Java Basic Features

SoftEng
http://softeng.polito.it

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

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 language
- 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

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

  ```
  switch(season){
  case "summer":
  case "spring": temp = "hot";
                 break;
  }
  ```
  - 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 logic values (`true`, `false`)
- Conditional constructs evaluate boolean conditions
  - Note well – It is not possible to evaluate this condition
    ```
    int x = 7; if(x){…} //NO
    ```
  - Use relational operators
    ```
    if (x != 0)
    ```

Passing parameters

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

- Note well: only the object reference is copied not the whole object

Elements in a OO program

Structural elements (types) (compile time)	Dynamic elements (data) (run time)
- Class	- Reference
- Primitive type	- Variable

Classes and primitive types

▪ Class	▪ type primitive
`class Exam {}`	`int, char,` `float`
▪ Variable of type reference	▪ Variable of type primitive

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

descriptor

instance

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

Class

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

      Exam e;   e | null |

- …allocates memory space for the *reference* ('pointer')

  …and *sometimes* it initializes it with null by default

- Allocation and initialization of the *object* value are made later by its constructor

      e = new Exam();   e | 0Xffe1 | ⟶ | Object Exam |

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`

Operators (integer and floating-point)

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

Logical operators

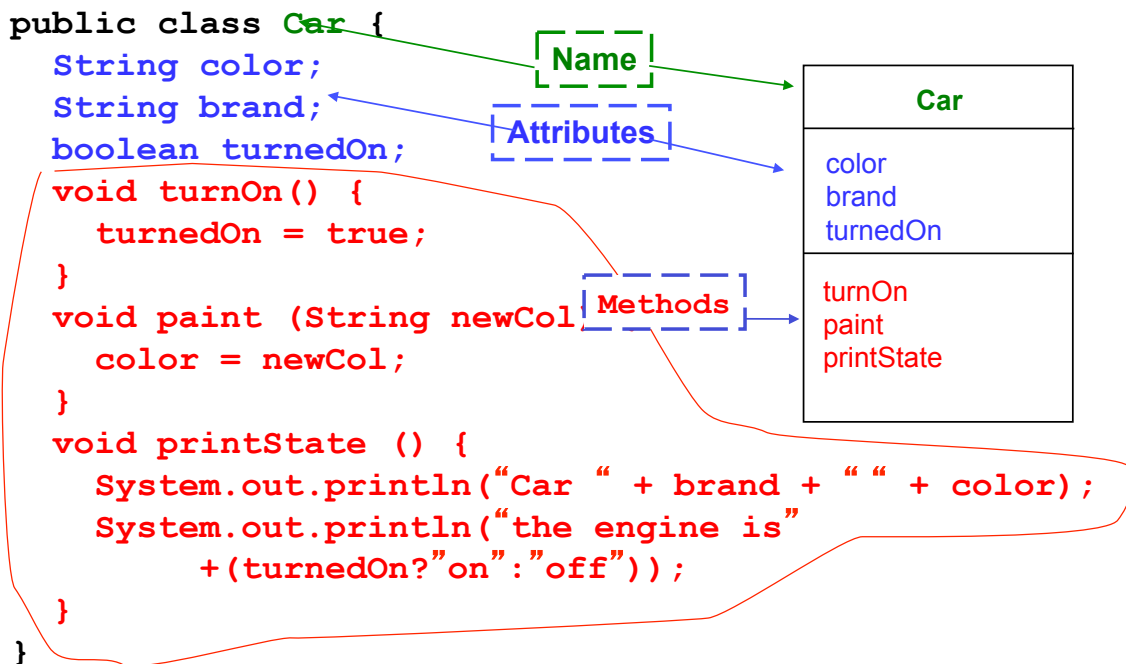- Logical operators follows C syntax:

  && || ! ^

- Note well: Logical operators work ONLY on booleans
  - ♦ Type int is NOT considered a boolean value like in C
  - ♦ Relational operators work with boolean values

CLASSES AND OBJECTS

Class

- Object descriptor
- It consists of attributes and methods

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"));
    }
}
```

Name

Attributes

Methods

Car
color
brand
turnedOn
turnOn
paint
printState

Methods

- Methods represent the messages that an object can accept:
  - ◆ turnOn
  - ◆ paint
  - ◆ printState
- Methods may have parameters
  - ◆ 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 (attributes values)
  - An internal unique identifier
- Zero, one or more reference can point to the same object

# 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 and a2 are uninitialized
a1 = new Car();
a1.paint("yellow");// Car "yellow" generated,
  // a1 is a reference pointing to "yellow"
a2 = a1; //now two references point to "yellow"
a2 = null; // a reference points to "yellow"
a1 = null;// no more references point to "yellow"
// Object exists but it is no more reachable
// then it will be freed by
// the garbage collector
```

- Note well: a reference IS NOT an object

Objects Creation

- Creation of an object is made with 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 heap
- Calls the constructor method of the object (a method without return type and with the same name of the Class)
- Returns a reference to the new object created
- Constructor can have parameters
    - String s = new String("ABC");

Heap

- A part of the 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 method contains operations (initialization of attributes etc.) we want to execute on each object as soon as it is created
- Attributes are always initialized
  - ◆ Attributes are initialized with default values
- If no constructor at all is declared, a default one (with no parameters) 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, 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

- In Java it is no longer a programmer concern
  - ◆ Managed memory language
- Before the object is really destroyed the method finalize, if existing, 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");
```

Caveat

- If a method is invoked from 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++)
            readPage(i);
   }
}
```

Caveat (cont'd)

- 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 attributes from local variables

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

Combining dotted notations

- Dotted notations can be combined

  ```java
  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

Operations on references

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

SCOPE AND ENCAPSULATION

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 (cont'd)

- Washing machine, design3
  - ◆ command:
    - Wash!
  - ◆ insert clothes, and the washing machine automatically select the correct program

- Hence, there are different solutions with different level of granularity / abstraction

Motivation

- Modularity = cut-down inter-components interaction
- Info 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**
  - Member is visible and accessible from instances of the same class only
- **public**
  - Member is visible and accessible from everywhere

Info hiding

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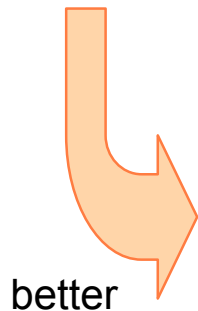class Car {
  public String color;
}
Car a = new Car();
a.color = "white";  // ok
```

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
```

Info 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

```java
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

```java
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

```java
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

```java
public class Student {

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

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

Example with getter/setter

```java
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;
    }
}
```

PACKAGE

Motivation

- Class is a better element of modularization than a procedure
- But it is still small, when compared to the size of an application
- For the sake of organization, 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

```
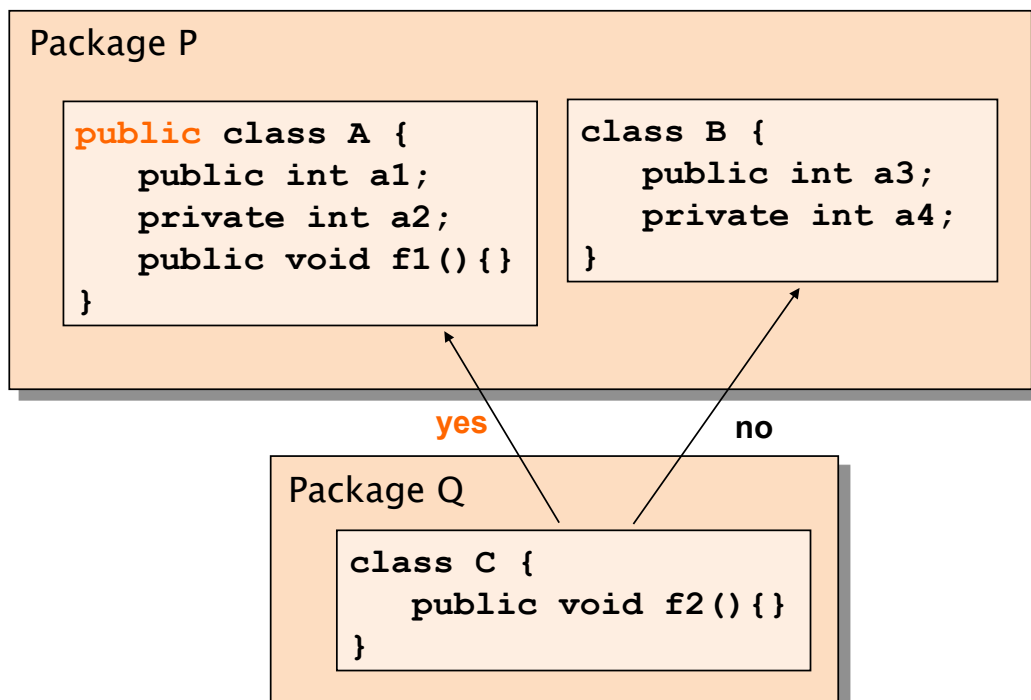Package P

  class A {

     public int a1;

     private int a2;          yes        class B {

     public void f1(){}                     public int a3;

                              no            private int
  }                                       a4;

                                          }
```
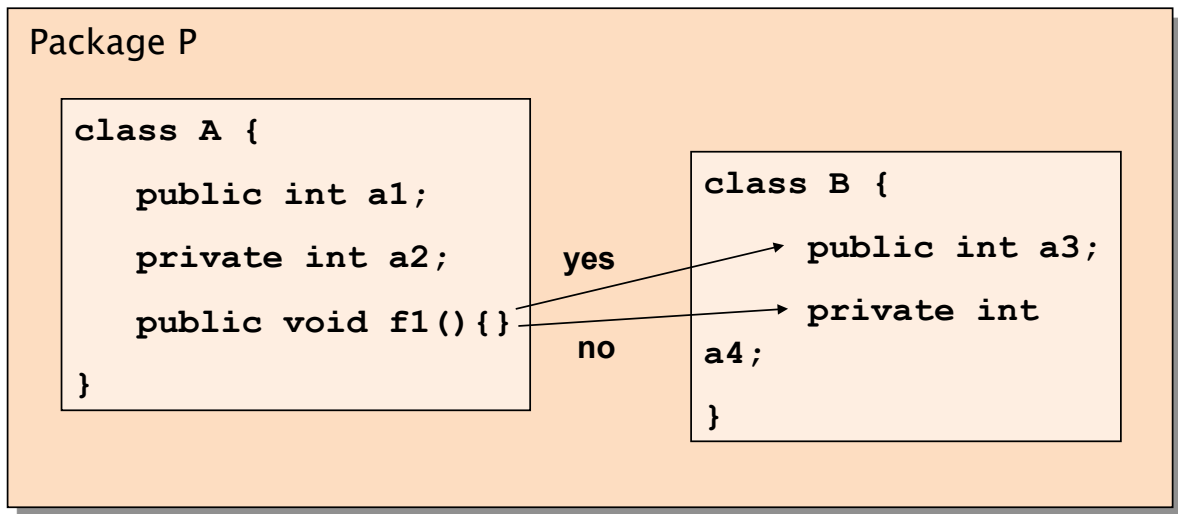
Visibility w/ multiple packages

- **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

Package P

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

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

no   **no**

Package Q

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

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(){}
}
```
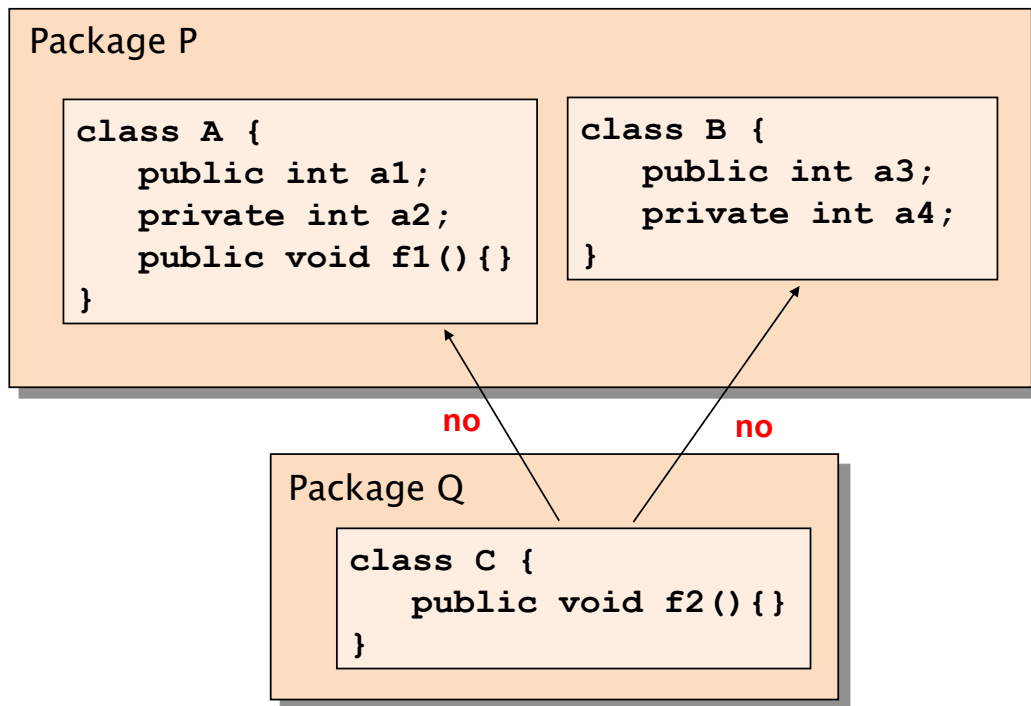
Access rules

	Method of the same class	Method of other class in the same package	Method of other 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

STRINGS

String

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

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 2 strings
  `"This string" + " is made by two strings"`

- Works also with other types (automatically converted to string)
  ```
  System.out.println("pi = " + 3.14);
  System.out.println("x = " + x);
  ```

String

- **`int length()`**
  - ◆ returns string length
- **`boolean equals(String s)`**
  - ◆ compares the values of 2 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
```

String

- **`String  valueOf(int)`**
  - ♦ Converts int in a String – available for all primitive types
- **`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

- **`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

- insert
- append
- delete
- reverse

WRAPPER CLASSES

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

Conversions

Example

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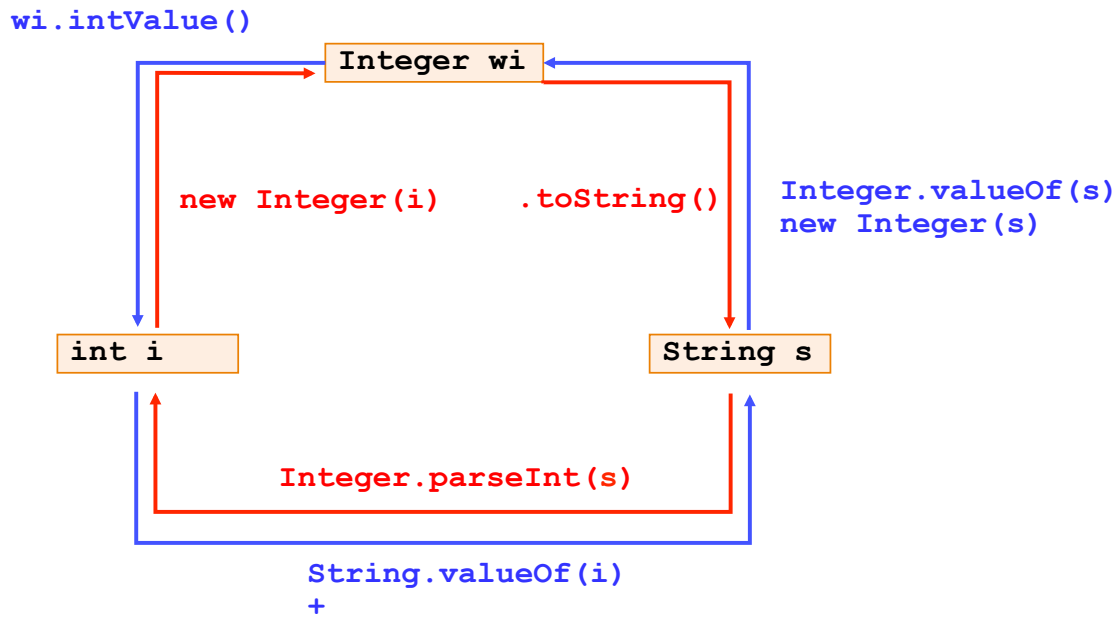Integer obj = new Integer(88);
String s = obj.toString();
int i = obj.intValue();


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

Autoboxing

- In 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

- Utility methods on the kind of char
  - **isLetter(), isDigit(), isSpaceChar()**
- Utility methods for conversions
  - **toUpper(), toLower()**

ARRAY

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

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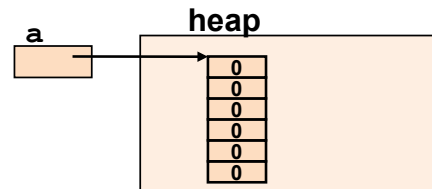int[] primes = {2,3,5,7,11,13};
Person[] p = { new Person("John"),
               new Person("Susan") };
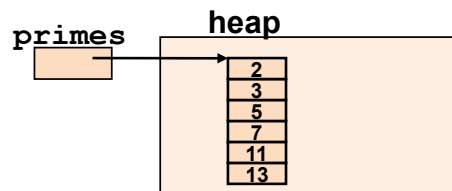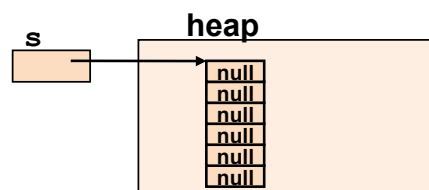```

# Example – primitive types

```
int[] a;
```

a

| null |

heap

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

a

heap

| 0 |
| 0 |
| 0 |
| 0 |
| 0 |
| 0 |

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

primes

heap
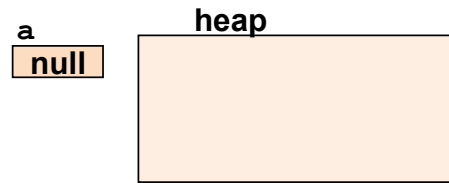
| 2 |
| 3 |
| 5 |
| 7 |
| 11 |
| 13 |

Example – object references

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

s

heap

| null |
| null |
| null |
| null |
| null |
| null |

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

s

heap

| null |
| null | → "abcd" |
| null |
| null |
| null |

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

p

heap
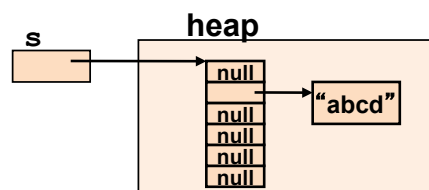
John

Susan
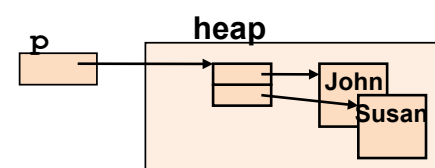
Operations on arrays

- 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){
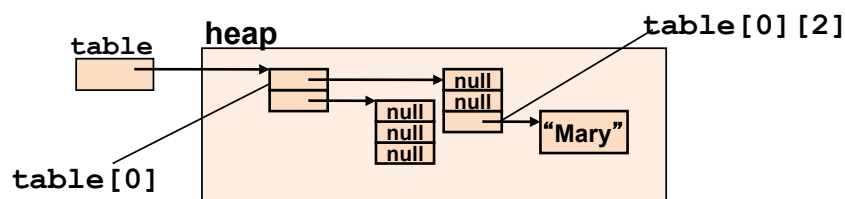  String arg= args[i];
  //...
}
```

Homework

- 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

Multidimensional array

- Implemented as array of arrays

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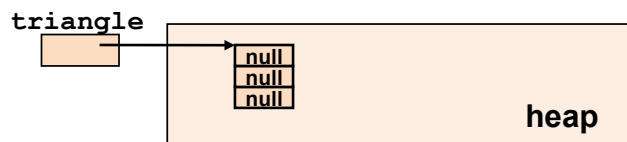table[0][2] = new Person("Mary");
```

Rows and columns

- As 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];
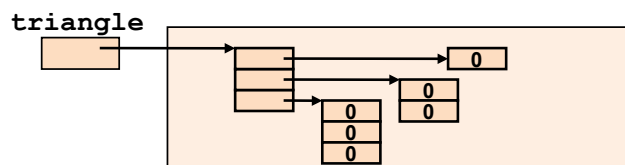balance[j] = temp;
```

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];
```

Tartaglia's triangle

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

```
1
1   1
1   2   1
1   3   3   1          4 = 3 + 1
1   4   6   4   1
1   5  10  10   5   1
1   6  15  20  15   6   1
```

Variable arguments

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

  method(Object ... args)

- The compiler assembles an Object array that can be used to scan the passed parameters

Variable arguments– example

```java
static void plst(String pre, Object...args){
   System.out.print(pre);
   for(Object o:args){
      if(o!=args[0]) System.out.print(", ");
      System.out.print(o);
   }
   System.out.println();
}
public static void main(String[] args) {
   plst("List:","A",'b',123, "ciao");
}
```

STATIC ATTRIBUTES AND METHODS

Class variables

- Represent properties which are common to all instances of an object
- They exist even when no object has been instantiated yet
- They are defined with the static modifier

```
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

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 more strict static checking compared to integer constants (used e.g. in C)

Enum

- Enum can be declared outside or inside a class, but NOT within a method
- Enums are not Strings or ints, but more like a kind of class but constructor can't be invoked directly, conceptually like this:

```
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) {…}
 }
```

Final Attributes

- 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

- Declared using final static modifiers
  - ◆ Not modifiable
  - ◆ Non redundant

```
final static float PI = 3.14;

…

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

```
  - ◆ Use uppercases (coding conventions)

Static initialization block

- Block of code preceded by `static`
- Executed upon class loading by the JVM

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

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...

static	
local (stack)	dynamic (heap)
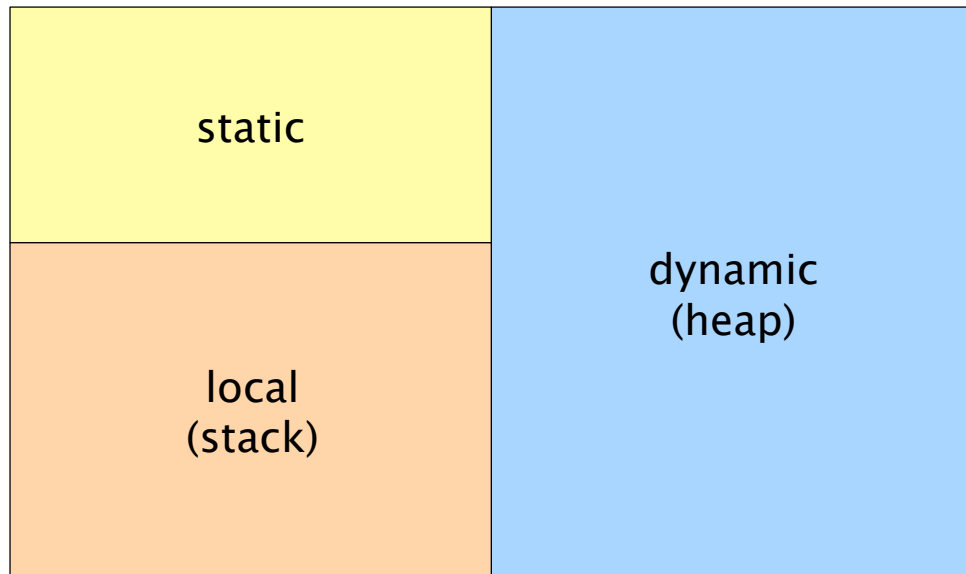
Example

```
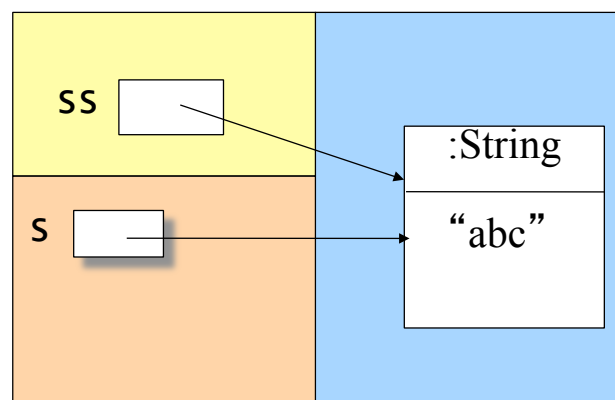static String ss;
.. main(){
  String s;

  s=new String("abc");

  ss = s;
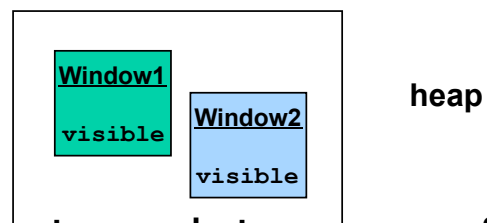}
```

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

Instance Variables

- Declared within a Class (attributes)

```
Class Window {
    boolean visible;
…
```


heap

Window1

visible

Window2

visible

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

Local Variables

- Declared within a method or a code block
- Stored in the stack
- Created at the beginning of the code block where they are declared
- Automatically destroyed at the end of the code 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 is stored in static memory
  - ♦ associated to a Class => also called Class variables

- Created/initialized during Class-loading in 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

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

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

NESTED CLASSES

(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
  - Full name includes the outer class:
    - `pkg.Outer.Inner`

Inner Class

- A.k.a. non-static nested class

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

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

Inner Class (example)

```
public class Outer {
   int i;
   class Inner {
      int step=1;
      void increment(){ i+=step; }
   }
   void m(){
      Inner in = new Inner();
      in.increment();
      in.step=4;
      in.increment();
   }
}
```

The inner instance is linked to `this` outer object

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());
}
```

1

- References to local variables are allowed
  - Replaced with "current" value

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());
}
```

1

What result should we expect?

- It is illegal to modify a local variable after is has been 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
  - ◆ Extend a class

- See: inheritance

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