Introduction to Java programming

Lecture 1

Paolo Ballarini, Celine Hudelot

CentraleSupélec

Aim of this course

- introducing the basics of the Java programming language
  - learning the basics of object oriented programming paradigm
- acquiring basic skills on developing Java programs within an Integrated Development Environment (IDE), like Eclipse

Required tools

You must install the following software tools on your computer:

- Java Standard Edition Development Kit (SE JDK) version 8:

  http://www.oracle.com/technetwork/pt/java/javase/downloads/jdk8-downloads-2133151.html

- Eclipse IDE:

  https://www.eclipse.org/ide/

- material of this course (slides, tutorials, homeworks): https://sites.google.com/site/pballarini/teaching/introjavabigdata

Outline

1. Programming languages
2. Java basics
3. Inheritance
4. Abstract classes, Interfaces
5. Collections
6. Summing up

Programming languages

Set of keywords and rules (i.e. syntax) to form sentences that can be transformed into an executable program (binary language).

Programming languages at different levels

- Low-level languages: elementary instructions very *close to the machine*
  - Assembler: V AX, [0110] (means to *copy the contents of 0110 in the AX register*)
  - machine language: instructions are binary sequences (e.g. 0110010, 1100100)
- High-level Languages: more abstract instructions
  - C, C++, Perl, Java, Python...

Programming paradigms: procedural programming

Procedural programming

- a program consists of a list of procedures (also named functions)
- each procedure consists of a sequence of computational steps (i.e. instructions)
- the basic (atomic) instruction is the variable assignment
- a program state corresponds to the values of its variables
- programs are executed
- execution of a procedure may depend on the current state of the program

Examples

FORTRAN, C, PASCAL

Procedural programming: a small example

Using the procedural kernel of Python:

```
def sum(l):
    s=0
    while l != []:
        s= s+ l[0]
        l = l[1:]
    return s
```

When using the ”sum” procedure :

```
>>> sum([4,5,6])
15
```

Programming paradigms: functional programming

Functional programming

- computations are treated as the evaluation of a mathematical function
- function evaluation avoids changing data (i.e. the state of the program)
- a program is built through functions yielding a result as output and taking in input the outputs of other functions.
- programs are evaluated
- recursion

Examples

CAML, SCHEME, LISP, ...

Functional programming: a small example

Using OCAML:

```
let rec sum l = match l with
  [] -> 0
  | x :: l’ -> x + sum(l’) ;;
```

When using the ”sum” function :

```
# sum([4;5;6]) ;;
- : int = 15
```

Programming paradigms: Object-oriented programming

OO-programming (OOP) lays on top of other programming paradigms, mainly for the purpose of variable encapsulation

Object-oriented programming

- In OOP we design frames (*classes*) that are used to build components (*objects*).
- Classes contain data (*attributes*) and actions (*methods*).
- Based on the principle that things (objects) have common points, similarities by themselves or how they act.
- OOP features: *abstraction*, *encapsulation*, *modularity*, *polymorphism*, *inheritance*

Examples

SMALLTALK, ...

Programming paradigms

Hybrid programming

Languages combining several programming paradigms.

Examples

C++, JAVA, PYTHON, PERL, RUBY, OCAML...

Programs execution

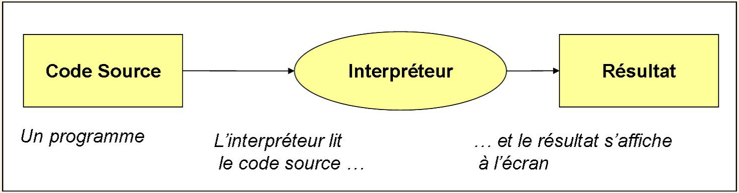
Execution of high-level programs

- Computers can only process machine code (i.e. binary coded instructions)
- Programming paradigms (procedural, functional, logic, OO,....) result in high-level languages with complex instructions expressed in natural language (i.e. english)
- How can we get a computer to execute programs written in high-level languages (e.g. C, C++, Java...) ?

Solutions

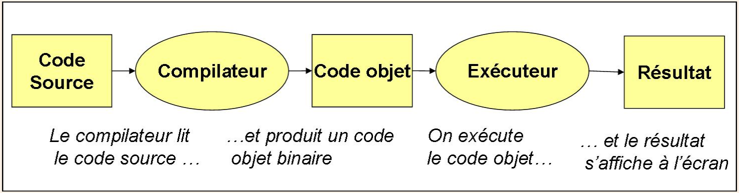
- programs interpretation
- programs compilation
- mixed techniques (virtual machines)

Programs execution: Interpretation


Code Source
Interpréteur
Résultat
Un programme
L'interpréteur lit le code source ...
... et le résultat s'affiche à l'écran

- An interpreter is a program that reads instructions of an high-level language and execute them on the computer (on-the-fly translation of the source code toward machine language)
- Instructions of the source-code are interpreted one-by-one as we go along the source-code lines
- Pros: flexibility of development (modifications to source code can be tested straight away)
- Cons: interpretation is normally not the fastest solution

Programs execution: Compilation


Code Source
Compilateur
Code objet
Exécuteur
Résultat
Le compilateur lit le code source ...
...et produit un code objet binaire
On exécute le code objet...
... et le résultat s'affiche à l'écran

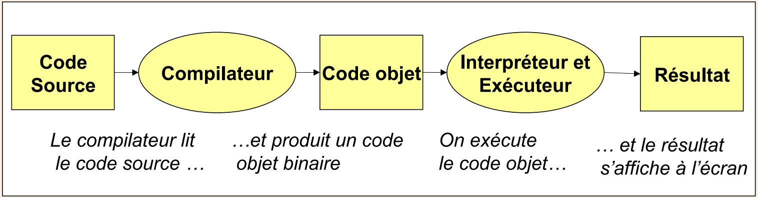
- The source program is translated into object code (i.e. machine code) by means of a compiler (lexical analysis, syntactic, semantic and object code generation)
- Pros: compiled code executes faster than interpreted one
- Cons: any change to the source code requires a new compilation of the entire program

Mixed approach: Interpretation of compiled bytecode

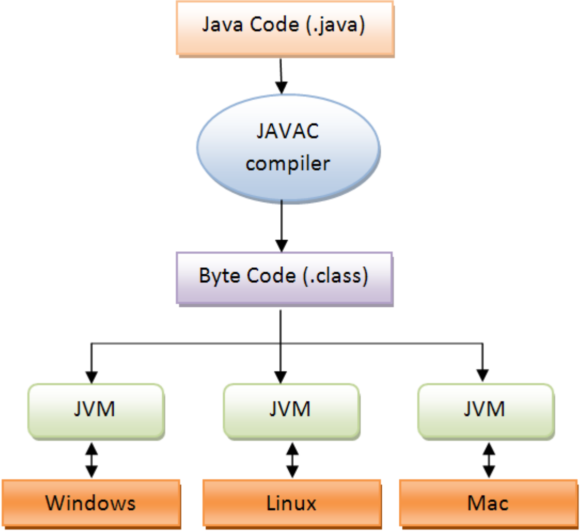
Due to interoperability issues (in particular with WEB technologies)

- Source-code is compiled into an intermediate code called bytecode
- Bytecode is interpreted through a so-called Virtual Machine (VM)
- Pros: good compromise between ease of development and speed of execution
- Pros: bytecode (intermediate code) is portable to any computer with a VM (only the VM must be re-implemented for different computers)

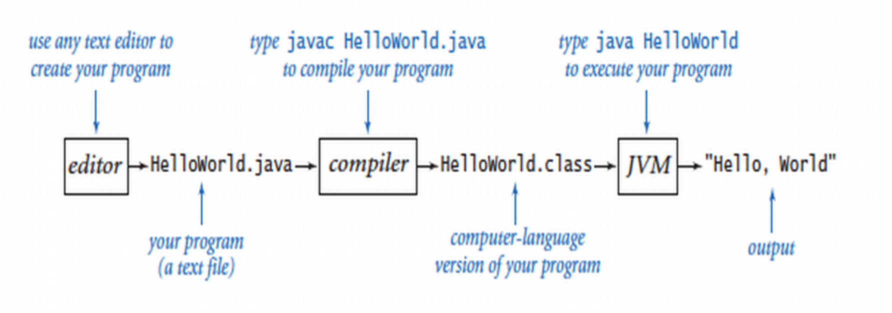
Java : typical example


Code Source
Compilateur
Code objet
Interpréteur et Exécuteur
Résultat
Le compilateur lit le code source …
…et produit un code objet binaire
On exécute le code objet…
… et le résultat s'affiche à l'écran

Java program execution



Developing a java program


use any text editor to create your program
type javac HelloWorld.java to compile your program
type java HelloWorld to execute your program
editor
HelloWorld.java
compiler
HelloWorld.class
JVM
"Hello, World"
your program (a text file)
computer-language version of your program
output

Developing a Java program

Java platform : overview

- Java language.
- Java compiler : checks your jave code against its syntax rules, then writes out bytecode in .class files.
- JVM : virtual machine that runs the Java bytecodes.
- Garbage Collector : implicit memory management by the JVM.
- Java Development Kit : tools such as compilers needed to develop the Java programs.
- Java Runtime Environment :the JVM, code libraries, and components that are necessary for running programs that are written in the Java language.
- Java IDE : Eclipse, Netbeans...

Setting up your Java development environment

Install the JDK

1. Browse to Java SE Downloads and click the Java Platform (JDK) box to display the download page for the latest version of the JDK.
2. Agree to the license terms for the version you want to download.
3. Choose the download that matches your operating system and chip architecture.

Setting up your Java development environment

Install and set up an IDE, e.g. Eclipse

1. Launch Eclipse from your local hard disk.
2. Choose the default workspace.
3. Close the Welcome to Eclipse window.
4. Select Preferences ¿ Java ¿ Installed JREs.

Eclipse IDE

Four main components

- Workspace : A folder where Eclipse stores yours progams and your projects (you can have different workspaces)
  - Good Practice : arrange your programs in different workspaces
- Projects
- Perspectives : way of looking at each project (debug, java, papyrus...)
- Views

Outline of this lecture

1. Programming languages
2. Java basics
   - Object Oriented Programming
   - Class declaration
   - Constructors and objects creation
   - Access modifiers: instance v class variables/methods
   - Encapsulation: private attributes, getters/setters
   - Methods overloading
   - Primitive types, reference types
   - Arrays and String
   - Control flow instructions
3. Inheritance
   - Overriding
4. Abstract classes, Interfaces
5. Collections
6. Summing up

Outline

1. Programming languages

2. Java basics
   - Object Oriented Programming
   - Class declaration
   - Constructors and objects creation
   - Access modifiers: instance v class variables/methods
   - Encapsulation: private attributes, getters/setters
   - Methods overloading
   - Primitive types, reference types
   - Arrays and String
   - Control flow instructions

3. Inheritance

4. Abstract classes, Interfaces

Object-oriented programming

OOP basic idea: a programming paradigm which allows to directly map real-life problems into a program

- it is based on the notion of obejct

an object is a data structure that contains:

- data: in form of variables called attributes or fields
- behaviour: in form of procedures called methods

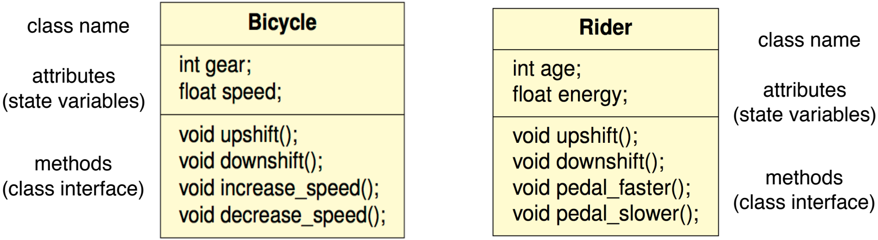
Real-world objects

real-world objects share two characteristics: they all have a state and a behaviour

examples of real-world objects

- Dog:
  - state: name, color, breed, hungry, ...
  - behaviour: barking, fetching, wagging tail, eating, ...
- Bicycle:
  - state: current gear, current pedal cadence, current speed, ...
  - behaviour: changing gear, changing pedal cadence, applying brakes, ...

Example: class “Bicycle” and class “Rider”



class name

attributes
(state variables)

methods
(class interface)

Bicycle

int gear;
float speed;

void upshift();
void downshift();
void increase_speed();
void decrease_speed();

Rider

int age;
float energy;

void upshift();
void downshift();
void pedal_faster();
void pedal_slower();

class name

attributes
(state variables)

methods
(class interface)

What is a (software) class ?

class: the *blueprint* characterising a category of objects

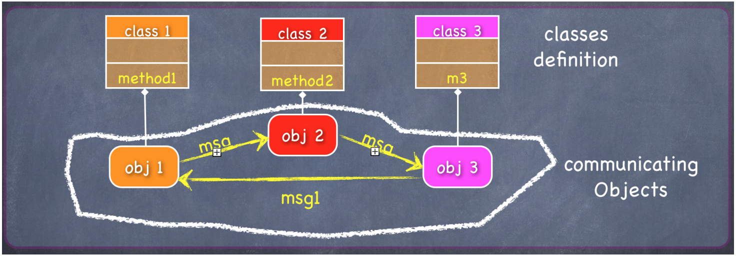
- defines the attributes representing the state of objects
- defines the methods representing the behaviour of objects

several objects can be instantiated from a given class

What an Object-Oriented program looks like?

an Object-Oriented program consists of:

- a collection of classes definitions
- a collection of objects' instances



computation: instantiated objects perform the desired computation by invoking each other methods (i.e. by exchanging messages)

What does a Java program consists of?

java program: a collection of classes declaration

```
public class MyClass1 {
      // ATTRIBUTES DECLARATION
      // CONSTRUCTORS
      // METHODS
}


public class MyClass2 {
      // ATTRIBUTES DECLARATION
      // CONSTRUCTORS
      // METHODS

}

...
```

remark: each class is stored in a dedicated file named after the class (e.g. MyClass1.java, MyClass2.java)

Class declaration

```
class MyClass1 {
        // ATTRIBUTES DECLARATION
        // CONSTRUCTORS DECLARATION
        // METHODS DECLARATION
}
```

a class declaration consists of:

- class header: defining the name of the class, its visibility, etc..
- class body: consisting of 3 parts enclosed within braces {}
  - attributes: variables that determine the state of the class and its objects
  - constructors: for initialising new objects
  - methods: to determine the behavior of the class

remark: the set of (`public`) methods is called the class interface

Example class declaration

```
public class MyClass1 {

        // ATTRIBUTES DEFINITION
        private int n; // a private integer variable
        public char c; // a public char variable
        protected boolean b; // a protected boolean variable
        ...

        // CONSTRUCTORS DEFINITION
        public MyClass1(...) {
         ...
        }

        ...
        // METHODS DEFINITION
        public void myMethod1(...) {
         ...
        }

}
```

Class header

```
public class MyClass1 extends MySuperClass1 implements MyInterface1 {
        // CLASS BODY
}
```

the class header consists of several elements

- modifiers (such as `public` or `private`)
- the `class` keyword
- class identifier (the class name)
- the name of the parent class (if any, we will see later)
- the name of the interfaces implemented by the class (if any, we will see later)

convention: a class identifier should start with a capital letter

Variables within a class body

Within a class body there are different kind of variables:

- member variables (also called attributes or fields)
- local variables: defined within a method's body or a block of code (within braces)
- parameters: defined (within brackets) in a method's declaration

```
class MyClass1 {
      public int n; // a member variable

      public int sum(int m){ // m is a parameter of method sum()
         int result; // result is a local variable of method sum()
         result = n +m;
         return result;
      }
}
```

Member variables declaration

a field's declaration is composed of 3 parts:

- zero or more modifiers (such as `public`, `private`...)
- the type of the field (e.g. `int`, `float`, `bool`...)
- the name of the field

```
class MyClass1 {
      public int n; // a member variable

      public int sum(int m){ // m is a parameter of method sum()
         int result; // result is a local variable of method sum()
         result = n +m;
         return result;
      }
}
```

Method declaration

a method's declaration is composed of 6 parts:

- zero or more modifiers (such as `public`, `private`...)
- the return type: the data type returned by the method (or `void` if method doesn't return a value)
- the name of the method
- the parameters of the method: a comma-separated list of typed variables enclosed with parenthesis
- an exception list (if any, see later on)
- the body of the method: sequence of instructions enclosed between braces {}

method's signature: a method's name and the parameter types form the signature of the method.

Example: syntax for defining methods

```
public class Bicycle {
      ...
       // nullary method which does not return a value
       public void upshift(){ gear++;}

       // unary method which does not return a value
       private void shift(int n){ gear=n;}

       // unary class method returning a floating point value
       public static float wheel_circ(float wheel_diam){
          return 2*Math.PI*wheel_diam;
       }
}
```

Constructor: a “special” method

constructor: a method which has the same name of the class and which is automatically invoked every time an object is instantiated

```
public class Bicycle {
      ...
      // constructor
      public Bicycle(...){
        ...
      }
      ...
}
```

characteristics:

- goal: to initialise the attributes of an object on creation
- it is automatically invoked through the `new` command
- several constructors can be defined for a given class
  - different constructors MUST have different parameters

Example: Bicycle constructors

```
public class Bicycle {
        // attributes declaration
        public int gear;
        public float speed;
        public static float wheel_diam;

        // constructor 1: three parameters
        public Bicycle(int gear, float speed, float wheel_diam){
            this.gear = gear;
            this.speed = speed;
            this.wheel_diam = wheel_diam;
        }

        // constructor 2: two parameters, speed initialised to 0
        public Bicycle(int gear, float wheel_diam){
            this.gear = gear;
            this.wheel_diam = wheel_diam;
        }
        ...
}
```

Object creation: new

```
Class_name obj_name = new Class_name(arg_list);
```

Example:

```
public class TestBicycle {
      ...
      public static void main(){

        // creates a 26-inches wheels bicycle
        // initially in gear 1 and initial speed 10
        Bicycle bicycle1 = new Bicycle(1,10,26);

        // creates a 28-inches wheels bicycle
        // initially in gear 3 and initial speed 0
        Bicycle bicycle2 = new Bicycle(3,28);
         ...
      }
}
```

Default constructor

default constructor: a *nullary* (zero-argument) constructor automatically generated by the compiler if no constructor has been defined for a class

what does the default constructor do?

- it implicitly invokes the nullary constructor of the *superclass*
- it automatically initialises all attributes to their default value:
  - integer: initialised to 0
  - floating-point: initialised to 0.0
  - boolean: initialised to `false`
  - reference: initialised to `null`

The main method: starting the computation

main() method: determines the computation to be performed

- when you “run” a class, the runtime system starts by calling the main() method (if any) contained in the class
- the main() method then calls all the other methods required by your application

The main method signature:

```
public static void main(String[] args){
   //method−body ...
}
```

- accepts a single argument: an array of elements of type String
- the array is used by runtime system for passing information to your application

Adding comments in a class definition

remark: commenting classes is a fundamental part of programming

```
/**
 * javadoc comments about the class
**/
public class CurrentClass {
      // comments about single lines
      private int var;

      /*
       * Comments about the constructor
       */
      public CurrentClass(...) {
       ...
      }

      ...

      public void method(...) {
       ...
      }

}
```

Identifiers (variables and methods names)

identifier: a sequence of characters used for naming variables, methods and classes

characteristics of identifiers:

- no restrictions on length
- case sensitive (`Circle` is different from `circle`)
- cannot start with a digit
- cannot include “/” or “-”
- cannot be a JAVA keyword

Naming conventions

java identifiers CANNOT begin with a digit

- `public class 1bicycle{...}` ⇐ syntax error
- `public bool 2nd_gear;` ⇐ syntax error
- `public void 1upshift(){...}` ⇐ syntax error

class names: should begin with a capital letter

- `public class Bicycle{...}` ⇐ OK
- `public class bicycle{...}` ⇐ discouraged

attributes and methods names: should begin with a smallcase letter

- `public bool secondGear;` ⇐ OK
- `public bool SecondGear;` ⇐ discouraged

JAVA keywords

List of JAVA keywords

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

Modifiers

Java modifiers: keywords added to change meaning of a definition (of a class, or a variable or a method)

Two categories of modifiers in Java:

- access modifiers: indicate the visibility of the declared element (a class, a variable or a method declaration)
- non-access modifierd: denote special properties of the declared element (a class, a variable or a method declaration)

Access modifiers

access modifier: a *keyword* that sets the accessibility of an *attribute*, a *methods* or a *class*

in JAVA there are four access modifiers: `public`, `protected`, `private`, default

access modifier can be used either:

- inside a class: to characterise the visibility of attributes, methods and nested-classes (we will see this later on)
- outside a class: to characterise the visibility of *top level* classes

Access modifiers

meaning of access modifiers for attributes/methods/nested-classes visibility:

- `public`: visible everywhere (outside/inside the class/package)
- `protected`: visible in the same package and in subclasses (wherever they are)
- `private`: visible only in the class where it is defined
- default (no-modifiers is given): visible to the package

meaning of access modifiers for class visibility:

- top-level classes can only have default or `public` visibility

Example: access modifiers

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

   // constructors
   Point(double x, double y){
      this.x = x;
      this.y = y;
   }
   Point(double x){
      this.x = x;
      this.y = 10;
   }

  // methods
  double getX(){
    return x;
  }
  double getY(){
    return y;
  }
}
```

Encaspulation

encaspsualtion: hiding of the variables of a class from other classes, and making them accessible only through the methods of their current class

what are the benefits of encapsulation ?

- the attributes of a class can be made read-only or write-only.
- a class can have total control over what is stored in its attributes.
- the users of a class do not know how the class stores its data. A class can change the data type of a field and users of the class do not need to change any of their code

Encapsulation in practice

what shall we do to achieve encapsulation ?

- declare all attributes as `private`: attributes are not accessible directly (encapsulation)
- provide public *setter* and *getter* methods to modify and view the variables values.

- *getter*: a method that returns the value of a (private) attribute
- *setter*: a method that assign a (private) attribute with a value taken as argument

Example: encapsulation + getters/setters

```
public class Point {
  // all private attributes
  private double x;
  private double y;

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

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

  // setters
  public void setX(double x) {this.x=x;}
  public void setY(double y) {this.y=y;}

}

public class TestPoint {
  public static void main(String[] args){
    Point p1 = new Point(1,9);
    Point p2 = new Point(2,4);
    // attributes x and y are read through getters thus TestPoint is robust wrt internal changes of class Point
    System.out.println("X distance between p1 and p2= " + (p1.getX() - p2.getX()));
    System.out.println("Y distance between p1 and p2= " + (p1.getY() - p2.getY()));

    p1.setX(10); // set the value of p1 x-coordinate to 10
  }
}
```

Non-access modifiers

non-access modifier: provide them special properties to attributes and/or methods and/or classes

in JAVA there are five non-access modifiers:

- `final`: for constants attributes and/or non-extendible classes
- `static`: for variables/methods that can be accessed without needing an object
- `transient`, `synchronized`, `volatile`: not in the scope of this course.

instance v class attributes/methods

- declaration of variables (also called attributes or fields)
  - instance variables: belong to a specific object
  - class variables: shared by all object instances (`static` variables)
- declaration of methods
  - instance methods: accessed exclusively through an object instance
  - class methods: invoked without needing to instantiate an object (`static` methods)

Example: a class with instance/class attributes/methods

```
public class Bicycle {

    // attributes declaration
    public int gear;
    public float speed;
    public static float wheel_diam; // the wheels diameter
    ...

    // methods declaration
    public void upshift(){ gear++;}
    public void downshift(){ gear−−;}
    public static float wheel_circ(){
            return 2*Math.PI*wheel_diam;
    }
    public static float wheel_circ(float d){
              return 2*Math.PI*d;
    }
    ...
}
```

Instance variables

an attribute of a class is called an instance variable if its declaration IS NOT PREFIXED by `static`

```
public class Bicycle {

      public int gear; // instance variable
      ...
}
```

characteristics:

- is automatically initialised to default values:
  - 0: for numeric types
  - `""`: for character types
  - `null`: for object references
- belongs to an object
  - each OBJECT has its own value for each instance variable

Class (static) variables

an attribute of a class is called a class variable if its declaration IS PREFIXED by `static`

```
public class Bicycle {
      ...
      public static float wheel_diam; // class variable
}
```

characteristics:

- is shared by all objects instantiated from the class
  - it exists only one copy of each class variable
- is the equivalent of “global” variables in C language

Instance method

instance method: a method whose declaration IS NOT PREFIXED by `static` keyword

```
public class Bicycle {
    ...
    public void upshift() {gear++;}; // instance method
    ...
    public static void main() {
        Bicycle mybicycle = new Bicycle(); // create a Bicycle object
        mybicycle.upshift(); // call instance method upshift() on mybicyle
    }
}
```

- accessed exclusively trough an object instance of the class

```
mybicycle.upshift();
```

Class method

a method of a class is called a class method if its declaration IS PREFIXED by `static`

```
public class Bicycle {
       ...
       // class method
       public static float wheel_circ(float d){return 2*Math.PI*d;}

       ...
       public static void main() {
           // invocation of class method wheel_circ() of class Bicycle
            System.out.println("circumference:" + Bicycle.wheel_circ(28));
       }
}
```

- no need to instantiate an object to invoke a class method
    - accessed directly by prefixing with CLASS name

```
Bicycle.wheel_circ(28);
```

A method's signature

methods signature: combination of method's name and *typed* list of parameters

Examples

```
public class myClass {
  ...

  // signature is: method1 + one int parameter
  public void method1(int i) {System.out.println(i);}

  // signature is: method1 + one char parameter
  public void method1(char c) {System.out.println(c);}

  // signature is: method2 + two parameters, one char and one float
  public String method2(char c, float f) {
    System.out.println(c + f);
    return "hello";
  }
  ...
}
```

Methods' overloading

methods' overloading: feature that allows a class to have two or more methods having same name given their signatures are different

- if a class contains several version of a method we say that method is OVERLOADED

motivation for overloading

- constructors: different ways for initialising objects
- methods: allowing same name for similar operations

Example 1 - overloading

```
class OverloadingEx1
{
    public void disp(char c){
         System.out.println(c);
    }
    public void disp(char c, int num){
         System.out.println(c + " " + num);
    }

    public static void main(String args[]){
       OverloadingEx1 obj = new OverloadingEx1();
       obj.disp('a');
       obj.disp('a',77);
   }
}
```

output:

Example 1 - overloading

```java
class OverloadingEx1
{
    public void disp(char c){
         System.out.println(c);
    }
    public void disp(char c, int num){
         System.out.println(c + " " + num);
    }

    public static void main(String args[]){
       OverloadingEx1 obj = new OverloadingEx1();
       obj.disp('a');
       obj.disp('a',77);
   }
}
```

output:

```
> a
> a 77
```

Primitive types versus reference types

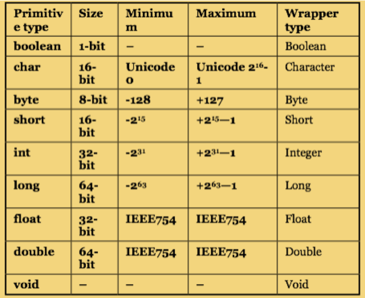
In JAVA we distinguish between primitive data types and reference data types

- primitive data types
  - are eight (`char`, `int`, `double`, `boolean`, ...)
  - are created by declaration (e.g. `int x; char c;` ...)
  - declaration of a variable allocates memory
- objects/reference data types:
  - are instances of classes
  - we manipulate them through a *reference*
  - a reference is like a "pointer" (in C/C++)
  - declaration of a reference does not allocate memory: e.g. `Bicycle b; Integer n;...`)
  - a reference is linked to an actual object on objects creation:

```
Bicycle b = new Bicycle();
```

Primitive types and Wrapper classes

wrapper classes: a class provided in the `java.lang` package to allow treating *primitive types* as objects.

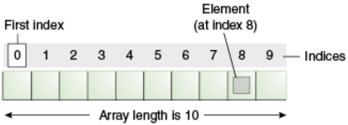


Primitive type	Size	Minimum	Maximum	Wrapper type
boolean	1-bit	–	–	Boolean
char	16-bit	Unicode 0	Unicode 2^{16}-1	Character
byte	8-bit	-128	+127	Byte
short	16-bit	-2^{15}	$+2^{15}-1$	Short
int	32-bit	-2^{31}	$+2^{31}-1$	Integer
long	64-bit	-2^{63}	$+2^{63}-1$	Long
float	32-bit	IEEE754	IEEE754	Float
double	64-bit	IEEE754	IEEE754	Double
void	–	–	–	Void

Example:

```
Integer n = new Integer(10);
  Float f = new Float(3.14);
 Double d = new Double("3.14");
```

Arrays in JAVA



- arrays are containers with fixed size
- the size is defined when the array is created
- they store elements of the same type (e.g. `int`, `String`, etc)
- elements are indexed starting from 0
- we distinguish between declaration, creation and initialisation

remark: in Java arrays are treated as *reference* type (they are objects!)

Array declaration and creation

declaration: an array is declared using square brackets (i.e. []):

```
    int [] v; // v is declared as an array of int
    String [] w; // w is declared as an array of String
```

creation: an array object is created through `new`:

```
    int [] v = new int[3]; // an int array of size 3 is created/assigned to v
    String [] w = new String [2]; //a String array of size 2 is created/assigned to w
```

remark: cannot use an array variable before creating the corresponding array object

```
public class MyClass {
    int [] v; // v is declared as an array of int

    public MyClass(){
        v[0] = 1; // ==> NullPointerException, v has not been created !
    }
}
```

Array initialisation

initialisation: an array object is initialised in either of two ways:

- explicitly after creation (by assigning values to its elements):

```
int v [] = new int [3]; // v is declared and created as an array of 3 int
v[0] = 10; // first element of v is assigned with 10
v[1] = 20; // first element of v is assigned with 10
v[2] = 30; // first element of v is assigned with 30
```

- while declared (by assigning a list of values in between braces {}):

```
int [] v = {10,20,30}; // v is declared, created and initialised
```

array constant: a list of values in between {}

remark: array constants can only be used at declaration

```
int v [] = new int [3]; // v is declared and created as an array of 3 int
v = {10,20,30} // compile−time error
```

Multi-dimensional arrays in JAVA

multidimensional array: an array whose components are themselves arrays

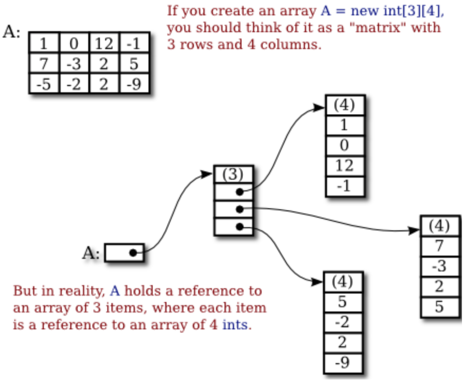
declaration: declared by concatenating square brackets

```
int [][] vv; // vv is declared as 2D array of int
int [][][] vvv; // vvv is declared as 3D array of int
```

creation: created through `new` and by specifying the size of each dimension

```
int [][] vv = new int[3][3]; // vv is a square matrix of size 3
int [][] a = {
    {1,0,12,-1},
    {7,-3,2,5},
    {5,-2,2,-9}}; // declaration/initialisation of 2D array
```

Multi-dimensional arrays: important remarks



A:

1	0	12	-1
7	-3	2	5
-5	-2	2	-9

If you create an array A = new int[3][4], you should think of it as a "matrix" with 3 rows and 4 columns.

But in reality, A holds a reference to an array of 3 items, where each item is a reference to an array of 4 ints.

remark: while creating a multi-dimensional array only the first dimension MUST be specified

```
int [] [] vv = new int [3][]; // correct declaration of a 2D array
int [] [] [] vvv = new int [3][][]; // correct declaration of a 3D array
int [] [] [] vvv = new int [3][2][]; // correct declaration of a 3D array
```

Multi-dimensional arrays: initialisation

initialisation of an *N*-dimensional array is done through *N*-nested loops:

```
// how to initialize two dimensional array in Java using for−loop
int[][] vv = new int[3][3];
for (int i = 0; i < vv.length; i++) {
  for (int j = 0; j < vv[i].length; j++) {
    vv[i][j] = i + j;
  }
}
```

Useful features of array objects

- an array object is provided with an attribute named `length` which is assigned with the length of the array at creation

```
int[] v = {1,2,3};
int len = v.length; // assign the value 3 to variable len
```

- an array object belongs to the `java.util.Arrays` class for which a number of useful methods are available
  - `void sort(int [] a)`: Sorts the specified array into ascending numerical order.
  - `int[] copyOf(int[] original, int newLength)`: Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.

Strings in JAVA

JAVA comes with the `String` class for representing sequences of characters

We declare `String` objects in several ways:

```
String s1 = "hello";// s1 is a reference to the "hello" string
String s2 = new String(" world");
```

basic operations on `String` objects:

```
String s3 = s1 + s2;// + concatenation
int x = s3.length(); // s3.length() returns the length of s3
```

Basic Input/Output in Java

OUTPUT on screen (through methods `print` of class `System`):

```java
int i = 10;
// displays on screen the value of i
System.out.print(i);
// displays on a line
System.out.println("i is equal to:  " + i);
```

INPUT from keyboard (through methods `next` of class `Scanner`):

```java
Scanner sc = new Scanner(System.in);
System.out.println("enter a word:  ");
String s = sc.nextLine();
System.out.println("enter an integer num:  ");
int i = sc.nextInt();
```

Operators

- **arithmetic**: `+, -, *, /,` % (modulo)
- **increment/decrement**: `++, --, +=, -=, *=, /=,` %=
- **relational**: `==, !=, >=, <=, >, <`
- **boolean**: `&&` (AND), `||` (OR), `!` (NOT)
- **bitwise**: `&` (AND), `^` (OR), | (XOR)
- **ternary**: `condition ? (expr1) : (expr2)`
- **access, method call**: `. [] ()`

Java packages

Pacakages: organisation of classes into *namespaces*

- namespace: a container for a set of identifiers

packages affect classes visibility

- classes in the same package can access each other
- classes with same name can exist in different packages

package naming:

- *default* anonymous package exists
- however package should be named with relevant names

Example: `organizationName.functionalityName`

```
fr.ecp.IS1220.2Dgeometry.Circle
fr.ens-cachac.2Dgeometry.Circle
```

homonymous classes `Circle` belonging to different packages

Declaring/importing packages

declaring pacakages: through the `myredpackage` keyword

```
package fr.ecp.IS1220.2Dgeometry
 ...
class Point{... }
class Line{... }
```

classes of external packages need to be imported using the import from keyword

packages are imported through keyword `import`

```
import java.util.*
import java.math.*
```

- * stands for: import all classes from that package
- classes from `java.lang` are imported automatically

Control flow instructions

- conditional
  - `if else`
- loops
  - `do while`
  - `while do`
  - `for`
- `switch`
- breaking loops/switch

if statement

if: executes a section of code only *if* a certain test is TRUE

form1 (with braces): the BLOCK of instructions within {}is executed only if `test` evaluates to TRUE

```
if (booleanExpr) {
  BLOCK
}
INSTRUCTION2
```

form2 (without braces): INSTRUCTION1 is executed only if `test` evaluates to TRUE

```
if (booleanExpr)
  INSTRUCTION1
INSTRUCTION2
```

if-else statement

if-else: like if but provides a secondary path of execution when test is FALSE

```
if(booleanExpr) {
      //do something
}
else {
      //do something else
}
```

it can be chained:

```
if(booleanExpr) {
      //do something
}
else if (booleanExpr2) {
      //do something else
}
else if (booleanExpr3) {
 ... }
else{
  ... }
```

while loop

continually executes a block of statements while a condition is true

```
while(booleanExpr)
{
    BLOCK //loops while booleanExpr is true
}
```

- The while statement evaluates `booleanExpr`, which must return a boolean value
- If `booleanExpr` evaluates to true, the while statement executes the statement(s) in the while block.
- It continues testing the `booleanExpr` and executing its block until `booleanExpr` evaluates to false

do-while loop

as while-loop but executes always at least once the block

```
do
{
  BLOCK //executes block once and then loops while booleanExpr is true
}
while(booleanExpr);
```

- do-while evaluates the booleanExpr at the bottom of the loop instead of the top.
- The statements within the do block are always executed at least once

for loop (simple)

Provides a compact way to iterate over a range of values.
It lists a set of instruction to be executed since a condition is true

```
for(int i=1;i<=n; i++) {
    BLOCK //executes BLOCK n times
}
```

- The initialization expression initializes the loop; it's executed once, as the loop begins
- When the termination expression evaluates to false, the loop terminates
- The increment expression is invoked after each iteration through the loop

equivalence `for`-loop and `while`-loop

```
for(int i=1;i<=n; i++) {
     BLOCK //executes loop body n times
}
```

Equivalent to the following one:

```
int i=1;
while(i<=n){
      BLOCK //executes loop body n times
      i++;
}
```

Also in this case the `for` syntax requires one statement that can be a single instruction or a block (in between {})

Example: simple for-loop

```
int [] array = {1,2,3,4,5,6,7};
for(int i=0; i<=array.length; i++) {
     System.out.println("array element" + array[i]);
}
```

Question: is there any issue with the above example of for-loop ?

Example: simple for-loop

```
int [] array = {1,2,3,4,5,6,7};
for(int i=0; i<=array.length; i++) {
    System.out.println("array element" + array[i]);
}
```

Question: is there any issue with the above example of for-loop ?

- it yields an `IndexOutOfBoundException`

for loop (enhanced)

Enhanced `for`: allows for iterating directly over elements of a *collection* (array, arrayList, sets...)

Example: enhanced `for`-loop (with an array):

```java
int [] array = {1,2,3,4,5,6,7};

for(int i: array){
    System.out.println("array element" + i);
}
```

Example: enhanced `for`-loop (with an ArrayList):

```java
ArrayList<String> list = new ArrayList<String>();
list.add("hello"); list.add("world");
for(String s:list){
     System.out.println("list element " + s);
}
```

switch

```
switch(intExpr)
{
     case intValue_1: instructions_1;
          break;
     case intValue_2: instructions_2;
          break;
     ...
     default: instructions;
}
```

- evaluates `intExpr` and executes all `instruction_i` following the matching `case intValue_i`
- `break` interrupts execution and jumps outside switch block
- eventually execute default instructions (unless break-ed)

Question

What will be printed?

```
int i=1;
switch(i)
{
      case 0: System.out.print(i);
      case 1: System.out.print(i);
      case 2: System.out.print(i);
      default: System.out.print("???");
}
```

Question

What will be printed?

```
int i=1;
switch(i)
{
      case 0: System.out.print(i);
      case 1: System.out.print(i);
      case 2: System.out.print(i);
      default: System.out.print("???");
}
```

answer

```
> 11???
```

- why? because `break`s are missing!

breaking control flow: break/continue

- break [optionalLabel]
  - usable within loop or a switch
  - exits the loop or switch (if no label)
  - exits current loop and jumps to optionalLable
- continue [optionalLabel]
  - usable within a loop or a switch
  - jumps to the remainder of the current iteration and continue with the next iteration
- labeled/unlabeled statements

enum type

enum: a variable's domain defined as a set of pre-defined constants

```
public enum Day {MONDAY,TUESDAY, ... , SUNDAY} // definition of Enum called Day

public EnumTest{
   Day day; // variable of enum type Day
   EnumTest(Day day){this.day = day;} // constructor

   public message(){ // display a different message depend on the day
       switch(this.day) {
          case MONDAY: System.out.print("I am sad");
                   break;
          case FRIDAY: System.out.print("I am happy!");
                   break;
       }
    }
}
```

Assigning values to enum variable:

```
Day day = Day.TUESDAY;
```

Basic JAVA packages

- `java.lang`: basic language functionalities, automatically imported
- `java.util`: collections and data structures
- `java.math`: multi precision arithmetic
- `java.net`: networking, sockets, ...
- `java.awt`: native GUI components
- `javax.swing`: platform independent rich GUI components

documentation `http://docs.oracle.com/javase/8/docs/api/`

Outline

1. Programming languages
2. Java basics
3. Inheritance
   - Overriding
4. Abstract classes, Interfaces
5. Collections
6. Summing up

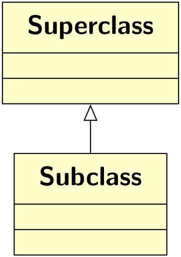
Inheritance

basic idea: we may need to represent similar classes which differ only in some parts

two possibilities

- we could re-define a NEW CLASS for each similar concepts (tedious and time-consuming, see Exercise 1 of Tutorial2)
- inheritance: we take an existing class, we ”clone” it, and make additions and modifications to the clone (see Exercise 2 of Tutorial2)

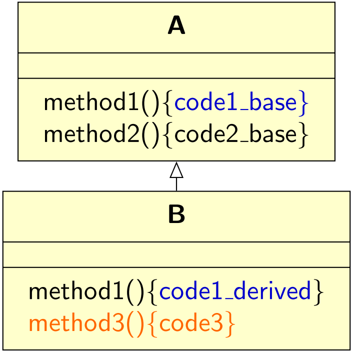
Inheritance: superclass, subclass



- subclass (or derived class): inherits the attributes and methods of its parent class superclass (or parent class)
- a subclass may add its own attributes and methods and override or extend its parent's methods

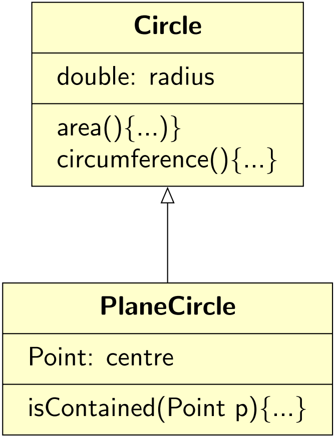
> **remark**: an object of the subclass IS ALSO an object of the superclass

overriding and extending a class



- overriding: the body of an inherited method is changed
  - method1() of class B overrides method1() of superclass A
- extending: new methods are added to the "derived" class
  - method3() of class B extends the interface of superclass A

Example

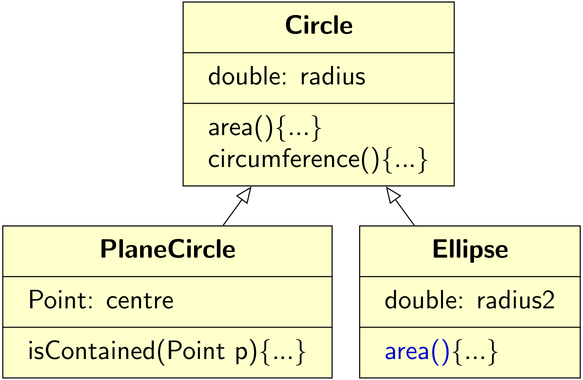


superclass **Circle**: a circle is characterised only by its radius

subclass **PlaneCircle**: a circle is characterised also by its centre

- inherits:
  - the `radius` attribute
  - the `area()` and `circumference()` methods
- introduces:
  - the `centre` attribute
  - the `isContained()` method

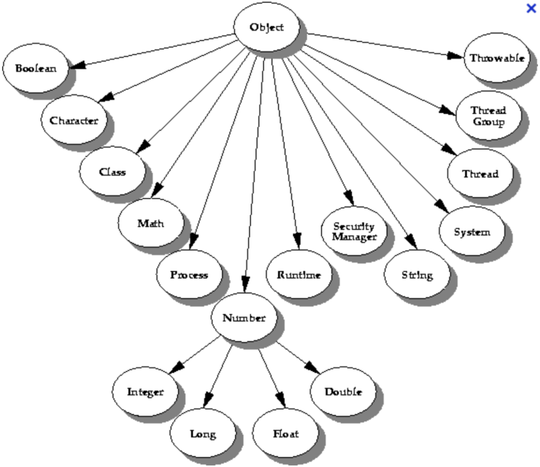
Example (continued)



subclass `Ellipse`:

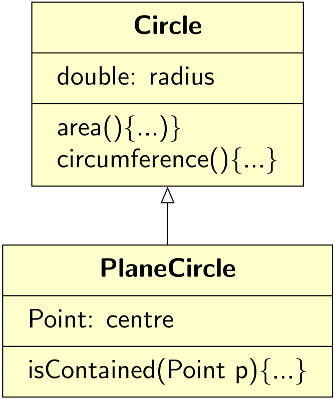
- inherits: `radius` attribute, `area()` and `circumference()` methods
- introduces: `radius2` attribute
- overrides: the `area()` method

Java.lang hierarchy tree



`Object`: is the root class, all other classes are descendants of `Object`

Characteristics of inheritance



Circle

double: radius

area(){...)}
circumference(){...}

PlaneCircle

Point: centre

isContained(Point p){...}

- a superclass object only sees the attributes and methods of the superclass
- a subclass object sees both the attributes/methods of the subclass as well as those of the superclass

- a Circle object state consists of its `radius` while its interface of the `area()` and `circumference()` methods
- a PlaneCircle object state consists of its `radius` and `centre` while its interface of the `area()`, `circumference()` and `isContained()` methods

How do we declare subclasses in Java ?

- `extends`: keyword used to declare a subclass
- `super`: keyword used to refer to the superclass from the subclass

```
public class Circle {
  private double radius;
  // constructor: initialise ”radius”
  public Circle(double radius){this.radius = radius};
  public double area(){return radius*radius*Math.PI();}
  public double circumeference(){return 2*radius*Math.PI();}
  }
}

// PlaneCircle is declared as a subclass of Circle
public class PlaneCircle extends Circle{
  private double x,y;
  //constructor of subclass: first thing call superclass constructor through super
  public PlaneCircle(double x, double y){
    super();
    this.x = x; this.y=y;
  }
}
```

Subclass constructor

remarks:

- a class constructor initialise the state of an object of the class
- a subclass object implicitly contains an object of the superclass

issue: what shall a subclass constructor do about the superclass attributes ?

two possibilities:

- invokes (one of the) superclass constructor(s) through `super(argList)`
- invokes the superclass default constructor through `super()`

Example: subclass constructor

super default constructor

```
public class Circle {
  private double radius;
  // superclass constructor
  public Circle(double radius){
     this.radius = radius;
  }
  ...
}

public class PlaneCircle extends Circle{
  private double x,y;

  // subclass constructor
  public PlaneCircle(double x, double y){
    super(); // invoking the default constructor of
              Circle
    this.x = x; this.y=y;
  }
  ...
}
```

super specific constructor

```
public class Circle {
  private double radius;
  // superclass constructor
  public Circle(double radius){
     this.radius = radius;
  }
  ...
}

public class PlaneCircle extends Circle{
  private double x,y;

  // subclass constructor
  public PlaneCircle(double x, double y, double r){
    super(r); // invoking the 1−arg constructor of
              Circle
    this.x = x; this.y=y;
  }
  ...
}
```

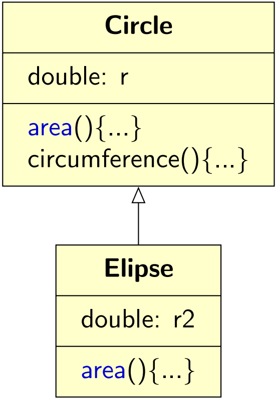
Remarks: subclass constructor

rule: the first instruction in a subclass constructor must be the call of superclass constructor (i.e. `super`)

why? because this will ensure that if you call any methods of the superclass within your subclass constructor, the superclass has already been set up correctly.

Method's overriding

overriding: when a class defines an instance method with same signature as a method in the superclass



- `area()` method in `Elipse` overrides `area()` method in `Circle`

Overriding properties

we can refer to overriden methods:

- from a method in the derived class
  - through `super`
- but NOT from an object of the derived class (even with cast the object into the superclass type)

Type conversion

type conversion: when a variable of one type is assigned to a variable of another type

- primitive types: `short`, `int`, `long`, `byte`, `bool`, `float`, `double`
- reference types: `class`, `interface`, [] (arrays)

what happen when we convert a variable of a type into a variable of another type ?

```
int x=1;
char c='a';
double y;
y = x; //??
x = c; //??
```

Type conversion: widening

type widening: from narrower to wider type, automatically converted

- widening conventions
  - `byte` to `short`,`int`,`long`, `float` or `double`
  - `short` to `int`,`long`, `float` or `double`
  - `int` to `long`, `float` or `double`
  - `long` to `float` or `double`
  - `float` to `double`

```
int x=1;
char c='a';
x = c; //no compile-time error automatically converted
```

Type conversion: narrowing

type narrowing: from wider to narrower type, requires casting

- narrowing conventions
  - `double` to `byte`, `char`, `short`,`int`,`long` or `float`
  - `float` to `byte`, `char`, `short`,`int` or `long`
  - `long` to `byte`, `char`, `short` or `int`
  - `int` to `byte`, `char` or `short`
  - `short` to `byte` or `char`
  - `byte` to `char`

```
int x=1;
double y;
x = (int) y; //requires casting
```

Explicit type-casting

type casting: the explicit conversion of a type into another through a specific syntax:

```
target_type var1= (target_type) var2;
```

example:

primitive types

```
public class MyClass {
  public static void main(String [] args){
    int x =10;
    double y;
    char c;
    ...
    y = (double) x; // downcasting an int to a double
    x= (int) c; // downcasting a char to an int
  }
}
```

reference types

```
public class MyClass {
  public static void main(String [] args){
    Circle c;
    PlainCircle pc = new PlainCircle(0,0,10);
    ...
    c = (Circle) pc; // downcasting a plaincircle into
            a circle
    ...
  }
}
```

Example: overriden methods

we cannot access the (overridden) superclass method from a DERIVED OBJECT

```
Circle c = new Circle(1);
Ellipse e = new Ellipse(1,3);

System.out.println("circle c area =" + c.area()); // area is = 3.14
System.out.println("ellipse e area =" + e.area()); // area is = 9.42

// attempting to call the area method of class Circle on the Eclipse object e
System.out.println("ellipse e area =" + ((Circle) e).area());
```

remark: when compiling `((Circle) e).area()` the compiler will automatically generate a call to the `Ellipse.area()` method and not to the `Circle.area()` method

Why overriding is useful?

because is one of basic mechanism to obtain modularity

- overriding is the practical means for polymorphism
- we can write code that is inherently working with objects of different nature

Example: polymorphic code (through overriding)

```
Circle [] arrayCircles = new Circle[2];

arrayCircles[0] = new Circle (1);
arrayCircles[1] = new Ellipse(1,3);

for (int i=0; i< arrayCircles.length; i++){
System.out.println(”the area of arrayCircles[” + i + ”] is: ” + arrayCircles[i].
      area());
}

// what is area is displayed here ? the Circle or Ellipse area ?
System.out.println(”e1 area casted into circle: ” + ((Circle)e1).area());
```

- we can declare an array of `Circle` objects and populate it with both `Circle` and `Ellipse` objects
- we can compute the area of each object in `arrayCircles` simply by invoking the `area()` method on each element of `arrayCircles`
- the correct method (i.e. the overriding or overridden) will be invoked automatically by the compiler

Outline

1. Programming languages
2. Java basics
3. Inheritance
4. Abstract classes, Interfaces
5. Collections
6. Summing up

Abstract class

- abstract class: A class that cannot be instantiated (it can be subclassed by *concrete subclasses*)
- abstract method: method without body

```
public abstract class Shape {
    private String name;

    public abstract double area();
    public void printName(){..}
    ...
}
public class Circle extends Shape{
    public double area(){return(Math.PI*radius*radius)};
}
```

Motivations:

- guarantees that all subclasses have the same methods
- let you make collections of mixed type (e.g. arrays of shapes)

Abstract class: summary

Relevant facts about abstract classes.

- If a class is declared `abstract` it cannot be instantiated.
- `abstract` classes may or may not contain *abstract methods*
- if a class have at least one `abstract` method, then the class **must be** declared `abstract`.
- to instantiate objects of an `abstract` class one must define a concrete subclass that implements all `abstract` methods
- a class that inherits from an `abstract` class and does not implement all `abstract` methods must be declared `abstract`

Interface

A group of related methods with empty bodies

- declared through `interface` keyword
- a class `implements` an interface if implements all of its methods

```
public interface Shape {
    public abstract double area();
        ...
}
public class Circle implements Shape {
    public double area(){
        return PI*radius*radius};
}
```

Interfaces form a contract between the class and the outside world

- guarantee that all subclasses have the same methods
- allows to make collections of mixed type (e.g. arrays of shapes)

Interfaces: main characteristics

- equivalent to a fully abstract class
- a class may implements one or more interfaces
  - a way to deliver multiple inheritance in Java
- an interface can extend one or more interfaces
- very useful for software design: specify what not how
- By default, in a Java `interface`
  - all methods are implicitly `abstract` and `public`
    all attributes are implicitly `public`, `static`, and `finel`

Interfaces: summary

Relevant facts about interfaces.

- an interface cannot be instantiated
- an interface cannot contain any constructor
- all methods of an interface are implicitly abstract
- an interface cannot contain *instance attributes*. Any attribute of an *interface* must be declared as both `static` and `final`.
- an interface is implemented (not extended) by a class
- an interface can extends many interfaces

Example: using Interfaces vs Abstract classes

Problem: design an OOP solution allowing for computing the area of different 2D shapes (e.g. rectangles, circles, etc.)

solution1: using `interface`

```
public interface Shape {
      public double area();
}

class Square implements Shape {
      private double side;
      public double area(){return side*side;}
}

class Circle implements Shape {
      private double radius;
      public double area(){return Math.PI*radius*
              radius;}
}
...
```

solution2: using `abstract`

```
public abstract class Shape {
      public abstract double area();
}

class Square extends Shape {
      private double side;
      public double area(){return side*side;}
}

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

Comparison: Interfaces vs Abstract classes

using `interface` makes the code more flexible:

- classes can directly extend only one abstract class
- classes can implement several interfaces

for example:

```
public class Square extends Rectangle implements Polygon, Shape {
          public double area(){return side*side;}
}
```

Outline

1. Programming languages
2. Java basics
3. Inheritance
4. Abstract classes, Interfaces
5. Collections
6. Summing up

The notion of collection in OOP

A *collection* is an object that groups multiple elements of the same type in a single unit

Java provides two families of collections:

- arrays: they have fixed size
    - may contain either primitive types or objects
- `Collection` framework: they have variable size
    - must contain objects (i.e. reference types) only

Collection framework

Collection framework:

unified architecture for representing and manipulating collections

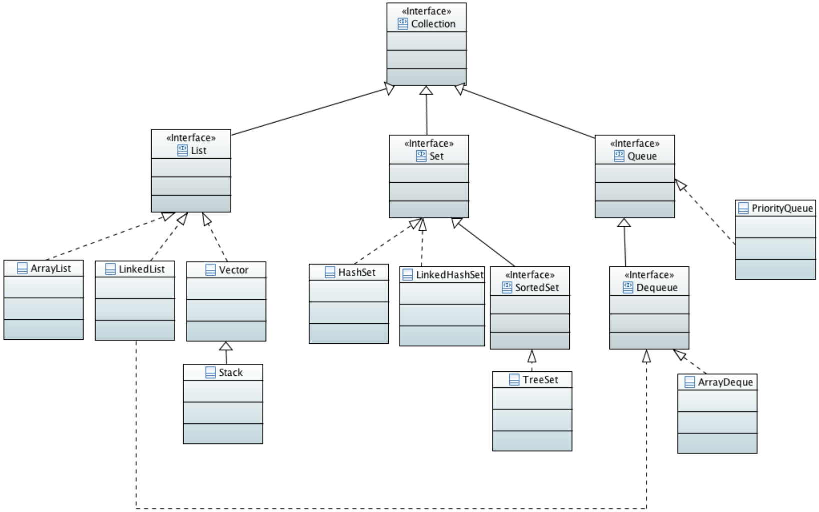
It consists of:

- Interfaces: abstract data types representing collections
- Implementations: concrete implementations of the collections interfaces
- Algorithms: methods for manipulating collections, like searching, sorting on objects of a collection

Why an Interface architecture?

because allows collections to be manipulated independently of their implementation

Java core `Collection` interfaces



Java core Collection interfaces are generics

remark: all interfaces of the Java collection framework are generics

```
// Java collections are declared (in java.util package) as generic interfaces
public interface Collection <E> ...
public interface List <E> ...
public interface Set <E> ...
public interface Queue <E> ...
```

remark: to create a collection object you must specify the type of object contained in the collection

```
// create an HashSet of String
Collection<String> c = new HashSet<String>();

// create an ArrayList of Point
List<Point> l = new ArrayList<Point>();
...
```

The Collection interface

- the interface at the root of the collection framework hierarchy
- is the most general type of collection

basic methods declared in `Collection`:

```
int size(); // returns the size of the collection
boolean isEmpty(); // returns true if the collection is empty
boolean contains(Object element); // true if collection contains element
boolean add(Object element); // add element to collection
boolean remove(Object element); // remove element from collection
```

remark: any implementation of `java.util.Collection` must concretise all above methods

Converting collections of different type

- the type Collection is used to pass around collections of objects where maximum generality is desired
- conversion constructor: all implementation of Collection have a constructor that takes a Collection as argument
- this constructor can be used to convert an Collection into any specific implementation of the Collection interface

Example: collection conversion through conversion constructor

```
// create a an HashSet of String and assign it to a Collection of String variable
Collection<String> c = new HashSet<String>();

// convert c into an List of String by invoking the conversion constructor of ArrayList
List<String> l = new ArrayListList<String>(c);
...
```

The List interface

- List is an ordered Collection that may contain duplicate elements
- Java platform has two implementation of List: ArrayList and LinkedList

in addition to Collection methods it also declares the following:

```
E get(int i); // returns the element at position i
E set(int i, E e); // replaces element at position i with e
int indexOf(E e); // returns index of 1st occurrence of e (−1 if not found)
```

Example: creating List objects

```
List<String> s = new ArrayList<String>(64); // an ArrayList with initial
      capacity at 64
List<String> s1 = new ArrayList<String>(); // an ArrayList with no initial
      capacity specified
```

ArrayList example

```
import java.util.ArrayList;

public class MyArrayList {

    public static void main(String[] args) {
        // Declare the List: the actual type is ArrayList
        List<String> var = new ArrayList<String>();

        // add a few Strings to it
        var.add(”Lars”);
        var.add(”Tom”);
        // Loop over it and print the result to the console
        for (String s : var) {
            System.out.println(s);
        }
    }
}
```

ArrayList example

```java
package collections;
import java.util.ArrayList;

public class MyArrayList {

    public static void main(String[] args) {
        // Declare the List concrete type is ArrayList
        List<String> var = new ArrayList<String>();

        // add a few Strings to it
        var.add("Lars");

        Integer myInteger = new Integer(5);
        var.add(myInteger); // Compile time error!!!!

        // Loop over it and print the result to the console
        for (String s : var) {
            System.out.println(s);
        }
    }
}
```

Remark: if you try put a non `String` object into the list `var` you get a compile-time error

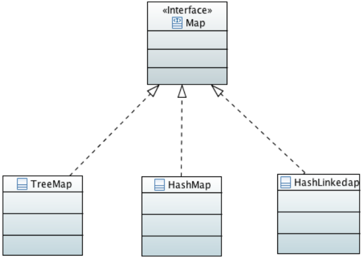
The Set interface

- Set is a Collection that cannot contain duplicate elements
- Java platform has one implementation of Set:
  - HashSet: stores elements into a hash table
- Set has only methods inherited from Collection

Example: creating Set objects

```
Set<String> s1 = new HashSet<String>(); // an HashSet with no
                                        // initial capacity
```

The Map interface



- Map: an interface that allows to handle groups of objects.
- Map is an interface separated from the Collection hierarchy

Map Interface

Map: an object that maps keys into values, cannot contain duplicated keys (a model of mathematical function)

basic operations of interface Map:

```
V put(K , V v); // associates v value with the specified key

V get(Object key); // returns the value to which the specified key is mapped

int indexOf(E e); // returns index of 1st occurrence of e (−1 if not found)

V remove(Object k) // removes the mapping for key k

boolean containsKey(Object key); // true if this map contains a mapping for k

boolean containsValue(Object value); // true if this map maps one or more
                                     // keys to the specified value

int size(); //returns the number of key−value mappings in this map
```

Map Interface

- More operations:
  - `boolean isEmpty()`: returns true if this map contains no key-value mappings
  - `Set<K> keySet()`: returns a Set view of the keys contained in this map
  - `Set<Map.Entry<K,V>> entrySet()`: returns a Set view of the mappings contained in this map
  - `Collection<V> values()`: returns a Collection view of the values contained in this map.
- Implementations:
  - HashMap, ...

Example:

```
Map<String, Integer> s = new HashMap<String,Integer>();
```

Initialization of a map of String and int

Map example

```
public class MapTester {
  public static void main(String[] args) {
    // Keys are Strings
    // Objects are also Strings
    Map<String, String> mMap = new HashMap<String, String>();
    mMap.put("Android", "Mobile");
    mMap.put("Eclipse", "IDE");
    mMap.put("Git", "Version control system");
    // Output
    for (String key : mMap.keySet()) {
      System.out.println(key +" "+ mMap.get(key));
    }
    // Delete from map
    mMap.remove("Android");

    System.out.println("New output:");
    // Output
    for (String key : mMap.keySet()) {
      System.out.println(key +" "+ mMap.get(key));
    }
  }
}
```

Outline

1 Programming languages

2 Java basics

3 Inheritance

4 Abstract classes, Interfaces

5 Collections

6 Summing up

Summing up

- Object-oriented programming
  - notion of class/object (state + behaviour), computation through message passing between objects
- Class declaration in Java
  - class header, class body (attributes, constructors, methods)
  - declaration of member variables
  - declaration of methods (method signature)
  - declaration of constructors
  - instantiating objects: keyword `new`
  - starting the computation: the `main()` method
- Access/non-access modifiers in Java
  - Instance variables/methods, class variables/methods

Summing up

- More basic elements of Java
  - Primitive types, reference types, wrapper classes
  - basic input/output operations
  - Arrays: containers with fixed size
    - Declaration, creation and initialisation of array objects
  - Strings
  - Packages: organising code into packages
  - Control-flow instructions: `if-else`, `switch`, `for`, `while`
  - Encapsulation: hiding of internal implementation of a class through `private`

Summing up

- Inheritance: defining of a new class as a modification of an existing one
  - overriding: a class defining an instance method with same signature as a method in the superclass
  - polymorphism: overriding yields concise polymorphic code
- abstract class versus interfaces
- using Collections in Java