Java:
Basic Concepts

SOftEng
http://softeng.polito.it

Learning objectives

- Learn the syntax of the Java language
- Learn the primitive types
- Understand how classes are defined and objects used
- Understand how arrays work
- Understand how modularization and scoping work
- Understand how to work with strings
- Learn about wrapper types
- Learn about memory management

Comments

- C-style comments (multi-lines)

```
/* this comment is so long
   that it needs two lines */
```

- Comments on a single line

```
// comment on one line
```

Code blocks and scope

- Java code blocks are the same as in C
- Each block is enclosed by braces { } and starts a new scope for the variables
- Variables can be declared both at the beginning and in the middle of block code

```
for (int i=0; i<10; i++){
   int x = 12;
   ...
   int y;
   ...
}
```

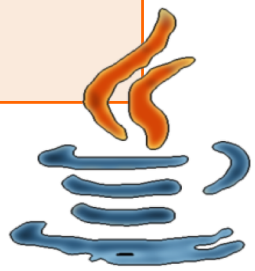
Control statements

- Similar to C
  - if-else,
  - switch,
  - while,
  - do-while,
  - for,
  - break,
  - continue

Switch statements with strings

- Since Java 7, String objects can be used as cases values

```
switch(season){
   case "summer":
   case "spring": temp = "hot";
   break;
}
```



Boolean

- Java has an explicit type (**`boolean`**) to represent logic values (**`true`**, **`false`**)
- Conditional constructs evaluate boolean conditions
  - Note: It is not possible to evaluate this integer condition

```
int x = 7; if(x){...} // NO
```

  - Use relational operators **`if(x != 0)`**
  - Avoid common mistakes, e.g., **`if(x = 0)`**

Passing parameters

- Parameters are always passed by value
- ...they can be primitive types or object references

- Note: only the object reference is copied not the value of the object

Elements in a OO program

Structural elements
(types)
compile time

- Class
- Primitive type

Dynamic elements
(data, instances)
run time

- Reference
- Variable

Classes and primitive types

- Primitive type(s)

```
int, char,
float
```

Type (Descriptor)

- Class

```
class Exam {}
```

- Variable of primitive type

```
int i;
```

Instance

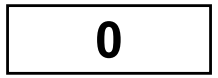
- Variable of type reference

```
Exam e;
e = new Exam();
```

Primitive type

- Defined in the language:
  - int, double, boolean
- Instance declaration:
  - Declares instance name
  - Declares the type
  - Allocates memory space for the reference

```
int i;
```



Primitive types

SOftEng
http://softeng.polito.it

Primitive types

- Have a unique dimension and encoding
  - Representation is platform-independent

Type	Dimension	Encoding
`boolean`	1 bit	-
`char`	16 bits	Unicode
`byte`	8 bits	Signed integer 2C
`short`	16 bits	Signed integer 2C
`int`	32 bits	Signed integer 2C
`long`	64 bits	Signed integer 2C
`float`	32 bits	IEEE 754 sp
`double`	64 bits	IEEE 754 dp
`void`	- -	-

Literals

- Literals of type **`int`**, **`float`**, **`char`**, strings follow C syntax
  - **`123  256789L  0xff34  123.75  0.12375e+3`**
  - **`'a'  '%'  '\n'  "prova"  "prova\n"`**
- Boolean literals (do not exist in C) are
  - **`true`**, **`false`**

Type conversion, typecasting

- Changing one data type into another
- To be used with caution: information can be lost (truncation, rounding, etc.)
  - Language-dependent rules
- Implicit (performed by the compiler)
  - `double d; int i;`
  - `if(d > i) ... d = i;`
- Explicit
  - `double x = 0.33; double y = 0.67;`
  - `int result = (int)x+(int)y;`

Operators (integer, float. point)

- Operators follow C syntax:
  - arithmetical `+ - * / %`
  - relational `== != > < >= <=`
  - bitwise (int) `& | ^ << >> ~`
  - assignment `= += -= *= /= %= &= |= ^=`
  - Increment `++ --`
- Chars are considered like integers (e.g. switch)

Logical operators

- Logical operators follows C syntax:

  `&&  ||  !  ^`

- Warning: logical operators work ONLY on **`boolean`** operands
  - Type **`int`** is NOT treated like a boolean: this is different from C
  - Relational operators return **`boolean`** values

Classes

SOftEng
http://softeng.polito.it

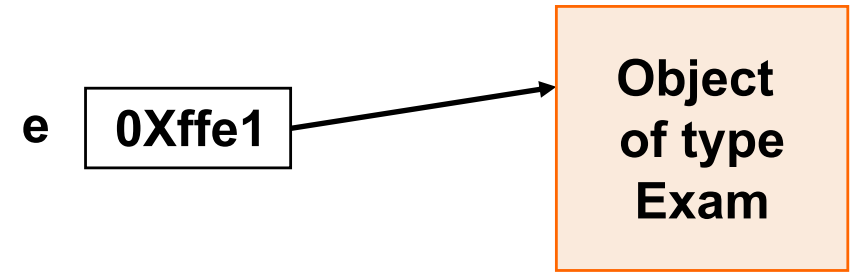
Class

- Defined by developer (e.g., **`Exam`**) or by the Java environment (e.g., **`String`**)
- The declaration

```
Exam e;
```

- allocates memory for the reference ("pointer")
  …and *sometimes* it initializes it with null
- Allocation (creation) and initialization of the object's value are made later by its constructor

```
e = new Exam();
```

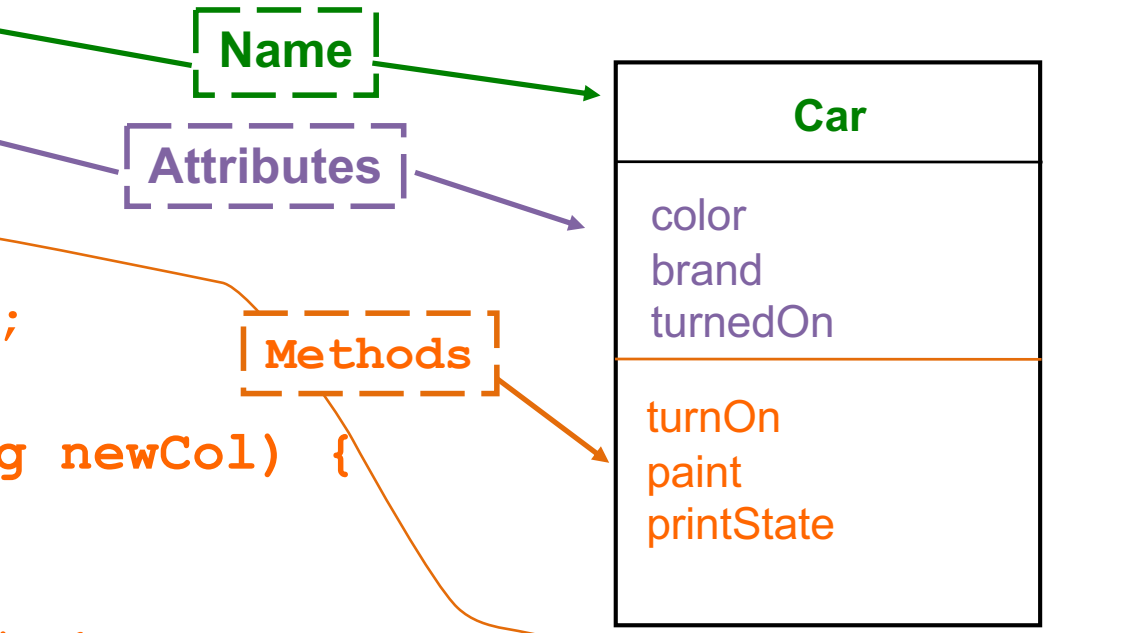


Class

- Object descriptor
  - Defines the common structure of a set of objects
- It consists of a set of members
  - Attributes
  - Methods
  - (Constructors)

Class – definition

```
public class Car {
   String color;
   String brand;
   boolean turnedOn;
   void turnOn() {
      turnedOn = true;
   }
   void paint (String newCol) {
      color = newCol;
   }
   void printState () {
      System.out.println("Car " + brand + " " + color);
      System.out.println("the engine is"
        +(turnedOn?"on":"off"));
   }
}
```



SOftEng
http://softeng.polito.it

Methods

- Methods represent the messages that an object can accept:
  - **`turnOn`**
  - **`paint`**
  - **`printState`**
- Methods may have parameters (may accept arguments)
  - **`paint("Red")`**

Overloading

- In a Class there may be different methods with the same name
- But they have a different signature
- A signature is made by:
  - Method name
  - ordered list of parameters types
- The method whose parameters types list matches, is then executed

```
class Car {
   String color;
   void paint(){
      color = “white”;
   }
   void paint(int i){}
   void paint(String
                newCol){
      color = newCol;
   }
}
```

Overloading

```
public class Foo{
   public void doIt(int x, long c){
      System.out.println("a");
   }
   public void doIt(long x, int c){
      System.out.println("b");
   }
   public static void main(String args[]){
      Foo f = new Foo();
      f.doIt(       5 ,(long)7 ); // "a"
      f.doIt( (long)5 ,       7 ); // "b"
   }
}
```

Objects

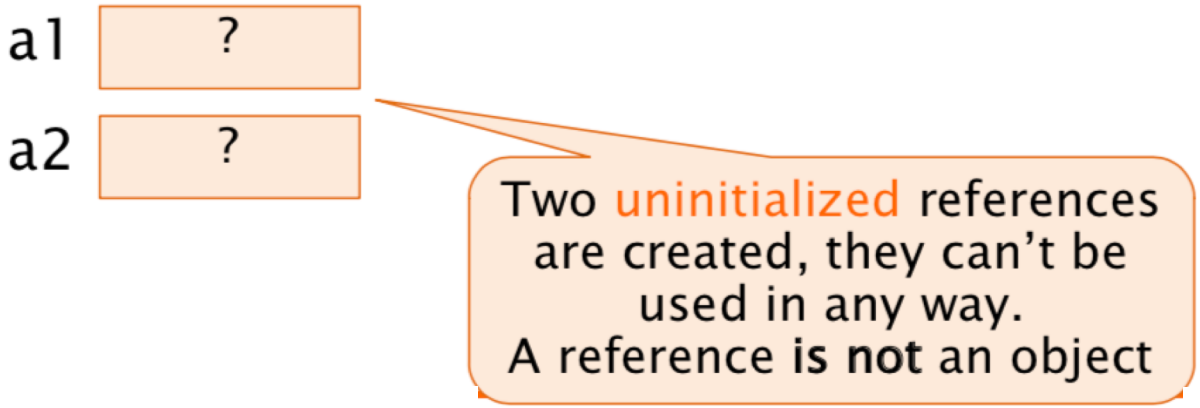
- An object is identified by:
  - its class, which defines its structure (attributes and methods)
  - its state (values of attributes)
  - an internal unique identifier
- Zero, one or more reference can point to the same object
  - Aliasing

Objects

```
class Car {
   String color;
   void paint(){
      color = “white”;
   }
   void paint(String newCol) {
      color = newCol;
   }
}
Car a1, a2;
a1 = new Car();
a1.paint(“green”);
a2 = new Car();
```

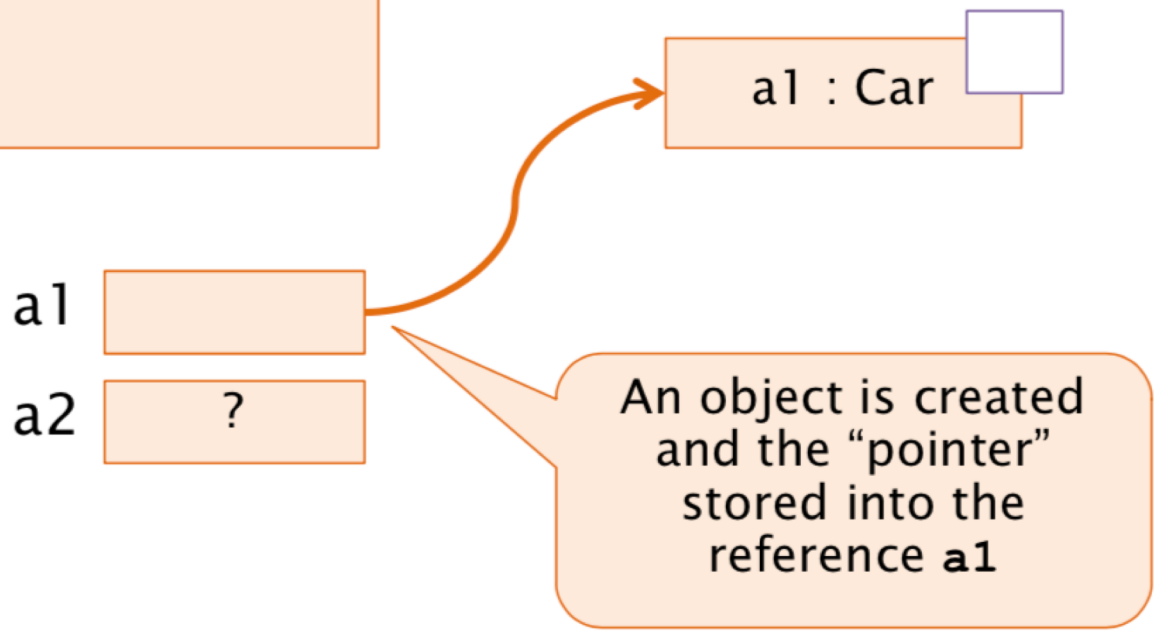
Objects and references

```
Car a1, a2;
a1 = new Car();
a1.paint("yellow");
a2 = a1;
a1 = null;
a2 = null;
```



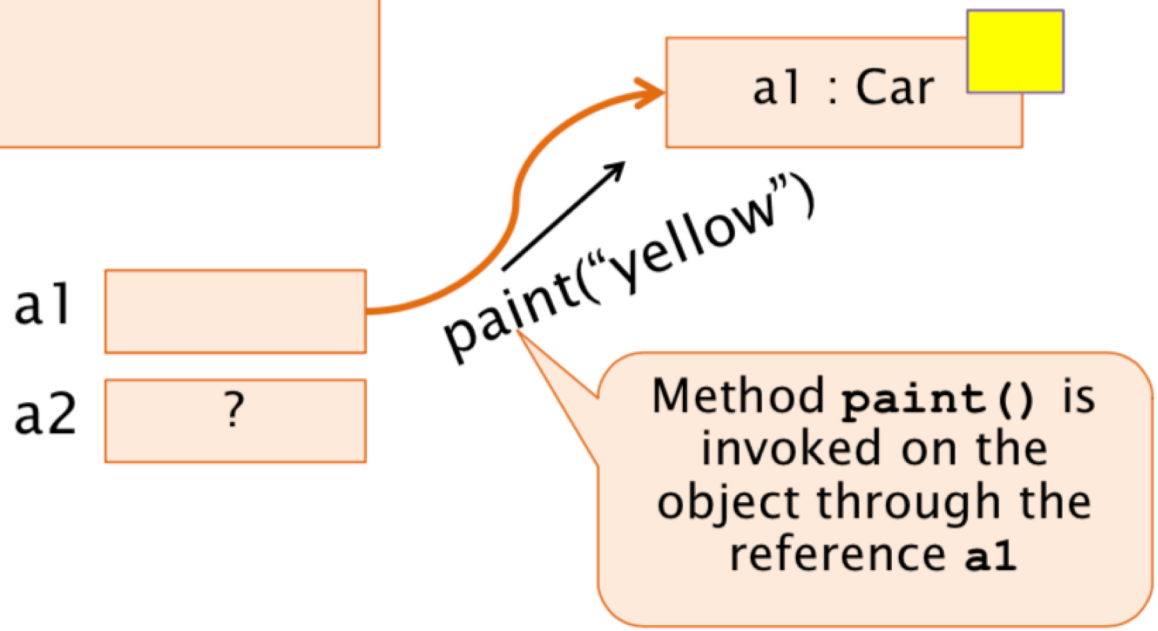
Objects and references

```
Car a1, a2;
a1 = new Car();
a1.paint("yellow");
a2 = a1;
a1 = null;
a2 = null;
```



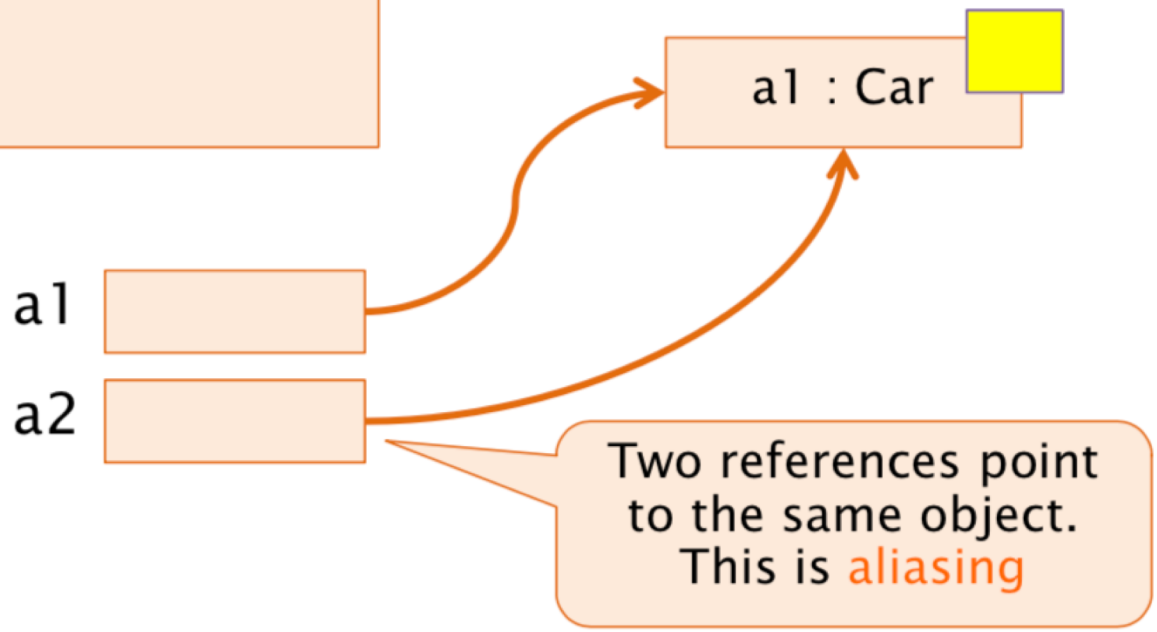
Objects and references

```
Car a1, a2;
a1 = new Car();
a1.paint("yellow");
a2 = a1;
a1 = null;
a2 = null;
```



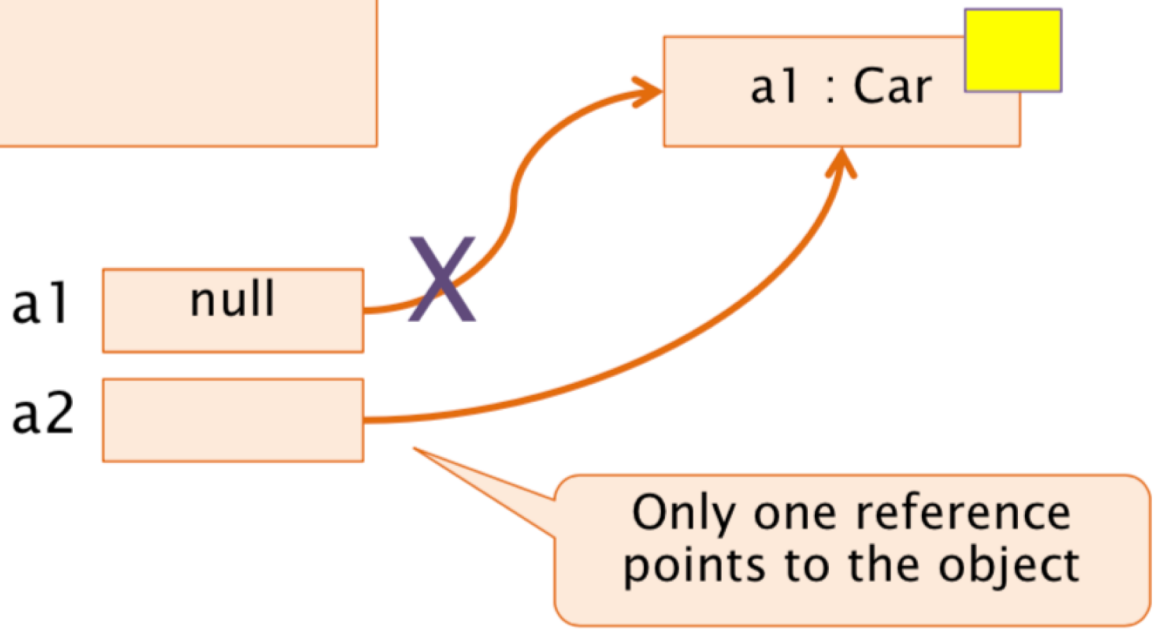
Objects and references

```
Car a1, a2;
a1 = new Car();
a1.paint("yellow");
a2 = a1;
a1 = null;
a2 = null;
```



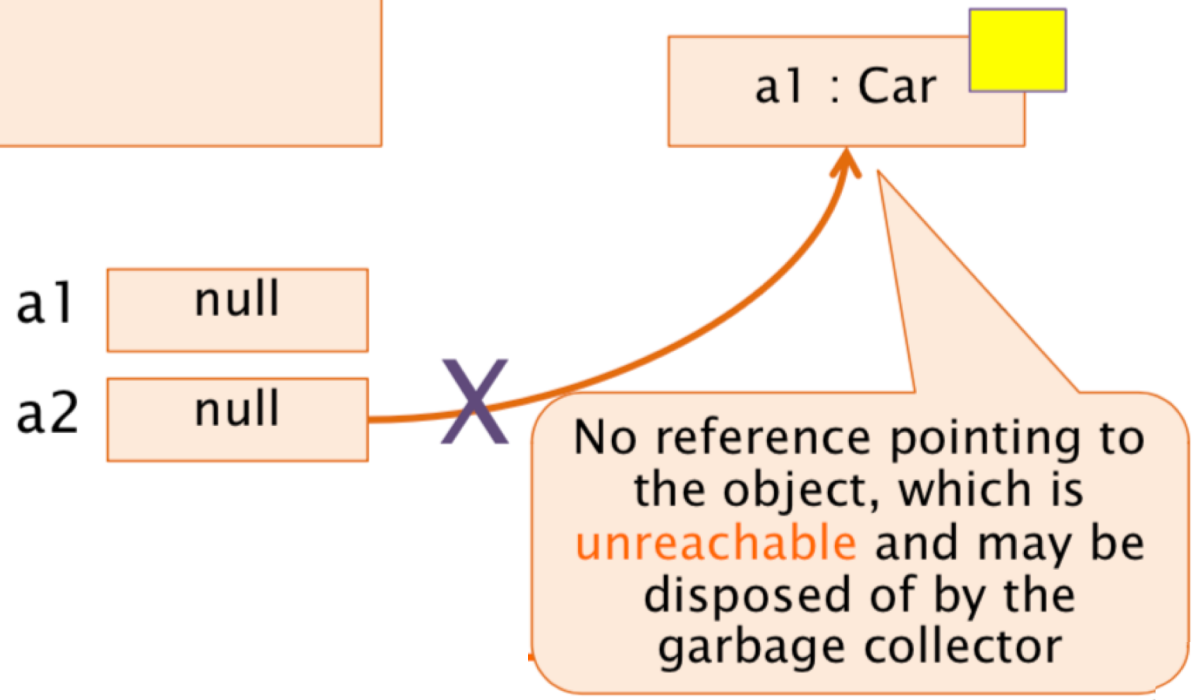
Objects and references

```
Car a1, a2;
a1 = new Car();
a1.paint("yellow");
a2 = a1;
a1 = null;
a2 = null;
```



Objects and references

```
Car a1, a2;
a1 = new Car();
a1.paint("yellow");
a2 = a1;
a1 = null;
a2 = null;
```



Objects creation

- Creation of an object is achieved using the keyword new
- It returns a reference to the piece of memory containing the created object

```
Motorcycle m = new Motorcycle();
```

The keyword **`new`**

- Creates a new instance of the specific class, and it allocates the necessary memory (in the so-called heap)
- Calls the constructor method of the object (a special method without return type and with the same name of the class)
- Returns a reference to the new object created
- Constructor can have parameters, e.g.
  - String s = new String("ABC");

Heap

- It is a part of the memory used by an executing program to store data dynamically created at run-time

- In C: **`malloc`**, **`calloc`** and **`free`**
  - Instances of types in static memory or in heap
- In Java: **`new`**
  - Instances (objects) are always in the heap

Constructor

- Constructor is a special method containing the operations (e.g., initialization of attributes, etc.) that one wants to be executes on each object as soon as it is created
- Attributes are always initialized
- If no constructor at all is declared, a default one (with no parameters) is defined
- Overloading of constructors is often used

Constructor

- Attributes are always initialized before any possible constructor is run
  - Attributes are initialized with default values
    - Numeric: `0` (zero)
    - Boolean: **`false`**
    - Reference: **`null`**
- Return type must not be declared for constructors
  - If present, it is considered as a method and it is not invoked upon instantiation

Constructors with overloading

```
class Car { // …
//   Default constructor, creates a red Ferrari
  public Car(){
     color = "red";
     brand = "Ferrari";
   }
// Constructor accepting the brand only     */
  public Car(String carBrand){
     color = "white”;
      brand = carBrand;
   }
// Constructor accepting the brand and the color
  public Car(String carBrand, String carColor){
     color = carColor;
     brand = carBrand;
   }
}
```

Destruction of objects

- Memory release, in Java, is no longer a programmer's concern
  - Managed memory language
- Before the object is really destroyed, the method **`finalize`**, if defined, is invoked:

```
public void finalize()
```

Current object – a.k.a `this`

- During the execution of a method it is possible to refer to the current object using the keyword `this`
  - The object upon which the method has been invoked
- This makes no sense within methods that have not been invoked on an object
  - E.g., the `main` method

Method invocation

- A method is invoked using dotted notation

```
objectReference.Method(parameters)
```

- Example:

```
Car a = new Car();
a.turnOn();
a.paint("Blue");
```

Note

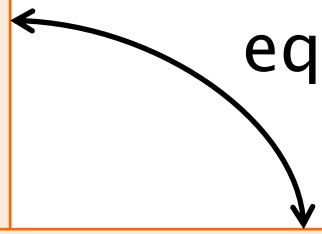
- If a method is invoked within another method of the same object, dotted notation is not mandatory

```
class Book {
   int pages;
   void readPage(int n) { …       }
   void readAll() {
         for(int i=0; i<pages; i++)
               this.readPage(i);
         }
}
```

Note

- In such cases **this** is implied
- It is not mandatory

```
class Book {
   int pages;
   void readPage(int n){…}
   void readAll() {
      for(…)
         readPage(i);
   }
}
```



equivalent

```
   void readAll() {
      for(…)
         this.readPage(i);
   }
```

Access to attributes

- Dotted notation

objectReference.attribute

  - Reference is used like a normal variable

```
Car a = new Car();
a.color = "Blue"; //what's wrong here?
boolean x = a.turnedOn;
```

Access to attributes

- Methods accessing attributes of the same object do not need to use the object reference

```
class Car {
   String color;

   void paint(){
      color = “green”;
      // color refers to current obj
   }
  }
```

Using `this` for attributes

- The use of `this` is not mandatory
- It can be useful in methods to disambiguate object's attributes from local variables

```
class Car{
   String color;
   ...
   void paint (String color) {
      this.color = color;
   }
}
```

Chaining dotted notations

- Dotted notations can be combined

```
System.out.println("Hello world!");
```

  - **`System`** is a class in package **`java.lang`**
  - **`out`** is a (static) attribute of **`System`** referencing an object of type **`PrintStream`** (representing the standard output)
  - **`println()`** is a method of **`PrintStream`** which prints a text line followed by a new-line
- Method invocations can be chained as well

Operations on references

- Only the relational operators == and != are defined
  - Note: the equality condition is evaluated on the (values of) the references and NOT on (the values of) the objects themselves!
  - The relational operators tell whether the references points to the same object in memory
- Dotted notation is applicable to object references
- There is NO pointer arithmetic

Array

SOftEng
http://softeng.polito.it

Array

- An array is an ordered sequence of variables of the same type which are accessed through an index
- Can contain both primitive types or object references (but no object values)
- Array dimension can be defined at run-time, during object creation (cannot change afterwards)

Array declaration

- An array reference can be declared with one of these equivalent syntaxes

```
int[] a;
int a[];
```

- In Java an array is an object (it is stored in the heap)
- Array declaration allocates memory space for a reference, whose default value is null

Array creation

- Using the new operator…

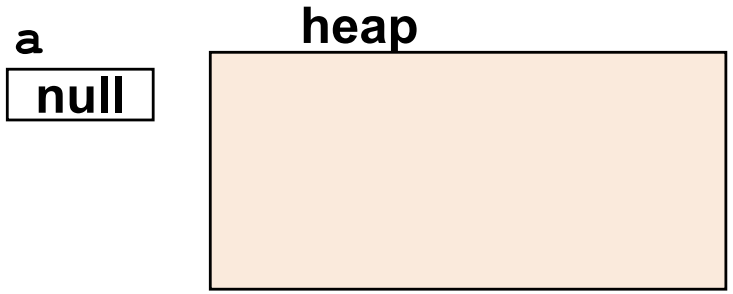
```
int[] a;
a = new int[10];
String[] s = new String[5];
```

- …or using static initialization, filling the array with values

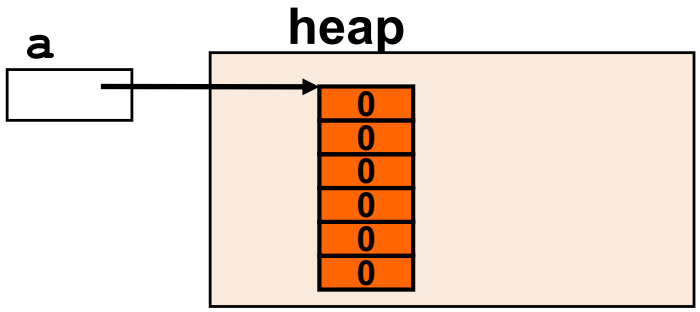
```
int[] primes = {2,3,5,7,11,13};
Person[] p = { new Person("John"),
               new Person("Susan") };
```

Example – primitive types

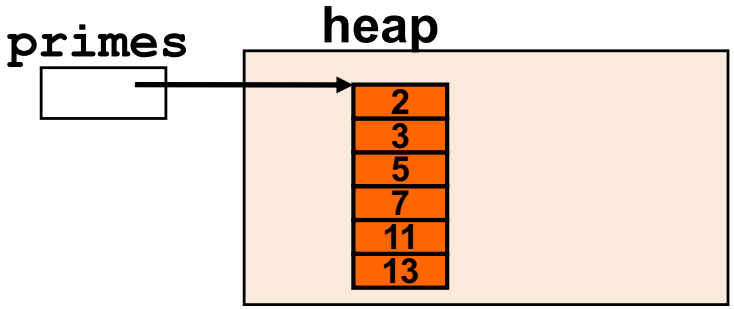
```
int[] a;
```




```
a = new int[6];
```




```
int[] primes =
  {2,3,5,7,11,13};
```



Operations on arrays

- Elements are selected with brackets [ ] (C-like)
  - But Java performs bounds checking

- Array length (number of elements) is given by attribute length

```
for (int i=0; i < a.length; i++)
    a[i] = i;
```

Operations on arrays

- An array reference is not a pointer to the first element of the array
- It is a pointer to the array object

- Arithmetic on pointers does not exist in Java

For each

- New loop construct:

```
for( Type var : set_expression )
```

  - Very compact notation
  - *set_expression* can be
    - either an array
    - a class implementing `Iterable`
  - The compiler can generate automatically loop with correct indexes
    - Less error prone

For each – example

- Example:

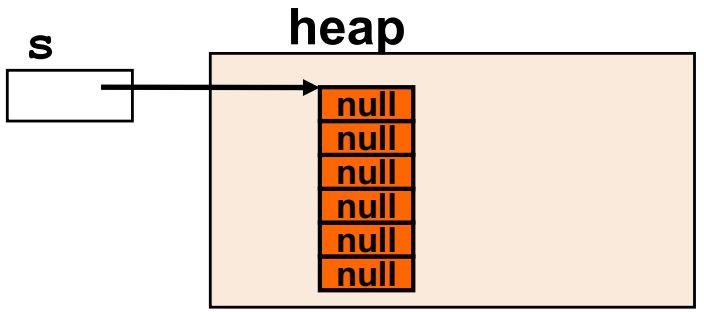
```
for(String arg : args){
   //...
}
```

  - is equivalent to

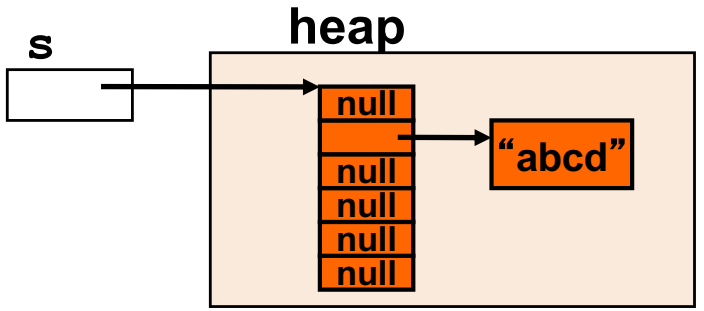
```
for(int i=0; i<args.length;++i){
   String arg= args[i];
   //...
}
```

Example – object references

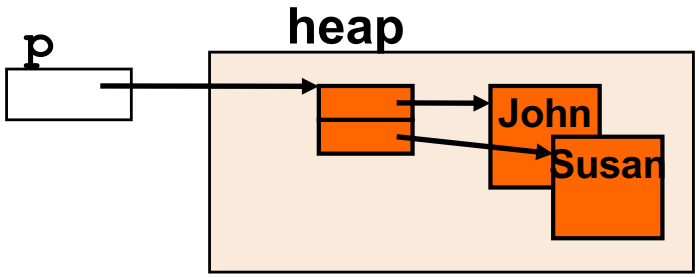
```
String[] s = new
  String[6];
```




```
s[1] = new
  String("abcd");
```



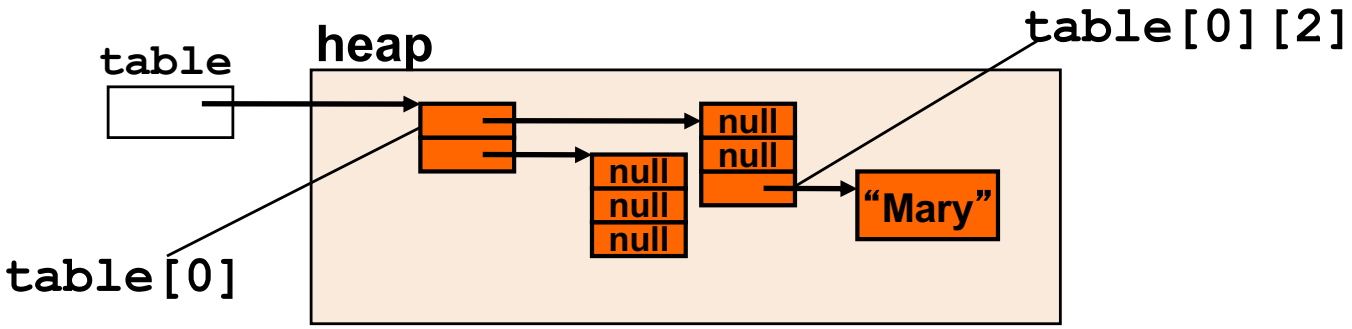

```
Person[] p =
{new Person("John") ,
 new Person("Susan")};
```



Multidimensional array

- Implemented as array of arrays

```
Person[][] table = new Person[2][3];
table[0][2] = new Person("Mary");
```



Rows and columns

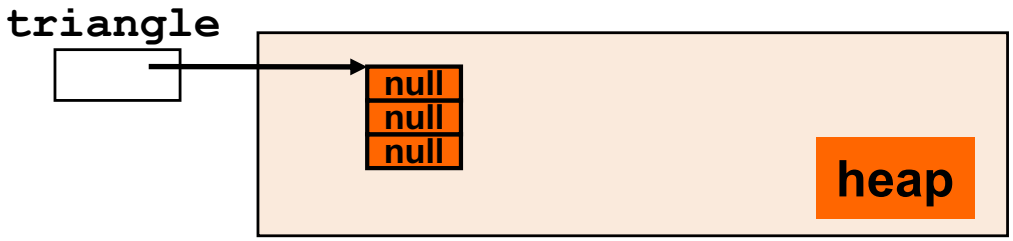
- Since rows are not stored in adjacent positions in memory, they can be easily exchanged

```
double[][] balance = new double[5][6];
...
double[] temp = balance[i];
balance[i] = balance[j];
balance[j] = temp;
```

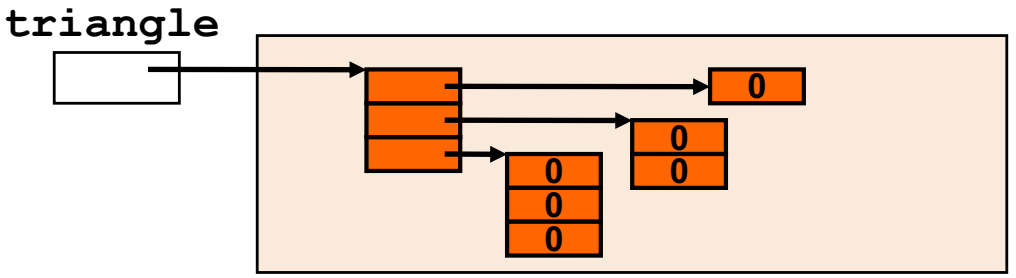
Rows with different length

- A matrix (bidimensional array) is indeed an array of arrays

```
int[][] triangle = new int[3][]
```




```
for (int i=0; i< triangle.length; i++)
    triangle[i] = new int[i+1];
```



Scope and encapsulation

SOftEng
http://softeng.polito.it

Example

- Laundry machine, design1
  - Commands:
    - time, temperature, amount of soap
  - Different values depending if you wash cotton or wool, ….
- Laundry machine, design2
  - Commands:
    - key C for cotton, W for wool, Key D for knitted robes

Example

- Washing machine, design3
  - Command:
    - Wash!
  - Insert clothes, and the washing machine automatically selects the correct program
- Hence, there are different solutions with different level of granularity / abstraction

Motivation

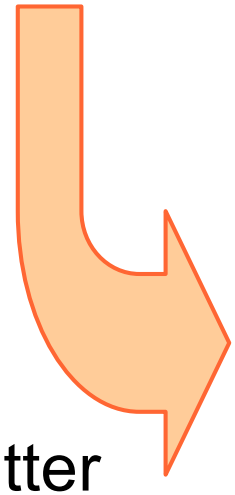
- Modularity = cut-down inter-components interaction
- Information hiding = identifying and delegating responsibilities to components
  - Components = classes
  - Interaction = read/write attributes
  - Interaction = calling a method
- Heuristics
  - Attributes invisible outside the class
  - Visible methods are the ones that can be invoked from outside the class

Scope and syntax

- Visibility modifiers
  - Applicable to members of a class
- `private`
  - Attribute/method visible and accessible from instances of the same class only
- `public`
  - Attribute / method accessible from everywhere
- Other modifiers (later)

Information hiding

```
class Car {
String color;
}
Car a = new Car();
a.color = “white”;  // ok
```



better

```
class Car {
  private String color;
  public void paint(String color)
    {this.color = color;}
}
Car a = new Car();
a.color = “white”;  // error
a.paint(“green”);   // ok
```

SOftEng
http://softeng.polito.it

Information hiding

```
class Car{
      private String color;
      public void paint();
}
```

no

yes

```
class B {
   public void f1(){
      ...
   };
}
```

Access

	Method in the same class	Method of another class
Private (attribute/ method)	yes	no
Public	yes	yes

Getters and setters

- Methods used to read/write a private attribute
- Allow to better control, in a single point, each write access to a private field

```
public String getColor() {
    return color;
}
public void setColor(String newColor) {
    color = newColor;
}
```

Example without getter/setter

```
public class Student {
   public String first;
   public String last;
   public int id;
   public Student(…){…}
}
```

```
public class Exam {
   public int grade;
   public Student student;
   public Exam(…){…}
}
```

Example without getter/setter

```
class StudentExample {
  public static void main(String[] args) {
     // defines a student and her exams
     // lists all student's exams
     Student s=new Student("Alice","Green",1234);
     Exam e = new Exam(30);
     e.student = s;
     // print vote
     System.out.println(e.grade);
     // print student
     System.out.println(e.student.last);
  }
}
```

Example with getter/setter

```
class StudentExample {
  public static void main(String[] args) {
    Student s = new Student("Alice", "Green",
                            1234);
    Exam e = new Exam(30);

    e.setStudent(s);
    // prints its values and asks students to
    // print their data
    e.print();
  }
}
```

Example with getter/setter

```
public class Student {

  private String first;
  private String last;
  private int id;

  public String toString() {
   return first + " " +
               last  + " " +
               id;
  }
}
```

Example with getter/setter

```
public class Exam {
    private int grade;
    private Student student;

    public void print() {
    System.out.println("Student " +
        student.toString() + "got " + grade);
    }

    public void setStudent(Student s) {
    this.student =s;
    }
}
```

Getters & setters vs. public fields

- Getter
  - Allow changing the internal representation without affecting it
    - E.g., can perform type conversion
- Setter
  - Allow performing checks before modifying the attribute
    - E.g., validity of values, authorization

Packages

SOftEng
http://softeng.polito.it

Motivation

- Class is a better element of modularization than a procedure
- But it is still “small”, when compared to the side of an application
- For the purpose of code organization and structuring, Java provides the package feature

Package

- A package is a logic set of class definitions
- These classes are made of several files, all stored in the same directory (folder)
- Each package defines a new scope (i.e., it puts bounds to visibility of names)
- It is then possible to use same class names in different package without name-conflicts

Package name

- A package is identified by a name with a hierarchic structure (fully qualified name)
  - E.g., **`java.lang`** (**`String`**, **`System`**, …)
- Conventions to create unique names
  - Internet name in reverse order
  - **`it.polito.myPackage`**

Examples

- **`java.awt`**
  - **`Window`**
  - **`Button`**
  - **`Menu`**

- **`java.awt.event`** (sub-package)
  - **`MouseEvent`**
  - **`KeyEvent`**

Creation and usage

- Declaration:
  - **`package`** statement at the beginning of each class file

```
package packageName;
```

- Usage:
  - **`import`** statement at the beginning of class file (where needed)

```
import packageName.className;
import java.awt.*;
```

Import single Class (class name is in scope)

Import all classes but not the sub packages

Access to a class in a package

- Referring to a method/class of a package

```
int i = myPackage.Console.readInt()
```

- If two packages define a class with the same name, they cannot be both imported
- If you need both classes you have to use one of them with its fully-qualified name:

```
import java.sql.Date;
Date d1; // java.sql.Date
java.util.Date d2 = new java.util.Date();
```

Default package

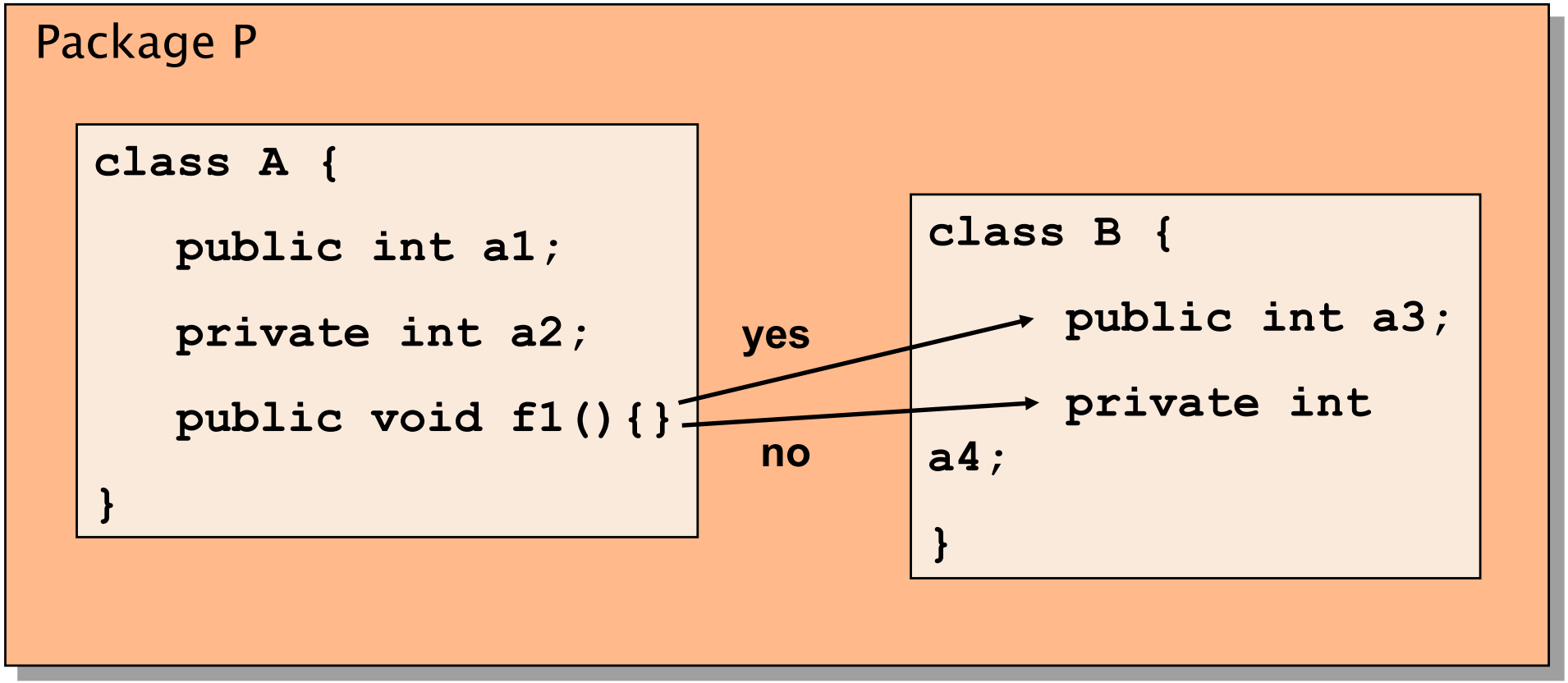
- When no package is specified, the class belongs to the default package
  - Default package has no name
- Classes in the default package cannot be accessed by classes residing in other packages
- Use of default package is a bad practice and it is discouraged

Package and scope

- Scope rules also apply to packages
- The “interface” of a package is the set of public classes contained in the package

- Hints
  - Consider a package as an entity of modularization
  - Minimize the number of classes, attributes, methods visible outside the package

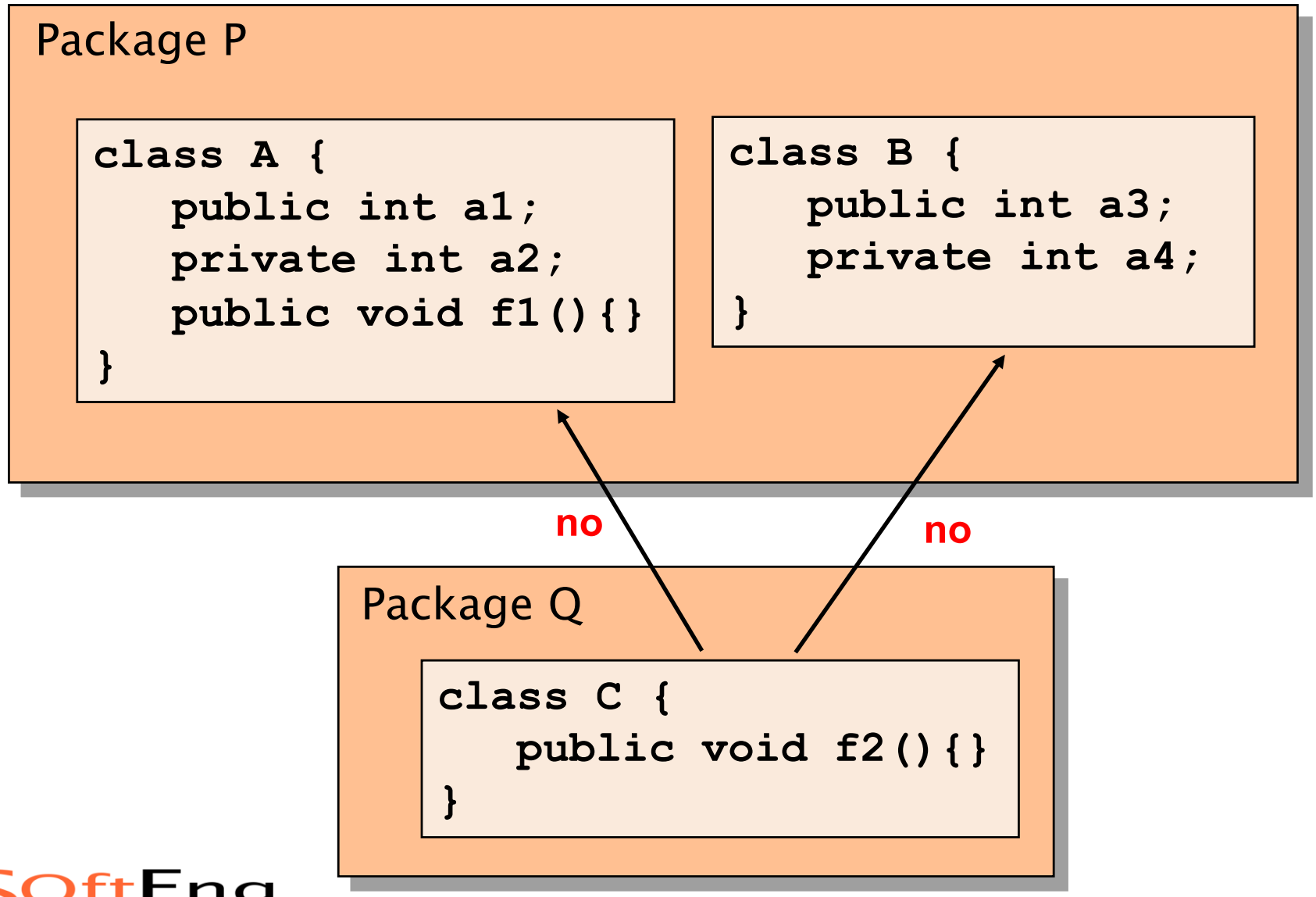
Package visibility



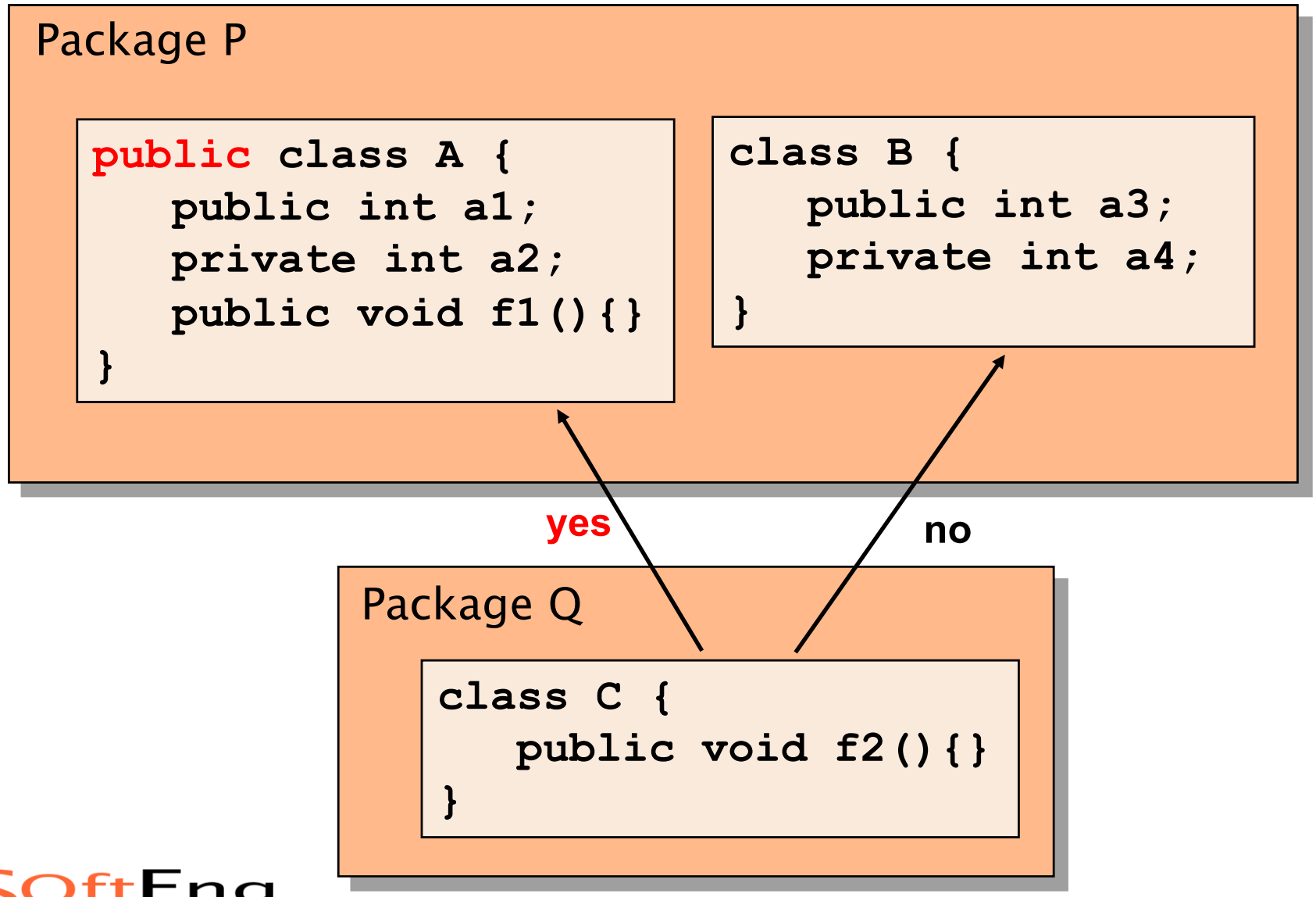
Visibility w/ multiple packages

- **`public class A { }`**
  - Class and public members of **`A`** visible from outside the package
- **`class A { }`**
  - Class and members of **`A`** not visible from outside the package (package visibility)
- **`private class A { }`**
  - Illegal: class and members of **`A`** visible to themselves only

Multiple packages



Multiple packages



Access rules

	Method of the same class	Method of other class in the same package	Method of other class in other package
Private attribute/ method	Yes	No	No
Package attribute / method	Yes	Yes	No
Public attribute / method in package class	Yes	Yes	No
Public attribute / method in public class	Yes	Yes	Yes

Strings

SOftEng
http://softeng.polito.it

String

- No primitive type to represent strings
- String literal is a quoted text
- In C
  - **`char s[] = “literal”`**
  - Equivalence between string and char arrays
- In Java
  - **`char[] != String`**
  - String class in **`java.lang`** package

String and StringBuffer

- Class **`String`** (**`java.lang`**)
  - Not modifiable / Immutable
- Class **`StringBuffer`** (**`java.lang`**)
  - Modifiable / Mutable

```
String s = new String("literal");

StringBuffer sb = new
  StringBuffer("literal");
```

Operator +

- It is used to concatenate two strings

```
“This string” + ” is made by two strings”
```

- Works also with other types (everything is automatically converted to the corresponding string)

```
System.out.println(”pi = ” + 3.14);
System.out.println(”x = ” + x);
```

String

- `int length()`
  - Returns string length
- `boolean equals(String s)`
  - Compares the content of two strings

```
String s1, s2;
s1 = new String("First string");
s2 = new String("First string");
System.out.println(s1);
System.out.println("Length of s1:"+ s1.length());
if (s1.equals(s2))  // true
if (s1 == s2) // false, in principle
```

String

- `String toUpperCase()`
- `String toLowerCase()`
- `String subString(int startIndex)`
- `int indexOf(String str)`
  - Returns the index of the first occurrence of *str*
- `String concat(String str)`
- `int compareTo(String str)`
- `String valueOf(int)`
  - Converts `int` in a `String` – available for all primitive types (static method)

SOftEng
http://softeng.polito.it

String

- **`String subString(int startIndex)`**
  - **`String s = "Human";`**
  - **`s.subString(2) returns "man"`**
- **`String subString(int start, int end)`**
  - Char '**`start`**' included, '**`end`**' excluded
  - **`String s = "Greatest";`**
  - **`s.subString(0,5)`** returns **`"Great"`**
- **`int indexOf(String str)`**
  - Returns the index of the first occurrence of *str*
- **`int lastIndexOf(String str)`**
  - The same as before but search starts from the end

StringBuffer

- Represents a string of characters
- It is mutable and allows operations that modify the content
- Can be converted to the corresponding String using the method **`toString()`**

StringBuffer

- **`StringBuffer append(String str)`**
  - Inserts str at the end of string
- **`StringBuffer insert(int offset, String str)`**
  - Inserts str starting from offset position
- **`StringBuffer delete(int start, int end)`**
  - Deletes character from start to end (excluded)
- **`StringBuffer reverse()`**
  - Reverses the sequence of characters

Performance issues

```
String s="";


for(i=0; i<N; ++i){
   s += i;
}
```

12 sec
N = 100k

```
StringBuffer sb=
 new StringBuffer();
for(i=0; i<N; ++i){
   sb.append(i);
}
```

2 ms
N = 100k

Three order of magnitudes difference!

Wrapper classes for built-in types

SOftEng
http://softeng.polito.it

Motivation

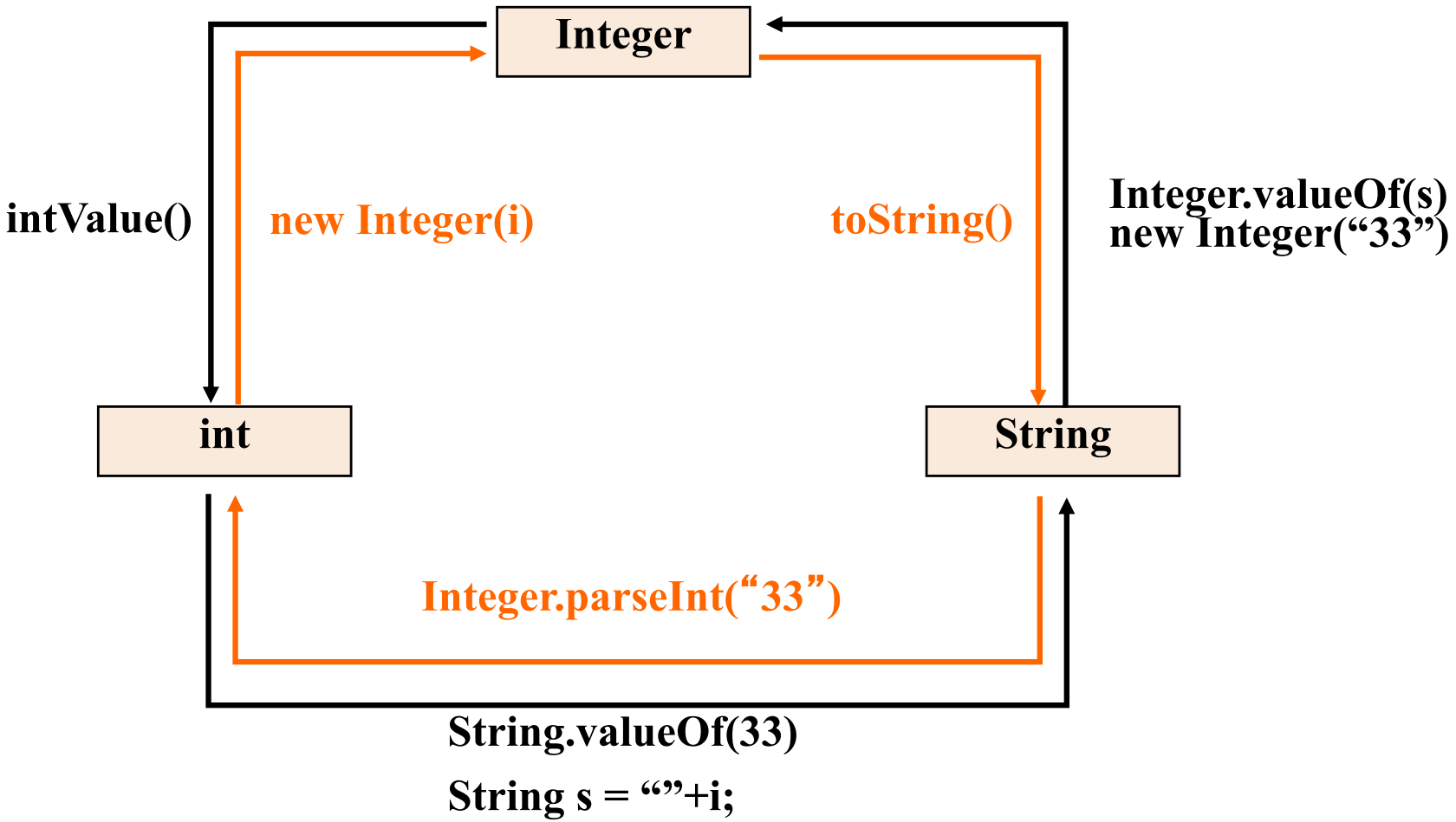
- In an ideal OO world, there are only classes and objects
- For the sake of efficiency, Java uses primitive types (`int`, `float`, etc.)
- Wrapper classes are "object versions" of the primitive types
- They define conversion operations between different types

Wrapper classes

Defined in `java.lang` package

Primitive type	Wrapper Class
`boolean`	`Boolean`
`char`	`Character`
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`void`	`Void`

Conversions



Example

```
Integer obj = new Integer(88);
String s = obj.toString();
int i = obj.intValue();

int j = Integer.parseInt("99");
int k =(new Integer(99)).intValue();
```

Autoboxing

- Since Java 5, an automatic conversion between primitive types and wrapper classes (autoboxing) is performed

```
Integer i= new Integer(2); int j;
j = i + 5;
  //instead of:
j = i.intValue()+5;
i = j + 2;
   //instead of:
i = new Integer(j+2);
```

Character

- Wrapper class encapsulating a single character
- Utility methods
  - `isLetter()`
  - `isDigit()`
  - `isSpaceChar()`
- Utility methods for conversions
  - `toUpper()`
  - `toLower()`

Static attributes and methods

SOftEng
http://softeng.polito.it

Static attributes: what

- Represent properties which are common to all instances of a class
  - A single copy of a static attribute is shared by all instances of the class
  - Sometimes called class attributes (class variables) as opposed to instance attributes
  - Static attributes exists before any object is created
  - A change performed by any object is visible to all instances at once
- They are defined with the `static` modifier

Static attributes: why/when

- Used to keep
  - a shared property (a count of created instances, a pool of all instances, etc.)
  - a common constant value

```
class Car {
   static int countBuiltCars = 0;
   public Car(){
      countBuiltCars++;
   }
}
```

Static methods

- Static methods are not linked to any instance
- They are defined with the **static** modifier
- Used to implement "functions"

```
public class HelloWorld {
   public static void main (String args[]) {
      System.out.println("Hello World!");
   }
}
```

```
public class Utility {
   public static double inverse(double n){
      return 1 / n;
   }
}
```

Static members access

- The name of the class is used to access the member:

```
Car.countCountBuiltCars
Utility.inverse(10);
```

- It is possible to import all static items:

```
import static package.Utility.*;
```

  - Then all static members are accessible without specifying the class name
    - Note: impossible if class is in default package

Example: `System` class

- Provides several utility functions and objects e.g.
  - **`static long currentTimeMillis()`**
    - Current system time in milliseconds
  - **`static void exit(int code)`**
    - Terminates the execution of the JVM
  - **`static final PrintStream out`**
    - Standard output stream

Final attributes

- An attribute declared as **`final`**:
  - cannot be changed after object construction
  - can be initialized inline or by the constructor

```
class Student {
   final int years=3;
   final String id;
   public Student(String id){
      this.id = id; }
}
```

Constants

- Use the final static modifiers
  - **`final`** implies not modifiable
  - **`static`** implies non redundant

```
final static float PI_GRECO = 3.14;

PI = 16.0;          // ERROR, no changes
final int SIZE;     // ERROR, init missing
```

- All uppercase (coding conventions)

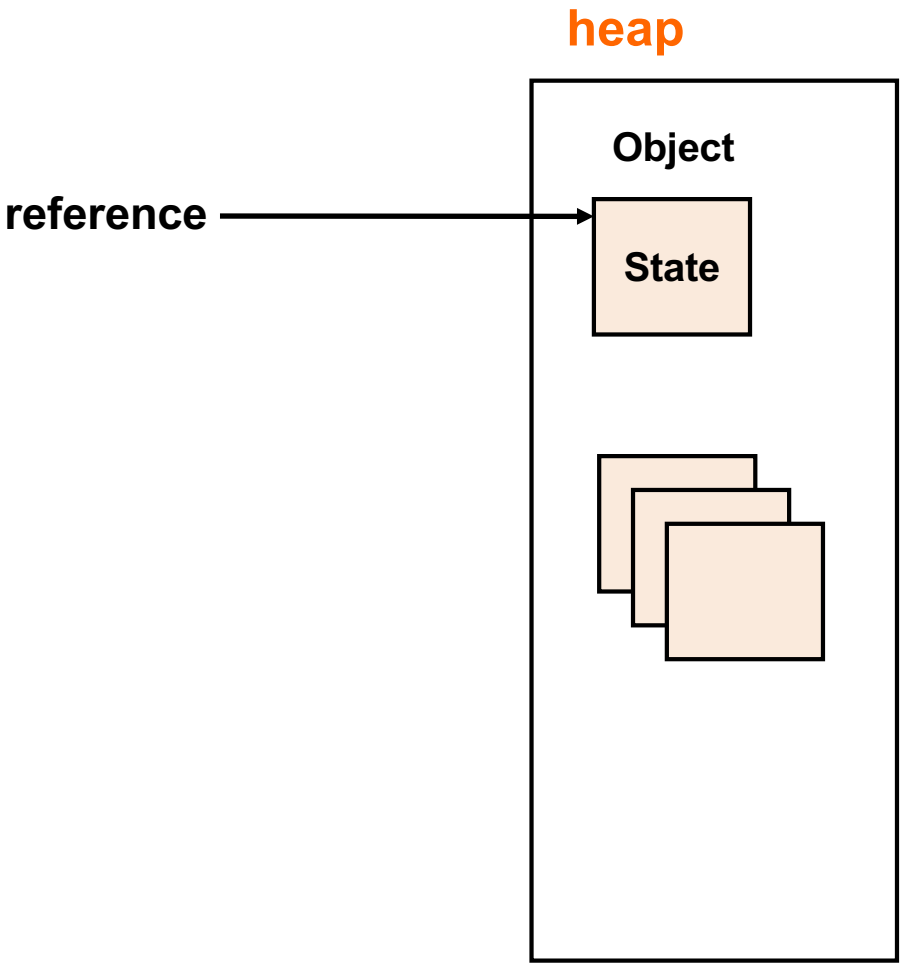
Memory management

SOftEng
http://softeng.polito.it

Memory types

- Depending on the kind of elements they host:
  - Static memory: elements living for all the execution of a program (class definitions, static variables)
  - Heap (dynamic memory): elements created at run-time (with “new”)
  - Stack: elements created in a code block (local variables and method parameters)

Objects are stored in the heap



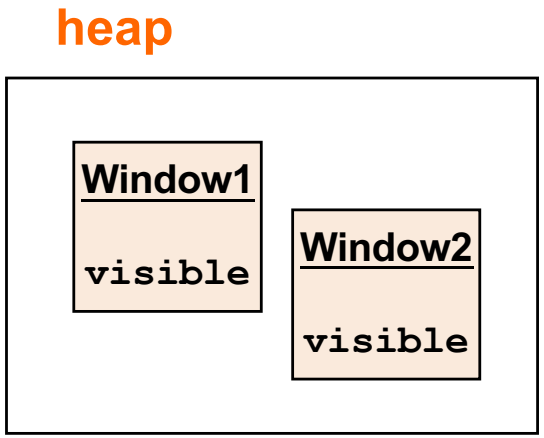
Types of variables

- Instance variables (or fields or attributes)
  - Stored within objects (in the heap)
- Local variables
  - Stored in the stack
- Static variables
  - Stored in static memory

Instance variables

- Declared within a class (attributes)

```
class Window {
    boolean visible;
    …
}
```



- There is a different copy in each instance of the class
- Created/initialized whenever a new instance of a class is created

Local variables

- Declared within a method or a code block
- Created at the beginning of the code block where they are declared
- Automatically destroyed at the end of the block

```
class Window {
    …
    void resize () {
        int i;
        for (i=0; i<5; i++) { … }
    } // here i is destroyed
}
```

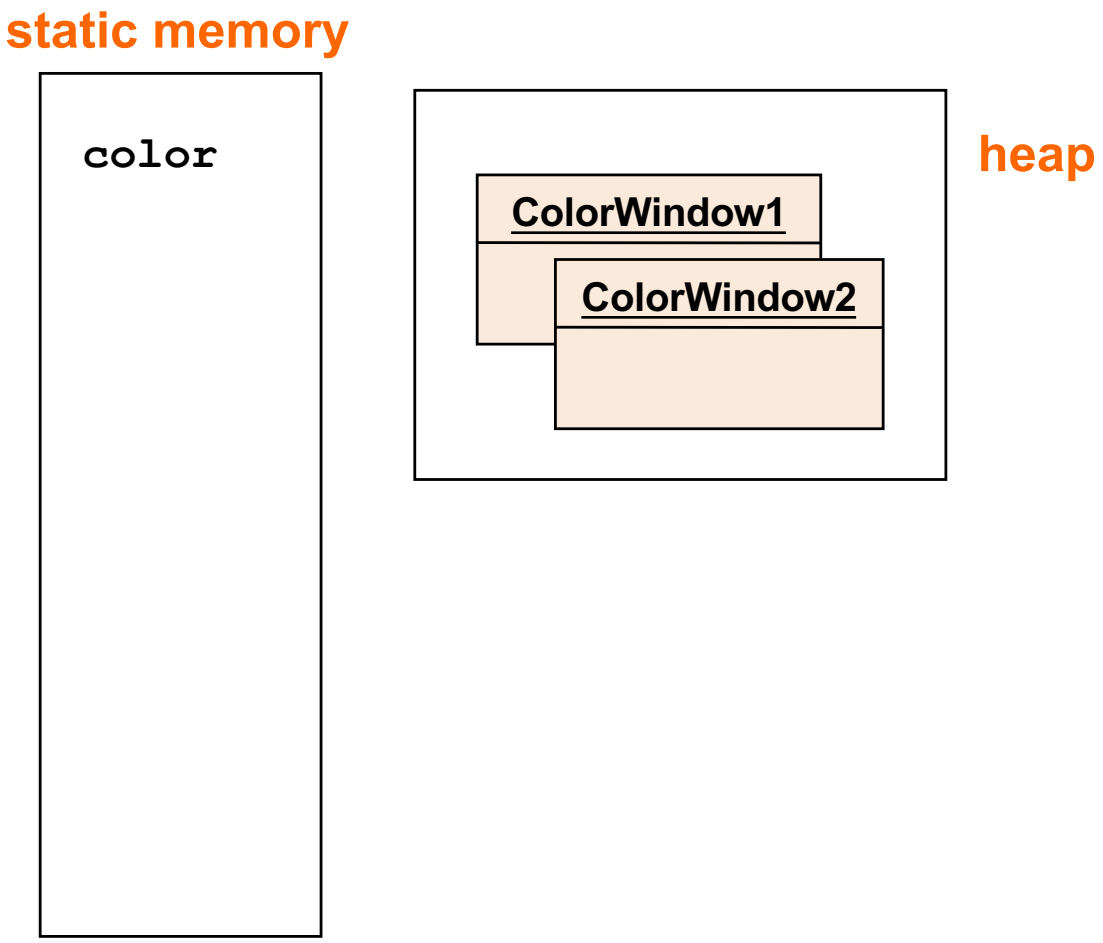
Static variables

- Declared in classes or methods with **`static`** modifier

```
class ColorWindow {
  static String color;
   …
}
```

- Only ONE copy
  - Associated to a class: a.k.a. class variables
- Created/initialized during class-loading in memory

Static variables



Object destruction

- It is not made explicitly but it is made by the JVM garbage collector when releasing the object's memory
  - Programmer must not worry about objects destruction (although there is no guarantee an object will be ever explicitly released)

Garbage collector

- Is a component of the JVM that cleans the heap from “dead” objects
- It periodically analyses references and objects in memory
- … and deallocates objects with no active references

Wrap-up

- Java syntax is very similar to that of C
- New primitive type: **`boolean`**
- Objects are accessed through references
  - References are disguised pointers!
- Reference definition and object creation are separate operations
- Different scopes and visibility levels
- Arrays are objects
- Wrapper types encapsulate primitive types