Introduction to Java (cont.)

Vũ Thị Hồng Nhạn

(vthnhan@vnu.edu.vn)

Dept. of Software Engineering, UET

Vietnam National Univ., Hanoi

Content

- Final, static fields/methods
- Composition
- Command input
- Input Scanner
- File Scanner
- Packages in Java

Final fields

- A field of a class can be described with the keyword **final**
- A final field is simply a constant variable
  - i.e., a variable that is only to be set once and is not allowed to change again over time
- A good example of a final field is defining math constant like **PI**

```
public class MathLib{
    public final double PI=3.14;
}
```

Final fields

- This basically means that even though the field is **public,** you are not allowed to change the value of PI anywhere (inside or outside of the class)

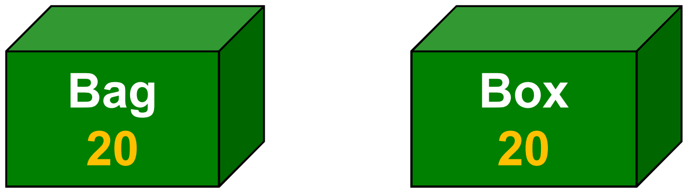
```
public static void main(String [] args){
        MathLib mathLib= new MathLib();
        mathLib.PI=0; // this is not allowed and will show a compiler
                                    error
}
```

Static

Object’s lifetime

- **Objects** that are created from a class don’t really last forever
- E.g.

Item	
String	name
double	weight



- Typically you’d create an object from a class, fills its fields with some values
- and maybe create another object and fill its fields with different values
- but then **eventuall**y both those object will get destroyed including every single value stored in those fields

Object’s lifetime…

- Typically, that would happen whenever the scope of that object ends
- E.g., inside the method, the variable **myItem** is an object of the type class Item
  - once the method ends, this variable doesn’t exists anymore, including all the values of all the fields inside it

```
public void method(){
        Item myItem = new Item();
        myItem.weight=10;
        ….
}
```

  - **myItem ???**

Static field

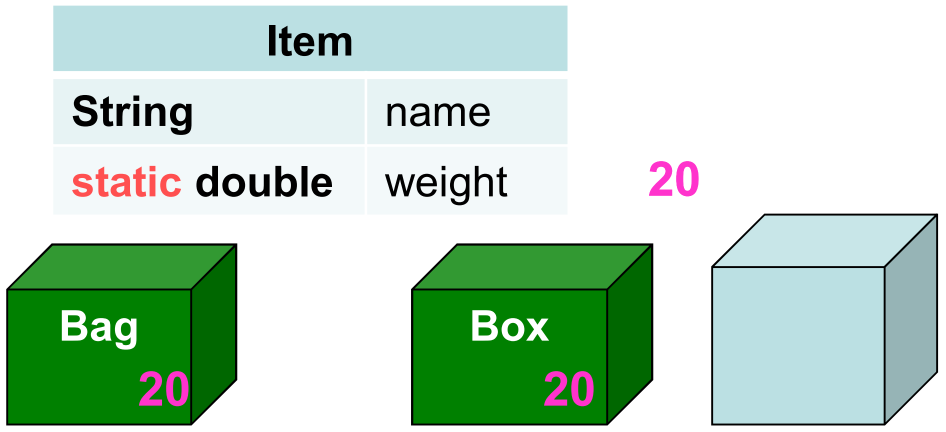
- In some occasions, you might want to store the value of a certain field even if there are **no objects** for that class
- In that case, you need to add the keyword **static** when declaring this field

Item	
String	name
static **double**	weight

- Declaring a field as **static** means that these values are..
  - no longer **within** the object itself
  - BUT **within** the class **instead,** meaning that **all** **objects of the class** will **share** that same exact value

Static field...

- And even if every single object of the class has been destroyed, the value is still stored **within** the class

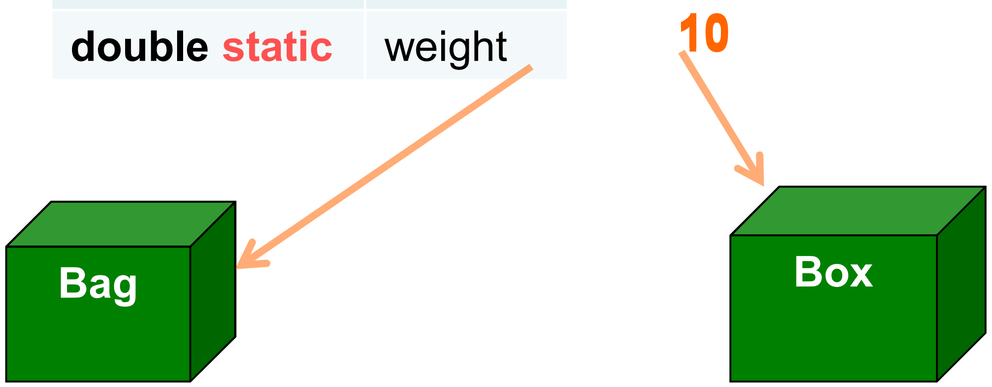


- If you decide to create a **new** object of the same class
  - then, it will end up using the same value that was stored in the class

Static field…

- Notice that
  - the **static** here doesn’t mean the *value doesn’t change*
- In fact, that value does change!
  - it will update it in *every single object* of that class again

Item	
String	name
double static	weight



Static field…

- Now because **static** fields **belong to** classes instead of object,
  - Java allows you to access a static field **directly from the class** instead of having to create an object of that class

- E.g., access the **weigh**t field from the class **Item** directly and set it to a value

```
public void method(){
        …
        Item.weight=10;
        ….
}
```

Static field...

example

```
public class Person{
    public static int count;
    Public Person(){ count++}
}
public class Main{
    public static void main(){
      for(int i = 0; i < 10; i++){
        Person person = new Person();
        System.out.println(person.count);
      }
    }
}
```

Static methods

- Just like static fields, static methods also belong to **the class** rather than the object
- It's ideally used to create a method that doesn't need to access any fields in the object
  - i.e., a method that is a standalone function
- A static method takes input argument and returns a result **based only on** those input values and nothing else
- However, **a static method** can still access **static fields**
  - that's because static fields also belong to the class and are shared among all objects of that class

Static methods...

Example

```
public class Calculator{
    public static int add(int a, int b){return a+b;}
    public static int substract(int a, int b){return a –b;}
}
```

- Since both add and subtract don't need any object-specific values, they can be declared **static** as seen above
  - and hence you can **call them directly** using **the class name** Calculator without the need to create an object variable at all
  - Calculator.add(3,3);

Static methods

- When should/shouldn't we declare fields/methods to be static
  - Most of the time, you won't declare them as static
  - But if you end up **creating a class that provides some sort of functionality** rather than have a state of its own, then it's a perfect case to use static for almost all of its methods and fields
- E.g., the Math class has a bunch of static methods like random()

Composition in Java

- Represents **part-of** relationship
- In composition, both entities are **dependent** on each other
- When there's a composition between 2 entities, **the composed object** *cannot exist without the other entity*
- **Reference variable** must be created by statement **new** or refers to another existing object

```
class Person{

        private String name;

        private MyDate birthday = new MyDate(1,1,2000);

}
```

get/set non-primitive field

```
class Person{
    ....
    public MyDate getBirthday(){
        return birthday;
    }
}
```

```
Person p=new Person();
MyDate d= p.getBirthday();
d.setYear(1990);
```

get/set with copy constructor

```
class Person{
        private String name;
        private MyDate birthday;
        public Person(String s, MyDate d){
                name=s; birthday = new MyDate(d);
        }
        public MyDate getBirthday(){
                return new MyDate(birthday);
        }
        public void setBirthday(MyDate d){
                birthday = new MyDate(d);
        }
}
```

Runtime input

- A useful application should be as **interactive** and **fun** as possible
  - i.e., allow the user to provide information at **runtime**
- E.g., for a contact manager application, it has *some useful methods*, but to use them we have to write all the code in the **main method** including all **your friends' contact details**
  - → This way, users have to write code and recompile it every time they want to make a change!
- Java allow us to accept input from the user while the program is running
  - i.e., write the main method in a way that ask the user to input their friends' names, phone numbers… then pass that information on to be stored.
- There are 4 different ways a java program can read input from the user
  - **Command line arguments**
  - **Runtime input**
  - **Files**
  - Graphical User Interface (wont be covered in this course)

Command input

- *CmdLineParas.java*

```
public class CmdLineParas{
    public static void main(String[] args){
        for(int i=0; i<args.length; i++)
            System.out.println(args[i]);
    }
}
```

- Example

```
#java CmdLineParas Hello 2019
Hello
2019
```

Input scanner

- You can ask the user to type in a message and then the java program can read it into a variable and use it
  - To do so, we use the java class called **Scanner** which is included in the **java.util** library
  - by typing this at the top of the file: **import java.util.Scanner**;
- **A Scanner** allows the program to read any data type from a particular input, if we create the scanner object like this
  - Scanner scanner = **new** Scanner(System.in)
  - This command can be used to read a **String, an integer**, or an **entire line**
  - The method **nextLine()** of the scanner object **returns a String**

Input scanner ...

- E.g.,
  - System.*out*.println("*Enter your address:*");
  - **Scanner** scanner = **new** Scanner(System.*in*)
  - **String** address = scanner.nextLine();
  - System.*out*.println("*You live at:*" + address);

- If you want to read a number into *an integer variable* instead of the entire line, then use the method **nextInt()**

  - System.*out*.println("*How old are you:*");
  - **Scanner** scanner= **new** Scanner(System.*in*);
  - **int** age = scanner.**nextInt();**
  - if(age>40)
        System.*out*.println("*Oh you're not young*!");
    else
        System.*out*.println("*You're still young* ^^*");

File scanner

- Another way of accepting runtime input is through files
  - Files can be plain text files
- To read a text file in java, you can also use the **Scanner class**,
  - but instead of reading the command line inputs by passing System.in as the argument,
  - you pass a **File object** which you can create by typing in the file name
  - **File** file = **new** File("test.txt");
  - **Scanner** fileScanner= **new** Scanner(file);
- Then, you can read lines the same way we did before (use **nextLine()**)
- To check *if the file still has more lines*, you can use **hasNextLine** method in case you want to load the entire file

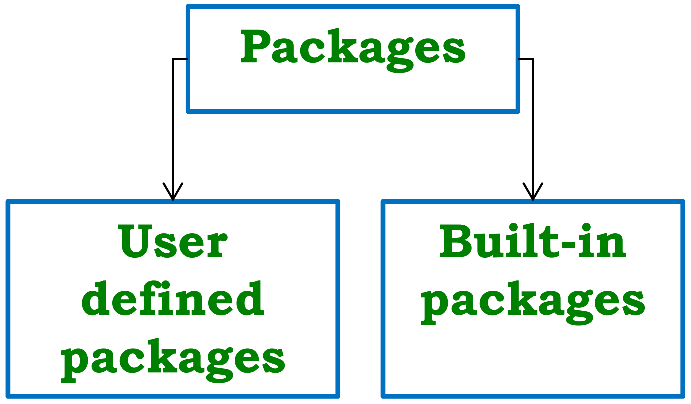
Packages

Packages in Java

- Provides a mechanism to encapsulate **a group** of *classes, sub-packages* and *interfaces*
- We'd better put **related classes** into **packages**
  - Can reuse *existing classes from the packages* as many times as we need in our program by importing a class from existing packages
- Package names and directory structure are closely related
  - E.g.:, *university.college.faculty* then there are thee directories *university, college, faculty*
- Subpackages are not imported by default
  - they have to be imported explicitly
  - E.g.: ***import*** *java.util.*; //import **all classes** from util package*
  - **util** is a subpackage created inside **java** package

Types of packages

- Built-in packages consist of a large number of classes that are a part of **Java API**
- Some common built-in packages



java.lang	contain classes for defining **primitive data types** & **math operations** (this package *imported automatically*)
java.io	support input/output operations
java.util	classes for implementing data structures like Linked List, Dictionary…,Date/Time operations
java.awt	classes for implementing the components for GUI like buttons, menu…

Types of packages…

- User-defined packages
  - First, create a directory **myPackage**
  - Then create the **MyClass** inside the directory with the first statement being the package names

```
package myPackage ;
public class MyClass{
        public void getMessage(String s){
                System.out.println(s);
        }
}
```

Types of packages…

- Now, we can use **MyClass** class in our program

```
Import myPackage.MyClass;

public class PrintName{
        Public static void main(String[] args ){
                String msg = "Test the newly built package"
                MyClass obj= new MyClass();

                obj.getMessage(msg);
        }
}
```

Handling name conflicts

- When a class name exists in *more than one package*, we need to use specific import statement
- E.g.,
  - import java.util.*;
  - import java.sql.*;
- If we declare: Date today; *//error! Because Date exists in both packages*
- Need to correct, e.g.,
  - import java.util.Date;
  - import.sql.*;
- We can use both and use in declare statement
  - java.util.Date today=new java.util.Date();
  - java.sql.Date tomorrow java.sql.Date();