Classes of objects and object oriented programming

Principle of object oriented programming
Using objects
Definition of classes
Subclasses and inheritance
Nested top-level classes and interfaces; inner classes

Principle of object oriented programming

classes and objects:

a class is a means of structuring some data
as well as the operations on these data (= for all the program)

class = group of variables
and methods (functions) which apply on these variables

example:

> circles are defined by their radius and the position of their center
> → based on these data, we can compute their circumference and area

a class is a general definition, a pattern → it is instantiated into objects

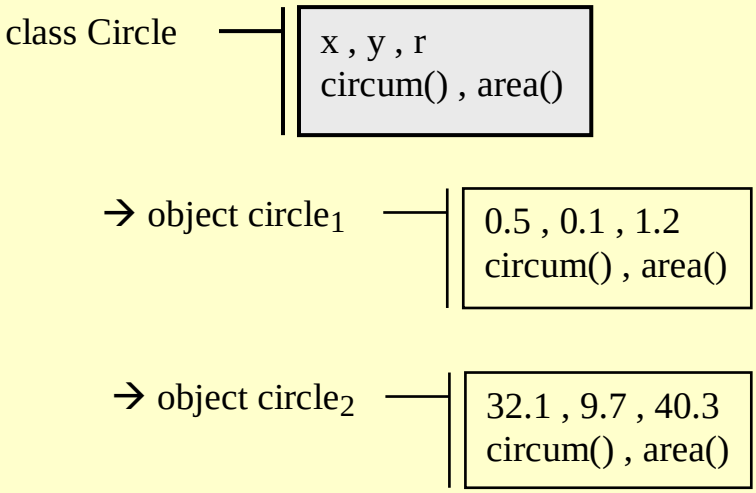
- in a class, we just declare (or define) the variables
- in each object instance of this class, we assign values to these variables
- the methods defined in the class are readily available for each object (they can be applied over it)

convention: we ALWAYS capitalize the first letter of the name of a class

example:

the class (or definition) of circles, named Circle, is instantiated into circles

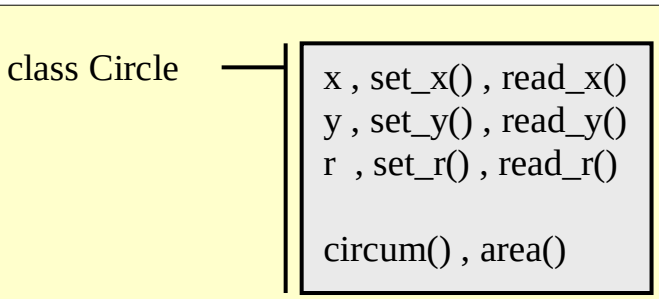
these circles are objects, instances of the class Circle



encapsulation:

inside the class, we access the variables directly
but outside the class, it is preferable to access them
only through some dedicated methods of the class

example:



→ to assign value to x of $circle_1$, apply set_x(value) over object $circle_1$

→ read_x() over object $circle_1$ returns the value of x of $circle_1$

instance variables and methods:

one instance of the variable or method
for each object (instance of the class)

→ characteristics of the object which differentiate it
from the other objects (radius of a given circle…)
; method to be applied over a certain object (area of a given circle…)

class (or static) variables and methods:

only one version of the variable or method
which is related to the class in general
and not to any of its object instances in particular

→ constant (PI of the class Math) or counter of instances of this class
(number of objects created by instanciation of the class)
; predefined methods (exp , sqrt) of the class Math …

subclass, superclass, inheritance:

a subclass is a less general, less vague, version of a certain class
→ contains more properties (more variables and methods)
which define it more precisely

the class from which a subclass originates is the superclass of this subclass

in Java, a class can have only ONE superclass
→ use interfaces instead of multiple inheritance

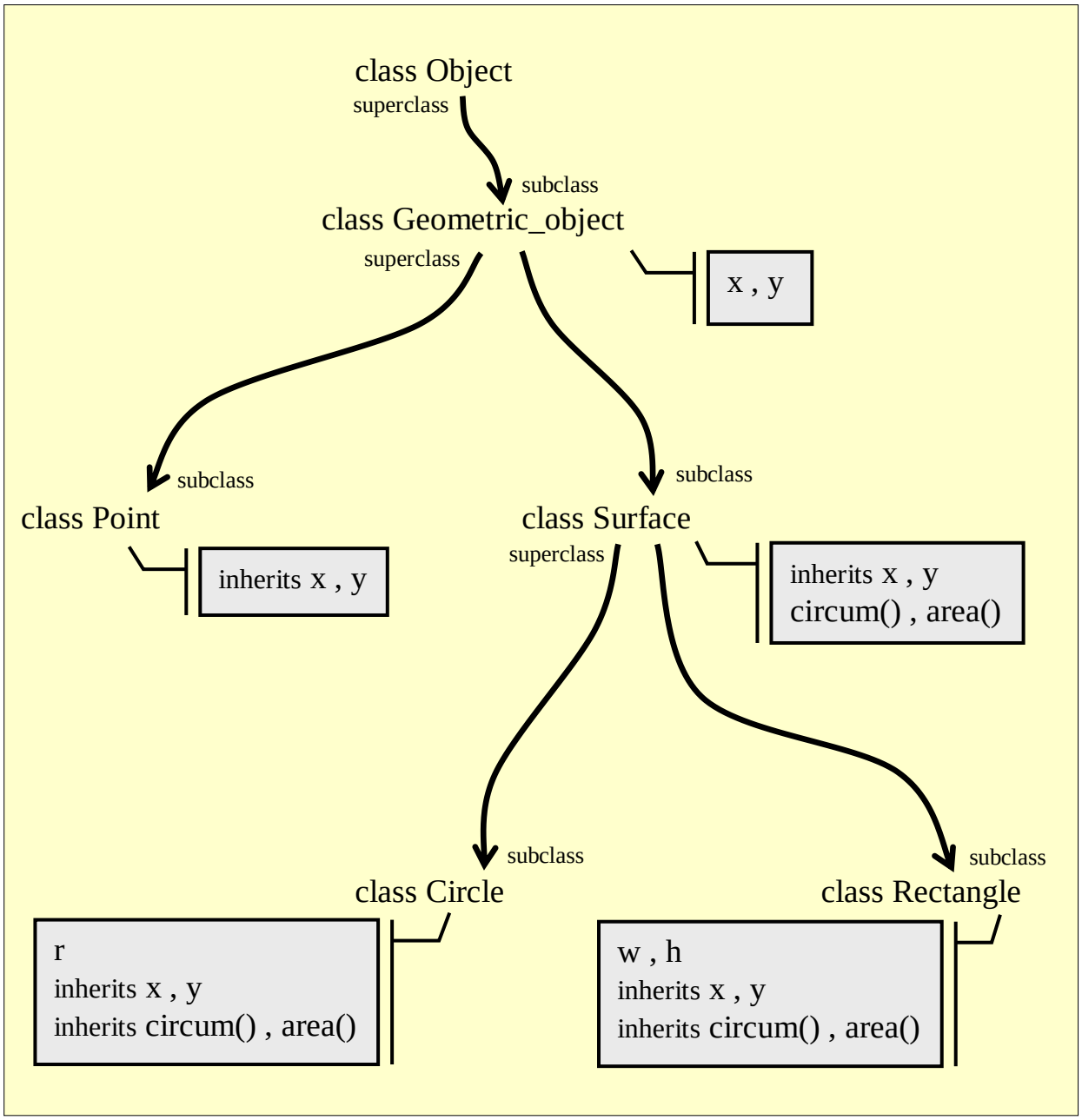
- a subclass inherits the variables and methods of its superclass, as well as those of the superclass of its superclass…
- in addition to these inherited variables and methods, the subclass defines some variables and methods which are particular to itself

hierarchy of classes:

the relation superclass – subclass creates an arborescence
along which classes inherit from their superclasses

- in Java, all classes are more or less direct subclasses of the root class **Object**

example:



polymorphism:

- method overloading (*static polymorphism* - at compile time)

  within the same class, several methods are defined with the same name but with different arguments

  → what differentiates these methods is not their names but their lists of arguments...

- method overriding (*dynamic polymorphism* - at run time)

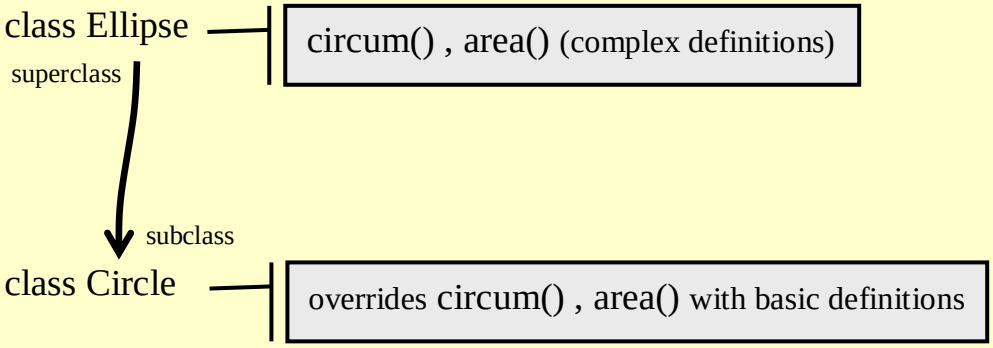
  the same method is defined with different versions,
  depending on the (sub)class of the object over which it is applied

method overriding:

a subclass may override -redefine- a method
of its (direct or indirect) superclass

example:

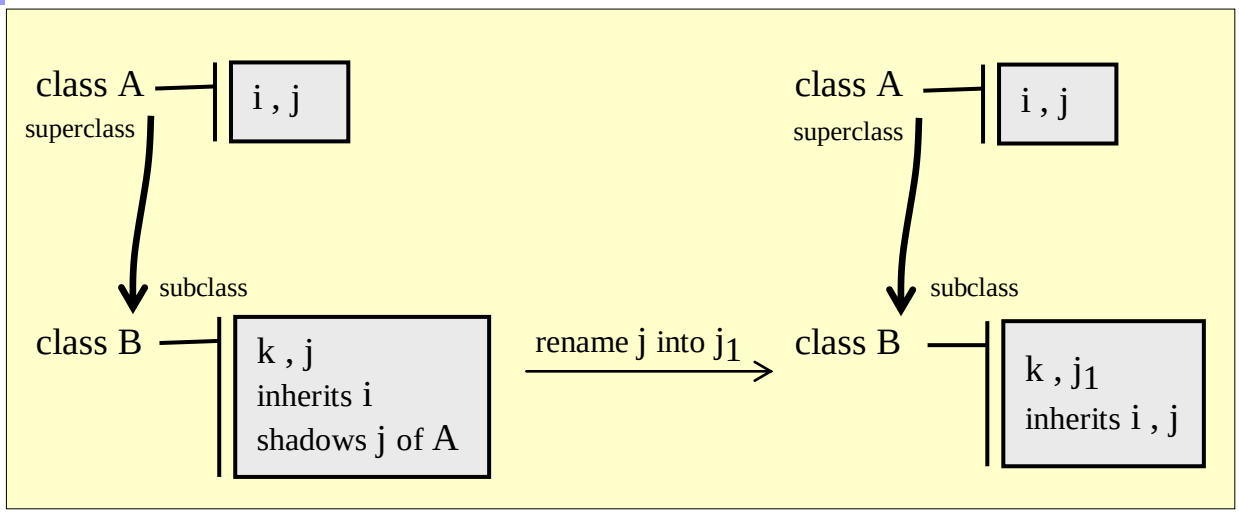
a circle is a particular case of an ellipse,
but its circumference and area can be computed easily with basic formulas



BUT do no let the subclass redefine a variable of its superclass
("shadowed", "hidden" variables) = possible but bad programming!

→ rename differently the variable of the subclass

example:



Using objects

3 steps:
- declaration of the variable containing the object reference
- creation of the object = allocation of a place in memory
- initialization of the object

declaration:

```
class_name  object_name;
```

creation + initialization: **new** followed by constructor method
= class_name + () or (arguments)

```
object_name = new class_name();

or   object_name = new class_name(arguments);
```

declaration and creation + initialisation:

```
class_name  object_name = new class_name();

or   class_name  object_name = new class_name(arguments);
```

each class can have more than one constructor method:
- default constructor with no arguments (performs no initialization)
- eventually other constructors with arguments (perform initialization)
- BUT all the constructors of a given class have the same name which is the name of the class (only the arguments differ) = overloaded methods

example:

```
Circle c1;    c1 = new Circle();      // creates a circle with no initialization
Circle c2 = new Circle(0.5 , 1.8 , 10.4);     // x = 0.5 ; y = 1.8 ; r = 10.4
Circle c3 = new Circle(c2);      // c3 is initialized as a copy of c2
```

access to the instance variables and methods of an object:

- outside the definition of the enclosing class:

```
object_name . variable_name
object_name . method_name(arguments)
```

- inside the definition of the enclosing class:

```
variable_name                 or   this . variable_name
method_name(arguments)        or   this . method_name(arguments)
```

this is the reference of the current object = implicit or explicitly written

in C language: **object_reference –> variable_name**

access to the class (or static) variables and methods of a class:

- outside the definition of the enclosing class:

```
class_name . variable_name
class_name . method_name(arguments)
```

- inside the definition of the enclosing class:

```
variable_name                 or   class_name . variable_name
method_name(arguments)        or   class_name . method_name(argu)
```

example:

```
Circle c = new Circle(0.5 , 0.0 , 1.0);
double a = c.area();   // the reference to c is passed as an extra argument to area()
System.out.println("radius = " + c.r + "  –>  area = " + a);
c.x = 2.5;     c.y = 3.6;     c.r = 10.0;
System.out.println("radius = " + c.r + "–> area = " + c.area());
System.out.println("pi=" + Math.PI + " ; sqrt(2)=" + Math.sqrt(2.0));
```

→ radius = 1.0 –> area = 3.141592653
radius = 10.0 –> area = 314.1592653
pi=3.141592653 ; sqrt(2)=1.41421356

finalizer:

method specific to a class
(most classes do not have a finalizer)

called after an object of this class is dereferenced
and before it is erased by the garbage collector

→ used to save data on the object or to close a file used by this object

BUT the program may end before the garbage collector erases the object

→ the finalizer will never be executed

→ before the end of the program, call **System.gc();**

= forces immediate garbage collection

copy of objects:

object1 = object2; copies in fact the reference of a single object!

→ the same object is now referenced
by the two variables **object1** and **object2**

for actually copying an object, we must create in memory
a new object that is identical to the copied object:

object_copy = object_copied.clone();

BUT:
- **clone()** is not defined for all predefined classes
- a priori, **clone()** is not applied recursively
  on other objects referenced by the copied object
  → we can redefine **clone()** ourselves so that it copies recursively
- the object returned by **clone()** is of type **Object**
  → maybe need of a cast

examples:

```
Circle a , b , c , d;
a = new Circle(0.5 , 0.5 , 10.0);
b = a;     // b and a refer to the same circle object

c = new Circle(0.5 , 0.5 , 10.0);   // c and d are both new circle objects
d = (Circle)a.clone();              // but they contain exactly the same data
                                    // as the circle object referenced by a and b
```

```
Hashtable ht1 = new Hashtable();

fill in ht1 . . . .

Hashtable ht2 = (Hashtable)ht1.clone();
```

tests on objects:

object1 == object2 compares in fact the references of the objects!

→ **true** if and only if the two variables **object1** and **object2** reference the same object in memory

for actually comparing two objects
, we must compare all the data contained by each of these objects

object1 . equals(object2) → **true** if and only if all the data of **object1** and **object2** all identical

BUT:
- **equals()** is not defined for all predefined classes
- a priori, **equals()** is not applied recursively on other objects referenced by **object1** and **object2**
  → we can define **equals()** so that it compares recursively

example:

```
class A
{  int x;    A(int x)  { this.x = x;  }
   public Object clone()  {  return new A(this.x);  }
   boolean equals(A a)  {  return this.x == a.x;  }  }

public class Test
{
   public static void main(String[] args)
   {  A a1 = new A(10) , a2 = new A(9) , a3 = (A)a1.clone();
      System.out.println( (a1==a1) + " " + (a1==a2) + " " + (a1==a3)
                                 + " " + a1.equals(a2) + " " + a1.equals(a3));
   }
}
```

→ true false false false true

object_name instanceof class_name → **true** if and only if the object is an instance of the class or of a subclass of the class

Definition of classes

the definition of a class encloses the definitions of its members:

- variables, methods
- nested top-level classes, nested top-level interfaces, member classes

```
<modifiers>  class class_name
{
   definition of member1
   ……
   definition of membern
}
```

ALL variables and methods must be defined within some classes!
= principle of object oriented programming

accessibility modifiers:

Accessibility of a class	
none	accessible only within its package
public	accessible anywhere

Accessibility of a class member	
none	accessible only within its package
public	accessible anywhere (if the class is also **public**)
private	accessible only within its class (or a class within its class) (and not within a subclass)
protected	accessible only within its package and within subclasses of its class (that may be outside its package)

definition of instance variables:

<access_modifier> <final> type variable_name;

if present, the **final** modifier indicates that the variable can be assigned a value only once (it is forbidden to modify its value afterwards)

definition of instance methods:

<access_modifier> <final> return_type method_name(arguments) { body }

if present, the **final** modifier indicates that the method cannot be overriden

this (object reference) is systematically passed
as an extra, implicit, argument to the instance method

→ if risk of confusion with a local variable or method argument
, designate the instance variable with **this . variable_name**

examples:

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

   public double circum()  {  return 2 * Math.PI * r;  }

   public double     area()  {  return Math.PI * r * r;  }

   public boolean bigger(Circle c)  {  return this.r > c.r  }

   public boolean contains_point(double x , double y)
   {
      double d = Math.sqrt(   (this.x – x) * (this.x – x)
                            + (this.y – y) * (this.y – y) );
      return d <= r;
   }
}
```

→ `Circle c , c1;  double x , y;`
`c.circum()`
`c.area()`
`c.bigger(c1)`
`c.contains_point(x , y)`

with encapsulation (**x** , **y** , **r** accessible directly only within **Circle**):

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

   public void set_x(double x)  {  this.x = x;  }
   public void set_y(double y)  {  this.y = y;  }
   public void set_r(double r)  {  this.r = r;  }

   public double read_x()  {  return x;  }
   public double read_y()  {  return y;  }
   public double read_r()  {  return r;  }

   public double circum()  {  return 2 * Math.PI * r;  }

   public double    area()  {  return Math.PI * r * r;  }

   public boolean bigger(Circle c)  {  return this.r > c.r  }

   public boolean contains_point(double x , double y)
   {
      double d = Math.sqrt(   (this.x – x) * (this.x – x)
                            + (this.y – y) * (this.y – y) );
      return d <= r;
   }
}
```

```
public class Circle
{
   public Position p;
   public double r;

   . . . . . .

   public boolean contains_point(Position p)
   {
      double d = Math.sqrt(   (this.p.x – p.x) * (this.p.x – p.x)
                            + (this.p.y – p.y) * (this.p.y – p.y) );
      return d <= r;
   }
}


class Position
{
   double x , y;
}
```

definition of static (class) variables:

<access_modifier> static <final> type variable_name;

if present, the **final** modifier indicates that the variable can be assigned a value only once (it is forbidden to modify its value)

static variable ~ global variable in C

static final variable ~ global constant in C (or preprocessor constant)

we ALWAYS write the name of a constant entirely in capital letters

definition of static (class) methods:

<access_modif> static <final> return_type method_name(arguments) { body }

static method ~ C function

if present, **final** means that the method cannot be shadowed in a subclass

NO extra, implicit, argument **this** passed to a class method
(since it is not related to one object in particular)!

example:

```
public class Circle
{
   public Position p;
       public double r;
   private static int number_circles = 0;
   private static final int MAX_NUMBER_CIRCLES = 10;

    . . . . . .

   public boolean bigger(Circle c) { return this.r > c.r }

   public static boolean bigger(Circle c1 , Circle c2)  {  return c1.r>c2.r  }

   public boolean contains_point(Position p)
   {   double d = Math.sqrt(   (this.p.x – p.x) * (this.p.x – p.x)
                             + (this.p.y – p.y) * (this.p.y – p.y) );
      return d <= r;  }

   public static boolean contains_point(Circle c , Position p)
   {   double d = Math.sqrt(   (c.p.x – p.x) * (c.p.x – p.x)
                             + (c.p.y – p.y) * (c.p.y – p.y) );
      return d <= c.r;  }
}


class Position
{
   double x , y;
}
```

→ **Circle c , c1; double x , y;**
c.bigger(c1) **Circle.bigger(c , c1)**
c.contains_point(p) **Circle.contains_point(c , p)**

how to increment Circle.number_circles at each creation of a circle?
→ do it in the constructor(s)

method overloading:

define two or more DIFFERENT methods which have the same name, but which differ by the number and / or type of their arguments

different return types are allowed only if the arguments are different!

overloading is possible only between methods of the same class

advice: only for methods that apply similar treatment
on input data presented slightly differently

example:

```
class Numbers
{
   static double maximum(double a , double b)
   {
      return (a>=b ? a : b);
   }

   static double maximum(double a , double b , double c)
   {
      double t = (a>=b ? a : b);
      return (t>=c ? t : c);
   }

   static int maximum(int a , int b)
   {
      return (a>=b ? a : b);
   }

   static int maximum(int a , int b , int c)
   {
      int t = (a>=b ? a : b);
      return (t>=c ? t : c);
   }
}
```

definition of several possible constructors:

the name of a constructor MUST be the class name
→ method overloading to define several constructors

a constructor implicitly returns a new instance of the class
→ no need to specify the return type, no **return** statement

```
<access_modifier>  class_name(arguments)
{  . . . . . .  (no return!)  }
```

the default constructor contains no arguments
= it performs no initialization on the new class instance
= provided by the compiler by default

if several constructors for the same class, it is possible to invoke
- AT THE FIRST LINE of the definition of a constructor -
another constructor with **this** :

```
< access_modifier>  class_name(arguments)
{
   this(other arguments);
   . . . . . .
}
```

example:

```
public class Circle
{
   private Position p;
       private double r;
   private static int number_circles = 0;
   private static final int MAX_NUMBER_ CIRCLES = 10;

   // replaces the default constructor:
   public Circle()  {  increment_and_check_number_circles();  }

   public Circle(double x , double y , double r)
   {   p = new Position();
      p.x = x;    p.y = y;    this.r = r;
      increment_and_check_number_circles();  }

   public Circle(Position p , double r) { this(p.x , p.y , r); }

   public Circle(Circle c)
   {  this(c.p.x , c.p.y , c.r);    // or  this(c.p , c.r);
      System.out.println("calling Circle(c)");  }

   // cannot be shadowed:
   private static final void increment_and_check_number_circles()
   {
      number_circles++;    // or  Circle.number_circles

      if (number_circles>MAX_NUMBER_ CIRCLES)
      {   System.out.println("too many circles!");
         System.exit(0);  }
   }

       . . . . . .
}

class Position
{
   double x , y;
}
```

instance initializers:

carries out the more or less complex initialization
of some instance variables each time an instance of the class is created
(these initializations could be placed in the constructor(s) as well)

like a constructor with no argument and no name:

```
{
   initialize some instance variables
}
```

one or several instance initializers within a class

executed automatically after the constructor of the superclass
and before the constructor of the current class

if several within the class, they are executed in their order of definition

static initializers:

carries out the more or less complex initialization
of some class (or static) variables ONCE, when the class is first loaded

```
static
{
   initialize some class (or static) variables
}
```

one or several static initializers within a class

if several within the class, they are executed in their order of definition

example:

```
public class Circle
{
   private Position p;
   private double r;

   // nb_points, da, cosine, sine, point can be assigned a value only once
   private static final int nb_points;
   private static final double da;
   static  {  nb_points = 100;    da = (2.0*Math.PI) / nb_points;  }

   private static final double[]  cosine = new double[nb_points]
                                  , sine    = new double[nb_points];

   static       // executed after the static initializer for nb_points and da
   {
      int i;    double a;
      for ( i=0 , a=0.0 ; i<nb_points ; i++ , a+=da)
      {  cosine[i] = Math.cos(a);    sine[i] = Math.sin(a);  }
   }

   private final Position[] point = new Position[nb_points];

   Circle(double x , double y , double r)
   {
      this.r = r;    this.p = new Position(x , y);

      for (int i=0; i<nb_points ; i++)
      {  point[i] = new Position();
         point[i].x = x + r * cosine[i];    point[i].y = y + r * sine[i];  }
   }

   . . . . . .

   public void draw_circle()
   {
      for ( int i=0 ; i<nb_points ; i++)
         draw_segment(  point[i].x     , point[i].y
                      , point[i+1].x , point[i+1].y );
   }
}
```

finalizer:

a class definition contains either no finalizer or only one finalizer

the name of a finalizer is always **finalize**

it takes no arguments and returns **void**

it must be declared **public** (or **protected**)

```
public void finalize ()
{
   save data related to the object
   close file used by the object

   possibility to use this to refer to the object being finalized
}
```

example:

```
class A
{
   int x;

   A(int x)  {  this.x = x;  }

   public void finalize()
  {  System.out.println("finished with x = " + this.x);  }
}

   . . . . . .

   A a =  new A(12);
   a.x += 3;
   a = null;
   System.gc();                  →   finished with x = 15
```

without **System.gc();** , the finalizer is not called in practice

example:

```
//      *******  linked-list implementation of a list of points  *******

class Point
{
   double x , y;

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

   void print()  {  System.out.println("x = " + x + " ; y = " + y);  }

   boolean equals(Point p)  {  return p.x == this.x  &&  p.y == this.y;  }
}


class Node
{
   Point p;
   Node next;

   Node(Point p , Node next)  {  this.p = p;    this.next = next;  }
}


class List
{
   Node head;

   List()  {  head = null;  }

   void add(Point p)
   {   head = new Node(p , head);  }

   void remove(Point p)
   {  for (Node n = head , prev_n = null  ;  n != null  ;  prev_n = n , n = n.next)
         if (n.p.equals(p))
            prev_n.next = n.next;  }

   void print_all()
   {  for (Node n = head  ;  n != null  ;  n = n.next)
         n.p.print();  }
}
```

example continued:

```
public class MyProg
{
   public static void main(String[] arg)
   {
      List l = new List();

      Point p0 = new Point(1.2 , 5.0);
      Point p1 = new Point(7.2 , -9.9);
      Point p2 = new Point(-5.7 , 2.3);

      l.add(p0);    l.add(p1);     l.add(p2);

      l.remove(p1);

      l.print_all();
   }
}
```

example:

```
//      *******  array-based implementation of a list of points  *******

class Point
{
   double x , y;

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

   void print()  {  System.out.println("x = " + x + " ; y = " + y);  }

   boolean equals(Point p)  {  return p.x == this.x  &&   p.y == this.y;  }
}


class List
{
   final static int N = 1000;
   Point[] array;
   List()  {  array = new Point[N];  }

   void add(Point p)
   {   for (int i = 0 ; i < N ; i++)
         if (array[i] == null)  {  array[i] = p;    break;  }  }

   void remove(Point p)
   {  for (int i = 0 ; i < N ; i++)
        if (array[i] != null  &&  array[i].equals(p))  array[i] = null;  }

   void print_all()
   {  for (int i = 0 ; i < N ; i++)
        if (array[i] != null)  array[i].print();  }
}
```

Subclasses and inheritance

definition of a subclass:

a subclass extends its ONLY superclass:

```
<modifiers> class subclass_name extends superclass_name
{
   additional variables and methods defining the class more precisely
}
```

a **final** class (indicated with modifier **final**) cannot be subclassed

within the subclass, we cannot access a **private** variable or **private** method of the superclass (although it is also by inheritance a member of the subclass)

example:

```
class A
{
   int i;    private int k;
   void set_k(int k)  {  this.k = k;  }
   double get_k()  {  return k;  }
}

class B extends A
{
   double x;
   void f()  {  System.out.println("x=" + x + "i=" + i + "k=" + get_k());  }
   // forbidden: void f()  {  System.out.println("k=" + k);  }
}
```

i and **get_k()** are inherited by **B**

k is also inherited by **B**, but it can be accessed within **B** only through a method of **A** (**get_k()** in this case)

an instance of a subclass is also an instance of the superclass
→ no need of a cast!

example:

```
class A
{ . . . . . }

class B extends A
{ . . . . . }

class C extends A
{ . . . . . }
```

→
```
A[] a = new A[4];
a[0] = new A();
a[1] = new B();
a[2] = new C();
a[3] = new B();
```

calling the superclass constructor:

before constructing what is specific to the subclass
, we must construct the general part related to the superclass

→ first of all, call the constructor of the superclass at the beginning of the constructor of the subclass:

```
<access_modifier> class_name(arguments)
{
   super(other arguments);
   . . . . . .
}
```

if this call is not specified then, BY DEFAULT, there is an IMPLICIT call to **super()** (with NO arguments) at the beginning of EVERY constructor

it is forbidden to use **super.super()**

example:

```
class A
{
   int k;

   A(int k)  {  this.k = k;  }

   . . . . . .
}

class B extends A
{
   double x;

   B(int k , double x)
   {  super(k);
      this.x = x;  }

   . . . . . .
}
```

overriding a method:

within a subclass, redefine an inherited method

a **static**, **private** or **final** method cannot be overriden
(a **static** method is shadowed instead...)

example:

```
class A
{
   int k = 15;
   void f()  {  System.out.println("k = " + k);  }
}

class B extends A
{
   int l = 21;
   void f()  {  System.out.println("k = " + k + " ; l = " + l);  }
}

class C extends A
{
   int m = 12;
}
```

→ A a = new A();
B b = new B();
C c = new C();

a.f(); → k = 15

b.f(); → k = 15 ; l = 21

c.f(); → k = 15

calling the overriden method:

super.method_name(arguments)

= only within the overriding version of the method!

example:

```
class A
{
   int k = 15;
   void f() { System.out.println("k = " + k); }
}

class B extends A
{                  // inherits f() from A
   int l = 21;     // calls to super.f() forbidden since f() is not overriden within B
}

class C extends B
{                  // inherits k , l from B ; overrides f() otherwise inherited from B
   void f()  {  super.f();    System.out.println("l = " + l);  }
}
```

→ A a = new A();
B b = new B();
C c = new C();

a.f(); → k = 15

b.f(); → k = 15

c.f(); → k = 15
l = 21

abstract methods and abstract classes:

if a class is too general, too vague, some of its methods cannot be defined precisely a priori

example:

class of all surfaces (circle , rectangles, ...)

→ we cannot define a priori the methods circum() , area() for any surface

→ abstract method: the body of the method is not defined (replaced by **;**):

<modifiers> abstract return_type method_name(arguments) ;

example:

public abstract double area() ;

a **static**, **private** or **final** method cannot be abstract

abstract class: contains at least one abstract method
(may also contain some normal methods)

= class too general to allow for complete definition of all its methods

```
<modifiers> abstract class class_name
{
   ……
   abstract method(s)
   ……
}
```

an abstract class CANNOT be instantiated (too general to create an actual object)
→ a subclass of an abstract class can be instantiated
ONLY if it overrides with complete definitions
ALL the abstract methods of the abstract class

example:

```
class Geometric_object
{
   Position p;
   Geometric_object(double x , double y)  {  p = new Position();
                                             p.x = x;    p.y = y;  }

   void move(double dx , double dy) {  p.x += dx;    p.y += dy  }
}

class Point extends Geometric_object
{
   Point(double x , double y) {  super(x , y);  }
}

abstract class Surface extends Geometric_object
{
   abstract double circum() ;
   abstract double     area() ;
}

class Circle extends Surface
{
   double r;
   Circle(double x , double y , double r)  {  super(x , y);    this.r = r;  }

   // implementation of the methods of the abstract class Surface:
   double circum()  {  return 2 * Math.PI * r;  }
   double     area()  {  return Math.PI * r * r;  }
}

class Rectangle extends Surface
{
   double w , h;
   Rectangle(double x , double y , double w , double h)
   {  super(x , y);   this.w = w;   this.h = h;  }

   // implementation of the methods of the abstract class Surface:
   double circum()  {  return 2 * (w + h);  }
   double     area()  {  return w * h;  }
}

final class Position  {  double x , y;  }    // cannot be subclassed
```

example continued:

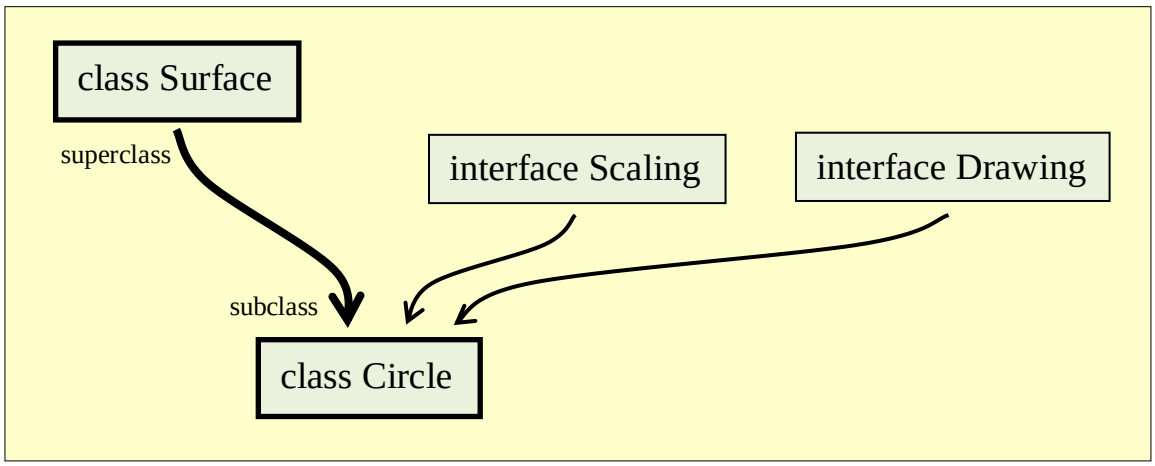
```
public static void main(String[] arg)
{
   Geometric_object[] go = new Geometric_object[5];

   go[0] = new Point(0.5 , 0.5);
   go[1] = new Circle(12.1 , 9.3 , 1.5);
   go[2] = new Rectangle(5.9 , 6.2 , 4.0 , 1.0);
   go[3] = new Point(1.2 , 2.5);
   go[4] = new Circle(-0.9 , -10.0 , 2.0);

   double total_area = 0.0;

   // sum the areas of only the go[i] which are instances of Surface
   // or, more precisely, of the subclasses Circle and Rectangle of Surface
   // → do not try to compute the area of an instance of Point!!!
   for (int i=0 ; i<go.length ; i++)
      if (go[i] instanceof Surface)
         total_area += ((Surface )go[i]).area();

   System.out.println("total area = " + total_area);
}
```

interfaces:

no multiple inheritance in Java:
a subclass CANNOT have more than one superclass

→ use instead a sort of degraded abstract class = interface

example:



interface = group of constants and ONLY abstract methods

no need to precise the modifier **abstract** for the interface
and its abstract methods:

```
<access_modifier>  interface  interface_name
{
   static final variables
   abstract methods (with no abstract modifier)
}
```

Accessibility of an interface
an interface is ***implicitely*** public and its elements are ***implicitely*** public BUT: the implementation of a method of an interface *must* be *explicitely* declared as public

in general, use an interface to describe a certain behavior of an object which further characterizes this object

example:

```
interface Moving
{
   void move(double dx , double dy) ;
}

interface Scaling
{
   void scale(double s) ;
}

interface Drawing
{
   void draw() ;
}
```

an interface can extend one or several other interfaces:

```
<access_modifier> interface interface_name extends interf_name1
                                                  , interf_name2
{
   // inherits all the constants and abstract methods of interf_name1 and interf_name2

   // proper to interface_name:
   static final variables
   abstract methods (with no abstract modifier)
}
```

example:

```
interface Changing extends Moving , Scaling
{
   // inherits move() and scale() from interfaces Moving and Scaling
}
```

a class may implement one or several interfaces

→ this class (or its subclasses) must provide complete definitions for all the abstract methods of the interface(s):

```
<modifiers> class class_name implements interface_name
{
   ……
   methods of the interface defined completely (with keyword public!)
}
```

```
<modifiers> class class_name implements interf_name1 , interf_name2
{ …… }
```

a class can extend another class and implement one or several interfaces:

```
<modifiers> class subclass_name extends superclass_name
                                implements interface_name
{ …… }
```

```
<modifiers> class subclass_name extends superclass_name
                                implements interf_name1 , interf_name2
{ …… }
```

example:

```
abstract class Geometric_object implements Moving , Drawing
{
   Position p;
   Geometric_object(double x , double y)  {  p = new Position();
                                             p.x = x;    p.y = y;  }

   // implementation of interface Moving:
   public void move(double dx , double dy)  {  p.x += dx;    p.y += dy  }
}

class Point extends Geometric_object
{
   Point(double x , double y)  {  super(x , y);  }

   // implementation of interface Drawing:
   public void draw()  {  draw_point(p.x , p.y , color);  }
}

abstract class Surface extends Geometric_object
                       implements Scaling
{
   abstract double circum() ;
   abstract double     area() ;  }

class Circle extends Surface
{
   double r;
   Circle(double x , double y , double r)  {  super(x , y);    this.r = r;  }

   // implementation of the methods of the abstract class Surface:
   double circum()  {  return 2 * Math.PI * r;  }
   double     area()  {  return Math.PI * r * r;  }

   // implementation of interface Scaling (inherited from class Surface):
   public void scale(double s)  {  r *= s;  }

   // implementation of interface Drawing (inherited from class Surface):
   public void draw()  {  draw_circle(p.x , p.y , r , color);  }
}

final class Position  {  double x , y;  }    // cannot be subclassed
```

example continued:

```
public static void main(String[] arg)
{
   Geometric_object[] go = new Geometric_object[5];

   go[0] = new Point(0.5 , 0.5);
   go[1] = new Point(12.1 , 9.3);
   go[2] = new Rectangle(5.9 , 6.2 , 4.0 , 1.0);
   go[3] = new Point(1.2 , 2.5);
   go[4] = new Rectangle(-0.9 , -10.0 , 2.0 , 4.1);

   // move, scale and draw the go[i]
   // in fact, scale only the go[i] which are instances of Surface
   // or, more precisely, of the subclasses Circle and Rectangle of Surface
   // → do not try to scale an instance of Point!!!
   for (int i=0 ; i<go.length ; i++)
   {
      go[i].move(1.2 , 10.1);

      if (go[i] instanceof Surface)  ( (Surface ) go[i] ).scale(0.5);

      go[i].draw();
   }
}
```

Nested top-level classes and interfaces; inner classes

classes and interfaces defined within a class or interface

nested top-level classes and nested top-level interfaces:

top-level, usual, classes and interfaces that are just defined
within another top-level class or interface
→ their only specificity is in their names which reflect this nesting:

enclosing_class . class

enclosing_class . interface or **enclosing_interface . interface**

nested top-level classes are defined with modifier **static**
but no need of **static** for nested top-level interfaces:

```
<access_modifier>  class  class_name
{
   static  <access_modifier>  class  ntl_class_name { . . . . . }

   <access_modifier>  interface  ntl_interface_name { . . . . . }
   . . . . . .
}
```

```
<access_modifier>  interface  interface_name
{
   <access_modifier>  interface  ntl_interface_name { . . . . . }
   . . . . . .
}
```

a nested top-level class or interface may use only the static members
of its enclosing class or interface (and NOT its instance variables and methods)

in general, use nested top-level classes and interfaces
as a way of grouping related classes and interfaces
(equivalent of a “micro-package”)

example:

```
class A
{
   static int k;

   static class B
   {
      . . .  k  . . .
   }

   interface I1
   {
      . . . . . .
   }

   interface I2
   {
      interface I3  {  . . . . . .  }
      . . . . . .
   }
}
```

→ A
- **A.B**
- **A.I1**
- **A.I2**
  - **A.I2.I3**

member classes:

inner class, defined within an enclosing class

every instance of a member class is related
to an instance of the enclosing class
and can use ALL its members (static as well as instance, or even private)

a member class is defined with no **static** modifier:

```
<access_modifier>  class  class_name
{
   <access_modifier>  class  member_class_name  {  . . . . . }
   . . . . . .
}
```

cannot have the same name as any of its enclosing class(es)

cannot have static members (because associated to an instance of class)

INSIDE the definition of the member class:

- instance of the member class: **this**
- instance of the enclosing class: **enclosing_class . this**

the creation of an instance of the member class is done relatively
to the enclosing object:

- inside the definition of the enclosing class: **new** or **this . new**
- outside the definition of the enclosing class: **enclosing_object . new**

example:

```
class A
{
   class B
   {
      int k;
      B(int n)  {  this.k = A.this.k + n;  }
   }

   B b1 , b2;

   private int k;

   A(int k)  {  this.k = k;  }

   B f(int m)  {  return this.new B(m);  }    // or  return new B(m);
}

public class Test
{
  public static void main(String[] arg)
  {
     A a = new A(10);
     a.b1 = a.f(1);
     a.b2 = a.new B(2);

     System.out.println("a.b1.k = " + a.b1.k + " ; a.b2.k = " + a.b2.k);
  }
}
```

→ a.b1.k = 11 ; a.b2.k = 12

local classes:

inner class defined and visible only locally,
within a block of code of an instance method of the enclosing class:

```
<modifiers>  class  class_name
{

  <modifiers>  return_type  method_name(arguments)
  {
    class  class_name1 { . . . . . }
    class_name1 object_name1 = new class_name1(arg);

    . . . . . .

    {
      class  class_name2 { . . . . . }
      class_name1 object_name2 = new class_name2(arg);

      . . . . . .
    }
  }

  . . . . . .

}
```

can use all the members of its enclosing class (static, instance, private)

can also read all the **final** local variables and **final** arguments
of the enclosing method (and not the non **final** ones!)


it is forbidden to modify the values of **final** local variables and arguments


cannot have the same name as any of its enclosing class(es)


cannot have static members (because associated to an instance of class)


NO modifier public, protected, private

INSIDE the local class:

- instance of the local class: **this**
- instance of the enclosing class: **enclosing_class.this**

DO NOT use **enclosing_object . new** or **this . new** since the creation MUST ANYWAY be within the block of code

example:

```
class A
{
   private int k;

   void f(int x , final int y)
   {
      final int m = 1;
      class B { int k;  B(int i) { this.k = A.this.k + i + m;} }

      k = x;
      for (int i=x ; i<y ; i++)
      {
         B b = new B(i);

         class C { int p = y; }
         C c = new C();

         System.out.println("b.k = " + b.k + " ; c.p = " + c.p);
      }
   }
}
```

→ **A a = new A();**
a.f(3 , 6);

→ b.k = 7 ; c.p = 6
b.k = 8 ; c.p = 6
b.k = 9 ; c.p = 6

anonymous classes:

inner class which is an unnamed local class

in fact, we simultaneously define a local class (without naming it)
and create an instance of this class

- *first case:* the anonymous class extends a superclass:

new class_name(arguments) { . . . . . }

→the arguments are passed to the constructor of the superclass class_name

in general, no arguments: **new class_name() { . . . . . }**

- *second case:* the anonymous class implements an interface:

new interface_name() { . . . . . }

→ the anonymous class is a subclass of the **Object** root class
which implements the interface interface_name

we cannot define a constructor for the anonymous class,
but it is possible to use an instance initializer inside the anonymous class definition

an anonymous class cannot have static members
(because it is associated to an instance of a class)

NO modifier public, protected, private

in general, an anonymous class encapsulates one or several methods
= equivalent of pointers of functions (allowed in C, but forbidden in Java)

example:

```
abstract class B {  int m;
                    B(int m) { this.m = m; }
                    abstract double f(double x) ;  }

abstract class C {  int q;
                    abstract int g(int k) ;  }

interface I {  double f(double x) ;
               double g(double x) ;  }

class A
{
   double z;

   A()
   {
      z = func(  new B(123) { double f(double x) { return x + m; } }

                , new C() { { q = 2; }
                            int g(int k) { return q + k; } }

                , new C() { { q = 3; }
                            int g(int k) { return q * k; } }

                , new I() { public double f(double x)  { return x + 1; }
                            public double g(double x) { return x - 1; } }

                , new I() { public double f(double x)  { return x * x; }
                            public double g(double x) { return 1.0 / x; } } );
   }

   static double func(B b , C c1 , C c2 , I i1 , I i2)
   {
      return    b.f(5.0)  +  c1.g(10)  +  c2.g(20)

              +  i1.f(30.0)  +  i1.g(40.0)  +  i2.f(50.0)  +  i2.g(60.0);
   }
}
```