An introduction to Java

January 2020

François de Coligny – Nicolas Beudez

INRAE - UMR AMAP
botAny and Modelling of Plant Architecture and vegetation

Java training - Contents

History

James Gosling and Sun Microsystems

- Java: May 20, 1995
- Java 1 → Java 8 (i.e. 1.8), March 2014
- Oracle since 2010

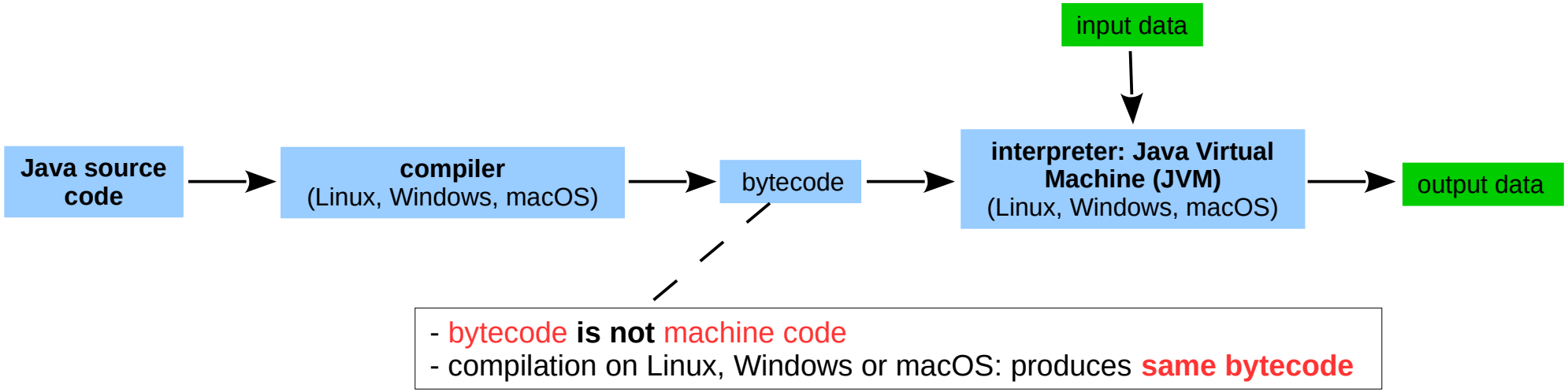
Specificities

Java is an object oriented language

object = a software brick (see later)

- **clean**, **simple** and **powerful**

- Java: **compiled** and **interpreted** language



Java is **portable** (Linux, Windows, macOS):
"write once, run everywhere" (**no recompilation** is needed)

- **static typing** (checks during compilation)

- simpler than C++ (automatic memory management, no pointers, no headers...)

Programming environment

Java environment

- JRE (Java Runtime Environment) — contains the **'java'** interpreter: JVM (Java Virtual Machine)
- JDK (Java Development Kit) — JRE + the **'javac'** compiler + ...

Several versions

- Jave SE (Standard Edition)
- Java EE (Enterprise Edition → Web)
- Java ME (Micro Edition)

Editors

- simple editors: **Notepad**++ (Windows), **TextPad** (Windows), **SciTE** (multi-platform), **gedit** (multi-platform) → syntax coloring...
- IDEs (Integrated Development Environment):
  - **Eclipse** (multi-platform) → completion, refactoring...

Installation

Windows / Linux

- download and install the **JDK (Java SE 8)**
- modify the **PATH environment variable**

  add the **java/bin/** directory (contains **javac** and **java** programs) at the beginning of the **PATH** variable

  e.g. C:/Program Files/Java/jdk1.8.0_102/bin *(Windows)*

  /home/beudez/applications/jdk1.8.0_102/bin *(Linux)*
- install **text editor**:

  TextPad or **Notepad++** *(Windows)*

  gedit, SciTE *(multi-platform)*

Check the installation

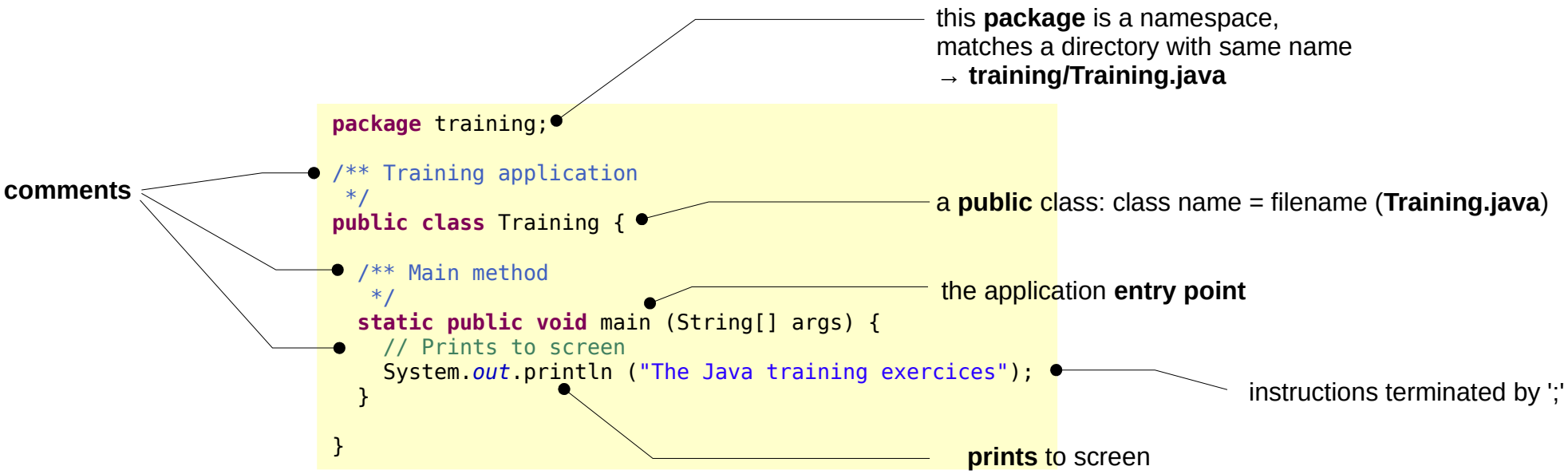
- in a terminal: **javac -version** and **java -version**

```
beudez@nicolas-HP:~$ java -version
java version "1.8.0_102"
Java(TM) SE Runtime Environment (build 1.8.0_102-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.102-b14, mixed mode)
beudez@nicolas-HP:~$
beudez@nicolas-HP:~$ javac -version
javac 1.8.0_102
beudez@nicolas-HP:~$
```

Bases

- a Java application
- the development process
- variables, simple types
- arithmetic operators
- boolean operators
- mathematical tools
- arrays
- conditions: if, else if, else
- loops: while, do... while
- loops: for
- loops: continue or break
- runtime exceptions
- exceptions management

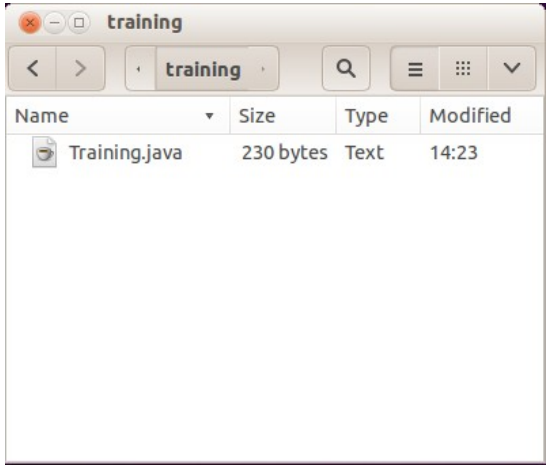
A Java application

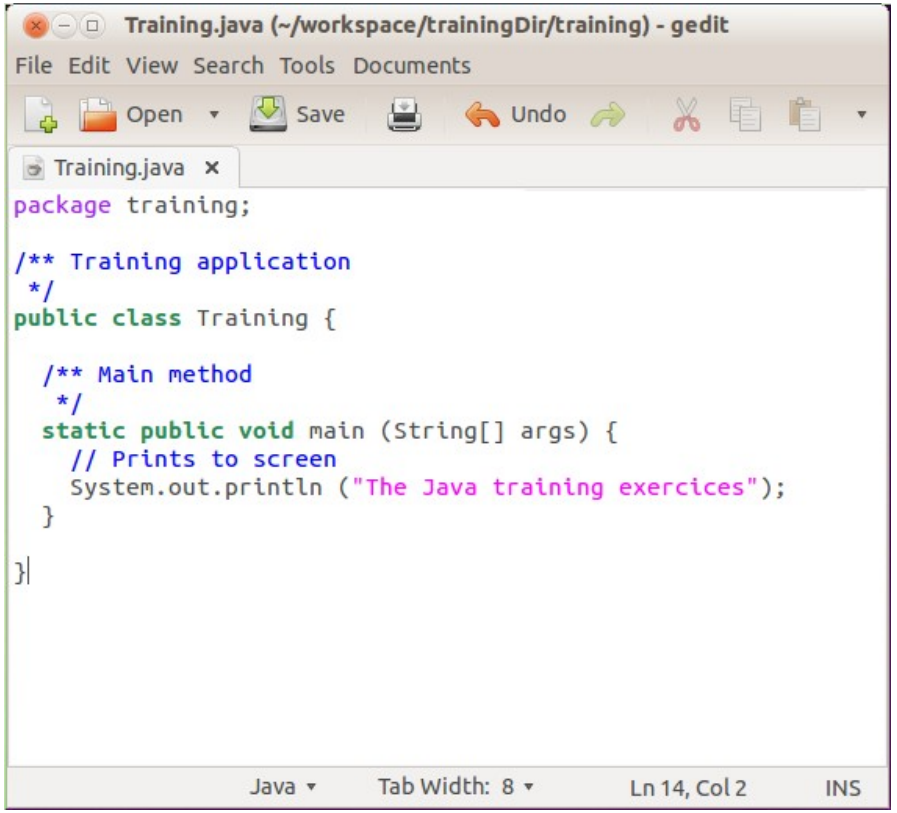



```
package training;

/** Training application
 */
public class Training {

 /** Main method
  */
  static public void main (String[] args) {
    // Prints to screen
    System.out.println ("The Java training exercices");
  }

}
```

A Java application

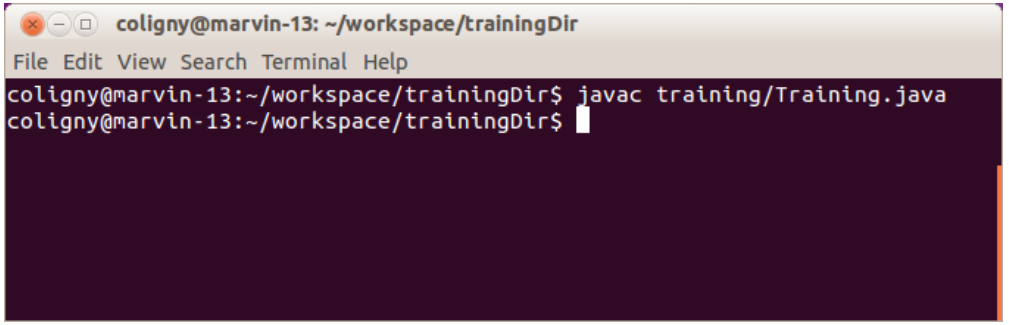
- Java programs are written with a text editor in files with a **'.java'** extension: **sources files**
- applications are .java files with a **public static void main(...) {...}** method

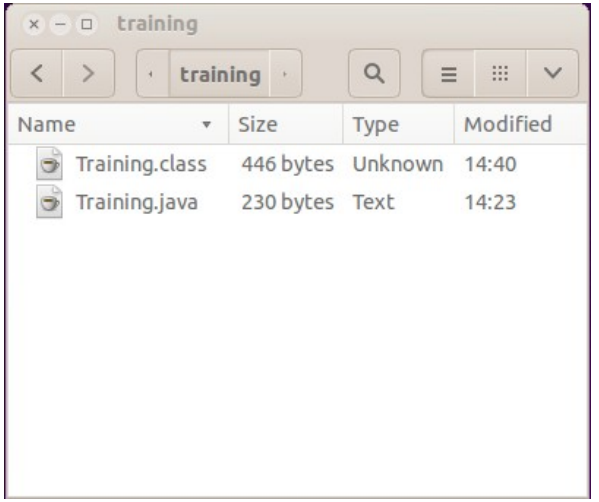




A Java application

- to compile a Java application, use the javac compiler (part of the JDK) in a terminal
- returns a Java bytecode file: Training.class

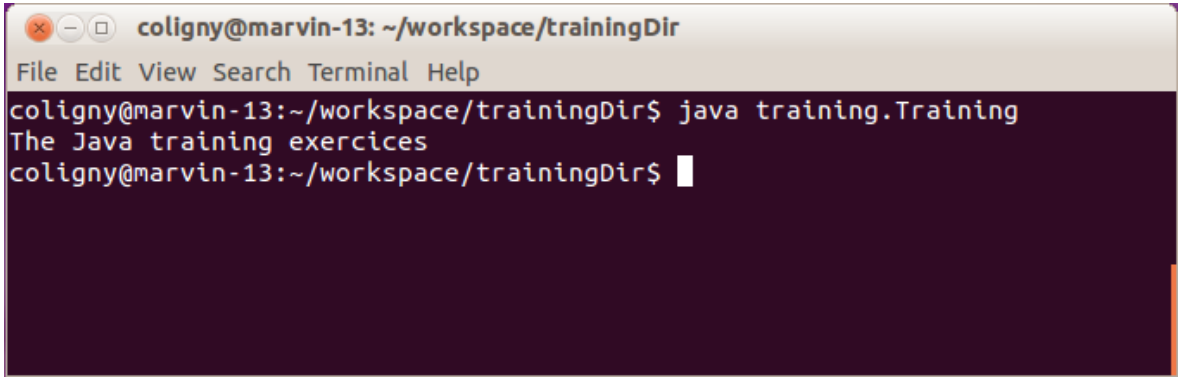




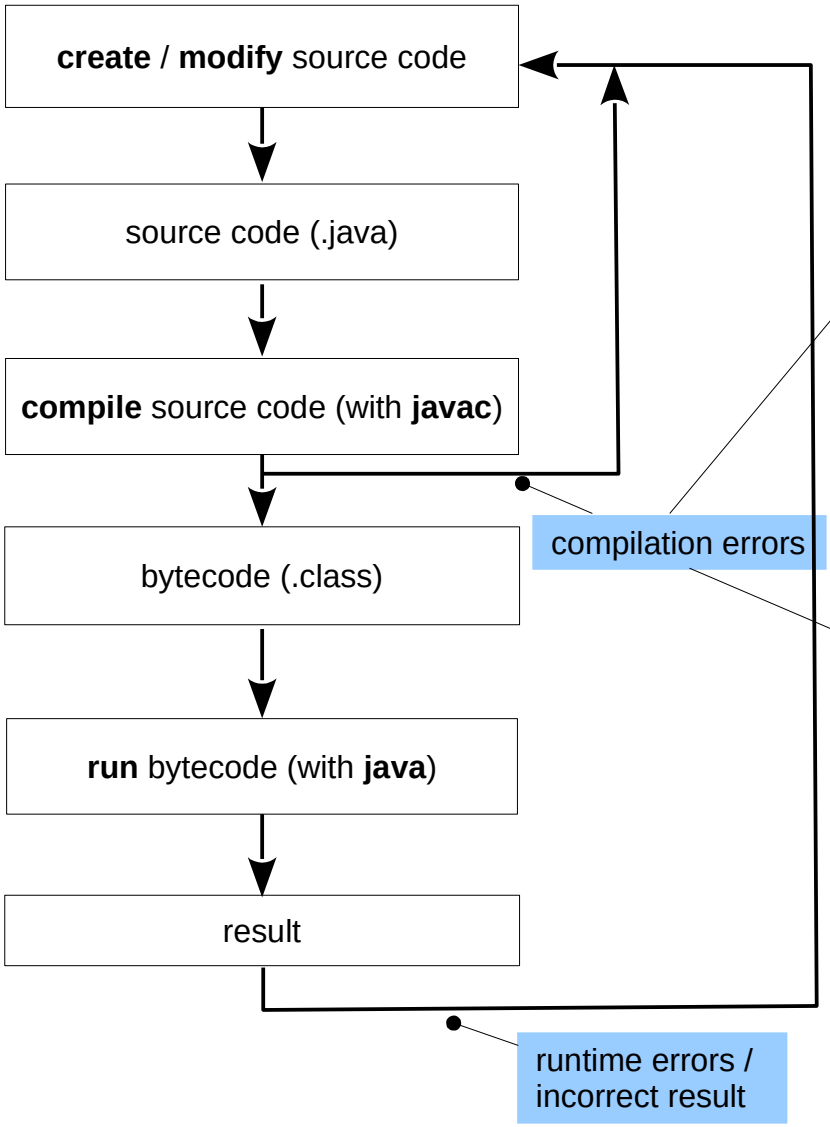
A Java application

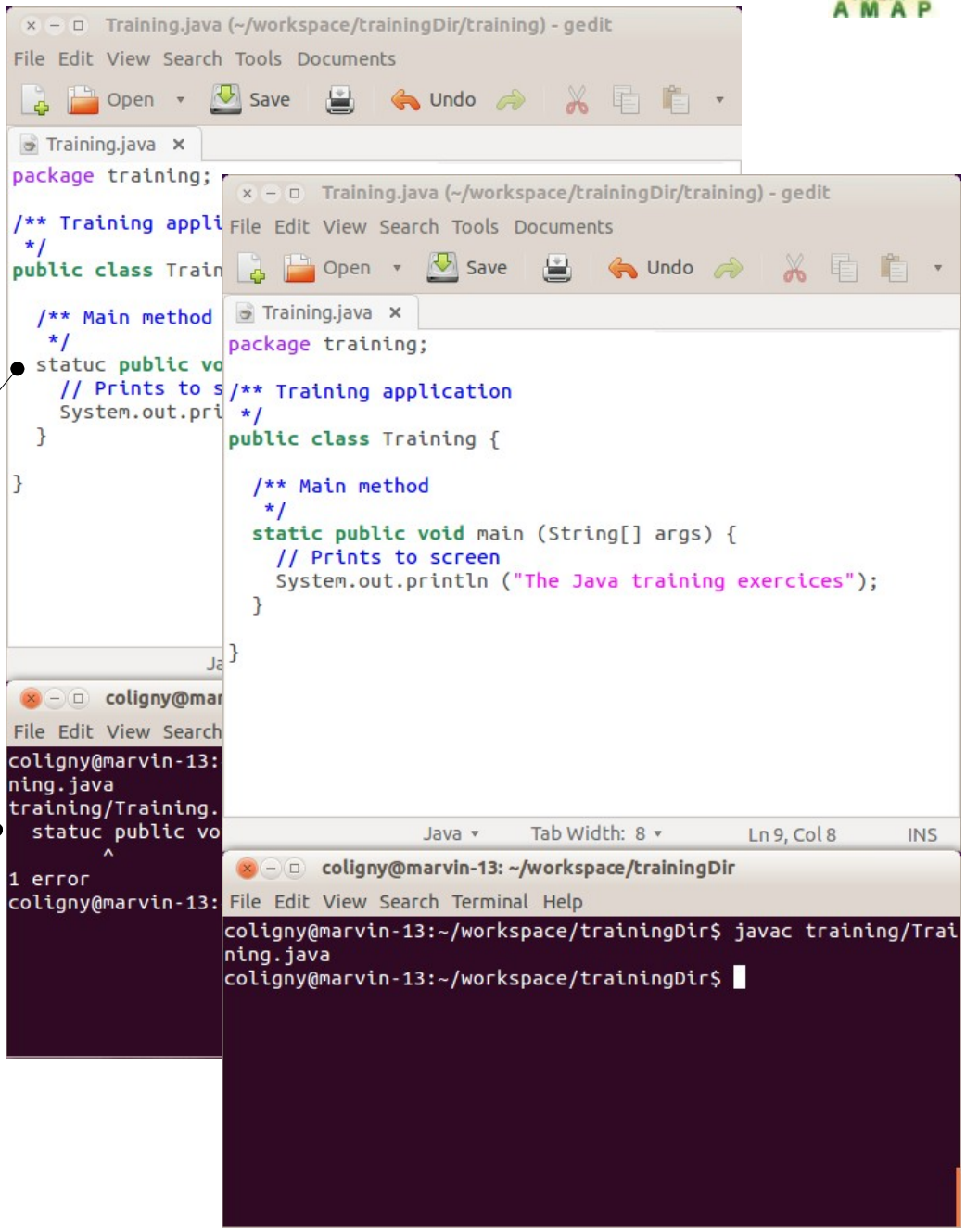
- to run a Java application, use the java interpreter (or Java Virtual Machine, JVM) in a terminal

the result



The development process





errors fixed, result is correct

Variables, simple types

Variable

- a variable has a **type** and holds a **value**
- a **variable name** starts with a lowercase letter (convention), e.g. myVariable

Integer types:

Type	Size (bits)	Minimum value	Maximum value	Example
`byte`	8	-128 (= $-2^8/2$)	127 (= $2^8/2-1$)	`byte b = 65;`
`short`	16	-32 768 (= $-2^{16}/2$)	32 767 (= $2^{16}/2-1$)	`short s = 65;`
`int`	32	-2 147 483 648 (= $-2^{32}/2$)	-2 147 483 647 (= $2^{32}/2-1$)	`int i = 65;`
`long`	64	-9 223 372 036 854 775 808 (= $-2^{64}/2$)	9 223 372 036 854 775 807 (= $2^{64}/2-1$)	`long l = 65L;`

Floating types:

Type	Size (bits)	Absolute minimum value	Absolute maximum value	Example
`float`	32	$1.40239846 \times 10^{-45}$	$3.40282347 \times 10^{38}$	`float f = 65f;`
`double`	64	$4.9406564584124654 \times 10^{-324}$	$1.797693134862316 \times 10^{308}$	`double d = 65.55;`

Character:

Type	Size (bits)	Example
`char`	16	`char c = 'A';`

Boolean:

Type	Size (bits)	Example
`boolean`	1	`boolean b = true;`

Declaration

value assignment

```
int i = 0;
double a = 5.3;
boolean found = false;
char letter = 'z';

String name = "Robert";
```

not a simple type (seen later)

Arithmetic operators

Arithmetic

- simple: **+**, **-**, *****, **/**, **%**
- increment / decrement: ++, --
- combined: **+=**, **-=**, ***=**, **/=**
- precedence with **parentheses**
- comparison: **<**, **<=**, **>**, **>=**, **==**, **!=**

index = index + 2;

i++; (same as: i = i+1;)

index += 2; (same as: index = index+2;)

(a + b) * c;

Beware of the int division

String concatenation:
“a string” **+** *something* turns *something* into a String and appends it

```
double r = 3d / 2d;
double s = 3 / 2;

System.out.println ("r: "+ r + " s: " + s);
```

```
coligny@marvin-13:~/workspace/trainingDir$ javac training/PrimitiveTypes.java
coligny@marvin-13:~/workspace/trainingDir$ java training.PrimitiveTypes
r: 1.5 s: 1.0
```

Caution

Boolean operators

Boolean variables are true or false boolean v = **true**;

Boolean calculation

- AND: **&&**
- inclusive OR: **||**
- NOT: **!**
- test equality: **==**
- test non equality: **!=**
- use **()** for precedence

(a<b) && (c<d)

is *true* if the two expressions *a<b* and *c<d* are both *true*, is *false* otherwise

(a<b) || (c<d)

is *true* if **at least** one of the two expressions *a<b* and *c<d* is *true*, is *false* otherwise

!(a<b)

is *true* if the expression *a<b* is *false*, is *false* otherwise (same value than *a>=b*)

```
boolean found = isFileInSystem("trees.txt");
if (found){
  readFile("trees.txt");
}
```

if (found) ⟺ if (found == true)

Mathematical tools

Constants

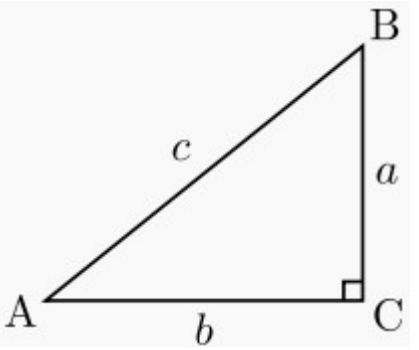
- Math.PI, Math.E

Trigonometry and other operations

- Math.cos(), Math.sin(), Math.tan()...
- Math.pow(), Math.sqrt(), Math.abs(), Math.exp(), Math.log()...
- Math.min(), Math.max(), Math.round(), Math.floor(), Math.ceil()...
- Math.toDegrees(), Math.toRadians()...

```java
// Square root
double a = 3;
double b = 4;
double c = Math.sqrt(a * a + b * b);

System.out.println("c: " + c);
```

```
coligny@marvin-13:~/workspace/trainingDir$ java training.PrimitiveTypes
c: 5.0
```



Arrays

- set of elements of **same type** (array of ‘int’, array of ‘double’,…) designated by a **unique name**
- 1, 2 or more dimensions
- managed by references
- memory allocation: with the **new** keyword
- **null** if not initialised
- can not be resized
- access elements with the **[ ]** operator
- indices begin at 0
- **size**: myArray.length

```
String[] a = new String[12];
a[11] = "Bob";

String[] b = {"Jack", "William", "Joe"};

int size = 4;
double[] c = new double[size];

double[][] d = new double[4][6];
d[0][2] = 3d ;
d[3][5] = 1d ;

// Index error: max is d[3][5]
System.out.println (d[4][6]);
```

null	null	null	null	null	null	null	null	null	null	null	Bob

Jack	William	Joe

0	0	0	0

2 dimensions

	0	1	2	3	4	5
0	0	0	3	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	1

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
        at training.Training.main(Training.java:31)
```

a runtime exception

Conditions: if, else if, else

Tests a simple condition

- can be combined

```
// Simple if
if (i == 10) {
  // do something
}

// Complex if
if (count < 50) {
  // do something
} else if (count > 50) {
  // do something else
} else {
  // count == 50
}

// Boolean expression
if (index >= 5 && !found) {
  System.out.println ("Could not find in 5 times");
}
```

Loops: while, do... while

Loop with condition

- while (condition) {…}

- do {...} while (condition);

Same results

while: condition is tested **first**

```
int count = 0;
while (count < 10) {
  count++;
}

System.out.println ("count: " + count);
```

count: 10

do... while: condition is tested at the **end**
→ always at least one iteration

test is at the end

```
int count = 0;
do {
  count++;
} while (count < 10);

System.out.println ("count: " + count);
```

count: 10

Different results

```
int count = 10;
while (count < 10) {
  count++;
}

System.out.println ("count: " + count);
```

count: 10

```
int count = 10;
do {
  count++;
} while (count < 10);

System.out.println ("count: " + count);
```

count: 11

Loops: for

Loop a number of times

- for (initialisation; stop condition; advance code) {...}

```
// With an array
int[] t = new int[12];
int sum = 0;
for (int i = 0; i < t.length; i++) {
  t[i] = i;
  sum += t[i];
}
```

from 0 to 11

sum: 66

Loops: continue or break

```
// Search an array
int[] t = new int[12];
int sum = 0;
int i = 0;

for (i = 0; i < t.length; i++) {

  if (t[i] == 0) continue;

  sum += t[i];

  if (sum > 50) break;

}
System.out.println ("i: " + i +" sum: " + sum);
```

from 0 to 11

```
i: 10 sum: 55
```

- an internal **continue** jumps to the next iteration
- an internal **break** gets out of the loop

- for all kinds of loops (for, while, do while)

Runtime exceptions

Something wrong during the execution

- could not be checked at compilation time
- e.g. try to **access an element outside the bounds of an array**
  → java.lang.ArrayIndexOutOfBoundsException
- e.g. try to **use an array that was not initialised**
  → java.lang.NullPointerException
- e.g. try to **read a file that could not be found**
  → java.io.FileNotFoundException

- exceptions stop the program if not managed...

Exceptions management

Exceptions can be managed everywhere

→ use a try / catch statement

this file does not exist

-1- this code raises an exception

```
String fileName = "wrongName";

try {

    BufferedReader in = new BufferedReader (new FileReader (fileName));
    String str;
    while ((str = in.readLine ()) != null) {
        //process (str);
    }
    in.close();

} catch (Exception e) {
    System.out.println ("Trouble: " + e);
}
```

-2- this code is skipped

-3- the catch clause is evaluated

-4- the trouble is reported
catch should never be empty!

Trouble: java.io.FileNotFoundException: wrongName (No such file or directory)

Object oriented programming (O.O.P.)

Java is an object oriented language...

- **encapsulation**
- vocabulary
- **class**
- properties
- **constructor**
- **instance(s)**
- **method**
- calling methods
- memory management
- **inheritance**
- specific references
- constructors chaining
- method overloading / overriding
- static variable and method
- **interface**
- abstract class
- the 'Object' superclass
- enums
- **polymorphism**
- cast using the ‘instanceof’ operator
- packages and import
- lifetime of variables
- Java reserved keywords
- Java modifiers

Not presented here:

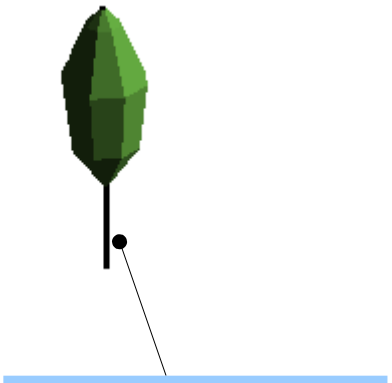
- static initializer
- nested class
- ...

Introduction to object oriented programming (O.O.P.)

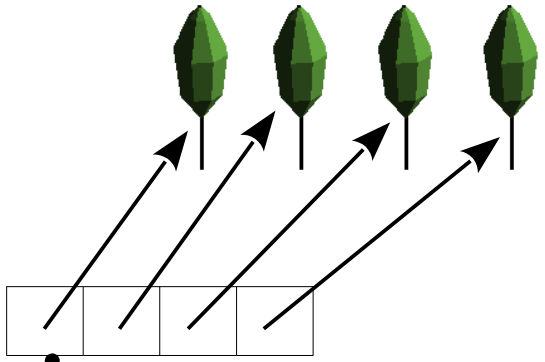
- The **O.O.P.**:
  - is based on structured programming
  - contributes to the **reliability** of softwares
  - makes it easy to **reuse** existing codes
  - introduces new concepts: object, encapsulation, class, inheritance

- In **O.O.P.** a program implements different **objects** (= a software brick).

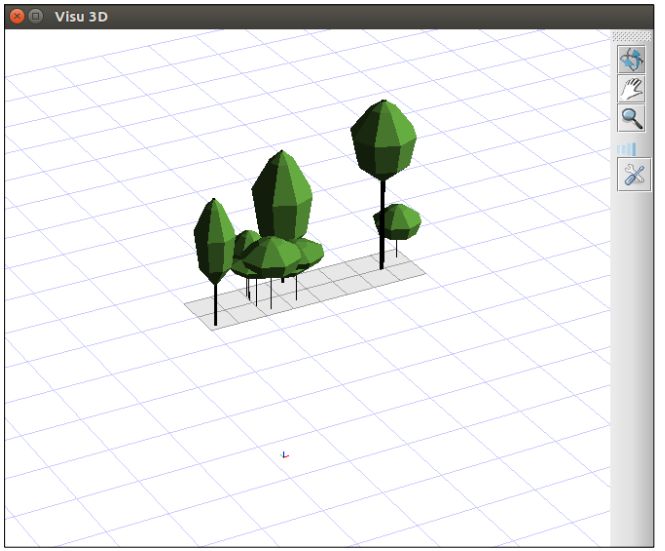
- Different kinds of objects:



a tree (‘physical’ object)



a list of trees (‘organizational’ object)



a tree 3D viewer (‘graphical’ object)

and many others...

Encapsulation

Bundle data and methods operating on these data in a unique container:

→ the object

Hide the implementation details to the users (developers) of the object, they only know its 'interface' (interface = the functions that one wishes to show to the user)

```
package training;

/**   A simple tree
 */
public class Tree {

  // diameter at breast height, cm
  private double dbh;

  public Tree () {}

  public void setDbh (double d) {
    dbh = d;
  }

  public double getDbh () {
    return dbh;
  }

}
```

data

methods operating on these data

Vocabulary

Class

- a class = a **new data type** → generalization of the concept of simple type (example: Tree)
- source files describe classes

Object

- instance of a class at runtime
- memory allocation
- several objects may be built with the same class (example: 3 instances of Tree class)

Instance variable (iv)

- variables of an object (example: dbh)
- (field, attribute, member data)

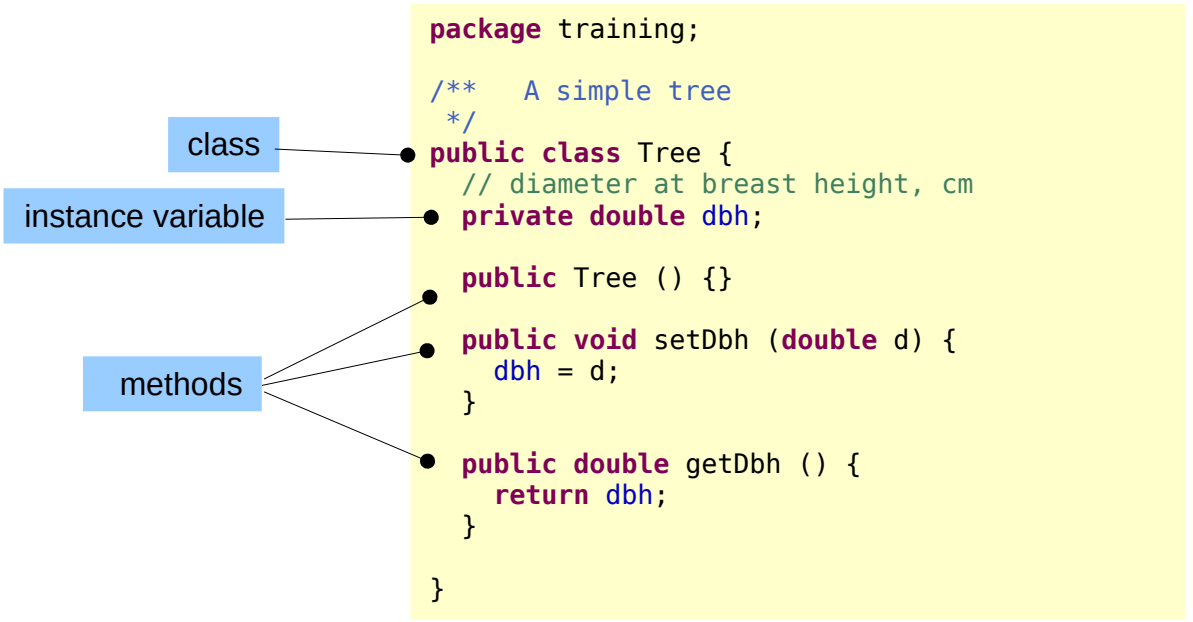
Method

- function of an object (example: setDbh(), getDbh())
- (procedure, member function)

Property

- instance variable or method

Class



A class is a new data type

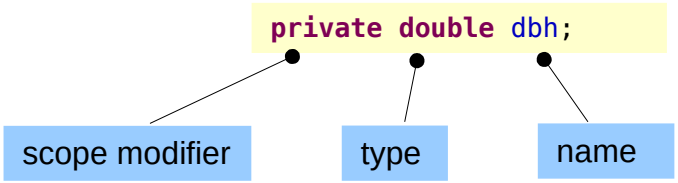
e.g. int, double, float, boolean, String, Tree...

Scope modifiers for the properties

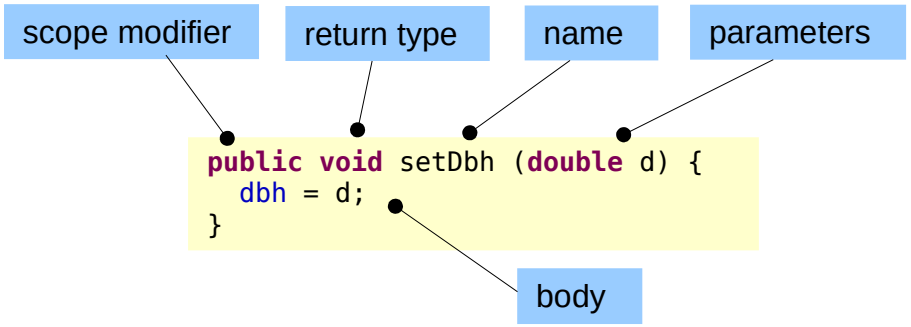
- **public** : visible by all (interface)
- **protected** : visible in the package (and in later seen subclasses...)
- **private** : scope is limited to the class (hidden to the others)

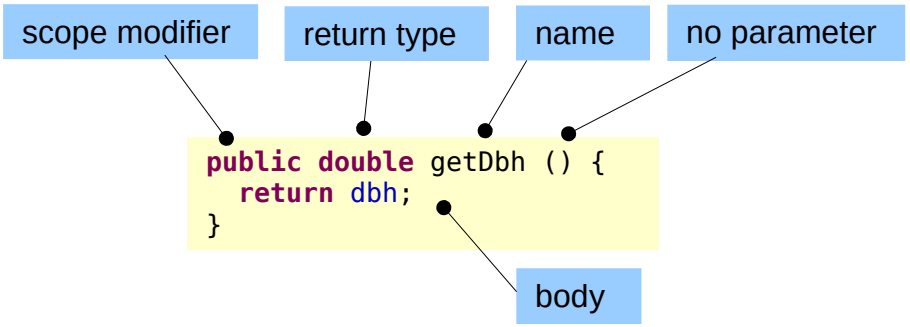
Properties

Instance variable



Method





A rule:

parentheses after the name → it is a method

Method

Classes contain instance variables and methods

- a class can contain several methods
- if no parameter, use **()**
- if no return type, use **void**

```
package training;

/**   A simple tree
 */
public class Tree {
  // diameter at breast height, cm
  private double dbh;

  public Tree () {}

  public void setDbh (double d) {
    dbh = d;
  }

  public double getDbh () {
    return dbh;
  }

}
```

constructors are particular methods without a return type

setDbh () method: 1 parameter
setSomething () is a **mutator**

getSomething () is an **accessor**
returns something

Constructor

- **particular method** called at object creation time
- **same name** than the class (starts with an uppercase letter)
- **no return type**
- deals with instance variables **initialisation**
- **several** constructors may coexist if they have different number and/or types of parameters

```
package training;

/**   A simple tree
 */
public class Tree {
  // diameter at breast height, cm
  private double dbh;

  public Tree () {}

  public Tree (double d) {
    dbh = d;
  }

  public void setDbh (double d) {
    dbh = d;
  }

  public double getDbh () {
    return dbh;
  }

}
```

a default constructor (no parameter)

another constructor (takes a parameter)

regular method with a parameter

Notes:
- this default constructor does nothing particular
  → 'dbh' is a numeric instance variable
  → set to 0 automatically
- the other constructor initializes 'dbh'

Instance

Vocabulary:
object = instance

Vocabulary:
the properties of the object
the properties of the class
→ instance variables + methods

Instanciation

- creates an instance of a given class
- i.e. an object

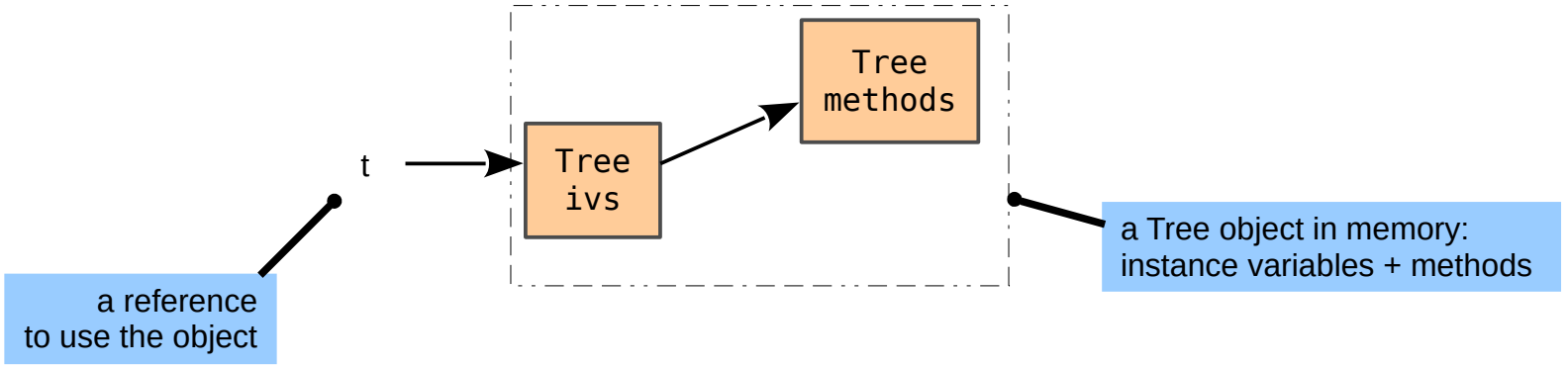
-1- declaration of a reference
type + name
no object created yet

```
// make an instance of Tree
Tree t;
t = new Tree ();

// same than
Tree t = new Tree ();
```

-2- creation of the object
new → instanciation
class name = constructor name

What happens in memory

- new → instanciation = memory reservation for the instance variables (ivs) + the methods
- the constructor is called (initialisations)
- returns a reference to the created object (a reference contains the address of an object)
- we assign it to the reference named 't'



Instances

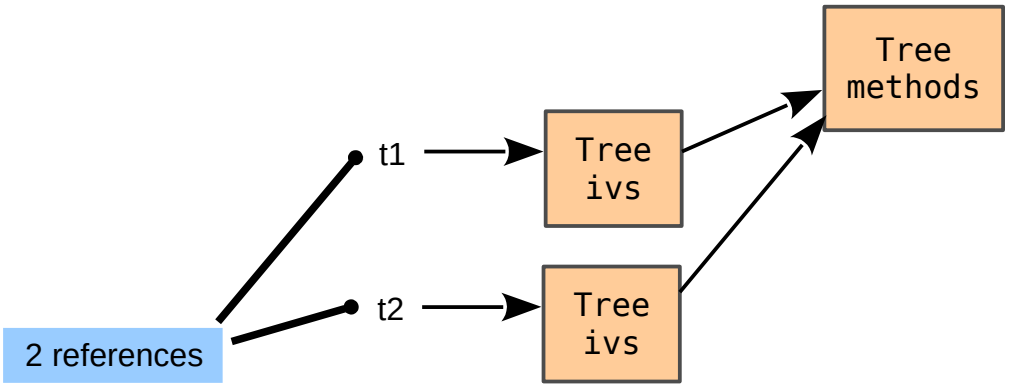
Creation of several objects

```
// Create 2 trees
Tree t1 = new Tree ();

Tree t2 = new Tree ();
```

2 times new → 2 objects

What happens in memory

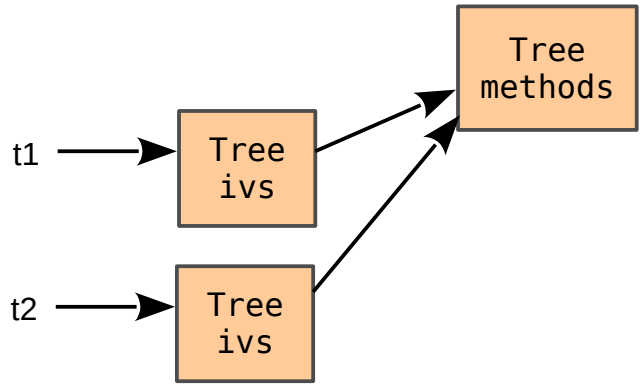
- 2 times ‘new’: 2 memory reservations for the instance variables of the 2 objects (their ‘dbh’ may be different)
- the constructor is called for each object
- the methods of the 2 objects are shared in memory
- each ‘new’ returns a reference to the corresponding object
- we assign them to 2 different references named 't1' and 't2'



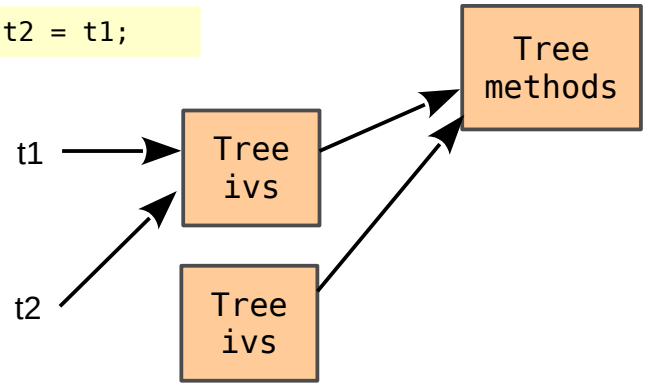
Instances

Using the references

```
// Create 2 trees
Tree t1 = new Tree ();

Tree t2 = new Tree ();
```

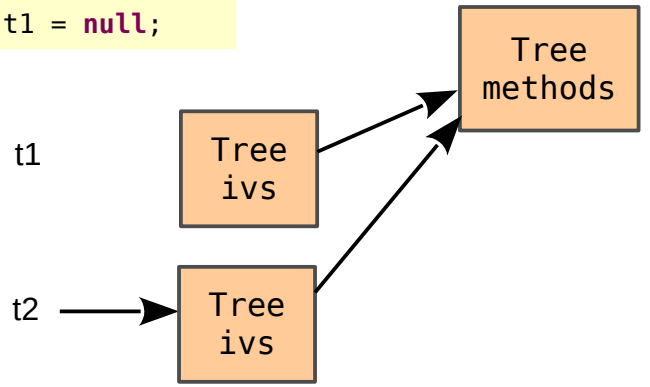



```
t2 = t1;
```



- both ‘t1’ and ‘t2’ point to the first tree
- the second tree is 'lost'

```
t1 = null;
```



- ‘t1’ points to nothing
- ‘t2’ points to the second Tree
- the first Tree is 'lost'

Calling methods

Definition of **Tree** class
(***Tree.java*** file)

```
package training;

/**   A simple tree
 */
public class Tree {
  // diameter at breast height, cm
  private double dbh;

  public Tree () {}

  public void setDbh (double d) {
    dbh = d;
  }

  public double getDbh () {
    return dbh;
  }

}
```

Use of Tree class
(***Training.java*** file)

```
// Create a tree
Tree t1 = new Tree ();

// Set its diameter
t1.setDbh (12.5);

// Print the diameter
double d1 = t1.getDbh ();

System.out.println ("t1 dbh: " + d1);
```

System is a class
out is a static public variable of type PrintStream
println () is a method of PrintStream

writing in out writes on the 'standard output'

Method returning nothing (void)

reference.method (parameters);

Method returning something

returnType variable = reference.method (parameters);

Memory management

- objects are instantiated with the keyword **new** → memory allocation
- objects are **destroyed** when there is no more reference on them → **garbage collecting**
  - this process is automatic
  - to help remove a big object from memory, set all references to null

```
// Declare two references
Tree t1 = null;

// Create an object (instanciation)
t1 = new Tree ();

// The object can be used
double v = t1.getDbh ();

// Set reference to null
t1 = null;
```

no object created yet

the object will be destroyed by the garbage collector

Inheritance

UML notation

Tree

SpatializedTree

How to create a spatialized tree ?

Simple manner results in **duplicates**...

```java
package training;

/**   A simple tree
 */
public class Tree {
  // diameter at breast height, cm
  private double dbh;

  public Tree () {}

  public void setDbh (double d) {
    dbh = d;
  }

  public double getDbh () {
    return dbh;
  }

}
```

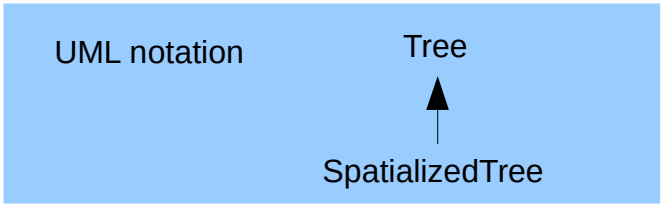
Tree.java file

```java
package training;

/**   A tree with coordinates
 */
public class SpatializedTree {
  // diameter at breast height, cm
  private double dbh;
  // x, y of the base of the trunk (m)
  private double x;
  private double y;

  /** Default constructor
   */
  public SpatializedTree () {
    setXY (0, 0);
  }

  public void setDbh (double d) {
    dbh = d;
  }

  public double getDbh () {
    return dbh;
  }

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

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

}
```

SpatializedTree.java file

No scalable and **no maintainable**

Inheritance



Reuse a class to make more specific classes
- e.g. a tree with coordinates
- inheritance corresponds to a '**is a' relation** — a spatialized tree **is a** tree (with coordinates)
- a **subclass** has all the instance variables and methods of its parent: the **superclass**
- all classes inherit from the **Object** class
- multiple inheritance is not allowed in Java

superclass

```java
package training;

/**   A simple tree
 */
public class Tree {
  // diameter at breast height, cm
  private double dbh;

  public Tree () {}

  public void setDbh (double d) {
    dbh = d;
  }

  public double getDbh () {
    return dbh;
  }

}
```

Tree.java file

subclass

```java
package training;

/**   A tree with coordinates
 */
public class SpatializedTree extends Tree {
  // x, y of the base of the trunk (m)
  private double x;
  private double y;

  /** Default constructor
   */
  public SpatializedTree () {
    super ();
    setXY (0, 0);
  }

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

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

}
```

inheritance keyword

calls constructor of the superclass

new methods

SpatializedTree.java file

```java
// SpatializedTree
SpatializedTree t3 = new SpatializedTree ();

t3.setDbh (15.5);
t3.setXY (1, 5);

double d = t3.getDbh ();  // 15.5
double x = t3.getX ();    // 1
```

inherited methods

Training.java file

Specific references

A keyword for the reference to the current class: this

- to remove ambiguities

A keyword for the reference to the superclass: super

```
package training;

/**   A tree with coordinates
 */
public class SpatializedTree extends Tree {
  // x, y of the base of the trunk (m)
  private double x;
  private double y;

  /** Default constructor
   */
  public SpatializedTree () {
    super ();
    setXY (0, 0);
  }

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

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

}
```

call to the constructor of the superclass

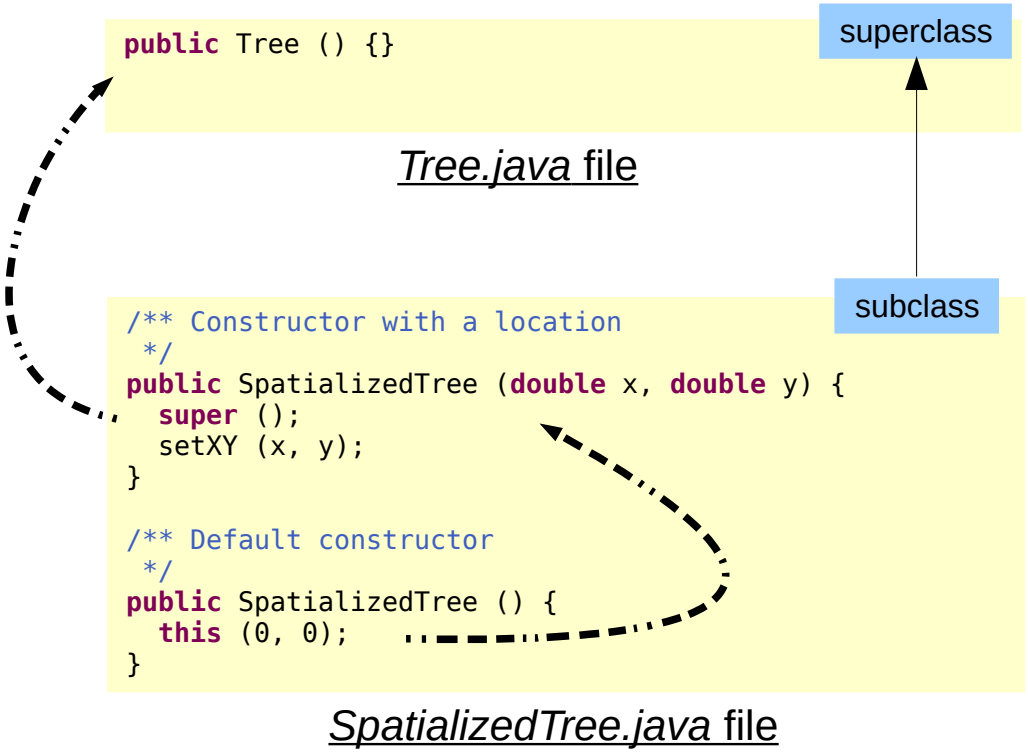
instance variable: this.x

a parameter

no ambiguity here

Constructors chaining

Chain the constructors to avoid duplication of code




```java
public Tree () {}
```

Tree.java file

```java
/** Constructor with a location
 */
public SpatializedTree (double x, double y) {
  super ();
  setXY (x, y);
}

/** Default constructor
 */
public SpatializedTree () {
  this (0, 0);
}
```

SpatializedTree.java file

```java
new Tree ();
// calls Tree ()

new SpatializedTree (1, 5);
// calls SpatializedTree (x, y)
// calls Tree ()

new SpatializedTree ();
// calls SpatializedTree ()
// calls SpatializedTree (x, y)
// calls Tree ()
```

Training.java file

Method overloading / overriding

Overload (“surcharge”)
- in the same class
- several methods with same name
and
- different types of parameters and/or
  a different number of parameters

BiomassCalculator

```
public double calculateBiomass (Tree t) {
  return t.getTrunkBiomass ();
}

public double calculateBiomass (TreeWithCrown t) {
  return t.getTrunkBiomass () + t.getCrownBiomass ();
}
```

Override (“redéfinition”)
- in a class and a subclass
- several methods with:
  - same signature i.e. same name and same types of parameters in the same order
  - and
  - same type of return value (or a derivated type since JDK 5.0)

superclass

```
public double getVolume () {
  return trunkVolume;
}
```

subclass

```
@Override
public double getVolume () {
  return trunkVolume + crownVolume;
}
```

e.g. if TreeWithCrown **extends** Tree

optional:
tell the compiler
--> it will check

Static variable and method

A common variable shared by all the instances of a class

- can be a constant: ‘Math.PI‘

```
public static final double PI = 3.14...;
```

- can be a variable

```
public static int counter;
```

e.g. ‘counter’ can be incremented each time the class is instancied

A method at the class level: no access to the instance variables

- no need to instanciate a class, example: the methods of the ‘Math‘ class like ‘Math.sqrt(double a)’
- a utility method: to reuse a block of code
- uses only its parameters (and not the instance variables)

example: in class **Tree**

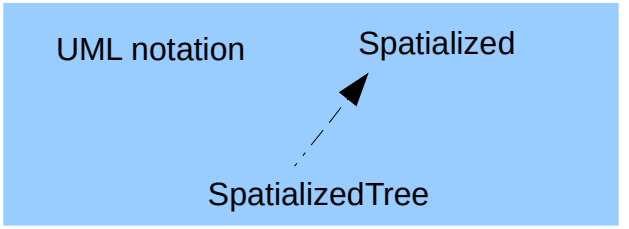
```
/**
 * Quadratic diameter
 */
public static double calculate_dg (double basalArea, int numberOfTrees) {
    return Math.sqrt (basalArea / numberOfTrees * 40000d / Math.PI);
}
```

- ‘basalArea’ and ‘numberOfTrees’ are the parameters
- their names have a local scope: they are only available in the method

```
double dg = Tree.calculate_dg (23.7, 1250);
```

ClassName.method (parameters)

Interface



A particular kind of class

- a list of methods without a body
- a way to **make sure** a class implements a set of methods
- a kind of **contract**
- classes extend other classes
- classes **implement** interfaces
- implementing several interfaces is possible

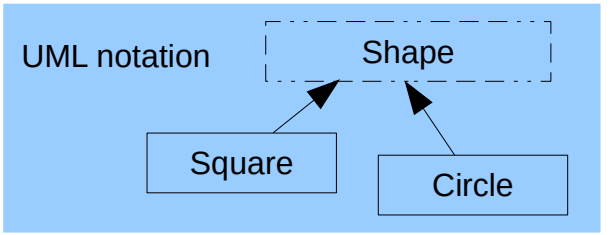
```
public interface Spatialized {

  public void setXYZ (double x, double y, double z);
  public double getX ();
  public double getY ();
  public double getZ ();

}
```

no method body in the interface

```
/**   A tree with coordinates
 */
public class SpatializedTree extends Tree implements Spatialized {
  ...

  public void setXYZ (double x, double y, double z) {
    this.x = x;
    this.y = y;
    this.z = z;
  }

  public double getX () {return x;}
  public double getY () {return y;}
  public double getZ () {return z;}

}
```

an **implementation is required** for the methods in the class or the subclasses

Abstract class



An incomplete superclass with common methods

- class 'template' containing **abstract methods** to be implemented in all subclasses (contains at least one abstract method)
- can also have regular methods (unlike an interface)
- each subclass implements the abstract methods
- can not be instanciated directly

Shape.java file

```
public abstract class Shape {
    private String name;
    ...
    public String getName () {return name;}
    public abstract double area ();  // m2

}
```

an **abstract class** (at least one abstract method): can not be instanciated

a regular method

an **abstract method**: no body

```
public class Square extends Shape {
    private double width;  // m
    ...
    @Override
    public double area () {
        return width * width;
    }
}
```

Square.java file

two subclasses:
they **implement** the abstract method

Circle.java file

```
public class Circle extends Shape {
    private double radius;  // m
    ...
    @Override
    public double area () {
        return Math.PI * radius * radius;
    }
}
```

```
// Example
Shape sh = new Shape (); // ** Compilation error

Square s = new Square ("square 1", 10);

Circle c = new Circle ("circle 1", 3);

String name1 = s.getName (); // square 1

double a1 = s.area (); // 100
double a2 = c.area (); // 28.27
```

Training.java file

Polymorphism

Write generic code to be executed with several types

- more abstract and general implementations

```
public abstract class Shape {

    public abstract double area ();  // m2

}
```

Shape.java file

```
public class Square extends Shape {
    private double width;  // m
    ...
    @Override
    public double area () {
        return width * width;
    }
}
```

Square.java file

```
public class Circle extends Shape {
    private double radius;  // m
    ...
    @Override
    public double area () {
        return Math.PI * radius * radius;
    }
}
```

Circle.java file

```
private float totalArea (Shape[] a) {
  double sum = 0;
  for (int i = 0; i < a.length; i++) {
    // the program knows what method to call
    sum += a[i].area ();
  }
  return sum;
}
```

this code is generic works with all shapes

Training.java file

several classes, all Shapes

Example of use

```
Shape[] a = {new Square (5), new Circle (3), new Square (10)};

float total = totalArea (a);
```

Training.java file

The 'Object' superclass

UML notation

If no 'extends' keyword...
… then the class extends Object
→ All classes extend Object

Tree.java file

extends Object

```
package training;

/**   A simple tree
 */
public class Tree {
  // diameter at breast height, cm
  private double dbh;

  public Tree () {}

  public void setDbh (double d) {
    dbh = d;
  }

  public double getDbh () {
    return dbh;
  }

  @Override
  public String toString () {
    return “Tree dbh: “ + dbh;
  }

}
```

note: native methods have a body in native language (e.g. C)
-> they are not abstract

Object.java file

a superclass for all classes

```
package java.lang;

public class Object {

  public final native Class<?> getClass();

  public native int hashCode();

  public boolean equals(Object obj) {
    return (this == obj);
  }

  protected native Object clone() throws
      CloneNotSupportedException;

  public String toString() {
    return getClass().getName() + "@" +
        Integer.toHexString(hashCode());
  }

  (...)

}
```

all these methods can be called on all objects

Training.java file

```
// Tree
Tree t = new Tree ();
t.setDbh (14.5);
System.out.println (“” + t);
```

appended to a String: i.e. t.toString ()

toString () can be overriden for a better result

```
training.Tree@37dd7056
```

```
Tree dbh: 14.5
```

Enum

Another particular kind of class: a type for enumerations

- an enum is a type with a limited number of values

Declaration

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

An example of use

```
private Day day;
...

day = Day.SUNDAY;
...
```

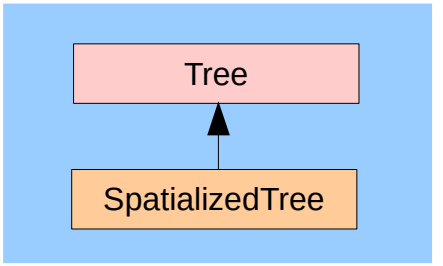
Cast

Cast of numbers

```
double d = 12.3;
int i = (int) d; // 12
```

In an inheritance graph

- a reference can have any supertype of the real type




```
Tree t = new SpatializedTree ();
```

type of the reference

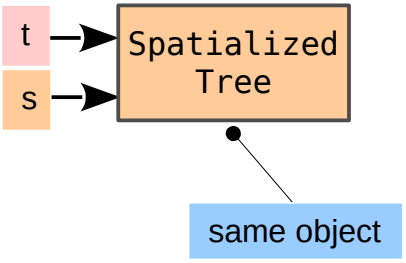
real type of the object

- we can only use the methods the reference knows

```
t.setDbh (10);    // ok
t.setXY (2, 10);  // ** compilation error: Tree does not define setXY ()
```

- to access the methods of the real type, we can create another reference

```
SpatializedTree s = (SpatializedTree) t;  // cast: creates another reference
s.setXY (2, 1);  // ok: SpatializedTree does define setXY ()
```



- example of use (with the 'instanceof' operator)

contains spatialized and non-spatialized trees

instanceof operator: checks the type of an object

calculates the rectangle enclosing the spatialized trees

```
List trees = forest.getTrees();

for (Object o : trees) {
  if (o instanceof SpatializedTree) {
    SpatializedTree s = (SpatializedTree) o;
    updateRectangle(s.getX(), s.getY());
  }
}
```

Packages and import

Packages

- namespaces to organize the developments: groups of related classes
- first statement in the class (all lowercase)
- match directories with the same names

e.g.

- **java.lang**: String, Math and other basic Java classes
- **java.util**: List, Set... (see below)
- **training**: Tree and SpatializedTree

The package is part of the class name: **java.lang.String**, **training.Tree**

```
package training;
```
first instruction

Import

- to simplify notation, import classes and packages
- instead of:

```
training.Tree t = new training.Tree ();
```

write:

```
import training.Tree;
...
Tree t = new Tree ();
```

Lifetime of variables

Lifetime of a variable: defined by the **scope** delimited by **{...}** in which the variable has been defined

- **instance variable** of a class: as long as the **object** it belongs is referenced (*lifetime = lifetime of the object*)

```java
package training;

/**   A simple tree
 */
public class Tree {
  // diameter at breast height, cm
  private double dbh;

  public Tree () {}

  public void setDbh (double d) {
    dbh = d;
  }

  public double getDbh () {
    return dbh;
  }

  public double getDbhSquared () {
    double res = dbh*dbh;
    return res;
  }
}
```

Tree.java file

scope = the body of the class

```java
// Before instanciation of
// Tree class

// Create a tree
Tree t1 = new Tree ();

// Set its diameter
t1.setDbh (12.5);

// t1 is no more referenced
t1 = null;
```

Training.java file

dbh does not exist

an object of type Tree is created and its reference is placed in t1: **dbh** (of t1) exists and is initialized to 0.0 (default value)

dbh has value 12.5

the created Tree is no more referenced: it becomes candidate to the garbage collector and **dbh does not exist anymore**

Lifetime of variables

Lifetime of a variable: defined by the **scope** delimited by **{…}** in which the variable has been defined

- **argument (parameter)** and **local variable** of a **method**: *exist only inside the method*

scope = the body of the method

```
package training;

/**   A simple tree
 */
public class Tree {
  // diameter at breast height, cm
  private double dbh;

  public Tree () {}

  public void setDbh (double d) {
    dbh = d;
  }

  public double getDbh () {
    return dbh;
  }

  public double getDbhSquared () {
    double res = dbh*dbh;
    return res;
  }
}
```

Tree.java file

scope = the body of the method

d is an argument of the setDbh() method: **d** exists only **inside** this method

```
// Create a tree
Tree t1 = new Tree ();

// Set its diameter
t1.setDbh (12.5);

// Call getDbhSquared() method
double dbhSquared = t1.getDbhSquared ();
```

Training.java file

d does not exist

d exists inside the method

d does not exist anymore

res does not exist

res exists inside the method

res does not exist anymore

res is declared in the getDbhSquared() method: **res** exists only **inside** this method

Lifetime of variables

Lifetime of a variable: defined by the **scope** delimited by **{...}** in which the variable has been defined

- **index** of a **loop**: *exists inside the loop (at least...)*

```
// With an array
int[] array = new int[12];
int sum = 0 ;
for (int i = 0; i < array.length; i++) {
  array[i] = i;
  sum += array[i];
}
```

i does not exist

i is created

i exists inside the loop

i does not exist anymore

scope = the body of the loop

lifetime of i = exists only inside the loop

sum has the same value with _i_ declared before the loop:

```
// With an array
int[] array = new int[12];
int sum = 0 ;
int i;
for (i = 0; i < array.length; i++) {
  array[i] = i;
  sum += array[i];
}
```

i is created

i exists inside the loop

i still exists and its value is 12

scope = the body of the loop + ...

lifetime of i = from its declaration + inside the loop + after the loop

Lifetime of variables

Lifetime of a variable: defined by the **scope** delimited by **{...}** in which the variable has been defined

- **local variable** of a **loop**: *exists only inside the loop*

```
// With an array
int[] array = new int[12];
int sum = 0 ;
for (int i = 0; i < array.length; i++) {
  int j = i+2;
  array[i] = i;
  sum += array[i];
}
```

j does not exist

j is created

j does not exist anymore

scope = the body of the loop

Names of variables

Use **explicit names** for:

- **instance variables**

```
package training;

/**   A simple tree
 */
public class Tree {
  // diameter at breast height, cm
  private double dbh;
  private int age;
  private double height;
  private String speciesName;

  public Tree () {}
}
```

- **local variables** having a **long range**

```
package training;

/**   A simple tree
 */
public class Tree {
  // diameter at breast height, cm
  private double dbh;

  public Tree () {}

  public void makeCalculations () {
     int anExplicitName;
     ...
     // some long calculations...
     anExplicitName = ...
     ...
     anExplicitName = ...
  }

}
```

Short names are authorized for variables having a **short range**:

```
// With an array
int[] array = new int[12];
int sum = 0 ;
for (int i = 0; i < array.length; i++) {
  array[i] = i;
  sum += array[i];
}
```

only 3 lines

Java reserved keywords

abstract
boolean
break
byte
case
cast
catch
char
class
const
continue
default
do
double
else
enum
extends
false
final
finally
float
for
goto (unused)
if
implements
import
instanceof
int
interface
long
native
new
null
package
private
protected
public
return
short
static
super
switch
synchronized
this
throw
throws
transient
true
try
void
volatile
while

Java modifiers

a final field cannot be changed e.g. Math.PI

a final class can not be subclassed e.g. java.lang.String

a final method can not be overriden

	class	interface	field	method	initializer	variable
abstract	x	x		x		
final	x		x	x		x
native				x		
none (package)	x	x	x	x		
private			x	x		
protected			x	x		
public	x	x	x	x		
static	x		x	x	x	
synchronized				x		
transient			x			
volatile			x			

Resources

- a focus on the collection framework
- the Collection interface
- ArrayList
- HashSet
- Map
- the tools in the Collections class
- how to iterate on objects in collections
- how to iterate on objects in maps
- collections and generics
- online documentation
- online documentation: javadoc
- online documentation: tutorials
- links to go further

A focus on the collection framework

A collection is like an array, but without a size limitation (size can vary during execution)

- contains references

- may have distinctive features
  - a **list** keeps insertion order
  - a **set** contains no duplicates and has no order

- the 8 simple types (int, double, boolean...) are not objects → need a **wrapper object**
  Byte, Short, Integer, Long, Float, Double, Character, Boolean
  Java helps: **Integer i = 12;** (autoboxing / unboxing)

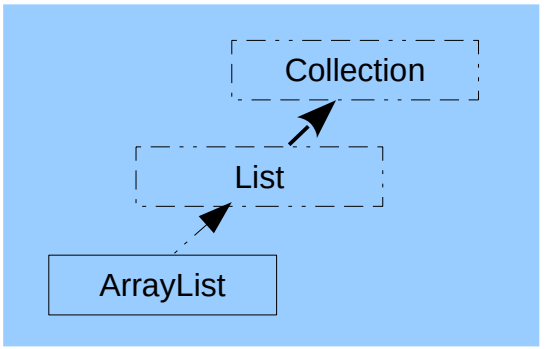
- all collections implement **the Collection interface**

The Collection interface

Implemented by all collections

```
public boolean add (Object o);        // adds o
public boolean remove (Object o);     // removes o

public void clear ();                 // removes all objects
public boolean isEmpty ();            // true if the collection is empty

public int size ();                   // number of objects in the collection
public boolean contains (Object o);   // true if o is in the collection
...
```

ArrayList



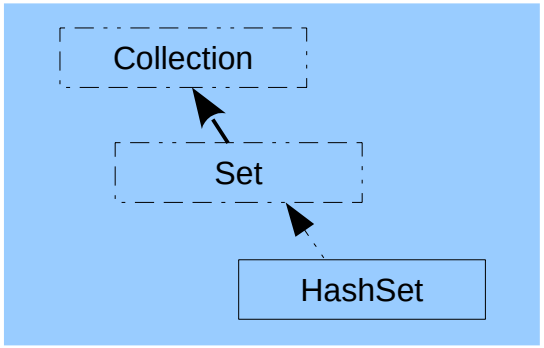
ArrayList

- implements the **List** interface
- keeps insertion order
- accepts duplicates
- specific methods added

```
public void add (int index, Object o);    // adds o at the given index (shifts subsequent elements)
public Object get (int index);            // returns the object at the given index
public int indexOf (Object o);            // returns the index of o
public Object remove (int index);         // removes the object at the given index
...
```

```
List l = new ArrayList ();
l.add ("Robert");  // add () comes from Collection
l.add ("Brad");
l.add ("Robert");

int n = l.size (); // 3
String s = (String) l.get (0);  // "Robert"
```

HashSet

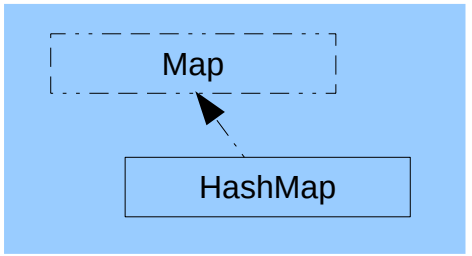


HashSet

- implements the **Set** interface
- does **not** keep insertion order
- does **not** accept duplicates

```
Set s = new HashSet ();

s.add ("one");
s.add ("two");
s.add ("one"); // duplicate, ignored

int n = s.size ();  // 2

if (s.contains ("one"))...    // true
if (s.contains ("three"))...  // false
```

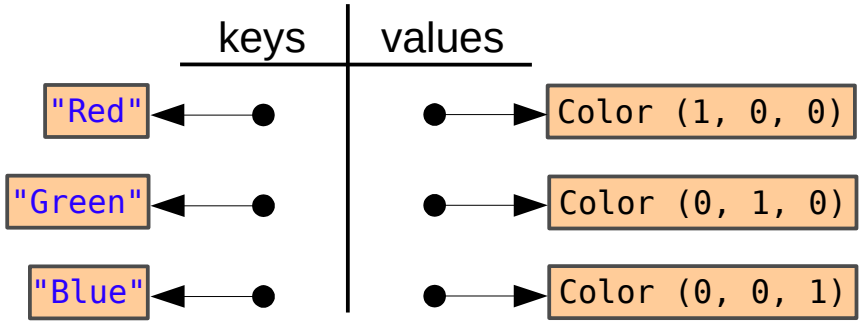
Maps



A Map associates a key with a value

- the common Map implementation is **HashMap**
- keys must be unique (like in a Set)
- keys and values are references

```
Map m = new HashMap ();

m.put ("Red", new Color (1, 0, 0));
m.put ("Green", new Color (0, 1, 0));
m.put ("Blue", new Color (0, 0, 1));

Color c = (Color) m.get ("Red");  // returns a color object

if (m.containsKey ("Blue"))...  // true

Set s = m.keySet ();  // set of keys: Red, Green, Blue
```



The tools in the Collections class

Tools for the collections are proposed in a class: Collections

```
public static final List EMPTY_LIST
public static final Set EMPTY_SET
public static final Map EMPTY_MAP

public static void sort(List list)
public static void sort(List list, Comparator c)

public static void shuffle(List list)
public static void reverse(List list)

public static Object min(Collection coll)
public static Object max(Collection coll)
```

empty collections and maps

sorting

changing elements order

```
// Random order
Collections.shuffle (list);
```

How to iterate on objects in collections

Two syntaxes to loop on a list

```
// List of Tree
List l = new ArrayList ();
l.add (new Tree (5.5));
l.add (new Tree (2.3));
l.add (new Tree (4.1));
...

// Loop with an Iterator
for (Iterator i = l.iterator (); i.hasNext ();) {
 Tree t = (Tree) i.next ();

  if (t.getDbh () < 3) {i.remove ();}

}
```

constructor takes a dbh

an Iterator + a cast

the iterator can remove the current element from the list

a cast is needed at iteration time

```
// Loop with a foreach
for (Object o : l) {
 Tree t = (Tree) o;

  t.setDbh (t.getDbh () * 1.1);
}
```

How to iterate on objects in maps

```
Map m = new HashMap ();
m.put ("Red", new Color (1, 0, 0));
m.put ("Green", new Color (0, 1, 0));
m.put ("Blue", new Color (0, 0, 1));

for (Object o : m.keySet ()) {
  String key = (String) o;
  //...
}

for (Object o : m.values ()) {
  Color value = (Color) o;
  //...
}
```

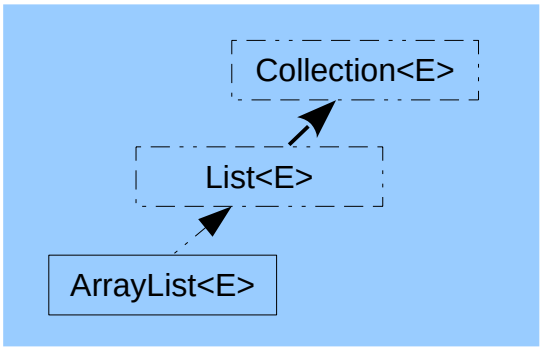
iterate on keys

iterate on values

Collections and generics

Collections are manipulated by generic classes that implement **Collection<E>**

E represents the type of the elements of the collection




```
// List of Tree
List<Tree> l = new ArrayList<Tree> ();
l.add (new Tree (1.1));
l.add (new Tree (2.5));
l.add (new Tree (3.4));

// Simplified foreach, no cast needed
for (Tree t : l) {

  t.setDbh (t.getDbh () * 1.1);

}
...

// Print the result
for (Tree t : l) {
  System.out.println ("Tree dbh: " + t.getDbh ());
}
```

longer: specify type

shorter: no cast

Online documentation

http://download.oracle.com/javase/8/docs/

Java Platform Standard Edition 8 Documentation

What's New
- Documentation Updates
- JDK 8 Release Notes

Tutorials and Training
- The Java Tutorials
- Java Training

More Information
- Java SE 8 Names and Version Numbers
- Java SE White Papers
- Documentation Accessibility
- Specifications
- Installation Instructions
- Certified System Configurations
- Compatibility Guide
- Known Issues
- JDK 8 Adoption Guide
- Troubleshooting Guide
- About Test / Sample Applications and Code

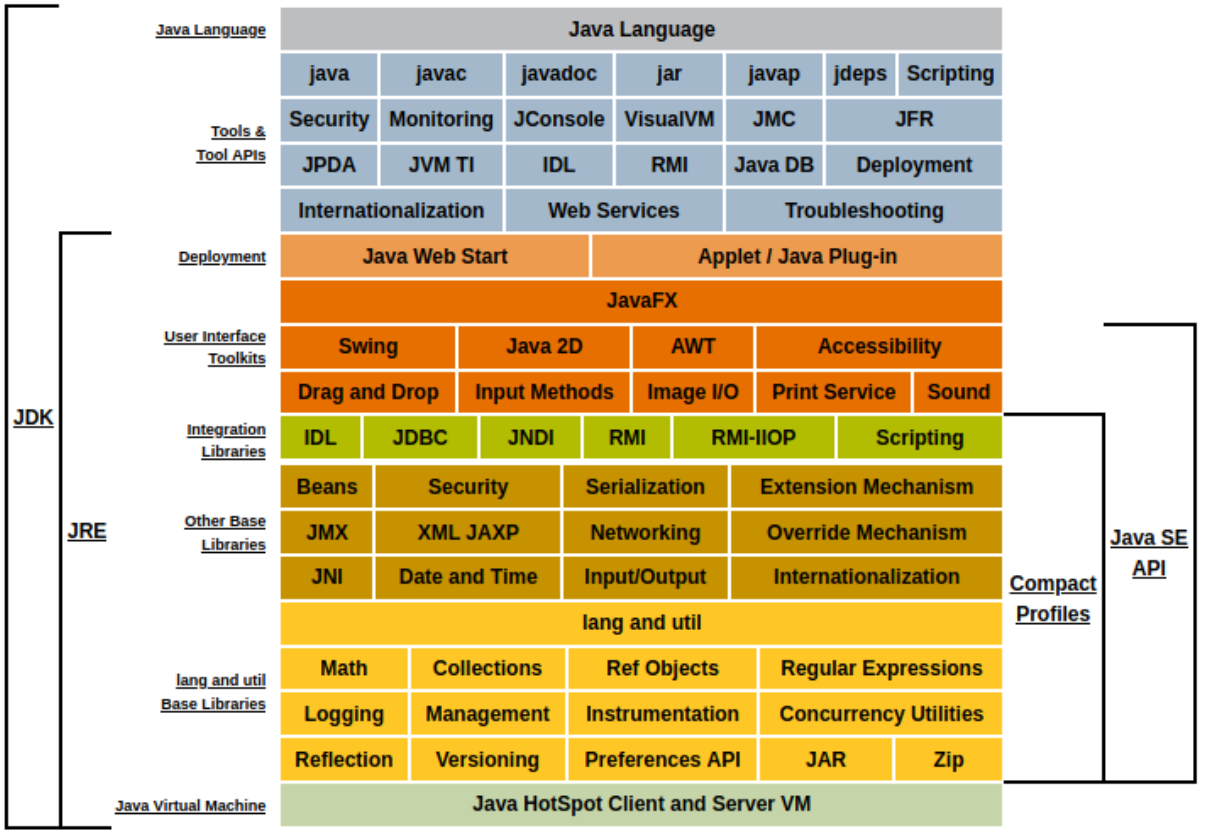
Downloads
- Java SE Downloads
- JDK 8 Documentation

Oracle has two products that implement Java Platform Standard Edition (Java SE) 8: Java SE Development Kit (JDK) 8 and Java SE Runtime Environment (JRE) 8.

JDK 8 is a superset of JRE 8, and contains everything that is in JRE 8, plus tools such as the compilers and debuggers necessary for developing applets and applications. JRE 8 provides the libraries, the Java Virtual Machine (JVM), and other components to run applets and applications written in the Java programming language. Note that the JRE includes components not required by the Java SE specification, including both standard and non-standard Java components.

The following conceptual diagram illustrates the components of Oracle's Java SE products:

Description of Java Conceptual Diagram



Online documentation: javadoc

http://download.oracle.com/javase/8/docs/api/

Java™ Platform Standard Ed. 8

All Classes All Profiles

Packages

java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event
java.awt.font
java.awt.geom
java.awt.im
java.awt.im.spi
java.awt.image
java.awt.image.renderable

NullPointerException
NullType
Number
NumberFormat
NumberFormat.Field
NumberFormatException
NumberFormatProvider
NumberFormatter
NumberOfDocuments
NumberOfInterveningJobs
NumberUp
NumberUpSupported
NumericShaper
NumericShaper.Range
NVList
OAEPParameterSpec
OBJ_ADAPTER
ObjDoubleConsumer
Object
Object
OBJECT_NOT_EXIST
ObjectAlreadyActive
ObjectAlreadyActiveHelper
ObjectChangeListener
ObjectFactory
ObjectFactoryBuilder
ObjectHelper
ObjectHolder
ObjectIdHelper
ObjectIdHelper
ObjectImpl
ObjectImpl
ObjectInput
ObjectInputStream
ObjectInputStream.GetField
ObjectInputValidation
ObjectInstance
ObjectName
ObjectNotActive
ObjectNotActiveHelper

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP — Java™ Platform Standard Ed. 8

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3
java.lang

Class Object

java.lang.Object

```
public class Object
```

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

Since:

`JDK1.0`

See Also:

`Class`

Constructor Summary

Constructors

Constructor and Description
`Object()`

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`protected Object`	`clone()` Creates and returns a copy of this object.
`boolean`	`equals(Object obj)` Indicates whether some other object is "equal to" this one.
`protected void`	`finalize()` Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
`Class<?>`	`getClass()` Returns the runtime class of this `Object`.
`int`	`hashCode()` Returns a hash code value for the object.
`void`	`notify()` Wakes up a single thread that is waiting on this object's monitor.

Online documentation: tutorials

http://docs.oracle.com/javase/tutorial/

The Java™ Tutorials

Search the online Java Tutor

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

The Java Tutorials are practical guides for programmers who want to use the Java programming language to create applications. They include hundreds of complete, working examples, and dozens of lessons. Groups of related lessons are organized into "trails".

Trails Covering the Basics

These trails are available in book form as *The Java Tutorial, Sixth Edition*. To buy this book, refer to the box to the right.

- Getting Started — An introduction to Java technology and lessons on installing Java development software and using it to create a simple program.
- Learning the Java Language — Lessons describing the essential concepts and features of the Java Programming Language.
- Essential Java Classes — Lessons on exceptions, basic input/output, concurrency, regular expressions, and the platform environment.
- Collections — Lessons on using and extending the Java Collections Framework.
- Date-Time APIs — How to use the `java.time` pages to write date and time code.
- Deployment — How to package applications and applets using JAR files, and deploy them using Java Web Start and Java Plug-in.
- Preparation for Java Programming Language Certification — List of available training and tutorial resources.

Creating Graphical User Interfaces

- Creating a GUI with Swing — A comprehensive introduction to GUI creation on the Java platform.
- Creating a JavaFX GUI — A collection of JavaFX tutorials.

Specialized Trails and Lessons

These trails and lessons are only available as web pages.

- Custom Networking — An introduction to the Java platform's powerful networking features.
- The Extension Mechanism — How to make custom APIs available to all applications running on the Java platform.
- Full-Screen Exclusive Mode API — How to write applications that more fully utilize the user's graphics hardware.
- Generics — An enhancement to the type system that supports operations on objects of various types while providing compile-time type safety. Note that this lesson is for advanced users. The Java Language trail contains a Generics lesson that is suitable for beginners.
- Internationalization — An introduction to designing software so that it can be easily adapted (localized) to various languages and regions.
- JavaBeans — The Java platform's component technology.
- JDBC Database Access — Introduces an API for connectivity between the Java applications and a wide range of databases and data sources.
- JMX— Java Management Extensions provides a standard way of managing resources such as applications, devices, and services.
- JNDI— Java Naming and Directory Interface enables accessing the Naming and Directory Service such as DNS and LDAP.
- JAXP — Introduces the Java API for XML Processing (JAXP) technology.
- JAXB — Introduces the Java architecture for XML Binding (JAXB) technology.
- RMI — The Remote Method Invocation API allows an object to invoke methods of an object running on another Java Virtual Machine.
- Reflection — An API that represents ("reflects") the classes, interfaces, and objects in the current Java Virtual Machine.
- Security — Java platform features that help protect applications from malicious software.
- Sound — An API for playing sound data from applications.
- 2D Graphics — How to display and print 2D graphics in applications.
- Sockets Direct Protocol — How to enable the Sockets Direct Protocol to take advantage of InfiniBand.

Links to go further

Oracle and Sun's tutorials
http://docs.oracle.com/javase/tutorial/
see the 'Getting Started' section

Learning the Java language
http://docs.oracle.com/javase/tutorial/java/index.html
https://openclassrooms.com/fr/courses/26832-apprenez-a-programmer-en-java

Coding conventions
http://www.oracle.com/technetwork/java/codeconvtoc-136057.html

Resources on the Capsis web site
http://capsis.cirad.fr

Learn to program in Java: Openclassroom web site
https://openclassrooms.com/fr/courses/26832-apprenez-a-programmer-en-java

Millions of **books**… including these references
"Java In A Nutshell", David Flanagan - O'Reilly (several editions)
"Programmer en Java", Claude Delannoy - Eyrolles