

# Java Basic Features

---

## Object-Oriented Programming



**SoftEng**  
<http://softeng.polito.it>

Version 2.5.0  
© Maurizio Morisio, Marco Torchiano, 2025








This work is licensed under the Creative Commons Attribution–NonCommercial–NoDerivatives 4.0 International License.

To view a copy of this license, visit

<http://creativecommons.org/licenses/by-nc-nd/4.0/>.

You are free: to copy, distribute, display, and perform the work

Under the following conditions:

-  **Attribution.** You must attribute the work in the manner specified by the author or licensor.
-  **Non-commercial.** You may not use this work for commercial purposes.
-  **No Derivative Works.** You may not alter, transform, or build upon this work.
  - For any reuse or distribution, you must make clear to others the license terms of this work.
  - Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

# Learning objectives

---

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

---

# JAVA SYNTAX

---

# 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 a block

```
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

---

- Strings can be used as cases values

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

*Note:* Compiler generates more efficient bytecode from switch using String objects than from chained if-then-else statements.

---



# Boolean

---

- Java has an explicit type (`boolean`) to represent logical values (`true`, `false`)
  - Conditional constructs require boolean conditions
    - ♦ Illegal to evaluate integer condition  
`int x = 7; if(x) {...} //NO`
    - ♦ Use relational operators `if (x != 0)`
    - ♦ Avoids 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 whole object

# Elements in an OO program

---

Structural elements  
(types)  
(compile time)

- Class
- Primitive type

---

Dynamic elements  
(instances)  
(run time)

- Reference
- Variable

# Classes and primitive types

---

## Type

- Class

```
class Exam {}
```

- type primitive

```
int, char,  
float
```

---

## Instance

- Variable of type reference

```
Exam e;
```

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

- Variable of type primitive

```
int i;
```

---

# PRIMITIVE TYPES

---

# Primitive type

---

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

`int i;`

0
---

# Primitive types

Type	Size	Encoding
<b>boolean</b>	1 bit	–
<b>char</b>	16 bits	Unicode UTF16
<b>byte</b>	8 bits	Signed integer 2C
<b>short</b>	16 bits	Signed integer 2C
<b>int</b>	32 bits	Signed integer 2C
<b>long</b>	64 bits	Signed integer 2C
<b>float</b>	32 bits	IEEE 754 sp
<b>double</b>	64 bits	IEEE 754 dp
<b>void</b>	–	

Logical  
size !=  
memory  
occupation

# 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`



# Operators (integer and f.p.)

---

- Operators follow C syntax:
  - ◆ arithmetical    +    -    \*    /    %
  - ◆ relational    ==    !=    >    <    >=    <=
  - ◆ bitwise (integers) &    |    ^    <<    >>    ~
  - ◆ 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 a key difference from C
  - ◆ Relational operators return **boolean** values

---

# CLASSES AND OBJECTS

---

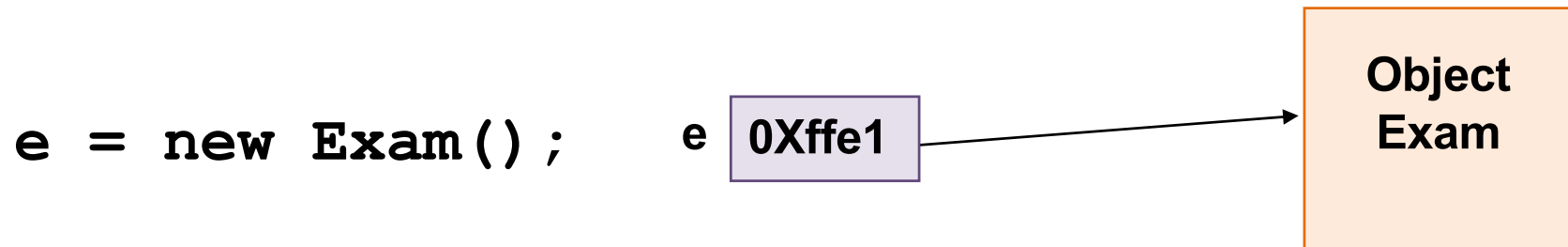
# Class

---

- Defined by developer (e.g., **Exam**) or in the Java runtime libraries (e.g., **String**)
- The declaration

**Exam e;**                      e null

- allocates memory for the *reference* (‘pointer’)  
...and *sometimes* it initializes it with **null**
- Allocation and initialization of the *object* value are made later by **new** and constructor

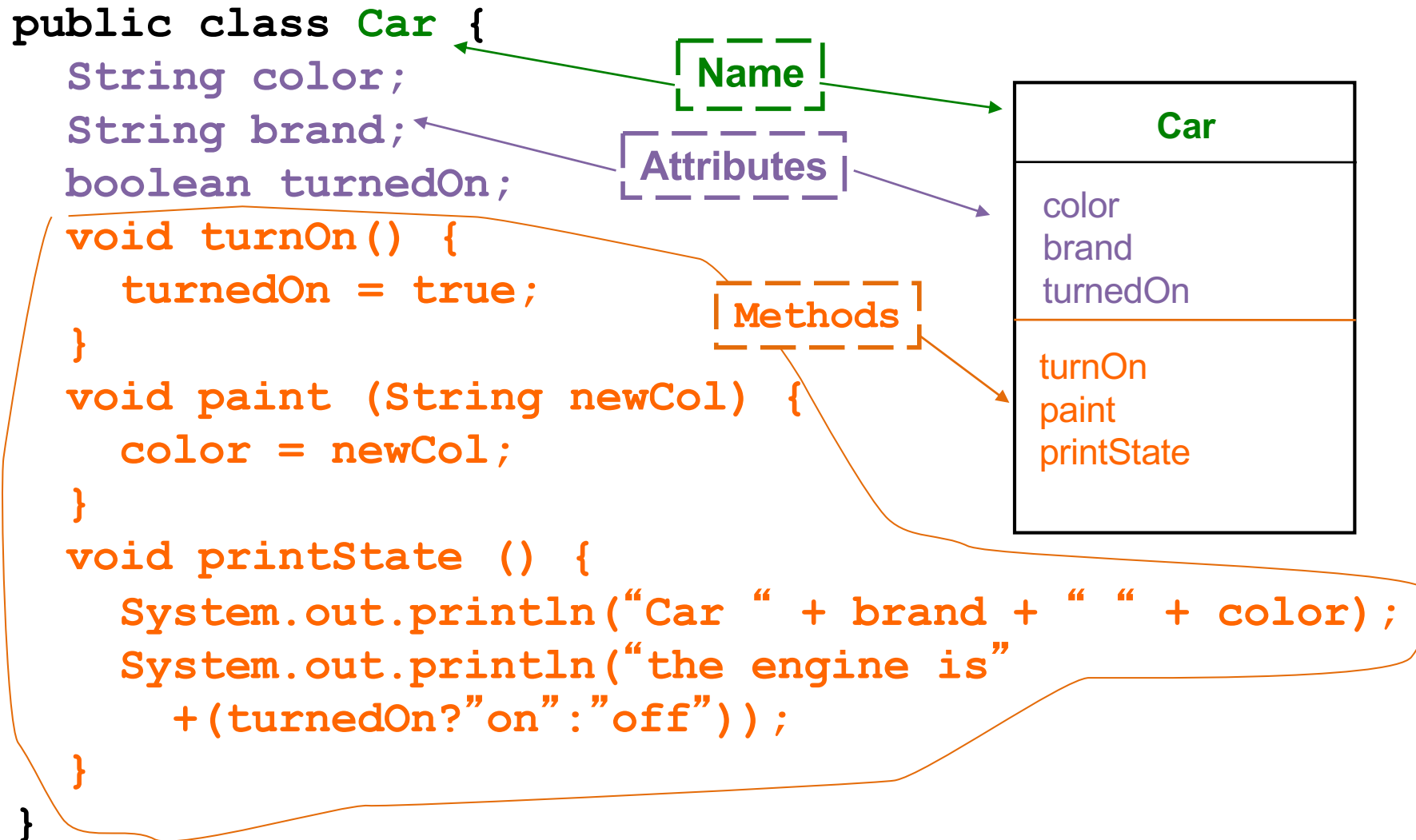


# Class

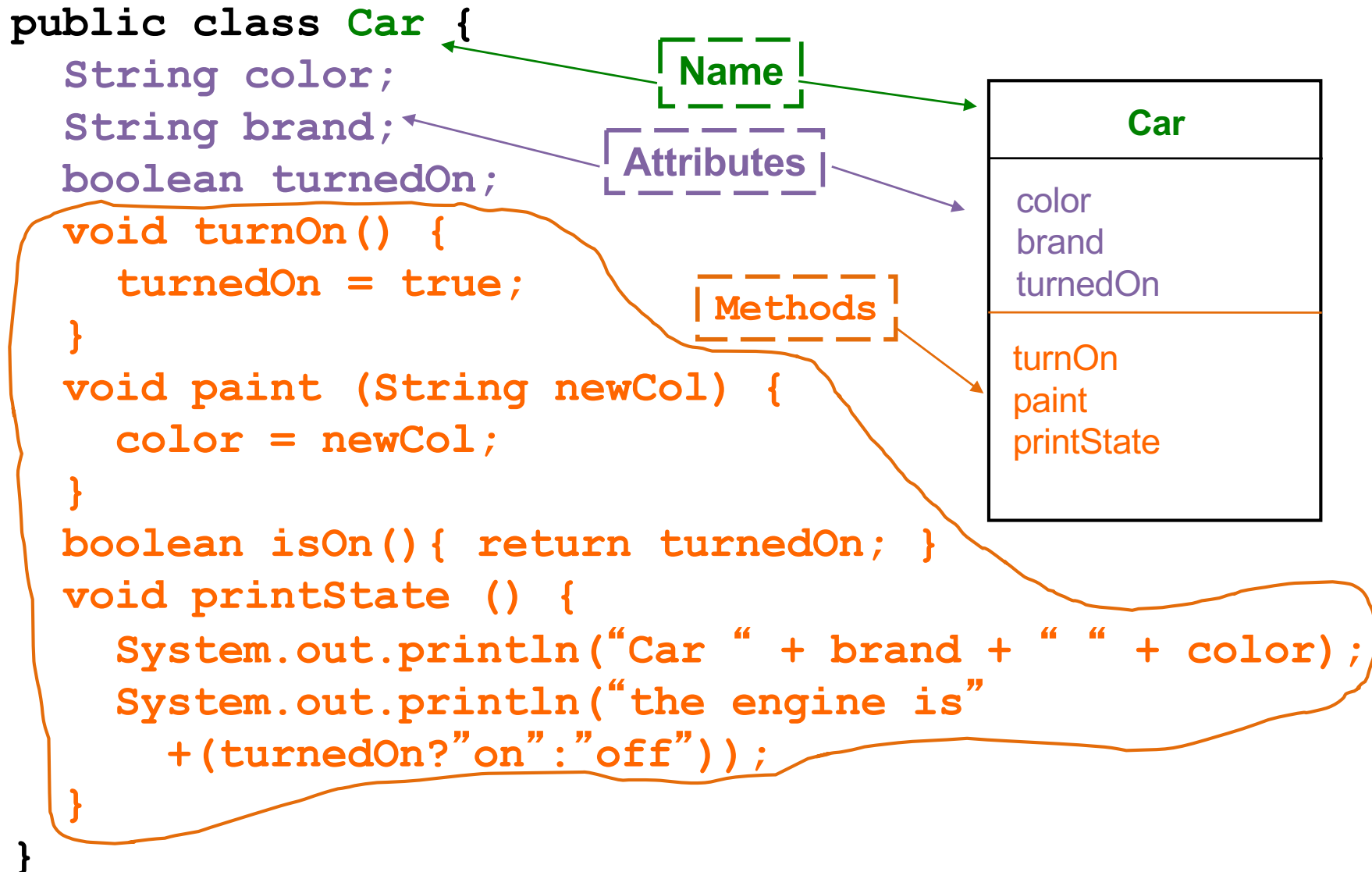
---

- Object descriptor
    - ◆ Defines the common structure of a set of objects
    - ◆ Similar to `typedef struct` in C
  - Consists of a set of **members**
    - ◆ Attributes
    - ◆ Methods
    - ◆ Constructors
-

# Class – definition



# Class – definition



# Attributes

---

- Attributes or Fields describe the data that can be stored within objects
  - They are like variables, defined by:
    - ♦ Type
    - ♦ Name
    - ♦ Similar to fields of struct in C
  - Each object has its own copy of the attributes
-



# Methods

---

- Methods represent the messages that an object can accept
  - ◆ `turnOn`
  - ◆ `paint`
  - ◆ `printState`
- They may accept arguments
  - ◆ `paint(String )`
- They may return values

# Objects

---

- An object is identified by:
  - ◆ Class, which defines its structure (in terms of attributes and methods)
  - ◆ **State** (values of attributes)
  - ◆ **Internal unique identifier**
- An object can be accessed through a **reference**
  - ◆ Any object can be pointed to by one or more references
    - Aliasing

# Objects

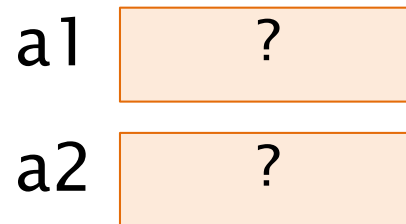
---

```
class ExLifeCycle() {  
    public static void main(String[] args) {  
        // declare reference  
        Car c;  
  
        // create object  
        c = new Car();  
  
        // use object  
        c.paint("yellow");  
  
    } // reference is lost  
}
```

# Objects and references

---

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

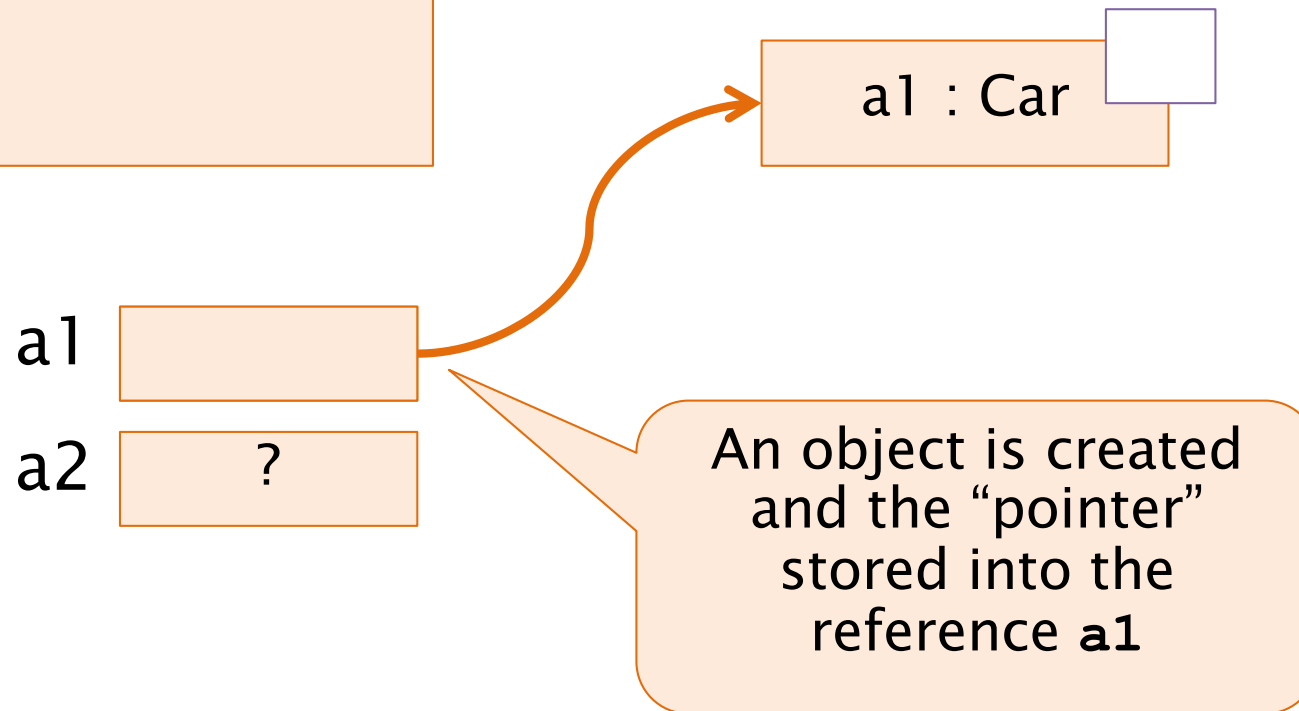


Two **uninitialized** references are created, they can't be used in any way.  
A reference is **not** an object

# Objects and references

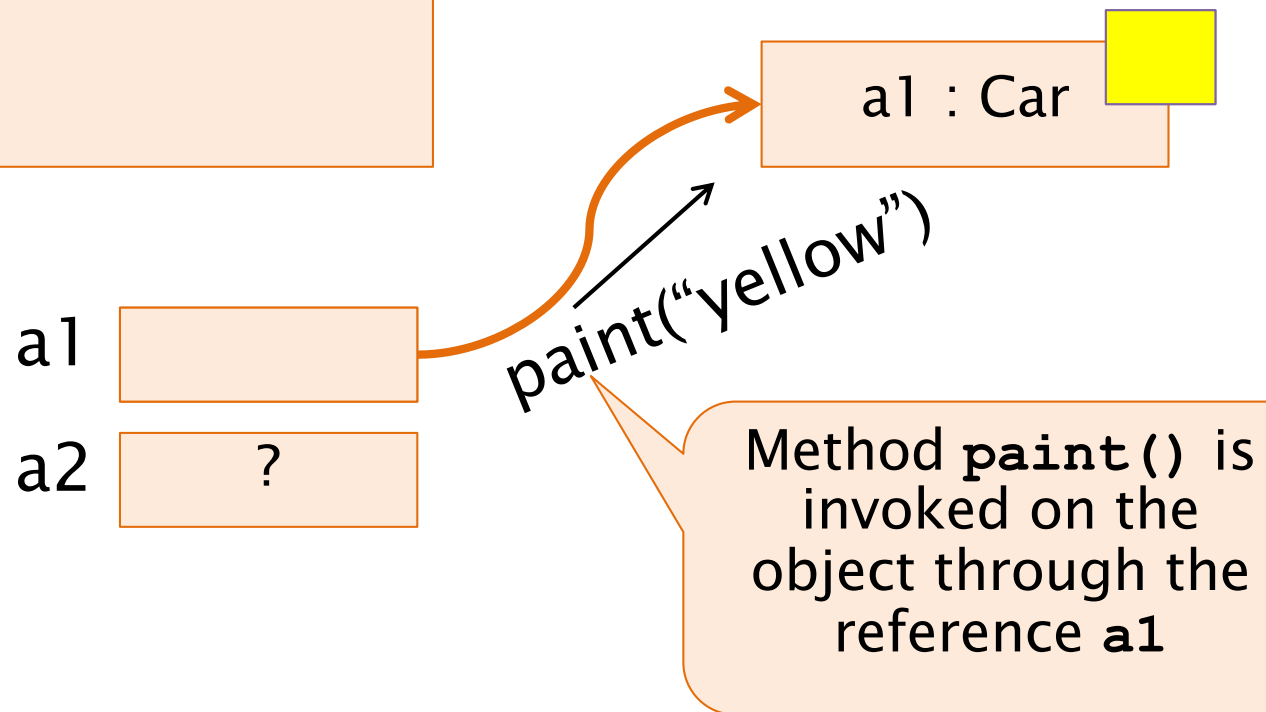
---

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



# Objects and references

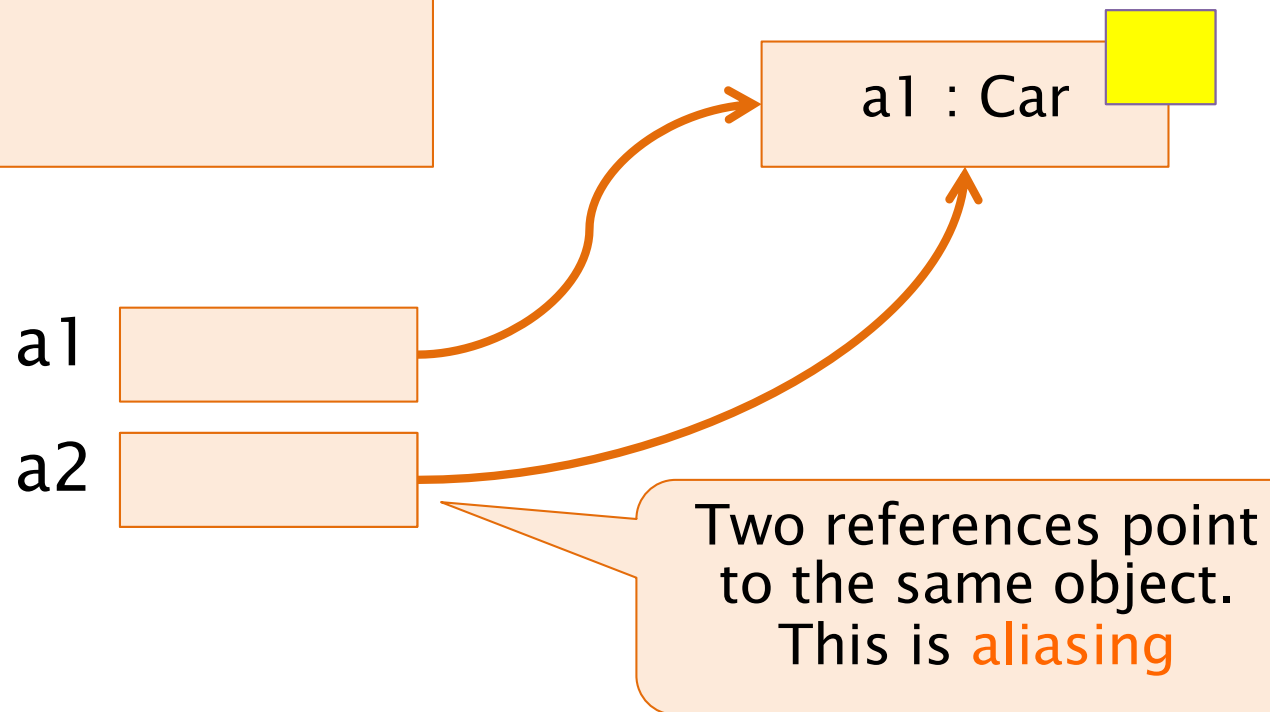
```
Car a1, a2;  
a1 = new Car();  
a1.paint("yellow");  
a2 = a1;  
a1 = null;  
a2 = null;
```



# Objects and references

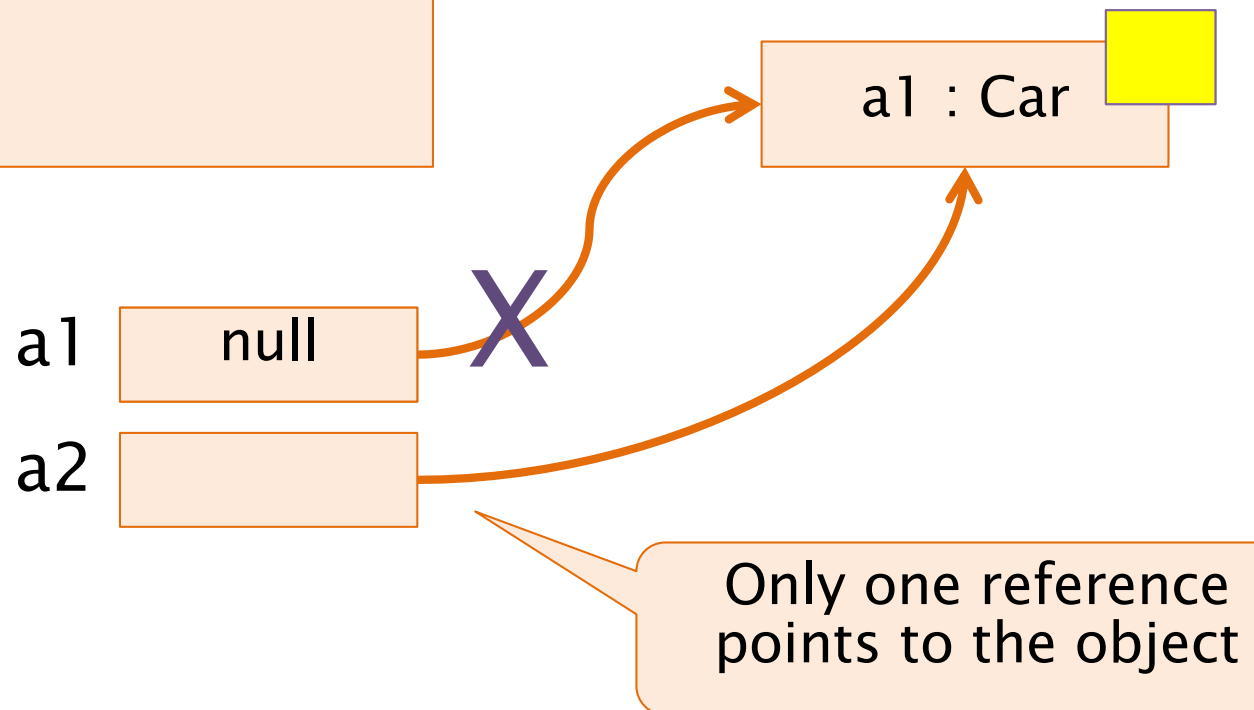
---

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



# Objects and references

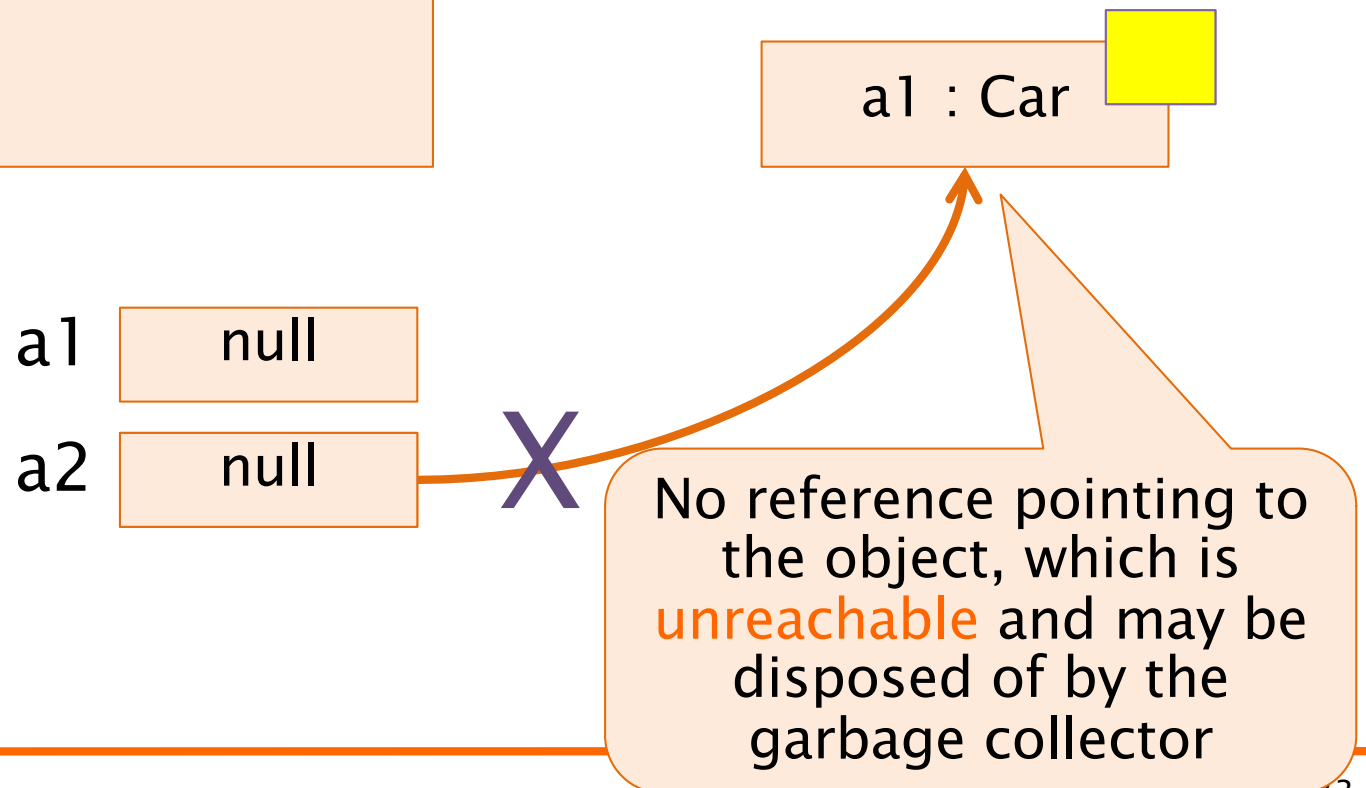
```
Car a1, a2;  
a1 = new Car();  
a1.paint("yellow");  
a2 = a1;  
a1 = null;  
a2 = null;
```





# Objects and references

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



# The keyword **new**

---

- Creates a new instance of the specific class
- Allocates the required memory in the heap
- Calls the **constructor** of the object
  - ◆ a special method without return type and named like the class
- Returns a reference to the new object

```
Car m = new Car();
```
- Constructor may have parameters,
  - ◆ e.g.

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

# Heap

---

- The part of the program memory used by an executing program to store data dynamically created at run-time
- C: **malloc**, **calloc** and **free**
  - ♦ Instances of types in static memory or in heap
- Java: **new**
  - ♦ Instances (Objects) are always in the heap

# Constructor (1)

---

- Constructor is a special method containing the operations (e.g. initialization of attributes) to be executed 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 arguments) is provided
- Overloading of constructors is often used

# Constructor (2)

---

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

# 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 from within another method of the **same object** dotted notation is not mandatory
- In such cases **this** is implied

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

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

equivalent



# Access to attributes

---

- Dotted notation

*objectReference.attribute*

- ♦ A 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” to disambiguate

---

- The use of this is not mandatory
- It can be used to disambiguate object attributes from local variables

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

# Chaining dotted notations

---

- Dotted notations can be combined in a single expression

```
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 Chaining

---

```
public class Counter {  
    private int value;  
    public Counter reset() {  
        value=0; return this;  
    }  
    public Counter increment(int by) {  
        this.value+=by; return this;  
    }  
    public Counter print() {  
        System.out.println(value);  
        return this;  
    }  
}
```

```
Counter cnt = new Counter();  
cnt.reset().print()  
    .increment(10).print()  
    .decrement(7).print();
```

# Operations on references

---

- The dot `.` operator is used to dereference object references
  - ♦ Obtain the object pointed by the reference
- Two comparison operators are defined
  - ♦ `==` and `!=`
- There is **NO** pointer arithmetic

# Comparing objects

---

- The relational operators check whether the references points to the same object in memory
    - ◆ They check identity
    - ◆ No check on the objects' contents
  - To compare contents an ad-hoc method must be defined
    - ◆ The criteria to compare the contests
-

# Overloading

---

- Several methods in a class can share the same name
  - Provided they have distinct **signature**
  - Signature of a method consists of:
    - ◆ Method name
    - ◆ Ordered list of argument types
      - Does not include the return type
-



# Overloading: disambiguation

---

- Invocation of an overloaded method is potentially ambiguous
  - Disambiguation is performed by the compiler based on actual parameters
    - ◆ The method definition whose formal argument types list matches the actual parameter list, is selected
-

# Overloading

---

```
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"
    }
}
```

# 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 eventually really destroyed the method **finalize**, if defined, is invoked:

```
public void finalize()
```

---

# SCOPE AND ENCAPSULATION

---

# Scope and Syntax

---

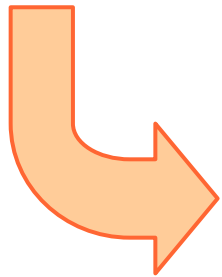
- Visibility modifiers
  - ◆ Applicable to members of a class
- **private**
  - ◆ Member is visible and accessible from instances of the same class only
- **public**
  - ◆ Member is visible and accessible from everywhere

# Visibility

---

```
class Car {  
    public 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
```



# Visibility

---

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

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

no

yes

# Access

---

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

# Information Hiding

---

- Information hiding is the approach of making information contained in objects non accessible from outside
  - It is implemented using the visibility modifiers
  - By default all attributes should be declared **private**
-

# 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
      - E.g. can perform type conversion
  - Setter
    - ◆ Allow performing checks before modifying the attribute
      - E.g. Validity of values, authorization
-

# Modifier vs. Query methods

---

- Modifiers

- ◆ Change the state of the object but do not return a value
  - e.g., setters

- Query

- ◆ Return a result and do not change the state of the object
  - ◆ No side-effects
    - e.g., getters
-

# Modifier / Query Separation

---

- Invocations to
  - ◆ **queries** can be added, removed, and swapped without affecting the overall behavior
  - ◆ **modifiers** cannot be touched without affecting the behavior
- Important to clearly separate them:
  - ◆ Queries return a value
  - ◆ Modifiers return **void**

See: <https://www.martinfowler.com/bliki/CommandQuerySeparation.html>

Original concepts in: B.Meyer, Object-Oriented Software Construction, Prentice-Hall, 1997

---

# Mutable vs. Immutable

---

- Classes that have only query methods are called **immutable**
    - ◆ Once initialized their status cannot be changed
  - Other classes that have modifiers conversely are called mutable
    - ◆ Their status can be changed
-

# Package

---

- Class is a better mechanism of modularization than a procedure
- But it is still small, when compared to the size 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 consist in several files, all stored in the **same folder**
- Each package defines a new **scope** (i.e., it puts bounds to visibility of names)
- It is therefore 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 single class  
(class name is in  
scope)

```
import java.awt.*;
```

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
    - ◆ The default package has no name
  - Classes in the default package cannot be accessed by classes residing in other packages
  - Usage of **default package is a bad practice** and 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

---

Package P

```
class A {  
    public int a1;  
    private int a2;  
    public void f1() {}  
}
```

yes

no

```
class B {  
    public int a3;  
    private int  
    a4;  
}
```

# Visibility w/ multiple packages

---

- **public** class A { }

- ◆ Class and public members of A are visible from outside the package

-  class B { }

Package visibility

- ◆ Class and any members of B are not visible from outside the package

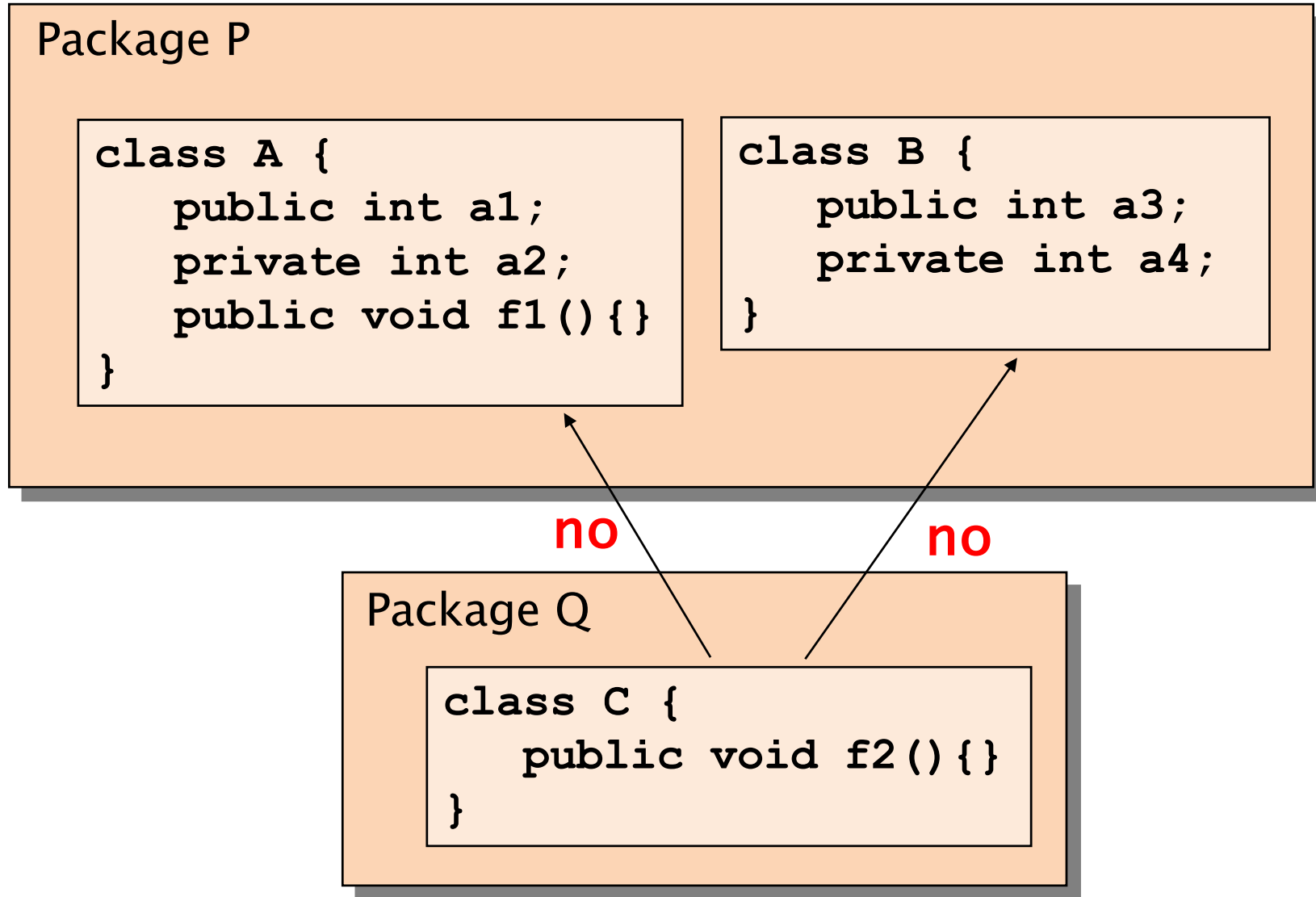
- **private** class A { }

- ◆ Illegal: why?

The class and its members would be visible to themselves only

# Multiple packages

---





# Multiple packages

---

Package P

```
public class A {  
    public int a1;  
    private int a2;  
    public void f1() {}  
}
```

```
class B {  
    public int a3;  
    private int a4;  
}
```

yes

no

Package Q

```
class C {  
    public void f2() {}  
}
```

# Access rules

---

	Method of the same class	Method of other class in the same package	Method of class in other package
Private member	Yes	No	No
Package member	Yes	Yes	No
Public member in package class	Yes	Yes	No
Public member in public class	Yes	Yes	Yes

---

# WRAPPER CLASSES

---

# String

---

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

See slide deck “Java Characters and Strings”

---

# Motivation

---

- In an ideal OO world, there are only classes and objects
- For the sake of efficiency, Java use 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

`boolean`

`char`

`byte`

`short`

`int`

`long`

`float`

`double`

`void`

## Wrapper Class

`Boolean`

`Character`

`Byte`

`Short`

`Integer`

`Long`

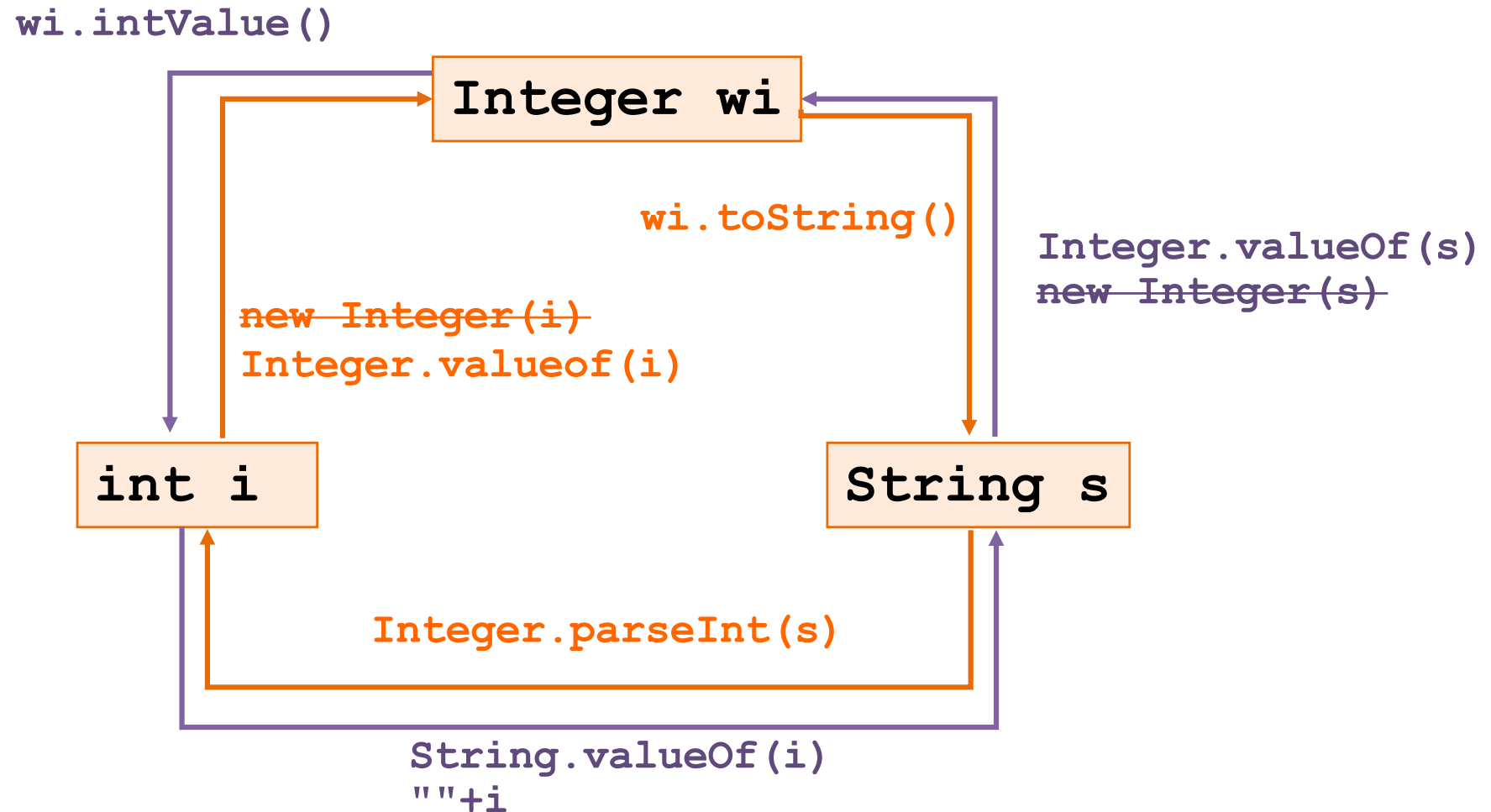
`Float`

`Double`

`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() ;
```



# Using Scanner

---

- Scanner can be initialized with a string

```
Scanner s = new Scanner("123");
```

- then values can be parsed

```
int i = s.nextInt();
```

- In addition a scanner is able to parse several numbers in the same string
-

# Autoboxing

---

- Since Java 5, the conversion between primitive types and wrapper classes is performed automatically (*autoboxing*)

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

# Character

---

- Utility methods on the kind of char
  - ◆ `isLetter()` , `isDigit()` ,  
`isSpaceChar()`
- Utility methods for conversions
  - ◆ `toUpper()` , `toLowerCase()`

---

# ARRAYS

---

# 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** and it is **stored in the heap**
- Array declaration allocates memory space for a **reference**, whose default value is null

a 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") };
```

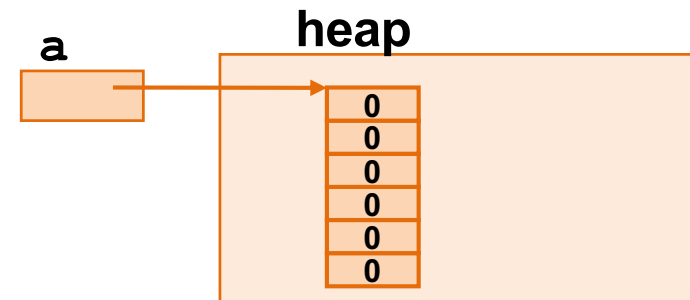
# Example – primitive types

---

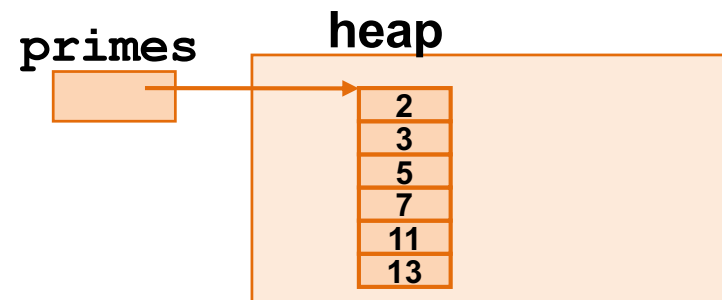
```
int[] a;
```



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



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

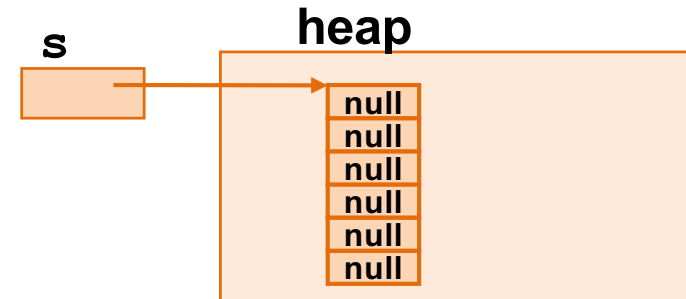




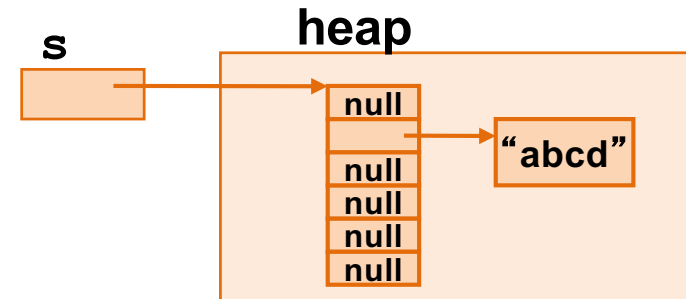
# Example – object references

---

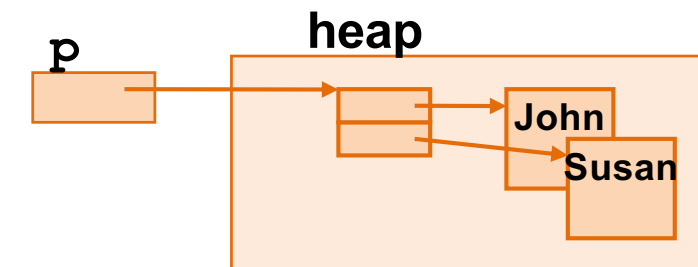
```
String[] s = new  
    String[6];
```



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



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



# Operations on arrays

---

- Elements are selected with brackets `[]` (C-like)
  - ◆ But Java makes 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 the loop with correct indexes
  - Less error prone

# For each – example

---

- Example:

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

◆ is equivalent to

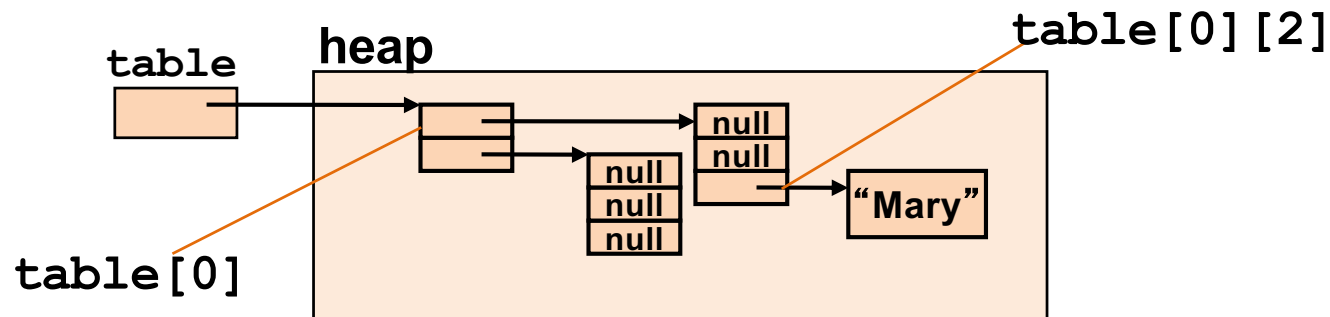
```
for (int i=0; i<args.length;++i) {  
    String arg= args[i];  
    //...  
}
```

# 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**

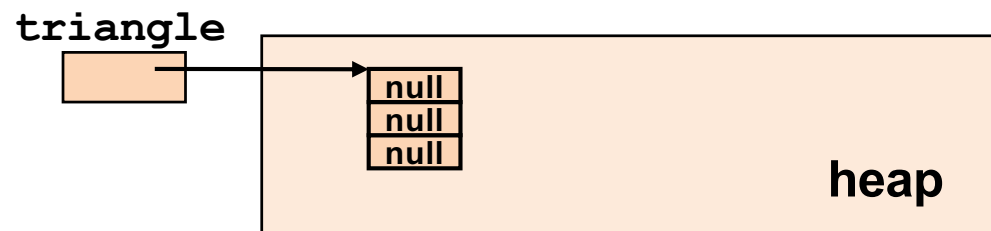
```
double[][] balance = new double[5][6];  
...  
double[] temp = balance[i];  
balance[i] = balance[j];  
balance[j] = temp;
```

# Rows with different length

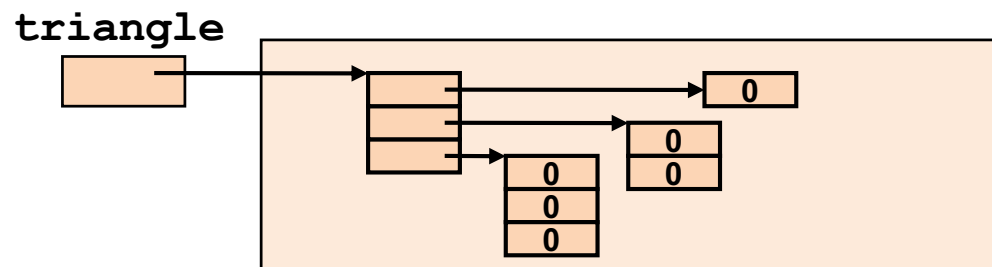
---

- 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];
```





# Exercise

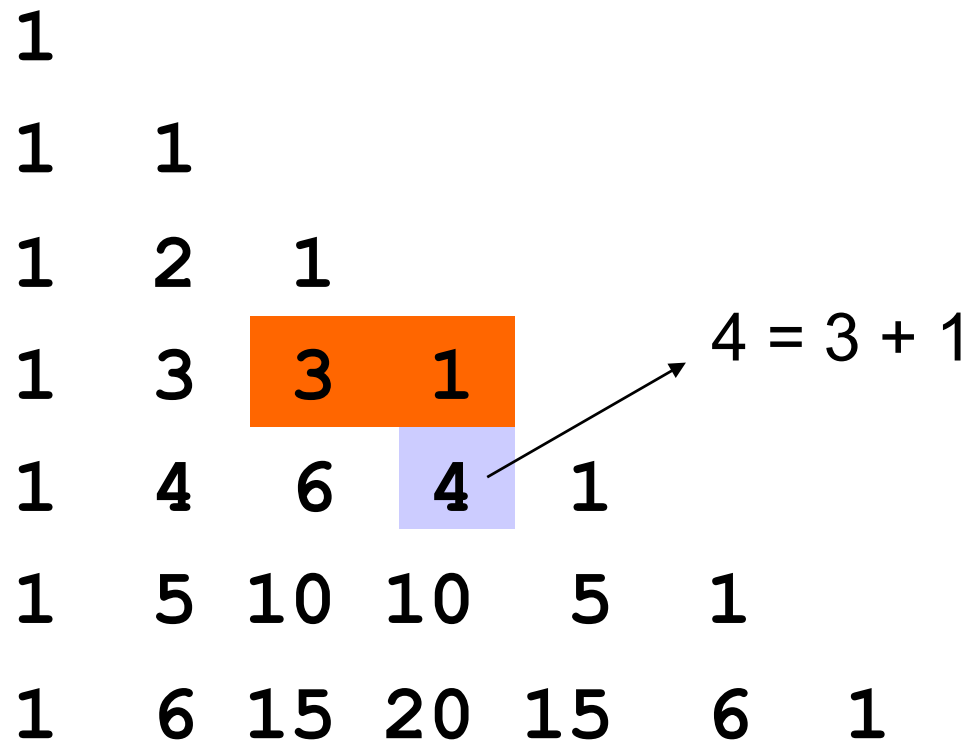
---

- Create an object representing an ordered list of integer numbers (at most 100)
- `print()`
  - ♦ prints current list
- `add(int)` and `add(int[])`
  - ♦ Adds the new number(s) to the list

# Tartaglia's triangle

---

- Write an application printing out the following Tartaglia's triangle



# Variable arguments

---

- It is possible to pass a variable number of arguments to a method using the **varargs** notation

`method( type  args )`

- The compiler assembles an array that can be used to scan the actual arguments
  - ◆ Type can be primitive or class

# Variable arguments– example

---

```
static int min(int... values) {  
    int res = Integer.MAX_VALUE;  
    for(int v : values) {  
        if(v < res) res=v;    }  
    return res; }  
  
public static void main(String[] args) {  
    int m = min(9,3,5,7,2,8);  
    System.out.println("min=" + m);  
    int[] numbers = {9,3,5,7,2,8};  
    min(numbers); // also ok!  
}
```

---

# STATIC ATTRIBUTES AND METHODS

---

# Static attributes

---

- 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** 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

---

- Used to keep a shared property
  - ◆ A count of created instances
  - ◆ A pool of all instances
- Keep a common constant value

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

# Static methods

---

- Static methods are not related 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 int 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 in default package



# Static methods: why

---

- Implement *functions*
    - ◆ Avoid creating an object just to invoke the method (see e.g., `main()`)
    - ◆ Collected in utility classes
  - Provide ideal factory method
    - ◆ Method to create an instance
-

# Function method

---

- A method whose return value depends only on the arguments
    - ◆ Typically defined as `static`
  - Often collected within a `utility` class
    - ◆ Class containing `static` function methods
  - Wrapper types include several *function* methods for conversion purposes
-

# Utility classes

---

- **System**
    - ♦ Interact with the operating system
  - **Math**
    - ♦ Mathematical functions
  - **Arrays**
    - ♦ Functions to operate on arrays
  - **Objects**
    - ♦ Functions to operate on object
-

# Class Math

---

- Defines several math-related function methods
    - ◆ Trigonometric functions
    - ◆ Min-max
    - ◆ Exponential and logarithms
    - ◆ Truncations
    - ◆ Random number generation
-

# Class **Arrays**

---

- Arrays utility functions
    - ◆ Binary search (**binarySearch()**)
    - ◆ Copy (**copyOf()**, **copyOfRange()**)
    - ◆ Equality (**equals()**, **deepEquals()**)
    - ◆ Fill-in (**fill()**)
    - ◆ Sorting (**sort()**)
    - ◆ String representation (**toString()**)
-

# Class System

---

- General purpose utilities
    - ◆ `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,
      - Also `err` for standard error
-

# Factory method

---

- A method used to create an object
    - ◆ Encapsulates an explicit object creation with the `new` operator
  - Can be used to:
    - ◆ Return objects from a pool
      - Requires immutable objects
      - Either pre-allocated or cached
    - ◆ Simplify creation
    - ◆ Maintain a collection of created objects
    - ◆ Control new objects allocation
      - See e.g., Singleton pattern
-



# Factory methods: Integer

---

- **valueOf(int)**
    - ◆ Replaces new Integer(int)
    - ◆ Cache values in the range -128 to 127
  - **valueOf(String)**
    - ◆ Returns the integer corresponding to the parsed string
    - ◆ Same as:  
`new Integer(Integer.parseInt(s))`
-

# 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;  
    }  
}
```

# Final variables / parameters

---

- Final parameters cannot be changed
    - ◆ Non final parameters are treated as local variables (initialized by the caller)
  - Final variables
    - ◆ Cannot be modified after initialization
    - ◆ Initialization can occur at declaration or later
-

# Constants

---

- Use **final static** modifiers

- ◆ **final** implies not modifiable

- ◆ **static** implies non redundant

```
final static float PI = 3.14;
```

```
...
```

```
PI = 16.0;           // ERROR, no changes
```

```
final static int SIZE; // missing init
```

- All uppercase (coding conventions)

# Static initialization block

---

- Block of code preceded by **static**
- Executed at class loading time

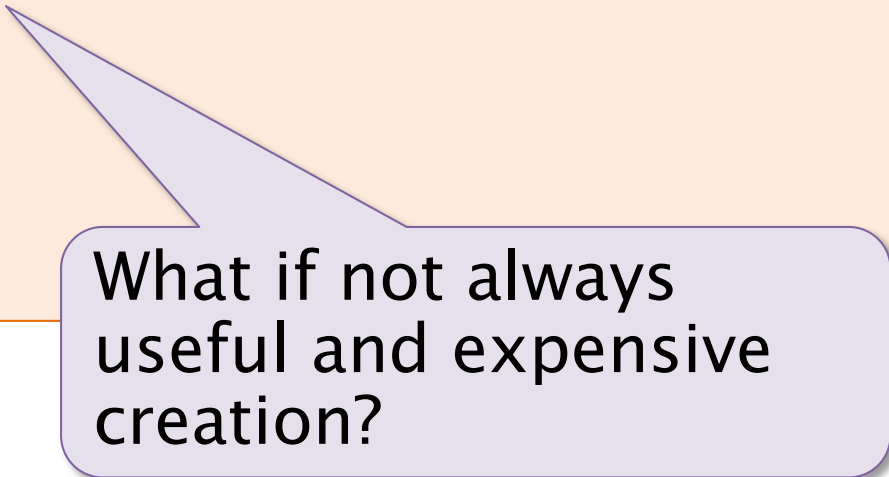
```
public final static double 2PI;  
static {  
    2PI = Math.acos(-1);  
}
```

# Example: Global directory (a)

---

- Manages a global name directory

```
class Directory {  
    public final static Directory root;  
    static {  
        root = new Directory();  
    }  
    // ...  
}
```



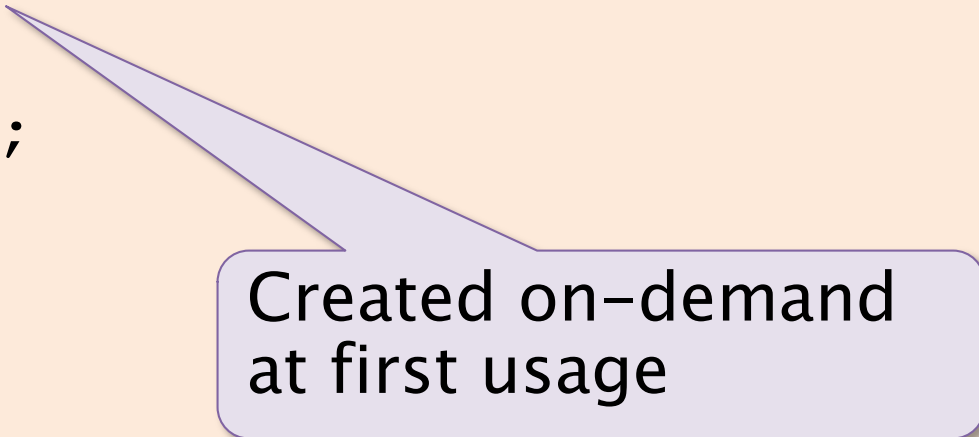
What if not always  
useful and expensive  
creation?

# Example: Global directory (b)

---

- Manages a global directory

```
class Directory {  
    private static Directory root;  
    public static Directory getInstance() {  
        if(root==null) {  
            root = new Directory();  
        }  
        return root;  
    }  
    // ...  
}
```



Created on-demand  
at first usage

# Singleton Pattern

---



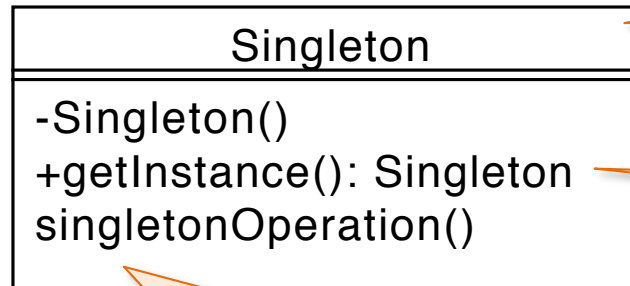
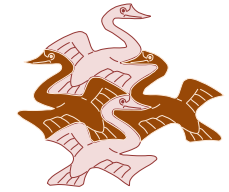
- Context:
  - ◆ A class represents a concept that requires a single instance
- Problem:
  - ◆ Clients could use this class in an inappropriate way

See slide deck on design patterns

---



# Singleton Pattern



Singleton class

Factory method

```
private Singleton() { }  
private static Singleton instance;  
public static Singleton getInstance() {  
    if(instance==null)  
        instance = new Singleton();  
    return instance;  
}
```

# Fluent Interfaces

---

- Method to design OO API based on extensive use of method chaining
- The goal is to improve readability
  - ◆ Code looks like prose
  - ◆ Often used to build complex objects
- Create a sort of Domain Specific Language (DSL) leveraging the syntax of the host language

# Example

---

- Usual non-fluent

$$10.40 \text{ kg} \cdot \text{m}^2 \cdot \text{s}^{-3}$$

```
Measure power = new Measure(10.4) ;  
power.addUnit("kg", 1) ;  
power.addUnit("m", 2) ;  
power.addUnit("s", -3) ;  
power.setPrecision(2) ;
```

- Fluent

```
Measure power = Measure.value(10.4) .  
    is("kg") .by("m") .squared() .by("s") .to(-3) .  
    withPrecision(2) .done() ;
```

---

# Measure

---

```
public class Measure {  
    private double value;  
    private Unit unit;  
    private int precision;  
    public Measure(double value) {  
        this.value = value;  
    }  
    public void setPrecision(int precision) {  
        this.precision = precision;  
    }  
    public void addUnit(String name, double exp) {  
        unit = new Unit(name,exp,unit);  
    }  
}
```

---

# Fluent Builder

---

```
public static  
Builder value(double v){  
    return new Builder(v);  
}
```

```
public static class Builder{  
    private Measure object;  
    private String unitName;  
    public Builder(double v){object = new Measure(v);}  
    public Builder is(String name) {  
        unitName = name;    return this;  
    }  
    public Builder by(String name) {  
        if(unitName!=null) {  
            object.addUnit(unitName, 1);  
        }  
        unitName = name;    return this;  
    }  
}
```

---

# Fluent Builder

---

```
public Builder squared() {  
    object.addUnit(unitName, 2);  
    unitName = null; return this;  
}  
public Builder to(double exponent) {  
    object.addUnit(unitName, exponent);  
    unitName = null; return this;  
}  
public Measure done() { return object; }  
public Builder withPrecision(int precision) {  
    object.setPrecision(precision);  
    return this;  
}  
}
```

---

---

# OTHER TYPES

---

# Enum

---

- Defines an enumerative type

```
public enum Suits {  
    SPADES, HEARTS, DIAMONDS, CLUBS  
}
```

- Variables of enum types can assume only one of the enumerated values

```
Suits card = Suits.HEARTS;
```

- ♦ They allow much stricter static checking compared to integer constants (e.g. in C)
-



# Enum

---

- Enum can are similar to a class that automatically instantiates the values

```
class Suits {  
    public static final Suits HEARTS=  
        new Suits ("HEARTS",0);  
    public static final Suits DIAMONDS=  
        new Suits("DIAMONDS",1);  
    public static final Suits CLUBS=  
        new Suits ("CLUBS", 2);  
    public static final Suits SPADES=  
        new Suits ("SPADES", 3);  
    private Suits (String enumName, int  index)  
    {...}  
}
```

# Record

---

- The record keyword allow defining an immutable class with compact notation

```
record Point(int x, int y) {}
```

- ♦ Provides getter methods with the same name of the attribute
    - `x()` and `y()`
  - ♦ Can include additional (non modifier) methods
  - ♦ Equivalent to a class but much more compact
    - Since Java 17
-

# Record vs. Class

---

```
record Point(int x, int y) {}
```

```
class Point() {  
    public final int x;  
    public final int y;  
    public Point(int x, int y) {  
        this.x=x;  
        this.y=y;  
    }  
}
```

---

---

# NESTED CLASSES

---

# Nested class types

---

- Static nested class
    - ◆ Within the container name space
  - Inner class
    - ◆ As above + contains a link to the creator container object
  - Local inner class
    - ◆ As above + may access (final) local variables
  - Anonymous inner class
    - ◆ As above + no explicit name
-

# (Static) Nested class

---

- A class declared inside another class

```
package pkg;  
class Outer {  
    static class Nested {  
    }  
}
```

- Similar to regular classes
    - ◆ Subject to usual member visibility rules
    - ◆ Fully qualified name includes the outer class:
      - `pkg.Outer.Inner`
-

# (Static) Nested class – Usage

---

- Static nested classes can be used to hide classes that are used only within another class
    - ◆ Reduce namespace pollution
    - ◆ Encapsulate internal details
    - ◆ Nested class lies within the scope of the outer class
-

# (Static) Nested class – Example

---

```
public class StackOfInt{  
    private static class Element {  
        int value;  
        Element next;  
    }  
    private Element head  
    public void push(int v) { ... }  
    public void pop() { ... }  
}
```

---



# Inner Class

---

- Linked to an instance
  - ♦ A.k.a. non-static nested class

```
package pkg;  
class Outer {  
    class Inner{  
    }  
}
```

- It is linked to instances of enclosing outer classes (i.e. it is non static)
-

# Inner Class

---

- Any inner class instance is associated with the instance of its enclosing class that instantiated it
  - Cannot be instantiated from
    - ◆ a static method
    - ◆ Other classes
  - Has direct access to that enclosing object methods and fields
-

# Inner Class (example)

---

```
public class Counter {  
    int i;  
    public class Incrementer {  
        private int step=1;  
        public void doIncrement(){ i+=step; }  
        Incrementer(int step){ this.step=step; }  
    }  
    public Incrementer buildIncrementer(int step){  
        return new Incrementer(step);  
    }  
    public int getValue(){  
        return i;  
    }  
}
```

inner instance is linked  
to this outer object


```
Counter c = new Counter()  
Incrementer byOne = c.buildIncrementer(1);  
Incrementer byFour = c.buildIncrementer(4);  
byOne.doIncrement();  
byFour.doIncrement();  
c.getValue(); // -> 5
```

# Local Inner Class

---

- Declared inside a method

```
public void m() {  
    int j=1;  
    class X {  
        int plus() { return j + 1; }  
    }  
  
    X x = new X();  
    System.out.println(x.plus());  
}
```



- ♦ References to local variables are allowed
    - Replaced with “current” value
    - Set of such local variables is called **closure**
-

# Local Inner Class

---

- Declared inside a method

```
public void m() {  
    int j=1;  
    class X {  
        int plus() { return j + 1; }  
    }  
    j++;  
    X x = new X();  
    System.out.println(x.plus());  
}
```

The diagram illustrates a variable 'j' being updated after an inner class method call. An orange arrow points from the 'j' in the 'plus()' method call to the 'j++' statement, indicating that the value of 'j' has changed. A red '1' is placed above the 'j' in the 'plus()' method call, indicating its value at that point in the execution.

What result should we expect?

- ♦ Local variable cannot be changed after being referred to by an inner class
-

# Local Inner Class

---

- Declared inside a method

```
public void m() {  
    final int j=1;  
    class X {  
        int plus() { return j + 1; }  
    }  
    j++;  
    X x = new X();  
    System.out.println(x.plus());  
}
```

- ♦ Local variables used in local inner classes should be declared **final**
    - Or **be effectively** final
-

# Anonymous Inner Class

---

- Local class without a name
  - Only possible with inheritance
    - ◆ Implement an interface, or
    - ◆ Extend a class
  - See: inheritance
-

---

# MEMORY MANAGEMENT

---



# Memory types

---

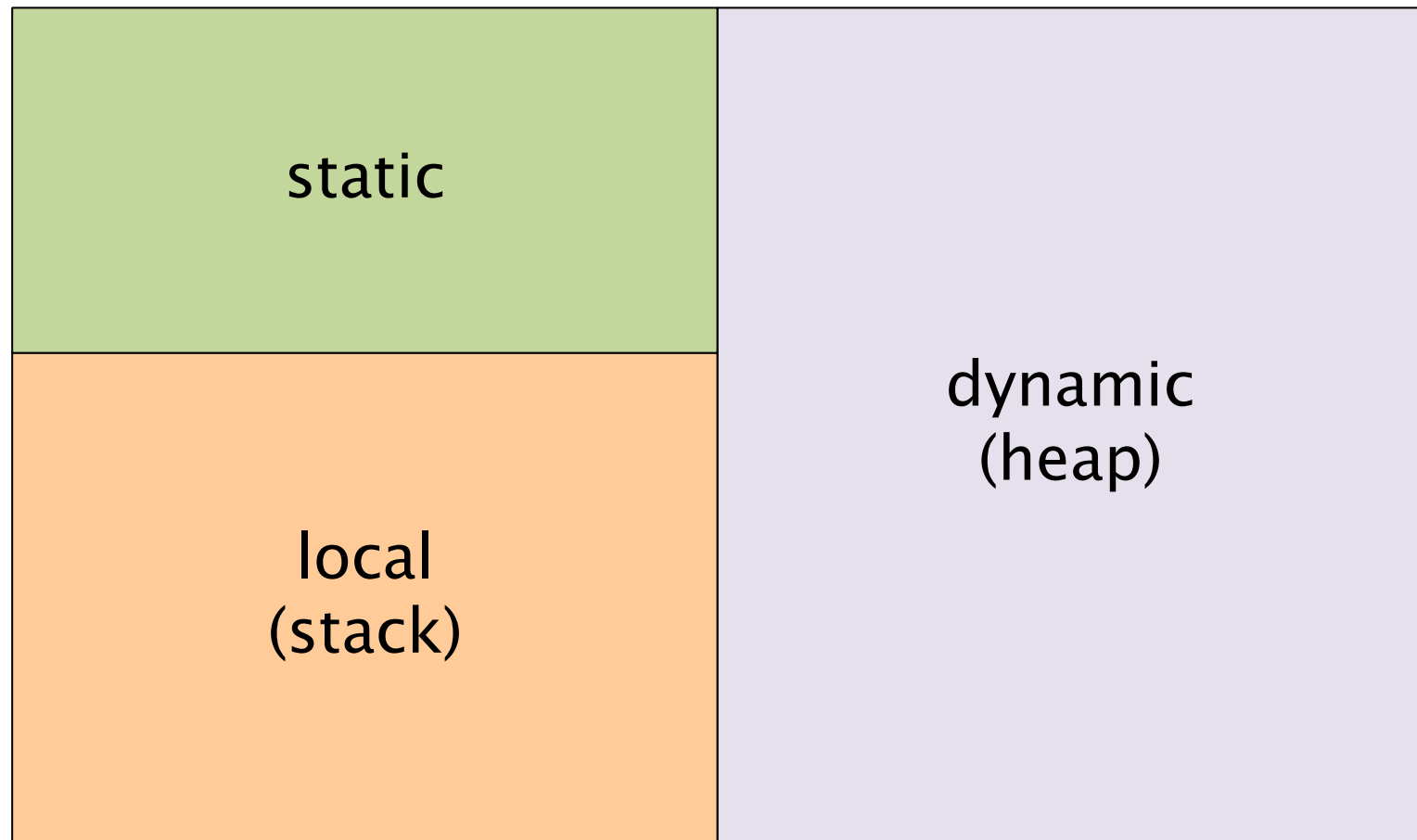
Depending on the kind of elements they include:

- 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)

# Memory types

---

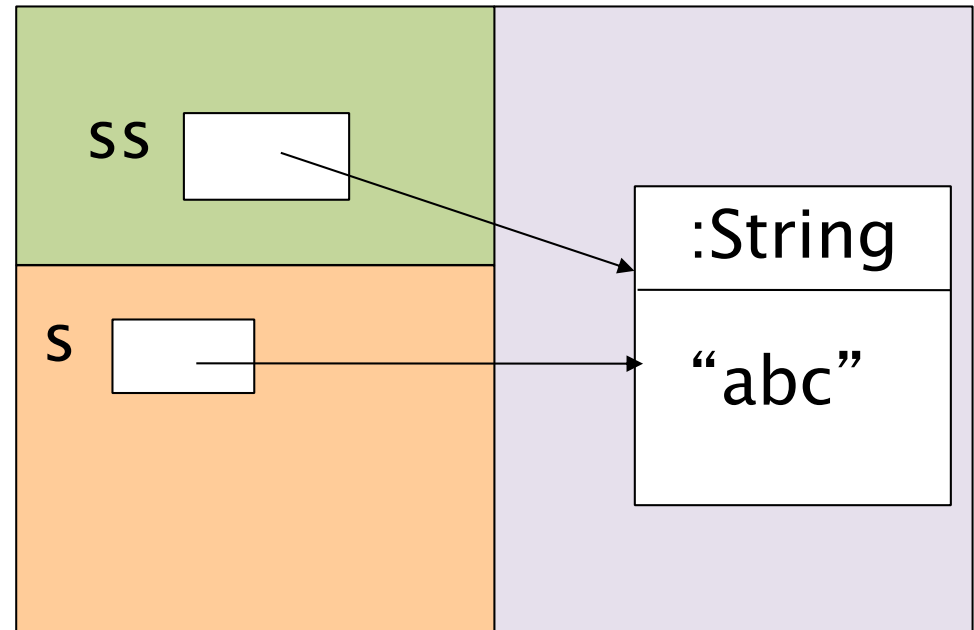
*Memoria est omnis divisa in partes tres...*



# Example

---

```
static String ss;  
.. main() {  
    String s;  
  
    s=new String("abc");  
  
    ss = s;  
}
```



# Types of variables

---

- **Instance variables**
  - ◆ Stored within objects (in the heap)
  - ◆ A.k.a. fields or attributes
- **Local Variables**
  - ◆ Stored in the Stack
- **Static Variables**
  - ◆ Stored in static memory

# Garbage collector

---

- Component of the JVM that cleans the heap memory from '*dead*' objects
- Periodically it analyzes references and objects in memory
- ...and then it releases the memory for objects with no active references
- No predefined timing
  - ◆ `System.gc()` can be used to *suggest* GC to run as soon as possible

# Object destruction

---

- It's not made explicitly but it is made by the JVM garbage collector when releasing the object's memory
  - ◆ Method `finalize()` is invoked upon release
- **Warning**: there is no guarantee an object will be ever explicitly released

# Finalization and garbage collection

---

```
class Item {  
    public void finalize() {  
        System.out.println("Finalizing");  
    }  
}
```

```
public static void main(String args[]) {  
    Item i = new Item();  
    i = null;  
    System.gc(); // probably will finalize object  
}
```

---

# 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