4. Static initialization



4. Static initialization

- Starting execution point in Java application is the main method of the class specified by Virtual Machine
- This method signature:

public static void main(String[] args)

- String[] args: is table of String objects containing argument from command line

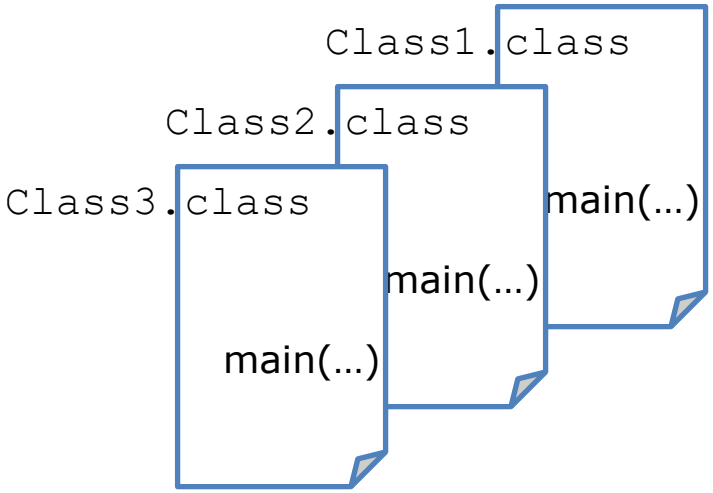
```
public class TestArgs {
  public static void main(String[] args) {
    System.out.println("Number of arguments : " + args.length);
    for (int i = 0; i < args.length; i++)
      System.out.println(" argument " + i + " = " + args[i]);
  }
}
```

java TestArgs arg1 20 35.0 ➔

```
Number of arguments : 3
 argument 0 = arg1
 argument 1 = 20
 argument 2 = 35.0
```

4. Static initialization

- Different classes in the same application can eventually has its own method **main()**
- At execution time, no hesitation on which method main() get executed
  - it is the one of the class indicated at JVM



- Advantage
  - possibility to define independently a unit test for each class of a system

4.1. Variable initialization

- The instance variables and class variables may have "initializers" associated with their declaration

```
Modifiers type variableName = expression;
```

```
private double x = 10;
private double y = x + 2;
private double z = Math.cos(Math.PI / 2);
private static int nbPoints = 0;
```

- Class variables initialized the first time the class is loaded
- Instance variables initialized when an object is created
- Initializations take place in the order of statements.

```
public class TestInit {
  private double y = x + 1;
  private double x = 14.0;
  ...
}
```

```
TestInit.java:2: error: illegal
forward reference
private double y = x + 1;
                   ^
1 error
```

4.2. Static initializer

▪If the necessary initialization for class variables can not be made directly with initializers (expression), JAVA provides method (algorithm) for initializing these: **static initializer**

xs a random number (double) in between 0 and 10
ys sum of n number drawn randomly,
n is integer part of xs
zs sum of xs and ys

```
private static double xs = 10 * Math.random();
private static double ys;
private static double zs = xs + ys;

static {
  int n = (int) xs;
  ys = 0;
  for (int i = 0; i < n; i++)
  ys = ys + Math.random();
}
```

- **Declaring a static initializer**
  - *static { code block }*
  - method
    - no parameter
    - no return value
    - no name

4.3. Invocation on Static initializer

- Invocation automatic and once when the class is loaded
- Invocation in the order of appearance in the class code

```
public class StaticInit {
  private static double xs = 10 * Math.random();
  private static double ys;
  static {
    int n = (int) xs;
    ys = 0;
    for (int i = 0; i < n; i++)
      ys = ys + Math.random();
  }
  private static double zs = xs + ys;

  public static void main(String[] args) {
    System.out.println("xs:"+xs);
    System.out.println("ys:"+ys);
    System.out.println("zs:"+zs);
  }
}
```

5.1. Revision

- Freeing memory allocated to objects is automatic
  - when an object is no longer referenced the "garbage collector" to recover the memory space t hat have been reserved.
- The “garbage collector" is a process (thread) that runs in the background task with a lower priority
  - It runs:
    - when there is no other activity (waiting for keyboard input or a mouse event)
    - when the Java interpreter has no more memory available
      - Only time it execute while other activities with highest priority are being executed (a nd therefore actually slows the system)
- May be less effective than explicit memory management but programming much easier and safe.

5.2. Finalize method

- "Garbage collector" manages automatically memory resources used by objects
- Sometimes an object can hold other resources (file descriptors, sockets ....) to be free when the object is destroyed.
- method of "finalization" for this purpose
    - closes opened files, complete network connections ... before the destruction of objects
- the "finalization" method:
    - Instance method
    - must be called "finalize()"
    - no parameter, return type void
    - is invoked just before "Garbage collector" does not recover memory space assigned to object,
- JAVA makes no guarantee about when and in what order the recovery the memory will be performed,
    - impossible to make assumptions about the order in which the methods finalization will be invoked

5.3. Example

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

  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
  public void translate(double dx, double dy) {
    x += dx;
    y += dy;
  }
  public String toString() {
    return "Point[x:" + x +
           ", y:" + y + "]";
  }

  public void finalize() {
    System.out.println("finalization of " + this);
  }
}
```

```
Point p1 = new Point(14, 14);
Point p2 = new Point(10, 10);
System.out.println(p1);
System.out.println(p2);
p1.translate(10, 10);
p1 = null;
System.gc();
System.out.println(p1);
System.out.println(p2);
```

```
Point[x:14.0, y:14.0]
Point[x:10.0, y:10.0]
null
finalisation de Point[x:24.0, y:24.0]
Point[x:10.0, y:10.0]
```

Test

Question	Possible answers	Correct Answer
1. Constructors:	a) Method that has no parameter b) Method that has same name as class name and return void c) Method that same name as class name without return type	
2. Overloading:	Overloading …… are the methods with the same name but differ ent …….. of parameters.	
3. Which one is class variable?	a) private int x=5; b) public static final int y=5; c) private static int z=5;	
4. Static initialization is done:	a) While creating class instance b) While loading class c) While using class	
5. Finalization method is named:	a) main() b) final() c) finalize() d) finally()	

Practice

No.	Exercise	Solution
1.	Create Time class containing hours, minutes and seconds as attributes and define some methods.	
2,	Create Currency class that is used as currency	
3,	Create Converter class that can converts currencies	

Summarize

- Constructors are special methods with the same name as class name without any return value. They are used to initialize attributes.
- Overloading methods are method with the same name but different number of arguments, type of arguments, or order of arguments.
- Class variables are shared variables, one copy for one class. Constants are class variables declared with keyword final.
- Static initializer is like a method without name, no argument and no return.
- To clean up well the memory we used, we create method finalize() which is call automatically by JVM before garbage collector.

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/