UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria Informàtica etsinf

Unit 4. Datatype classes

Introduction to Computer Science and Computer Programming
Introducción a la Informática y la Programación (IIP)

Year 2017/2018

Departamento de Sistemas Informáticos y Computación

Contents

Contents

Basic class structure: attributes and methods

- ***Object***: group or collection of ***data*** and ***operations*** with a given structure
- ***Class***: describes the behaviour of each object (instance) of the class
- Three different types of classes:
  - ***Datatype classes***: objects
  - ***Program classes***: executables
  - ***Utility classes***: operations

Basic class structure: attributes and methods

General class structure

```
[visibility] class ClassName  [ extends OtherClass ] {

// Attributes definition
  [[visibility] [nature] typeVar varName1;
   [visibility] [nature] typeVar varName2;
       ...          ...   ...
   [visibility] [nature] typeVar varNameN; ]

  // Methods definition
  [[visibility] [nature] typeMet methodName1 ([paramList]) { body }
   [visibility] [nature] typeMet methodName2 ([paramList]) { body }
       ...          ...       ...
   [visibility] [nature] typeMet methodNameM ([paramList]) { body } ]
}
```

Basic class structure: attributes and methods

General class structure

Particularities of this structure with respect to the type of class:

	Datatype	Program	Utility
Object attributes and methods	Yes	No	No
Class attributes and methods	Yes	Yes	Yes
`main` method	No	Yes	No

Basic class structure: attributes and methods

Attributes

- ***Attributes***: represent internal information of the object (instance variables) or class (class variables)

- Attributes are declared as a determined datatype, with the usual effect on values and operations

```
// Attributes definition
[[modifiers] type varName1;
 [visibility] [nature] typeVar varName2;
...  ...  ...
 [visibility] [nature] typeVar varNameN;
]
```

```
public class RealPoint
{
   double x, y;
   char shape;
   ...
}
```

Basic class structure: attributes and methods

Methods

- ***Methods***: define the operations of the objects or the class
  - ***Header*** (profile): visibility (optional), class modifier (optional), return datatype, name, list of parameters (optional)
  - ***Body***: sequence of instructions to be executed when the method is called
- Return datatype: `void` when the method does not produce a result
- `return` statement: gives the result of the method (when it has)

```
public class RealPoint {
  ...
  void setPoint(double nX, double nY) { x = nX; y = nY; }
  boolean origin() { return (x==0.0) && (y==0.0); }
}
```

Basic class structure: attributes and methods

Visibility modifiers: `private` and `public`

- Attributes and methods can have ***visibility modifiers***: ***public*** and ***private***
- ***Visibility modifiers*** define which objects can use the attributes and methods
- ***private*** visibility:
  - Attribute/method cannot be accessed from a different class
  - Access to `private` elements from a different class causes a compilation error
- ***public*** visibility: attribute/method can be accessed from any class

Special modifiers:

- ***friendly***: when no modifier is specified, access in the same package
- ***protected***: access from derived classes

Basic class structure: attributes and methods

Class/instance modifier: `static`

- Instance attribute/methods: default syntax (no extra modifiers)
- Class attribute/methods:
  - ***static*** modifier
  - Class attribute/methods are accessed by using class name (not object name)
  - Instance attributes in a class cannot be used in class methods of the class

Thus, the `main` method:

```
public static void main(String [] args)
```

- Is ***public***
- Is a ***class method*** (`static`)
- Does not return any value (`void`)
- Has as parameter `args`, of datatype `String []`

Contents

Object and class methods

- In a datatype class:
  - Object attributes: internal structure of the objects
  - Object methods: operations that can be performed on the objects
  - Class attributes: data of the class (usually constants)
  - Class methods: operations that can be performed on the whole class
- In an utility class:
  - Class attributes: constants offered by the class
  - Class methods: operations offered by class
- In a program class:
  - Class attributes: global data for all the methods
  - Class methods: subprograms and `main`

Object and class methods

Example of datatype class

```
public class Circle {
     private double radius; private String color; private int centerX, centerY;

     public Circle() { radius = 50; color = "black"; centerX = 100; centerY = 100;}

     public double getRadius() { return radius; }

     public void setRadius(double newRadius) { radius = newRadius; }

     public void decrease() { radius = radius / 1.3; }

     public double area() { return 3.14 * radius * radius; }

     public String toString() {
       String res = "Circle with radius "+radius+" and color "+color;
       res += " and center ("+centerX+","+centerY+")";
       return res;
     }
     // And more methods...
}
```

Object and class methods

Example of utility class

```
public class ThreeNumbers {

     public static double max(double x1, double x2, double x3) {
       if ((x1>=x2) && (x1>=x3)) return x1;
       if ((x2>=x1) && (x2>=x3)) return x2;
       if ((x3>=x1) && (x3>=x2)) return x3;
     }

     public static double min(double x1, double x2, double x3) {
       if ((x1<=x2) && (x1<=x3)) return x1;
       if ((x2<=x1) && (x2<=x3)) return x2;
       if ((x3<=x1) && (x3<=x2)) return x3;
     }

     public static double sum(double x1, double x2, double x3) {
       return x1+x2+x3;
     }
}
```

Object and class methods

Example of program class

```
public class ProgramTriangle {
  public static void main(String args[]) {
    Point p1 = new Point(); p1.x = 2.5; p1.y = 3;
    Point p2 = new Point(); p2.x = 2.5; p2.y = -1.2;
    Point p3 = new Point(); p3.x = -1.5; p3.y = 1.4;
    double side12, side23, side13, perimeter;

    System.out.println("Triangle with vertexes:\n(" + p1.x + "," + p1.y + ")");
    System.out.println("(" + p2.x + "," + p2.y + ")\n(" + p3.x + "," + p3.y + ")");
    side12 = dist(p1,p2); side23 = dist(p2,p3); side13 = dist(p1,p3);
    perimeter=side12+side23+side13; System.out.println("Perimeter: "+perimeter);
  }

  /** Calculate distance between two points */
  public static double dist(Point p, Point q) {
    double dx=p.x-q.x; double dy=p.y-q.y;
    return Math.sqrt(dx*dx + dy*dy);
  }
}
```

Object and class methods

Methods in datatype classes

According to their function with respect to the object, methods can be classified as:

- ***Constructors***: create the object
- ***Consultors***: retrieve (without altering) the state of the object
- ***Modifiers***: modify the state of the object

When defining a datatype class, the steps are:

1. Declare the class (`public class Name { ... }`)
2. Declare the attributes
3. Implement the constructor methods
4. Implement the ***get*** consultor methods
5. Implement the ***set*** modifier methods
6. Implement the rest of methods

Object and class methods

Class attributes

- Datatype classes usually have only ***object attributes***
- ***Class attributes***: usually constant (`final`) and public attributes
- They are employed in all objects defined for the class and by external classes

Example: `RealPoint` class

```
public class RealPoint {
     private double x, y;
     private char shape;
     public static final char CIRCLE='O', STAR='*', CROSS='x';

     ...
}
```

Object and class methods

Constructor methods

Sequence of events when an object is created (`new`):

1. Attributes get the default value:
   - Numerical (`byte`, `short`, `int`, `long`, `float`, `double`): zero
   - `char`: char of code zero
   - `boolean`: `false`
   - References: `null`
2. Attributes get initialisation value (when they have)
3. Corresponding ***constructor*** is called

A ***constructor*** is a method called when the object is created by `new` and it is responsible for giving the object a coherent initial state

Object and class methods

Constructor methods

Constructor syntax:

```
public ClassName ( [ parameter_list ] ) { ... }
```

A class can have different constructors with different parameters (***overload***)

Examples: `RealPoint` class

```
public class RealPoint {
     private double x, y;
     private char shape;
     public static final char CIRCLE='O', STAR='*', CROSS='x';

     public RealPoint() {
       x = y = 0.0; shape=CIRCLE;
     }

     public RealPoint(double nx, double ny) {
       x=nx; y=ny; shape=CIRCLE;
     }
     ...
}
```

Object and class methods

get consultor methods

In datatype classes ***attributes must be declared as private***

Access to attributes is via ***get consultors*** and ***set modifiers***

get methods are defined to each object attribute

Basic consultor syntax: for `private datatype attrName;`

```
public datatype getAttrName ( ) { return attrName; }
```

Examples: `RealPoint` class

```
public class RealPoint {
     private double x, y;
     private char shape;

     public double getX() { return x; }
     public double getY() { return y; }
     public char getShape() { return shape; }
     ...
}
```

Object and class methods

set modifier methods

set methods are defined to each object attribute

Basic modifier syntax: for `private datatype attrName;`

```
public void setAttrName ( datatype newValue ) {
    attrName = newValue;
}
```

For reference attributes, assignment must be attribute by attribute

Examples: `RealPoint` class

```
public class RealPoint {
     private double x, y;
     private char shape;

     public void setX(double nx) { x = nx; }
     public void setY(double ny) { y = ny; }
     public void setShape(char ns) { shape = ns; }
     ...
}
```

Object and class methods

A complete class: `RealPoint`

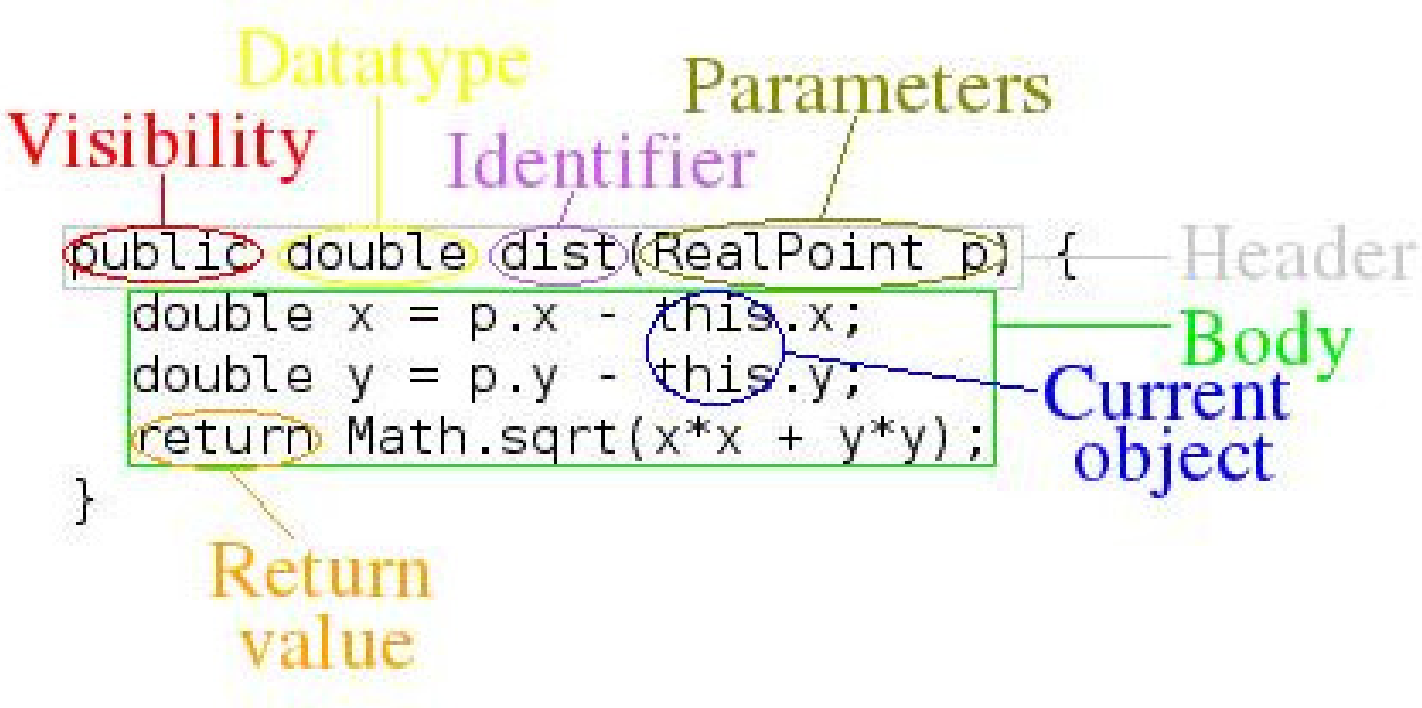
```
public class RealPoint {
     private double x, y;
     private char shape;
     public static final char CIRCLE='O', STAR='*', CROSS='x';

     public RealPoint() { x = y = 0.0; shape=CIRCLE; }
     public RealPoint(double nx, double ny) { x = nx; y = ny; shape=CIRCLE; }
     public RealPoint(double nx, double ny, char ns) {
       x = nx; y = ny; shape = ns;
     }

     public double getX() { return x; }
     public double getY() { return y; }
     public char getShape() { return shape; }

     public void setX(double nx) { x = nx; }
     public void setY(double ny) { y = ny; }
     public void setShape(char ns) { shape = ns; }
     ...
}
```

Contents

Dynamic methods features

Definition of methods:

```
[ modifiers ] ReturnType methodName ( [ ParameterList ] ) {
  // Instructions of the body of the method
}
```

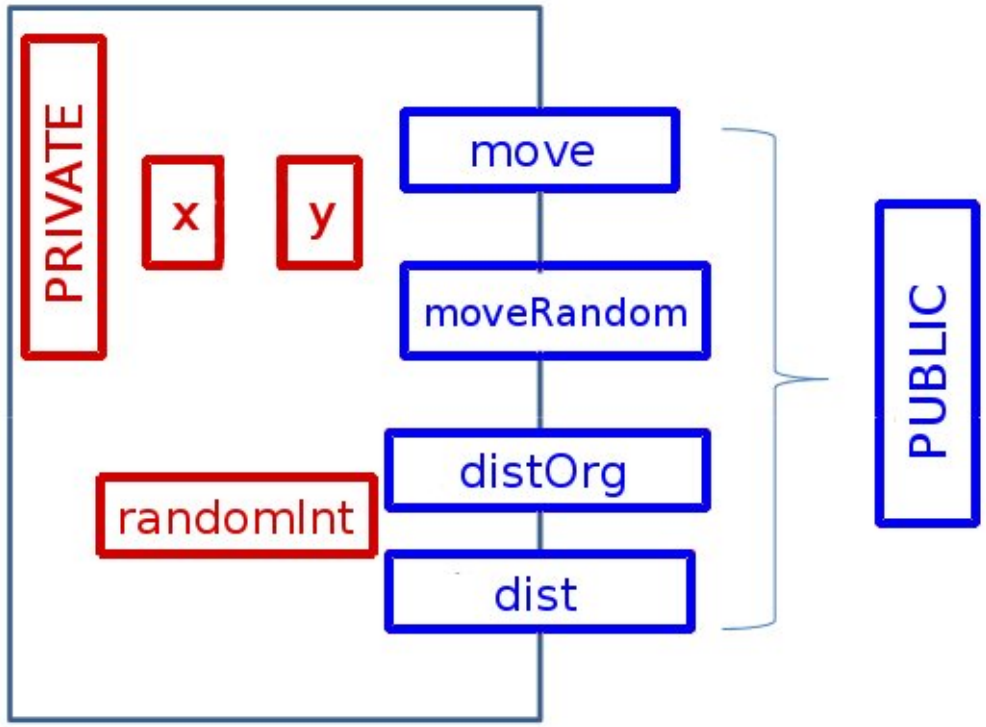


Dynamic methods features
Visibility modifiers

```
public class RealPoint {
   private double x;  // abscissa atribute
   private double y;  // ordenate atribute

   public void move(double px, double py) {
      x += px; y += py;
   }
   private int randomInt() {
      return (int)(Math.random()*(2*distOrg()+1));
   }
   public void moveRandom() {
      x += randomInt(); y += randomInt();
   }
   public double distOrg() {
      return Math.sqrt(x*x + y*y);
   }
   public double dist(RealPoint p) {
      double x = p.x - this.x; double y = p.y - this.y;
      return Math.sqrt(x*x + y*y);
   }
}
```

Dynamic methods features

Visibility modifiers



Dynamic methods features

Return datatype

- ***Return datatype***:
  - Datatype of the value returned by the method
  - `void` indicates that no value is returned by the method
  - The return datatype can be any Java datatype, primitive or reference
- Return value:
  - `return` specifies the returned value

```
return expr;
```

  - `expr` must be compatible with the method datatype
  - `return` ***terminates the method***
- `void` methods do not have return values
  - only `return;`
  - no `return` at all

Dynamic methods features

Parameters

- ***Parameters*** in the method header are ***formal parameters*** (no real value is associated to them until the method call)

- Syntax:

$$\texttt{type}_1\ \texttt{param}_1\texttt{,}\ \texttt{type}_2\ \texttt{param}_2\texttt{,}\ \texttt{...,}\ \texttt{type}_n\ \texttt{param}_n$$

  Warning!: do not get confused with variable declaration

  i.e., $\texttt{type}_1\ \texttt{param}_1\texttt{,}\ \texttt{param}_2\texttt{,}\ \texttt{...}$ is an incorrect parameter list

- Parameters values are usually given from the call point

- Methods ***without parameters*** can be declared as `methodName()`

Dynamic methods features

Parameters

```
public class RealPoint {
   private double x;  // abscissa atribute
   private double y;  // ordenate atribute

   ...

   public void move(double px, double py) {
      x += px; y += py;
   }

   public double dist(RealPoint p) {
      double ix = p.x - this.x; double iy = p.y - this.y;
      return Math.sqrt(ix*ix + iy*iy);
   }
}
```

Dynamic methods features

Variable scope

- When implementing a class, variables can be classified into:
  - Local variables: declared inside a method body
  - Parameters: declared in a method header
  - Attributes or global variables: declared outside all the methods

- ***Variable scope***: parts of the code where it can be used
  - Corresponds with the block where it is declared and all the internal blocks
  - Consequently:
    * Local variables: accesible only in the method where they are declared
    * Parameters: accesible only in the method where they are in the header
    * Attributes or global variables: accessible in all the methods of the class

Dynamic methods features

Variable scope

```
public class RealPoint {
   private double x;  // abscissa attribute
   private double y;  // ordenate attribute
      ...
   public void move(double px, double py) {
     x  +=  px; y += py;
   }

   public double dist(RealPoint p) {
     double ix = p.x - x; double iy = p.y - y;
     return Math.sqrt(ix*ix + iy*iy);
   }
}
```

Dynamic methods features

Body

- Body of a method is a **block of code**
- Includes any sequence of instructions
- Accessible data items:
  - Local variables
  - Parameters
  - Attributes (object and class)
  - `this` reference
- Non `void` methods must include a `return` sentence

Dynamic methods features

Body

Maximum proximity principle

- Local variables have preference on global variables
- `this` reference can be used to override this principle

```
public class RealPoint {
   private double x, y;
   ...
   public void move(double x, double y) {
      this.x += x; this.y += y;
   }
}
```

Dynamic methods features

Body

An example of maximum proximity principle:

```
public class RealPoint {
  private double x;
  private double y;
  ...
  public void move(double px, double py) {
    x+=px; y+=py;
  }

  // Wrong code!!                          // Correct code
  public double dist(RealPoint p){ |  public double dist(RealPoint p){
    double x, y;                   |    double x, y;
    x = p.x - x;                   |    x = p.x - this.x;
    y = p.y - y;                   |    y = p.y - this.y;
    return Math.sqrt(x*x + y*y);   |    return Math.sqrt(x*x + y*y);
  }                                |  }
}
```

Dynamic methods features

Use of methods

Method call syntax:

```
[object.]methodName(param_1, param_2, ..., param_n)
```

Where:

- `object`
  - It is the object on which the method is applied
  - It can be omitted for methods of current class (default object is `this`)
- `methodName` is the method identifier
- $\texttt{param}_1$, $\texttt{param}_2$, ..., $\texttt{param}_n$
  - It is the list of input parameters
  - They are ***actual parameters***
  - They are ***expressions*** that must evaluate to a specific value

> Formal and actual parameters must keep ***concordance*** with respect to its ***amount***, its ***order***, and its ***datatypes***

Dynamic methods features

Use of methods

Examples:

```
public class RealPoint {
  private double x;  // abscissa atribute
  private double y;  // ordenate atribute
  public void move(double px, double py) { x += px; y += py; }
  ...
  private int randomInt() {return (int)(Math.random()*(2* distOrg() +1)); }
  public double distOrg() { return Math.sqrt(x*x + y*y); }
  ...
}
```

```
RealPoint p = new RealPoint(); int x = 3;    // p is (0,0)
p.move(x, x+1) ;                             // p is (3.0, 4.0)
double d = 3 * p.distOrg() ;                 // d = 15.0
RealPoint q = new RealPoint(). move(d, p.distOrg()) ;    // q is (15.0, 5.0)
```

Dynamic methods features

Use of methods

Actual parameter

- Expression which evaluates to a datatype compatible with the corresponding ***formal parameter***
- The evaluation of the actual parameter is given to the formal parameter
- A method call is evaluated to the value returned by the method
- A method can be called in any expression that can employ its return value

Pass of parameters

- Java employs ***pass by value***: modifications in parameters not visible from call point
- For reference parameters, internal modificacions are visible from call point

Dynamic methods features

Constructors

Declaration syntax:

```
public ClassName( [ ParameterList ] ) { /* Instructions */ }
```

Features:

- Without datatype
- With the same name than the class
- Called by using `new`: `new ClassName(`$\texttt{param}_1$`, `$\texttt{param}_2$`, ..., `$\texttt{param}_n$`);`
- Result: the created object

```
public class RealPoint {
  ...
  public RealPoint() { x=0; y=0; }
  public RealPoint(double a,double o){
    x = a; y = o;
  }
  ...
}
```

```
RealPoint p = new RealPoint();
RealPoint q = new RealPoint(5,10);
double dist = q.dist(new RealPoint(2,6));
```

Dynamic methods features

Constructors

Using `this` in the constructor body:

```
public class RealPoint {
  private double x;
  private double y;
  public RealPoint(double ab,double or){
    this.x = ab; this.y = or;
  }
  public RealPoint() {
    this(0,0);
  }
  public RealPoint(RealPoint p) {
    this(p.x,p.y);
  }
  ...
}
```

Dynamic methods features

Method overload

Methods with the same name and datatype, but different list of parameters

```
public class RealPoint {
  private double x;
  private double y;
  public RealPoint(double ab,double or) {
    this.x = ab; this.y = or;
  }
  public RealPoint() {
    this(0,0);
  }
  public RealPoint(RealPoint p) {
    this(p.x,p.y);
  }
}
```

It can be applied in any kind of methods (not only constructors)

Contents

Static methods features

- ***static*** or ***class method***: pertains to the class (not to an object) and gives functionalities to the whole class
- Examples:
  - `main` method
  - Subprograms in program classes
  - Methods of the `Math` class
- Definition:
  - As instance methods but with `static` modifier
  - `this` and object attributes/methods cannot be used (no object associated)
- Syntax call:

```
[ClassName.]methodName(param1, param2, ..., paramn);
```

Example:

```
Math.sqrt(17.6);
```

Static methods features

Utility classes

Utility classes:

- Group static methods for general utilities on previously defined datatypes
- Attributes are unusual
- Default constructor creates empty objects (usually overriden as `private`)
- No `main` method is included
- Utility classes with related functionalities are usually grouped into `packages`
- Java provides several predefined utility classes in the `java.lang` package, e.g., `Math`

Contents

Overriding `Object` methods

`Object` class

- ***Inheritance***: new classes can be defined by the extension or restriction of the functionalities of previously defined classes
- Inheritance is the basic mechanism in OOL for code reusing
- Inheritance allows to model hierarchical relations between classes
- Inheriting class (derived) has the same features than the origin class (parent), but maybe is refined for a special case
- The Java class library is hierarchically organised
- `Object` is the base class for all the Java hierarchy

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Container
          └ java.awt.Window
              └ java.awt.Frame
                  └ javax.swing.JFrame
```

Overriding `Object` methods

`equals` and `toString`

- ***Object*** defines a common behaviour for all the objects of the language
- An `Object` instance does not have internal structure (empty object)
- Any object in Java is an instance of `Object` and inherits several methods, among others:

```
public boolean equals(Object o)
```

Checks whether the current object and `o` are the same object in the heap

```
public String toString()
```

Returns a `String` with the class of the object and a numerical code

Overriding `Object` methods

`equals` and `toString`

- Inherited methods can be ***overriden*** (write a new code) when necessary
- Example: class `RealPoint`

```
public boolean equals(Object o){
   return o instanceof RealPoint
          && this.x==((RealPoint)o).x
          && this.y==((RealPoint)o).y ;
}

public String toString(){
   return "("+this.x+","+this.y+")" ;
}
```

The standard recommends for `equals` to check whether `o` is an instance of the class (operator `instanceof`) or not, as well as checking attribute-by-attribute equality

- When Java finds an overriden method, it uses the new internal code
- Otherwise, it uses the code of the method in the `Object` class

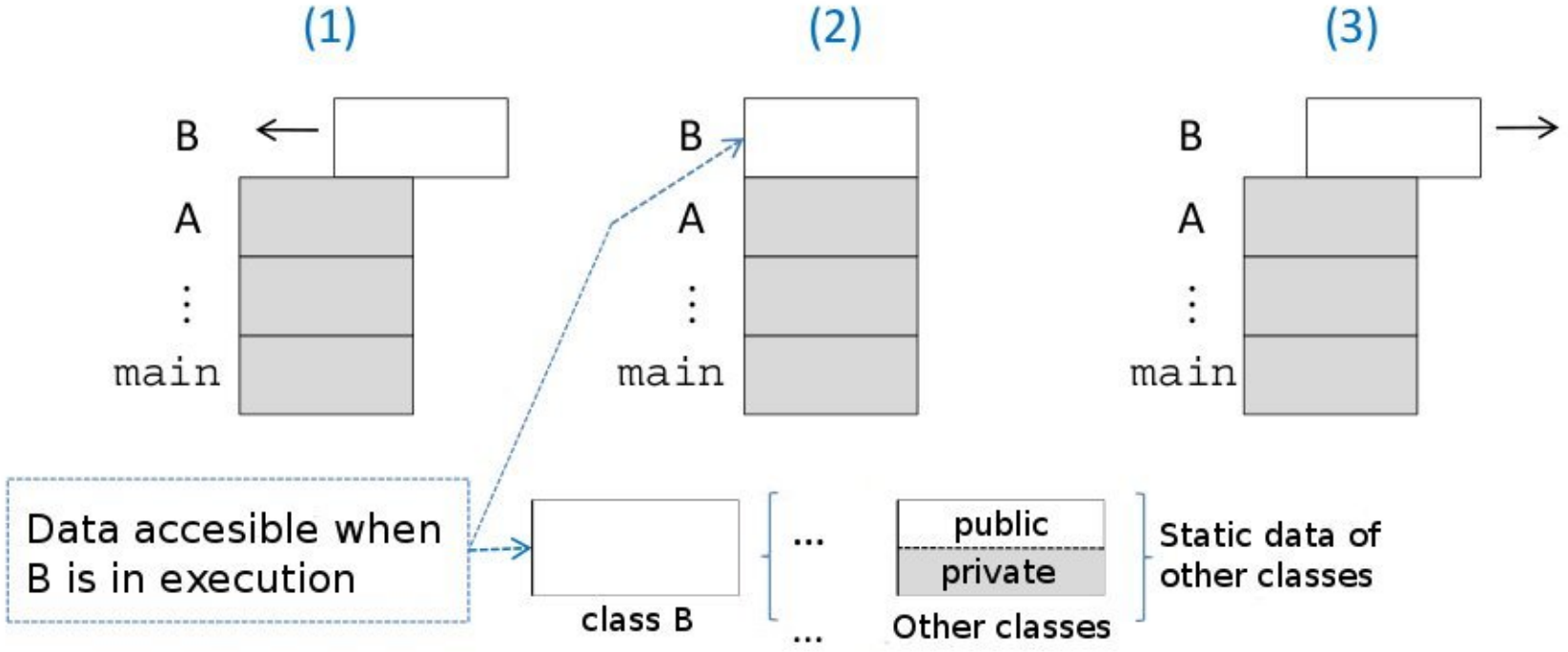
Contents

The call stack

- Java executes only the ***active method*** in a given moment
- When a method A calls to a method B:
  - The execution of A is suspended
  - The state of A is saved in its activation register
- Any activation register is destroyed only when its method finishes
- The memory may store many registers:
  - The ***active register***
  - A register for each suspended method

The call stack

- ***Call stack***: mechanism used for managing the registers
- Registers are ordered by oldness:
  - When A calls to B, register of B is situated on the top of the stack ***(1)***;
  - When finishing, the register of B pops ***(3)***
- Active method can only access data in the stack top register ***(2)***
- Global data (attributes) are apart and are accessible for the active method



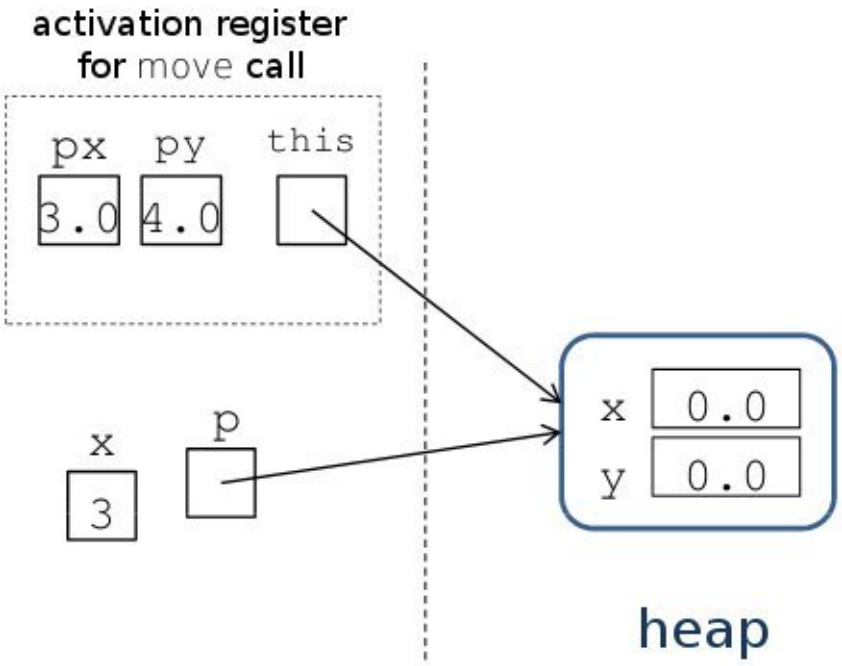
The call stack

Call process: when from method A a call to method B is executed

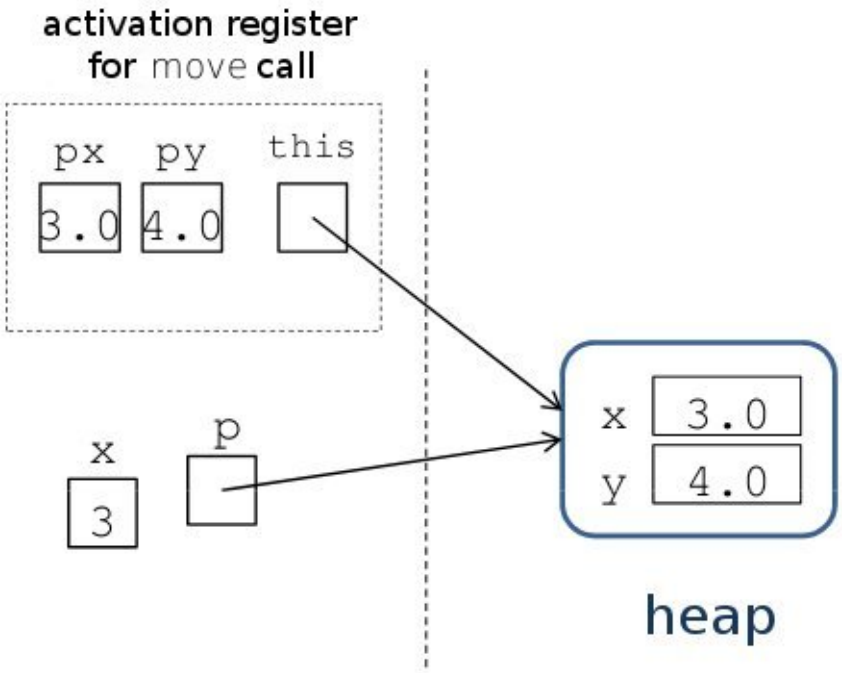
1. The actual parameters in A get evaluated
2. The memory for the activation register for method B is reserved
3. The parameters of B (present in the register) are given the values of the evaluation of the actual parameters in A, along with the `this` implicit parameter
4. The instructions in B get executed; the method finishes with `return` or by arriving to the last instruction
5. The execution returns from B to A, in the point of the call to B
6. The activation register for B is freed

The call stack

```
RealPoint p = new RealPoint(); int x = 3;    // p is (0,0)
p.move(x, x+1);                              // p is (3.0, 4.0)
double d = 3 * p.distOrg();                  // d = 15.0
RealPoint q = new RealPoint().move(d, p.distOrg()); // q is (15.0, 5.0)
```



At the beginning of `move`

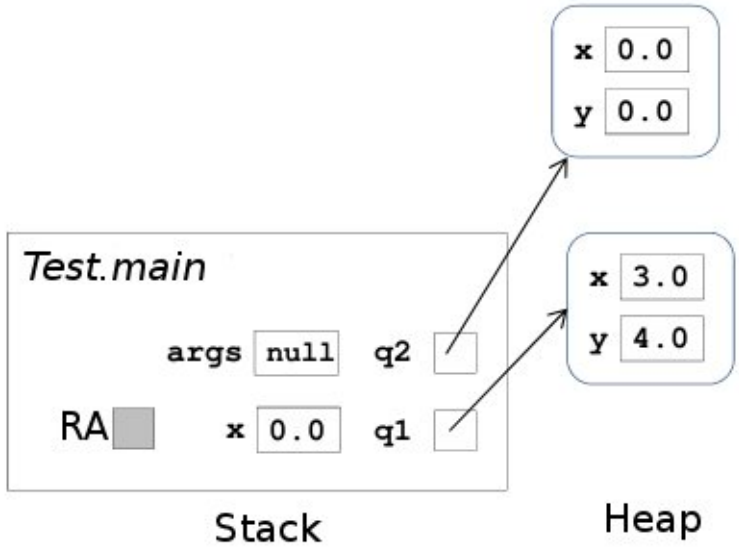


At the end of `move`

The call stack

- Java internal variables in each register (among others):
  - Return value (RV): stores value to be returned (except for `void` methods)
  - Return address (RA): address of the instruction of A where B was called
- E.g.: `main` method in `Test` class calls to `dist` method in `RealPoint` class

```
public double dist(RealPoint p){
    double x = p.x - this.x;
    double y = p.y - this.y;
    return Math.sqrt(x*x + y*y);
}
public static void main(String[] args){
    double x = 0.0;
    RealPoint q1 = new RealPoint(3.0,4.0);
    RealPoint q2 = new RealPoint();
    x = q2.dist(q1);        // <---
    System.out.println(x);
}
```

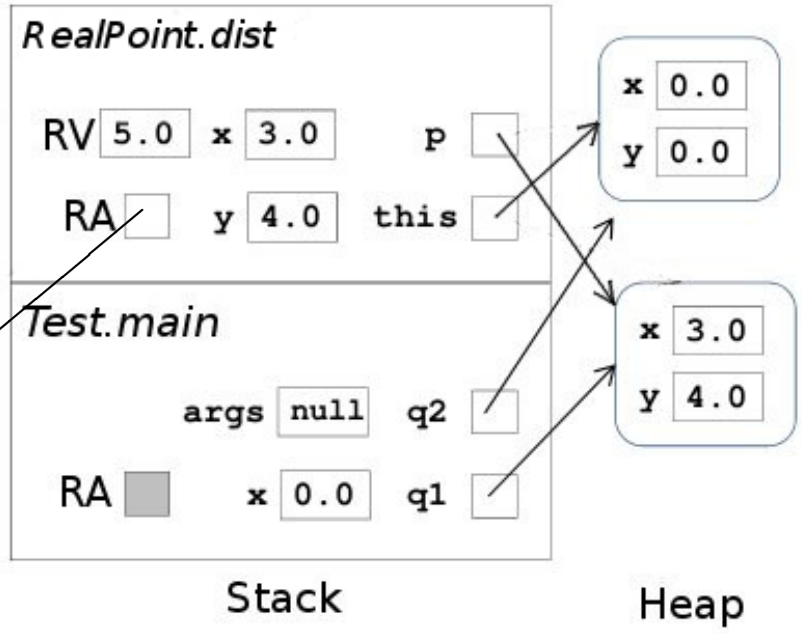


The call stack

Example (continued): the `dist` method calculates its result in its internal RV

JVM is going to return to `main` to the address pointed by RA

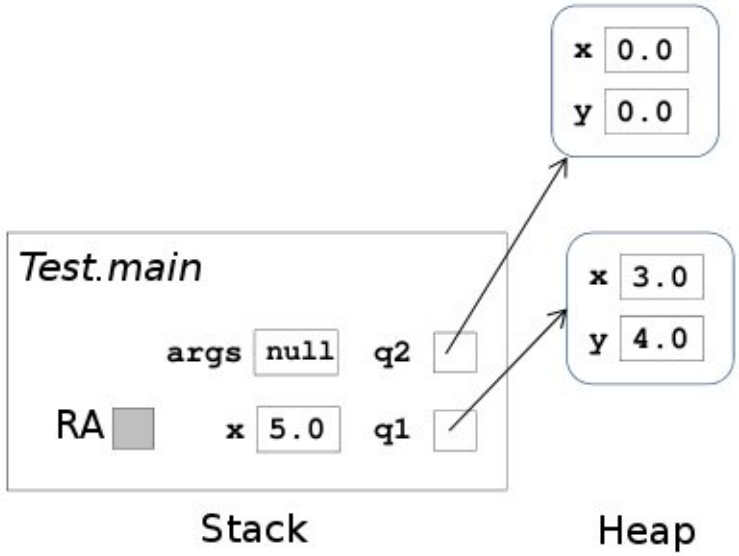
```
public double dist(RealPoint p){
    double x = p.x - this.x;
    double y = p.y - this.y;
    return Math.sqrt(x*x + y*y);  // <---
}
public static void main(String[] args){
    double x = 0.0;
    RealPoint q1 = new RealPoint(3.0,4.0);
    RealPoint q2 = new RealPoint();
    x = q2.dist(q1);
    System.out.println(x);
}
```



The call stack

Example (continued): `main` stores in `x` the result of the call to `dist`

```
public double dist(RealPoint p){
    double x = p.x - this.x;
    double y = p.y - this.y;
    return Math.sqrt(x*x + y*y);
}
public static void main(String[] args){
    double x = 0.0;
    RealPoint q1 = new RealPoint(3.0,4.0);
    RealPoint q2 = new RealPoint();
    x = q2.dist(q1);
    System.out.println(x);  // <---
}
```



The call stack

Call process for a constructor:

1. Actual parameters get evaluated, activation register is created, values assigned to the formal parameters

2. A memory block is reserved in the heap to keep the object (`this`)

3. Attributes get initialised (default or specified value)

4. Instructions of the body get executed

5. `this` is returned (implicitly)

Contents

Other issues on Java objects

Package organisation in Java

- A ***package*** organises and allows the use of previously defined classes
- Can be used to define and use new classes
- A package is group of classes that can be imported and used in other classes

```
package libUtil;
import javax.swing.*;
import java.awt.*;
public class BlackBoard extends JFrame {
  .....
}
```

- In Java, all classes are structured into packages
- Default package: `anonymous`
- The `java.lang` package is imported by default
- `java.lang` contains (among others) classes `Object`, `String`, and `Math`

Other issues on Java objects

Code documentation

- Documentation of methods in a class is used:
  1. ***Previously to implementation***: specifies method features desired result; the method must be implemented according to this specification
  2. ***Posterior to and independently of implementation***: indicates how to use the methods (profile, conditions of parameters, results)

- References to implementation must be avoided

- Comments on implementation must be in the body of the method

Other issues on Java objects

Code documentation

Java has a documentation standard that defines which comments must be included in the source code

- ***javadoc*** can generate HTML documentation from standard comments
- Basic format of standard class comments:

```
/** ClassName class: description of the class
  * @author Author Name
  * @version Version Number/Date
  */
```

- Basic format of standard method comments:

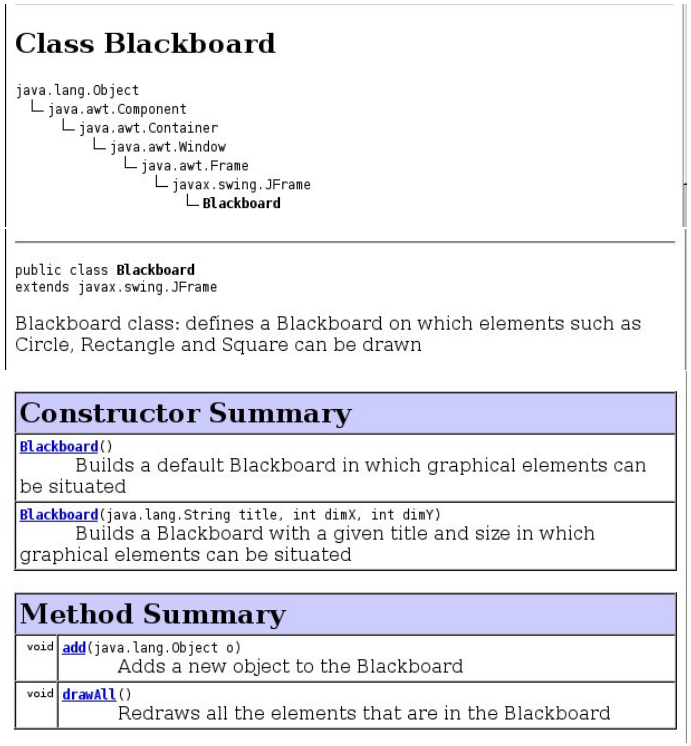
```
/**      Description of the method, including
 *       data conditions and special cases
 *       @param  param1   type1
  ......            ......
 *       @param  paramN   typeN
 *       @return returnType returned value
 */
```

Parameter description

Description of return value (for non `void` methods)

Other issues on Java objects

Code documentation



Class Blackboard

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Container
          └ java.awt.Window
              └ java.awt.Frame
                  └ javax.swing.JFrame
                      └ Blackboard
```

public class **Blackboard**
extends javax.swing.JFrame

Blackboard class: defines a Blackboard on which elements such as Circle, Rectangle and Square can be drawn

Constructor Summary

Blackboard() Builds a default Blackboard in which graphical elements can be situated
Blackboard(java.lang.String title, int dimX, int dimY) Builds a Blackboard with a given title and size in which graphical elements can be situated

Method Summary

void	**add**(java.lang.Object o) Adds a new object to the Blackboard
void	**drawAll**() Redraws all the elements that are in the Blackboard

```
import javax.swing.*;
import java.awt.*;
/**
 * Blackboard class: defines a Blackboard on which elements
 * such as Circle, Rectangle and Square can be drawn
 * @author IIP-PRG Book
 * @version 2011
 */
public class Blackboard extends JFrame {
    // Blackboard default elements:
    private static int DIM_X = 200;
    private static int DIM_Y = 200;
    private static int NUM_MIN = 8;
    // Graphical objects storing attributes
    private Object gOL[] = new Object[NUM_MIN];
    private int numGO = 0;
    // Consts for possible shapes
    private final static int UNKNOWN = -1;
    private final static int CIRCLE = 0;
    private final static int RECTANGLE = 1;
    private final static int SQUARE = 2;
    // Constant arrays for colors
    private static final String NAME_COLS[] =
            {"red","yellow","green","blue","orange","black"};
    private static final Color COLS[] =
            {Color.red, Color.yellow, Color.green, Color.blue,
             Color.orange, Color.black};
    private static final Color DEFAULT_COLOR = Color.black;
    /**
     * Builds a default Blackboard in which graphical elements
     * can be situated
     */
    public Blackboard() {
        super("Default blackboard"); setSize(DIM_X,DIM_Y);
        setContentPane(initPanel()); setVisible(true);
    }
}
```