Basics of the Object Oriented Programming
Classes, Nested Classes, Inheritance

Associate Professor Viorica Rozina Chifu
viorica.chifu@cs.utcluj.ro

Controlling access to members of a class

- Access level modifiers
  - Specifies if a particular field/method can be used by other classes
  - There are two levels of access control:
    - » At the level of class
      - **public**, or **package-private** (no explicit modifier)
    - » At the level of class members
      - **public**, **private**, **protected**, or **package-private** (no explicit modifier)

Controlling access to members of a class

- Modifiers at the classes level
  - **public**
    - » The class is visible to all classes everywhere
  - **default**
    - » If a class has no modifier, it is visible only within its own package

Controlling access to members of a class

- Modifiers at the member (i.e. field/method/constructor) level
  - **public**
    - » The member is visible to all classes everywhere
  - **package-private** (no explicit modifier)
    - » The member is visible only within its own package
  - **private**
    - » The member can only be accessed in its own class
  - **protected**
    - » The member can only be accessed within its own package and, by a subclass of its class in another package

Controlling Access to members of a class

- When a class is written, the access level of every member variable/ method must be decided
- When are used classes from another package, access levels determine which members of those classes can be used by the own classes

Modifier for a member of a class	Visibility of that member at class level	Visibility of that member at package level	Visibility of that member at subclass level	Visibility of that member at world level
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
no modifier	Yes	Yes	No	No
private	Yes	No	No	No

Controlling access to members of a class

- As example we consider a collection of classes, and we want to see how access levels affect the visibility

```
package one;
    class Alpha{}
    class Beta{}
package two{
   class AlphaSub extends Alpha{}
   class Gamma{}
```

Modifier	Alpha	Beta	Alphasub	Gamma
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
no modifier	Yes	Yes	No	No
private	Yes	No	No	No

  - If we have a member (e.g., a variable/method) defined in **Alpha** class which is declared **public** it will be visible in **Alpha**, **Beta**, **AlphaSub**, and **Gamma** (see the table)
  - If we have a member (e.g., a variable/method) defined in **Alpha** class which is declared **protected** it will be visible in **Alpha**, **Beta**, and **AlphaSub** (see the table)

Controlling access to members of a class

- As example we consider a collection of classes, and we want to see how access levels affect the visibility (cont.')

```
package one;
    class Alpha{}
    class Beta{}
 package two{
   class AlphaSub extends Alpha{}
   class Gamma{}
```

Modifier	Alpha	Beta	Alphasub	Gamma
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
no modifier	Yes	Yes	No	No
private	Yes	No	No	No

  - If we have a member (e.g., a variable/method) defined in Alpha class which have **no modifier** declared it will be visible in **Alpha**, and **Beta** (see the table)
  - If we have a member (e.g., a variable/method) defined in **Alpha** class which is declared **private** it will be visible only in the **Alpha** class (see the table)

Controlling access to members of a class

- Recommendation for choosing an access level:
  - Use the most restrictive access level that makes sense for a particular member
  - Use **private** unless you have a good reason not to
  - Avoid **public** fields except for constants
- **public** fields
  - Link to a particular implementation
  - Limit the flexibility in changing the code

Instance and Class Members

- Instance fields
  - Specific to objects
  - Each object created from the same class, have its own distinct copies of instance fields (variables)
- Static (class) fields
  - Have the **static** modifier in their declaration
  - Are associated with the class
    - » Are common to all objects of the class
  - Is shared by every object, instance of the class
  - Has a one fixed location in memory
  - The value of a static field can be changed by any object of the class
  - Can be manipulated without creating an object, instance of the class

Instance and Class Members

```
public class Student{
  private String name;
  private int id;
  private static int noStudents = 0;
  public Student(String n)
    { name = n;
      //increment number of Students and
      //assign ID number
      id = ++noStudents;
     }
   //new method to return the ID instance variable
  public int getID() { return id; }
  public static int getNoStudents()
            {return noStudents; }

}
```

- Example
  - **name** and **id** are instance variables in Student class
    - » Each Student object
      - Has its own values for these variables
      - Store the values in different memory locations
  - **noStudents** is a static variable
    - » Class variable are referenced to from another class by the class name
      - e.g., **Student.noStudents**
  - The constructor is used to:
    - » Set the **name** and the **id** instance variable
    - » Increment the **noStudents** class variable

Instance and Class Members

- Class methods
  - Have the **static** modifier in their declarations
  - Are invoked with the class name - this means that it is not necessary to create an instance of the class

ClassName.methodName(args)

  - Can be used to access **static** fields
    - e.g., In the **Student** class we have declared a static method, namely the **getNoStudents**() method to access the static field, **noStudents**
    - We can access this method outside from the class **Students** as in example bellow:

```
class Classroom{
  public static void main(String args[]){
    System.out.println("the number of students from this classroom is"+ Student.getNoStudents()); }
```

Instance and Class Members - Example

```
public class Car{
  private int speed;
  private int id;
  private static int noCars = 0;
  public Car(int startSpeed)
    { speed = startSpeed;
      id = ++noCars;}
  public int getID()
    { return id; }
  public static int getNoCars()
    {return noCars; }
  public int getSpeed(){return speed;}
  public void applyBrake(int decrement)
    {speed – = decrement;}
  public void speedUp(int increment)
      {speed += increment; }
}
```

Instance and Class Members

- Not all combinations of instance and class variables and methods are allowed:
  - Instance methods can access instance variables and instance methods directly
  - Instance methods can access class variables and class methods directly
  - Class methods can access class variables and class methods directly
  - Class methods **cannot** access instance variables or instance methods directly
    - They must use an object reference
  - Class methods cannot use **this** keyword as there is no instance for **this** to refer to

Instance and Class Members - Example

```
class Student{
  private int id;
  private String name;
  private static int noStudents = 0;
  public Student(String n)
   {  name = n;
      id = ++noStudents; }

   public int getID() { return id; }

   public String toString (){
     return “Nume student:”+ nume +”, id :”+id;
   }
   public void  print()
    {
   System.out.println (“Informatii student:” +
   toString());
   }
```

```
public static int getNoStudents()
          { return noStudents; }
public static void main(String args[]){
     // print() – incorrect call
      Student st = new Student(“ana”);
      st.print();
      int i = getNoStudents();
      System.out.println(“ total number of
    students:”+i);
    }
```

Instance and Class Members

- **Constants**
  - Defined by using **static** modifier, in combination with the **final** modifier
    - The purpose to use the **static** modifier is to manage the memory
      - It also allows the variable to be available without loading any instance of the class in which it is defined
    - The **final** modifier represents that the value of the variable cannot be changed
  - Cannot be reassigned, and it is a compile-time error if your program tries to do so
  - e.g., variable declaration for a constant PI:

static final double PI = 3.141592653589793;

Initializing Fields

- Example of initializing fields

```
public class BedAndBreakfast {
        public static int capacity = 10;  //initialize to 10
        private boolean full = false; //initialize to false
    }
```

  - For a field, an initial value in its declaration can be provided
  - Instance variables can also be initialized in constructors

Initializing Static Variables - Static Initialization Blocks

- Ways to initialize static variables:
  - Static Initialization Blocks
  - Private static methods

Initializing Static Variables - Static Initialization Blocks

- Is a block of code enclosed in braces, {}, and preceded by the **static** keyword

  static {//code}

- These blocks are only executed once when the class is loaded
- A class can have any number of static initialization blocks
  - The initialization blocks can appear anywhere in the class body
    - » Static initialization blocks are called in the order that they appear in the source code

- A static initializer block resembles a method with no name, no arguments, and no return type
  - It doesn't need a name, because there is no need to refer to it from outside the class definition
  - Like a constructor, a static initializer block cannot contain a **return** statement

Initializing Static Variables - Static Initialization Blocks

- Example of static initialization block:

```
public class Test {
  static int x = 0, y, z;
  static {
        System.out.println("Hi, I'm a Static Block!");
        int t =1;
        y = 2;
        z = x + y + t;
 }
```

Initializing Static Variables - Static Initialization Blocks

- Example of static initialization block:

```
public class Demo {
   static int[] numArray = new int[10];
   static {
      System.out.println("Running static initialization block.");
      for (int i = 0; i < numArray.length; i++) {
         numArray[i] = (int) (100.0 * Math.random());
      }
   }
   void printArray() {
      System.out.println("The initialized values are:");
      for (int i = 0; i < numArray.length; i++) {
         System.out.print(numArray[i] + " ");
      }
      System.out.println();
   }
   public static void main(String[] args) {
      Demo obj1 = new Demo();
      System.out.println("For obj1:");
      obj1.printArray();
      Demo obj2 = new Demo();
      System.out.println("\nFor obj2:");
      obj2.printArray();
   }
}
```

>>output:

```
Running static initialization block.
For obj1:
The initialized values are:
40 75 88 51 44 50 34 79 22 21

For obj2:
The initialized values are:
40 75 88 51 44 50 34 79 22 21
```

Initializing Static Variables - Private static methods

- Private static method
  - Is an alternative to static blocks
  - The advantage is that it can be reused later if you need to reinitialize the class variable

```
public class InitializationWithPrivateStaticMethod{
  public static int staticIntField = privStatMeth();

  private boolean instanceBoolField = true;

  private static int privStatMeth() {
     //compute the value of an int variable 'x'
     return x;
}
}
```

Initializing Instance Members

- Ways to initialize instance variables:
  - Instance initialization blocks
  - Final methods

Initializing Instance Members – Instance initialization blocks

- Instance initialization blocks
  - Look just like static initializer blocks, but without the static keyword
  - Initialization blocks are executed whenever the class is initialized and before constructors are invoked
  - There can be multiple instance initialization blocks in a class, and they are executed in the order they appear
  - The initialization of the instance variable can be done directly, but there can be performed extra operations while initializing the instance variable in the instance initializer block
  - Why use instance initializer block?
    - » Suppose I have to perform some operations while assigning value to instance data member e.g., a for loop to fill a complex array

Initializing Instance Members – Instance initialization blocks

- Example of instance initialization blocks:

```
public class Bike {
  int speed;
  Bike(){
     System.out.println("in constructor : "+speed);
  }

  {    System.out.println("in initialization bloc");
        speed=100;
  }

public static void main(String[] args) {
    Bike b1= new Bike();
    System.out.println("#########");
    Bike b2= new Bike();
}
}
```

Output >>
in initialization bloc
in constructor : 100
#########
in initialization bloc
in constructor : 100

Initializing Instance Members – Instance initialization blocks

- Initializer blocks of instance variables
  - Rules for instance initializer block:
    - The instance initializer block is created when instance of the class is created
    - The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call)
    - The instance initializer block comes in the order in which they appear

Initializing Instance Members – Final methods

- Final methods can be used to initialize instance variables
  - Are methods that can not be overridden in a subclass
  - The purpose of making a method **final** is to prevent modification of a method from outside (child class)

Initializing Instance Members – Final methods

- Example of initializing using final methods:

```
public class FinalMethods {
    String name = getName();

 protected final String getName() {
      name="Ana";
      return name;
   }

   public void display(){
      System.out.println("Name value is: " +this.name);

   }
   public static void main(String args[]){
      FinalMethods obj = new FinalMethods();
      obj.display();
   }
}
```

```
>>output: Name value is: Ana
```

Nested Classes

- Are member of its enclosing class
- Can be declared as:
  - **private**, **public**, **protected**, or **package private**
- Are divided into two categories:
  - Static nested classes
    - » Nested classes that are declared **static**
    - » Do not have access to other members of the enclosing class
  - No - static nested classes
    - » Are also called **inner** *classes*
    - » Have access to other members of the enclosing class, even if they are declared **private**

Nested Classes

- Example of nested classes

```
class OuterClass
  { ...
    static class StaticNestedClass
        {...}
    class InnerClass
      {...}
    }
```

Nested Classes

- Are used because:
  - Is a way of **logically grouping classes** that are only used in one place
    - If a class is useful to only one other class, then it is embedded in that class and keep the two together
  - **Increases encapsulation**
    - Consider two top-level classes, **A** and **B**, where **B** needs access to members of **A** that are declared **private**
      - By hiding the **B** class within the **A** class (**B** is declared as inner class of **A**), **A**'s members can be declared private, and **B** can access them
        - In addition, **B** itself can be hidden from the outside world.
  - Can lead to **more readable** and **maintainable code**
    - Nesting small classes within top-level classes places the code closer to where it is used

Nested Classes - Static nested class

- Static nested classes are declared in Java like this:

```
public class Outer {
  public static class Nested {
  }
}
```

- In order to create an instance of the static nested class you must reference it by prefixing it with the Outer class name, like this:

```
Outer.Nested instance = new Outer.Nested();
```

- Cannot refer directly to instance variables or methods defined in its enclosing class
  - It can use them only through an object instance of that class
- Interacts with the instance members of its outer class (and other classes) just like any other class

Nested Classes - Static nested class

- Static nested classes example:

```
class OuterClass
{
    // static member
    static int outer_x = 10;

   // instance(non-static) member
    int outer_y = 20;

    // private member
    private static int outer_private = 30;

// static nested class
    static class StaticNestedClass
    {
        void display()
        {
            // can access static member of outer class
            System.out.println("outer_x = " + outer_x);

            // can access display private static member of outer class
            System.out.println("outer_private = " + outer_private);

            // The following statement will give compilation error
            // as static nested class cannot directly access non-static members
            // System.out.println("outer_y = " + outer_y);

            //access instance member
            System.out.println("outer_y = " + new OuterClass().outer_y);
         } }}
```

Nested Classes – Non- Static nested class

- Non-static nested classes are declared in Java like this:

class *OuterClass* {

... **class** *InnerClass* { ... }

}

- Are associated with an instance of its enclosing class
- It cannot define any static members itself
- Objects that are instances of an inner class exist within an instance of the outer class
  - Thus, you must first create an instance of the enclosing class to create an instance of an inner class

OuterClass outerObject= **new** *OuterClass*()

OuterClass.InnerClass innerObject = outerObject.**new** *InnerClass*();

Nested Classes – Non- Static nested class

- Inner class has direct access to methods and fields of its enclosing instance, even if they are declared private
- Example of accessing a private field:

```
public class Outer {
    private String text = "I am private!";
    public class Inner {
        public void printText() {
            System.out.println(text);
        }
    }
}
     Outer outer = new Outer();
     Outer.Inner inner = outer.new Inner();
     inner.printText();
```

Nested Classes – Non- Static nested class

- **Inner Class Shadowing**
  - If a **Java inner class** declares fields or methods with the same names as field or methods in its enclosing class, the inner fields or methods are said to **shadow** over the outer fields or methods

```
public class Outer {
    private String text = "I am Outer private!";
    public class Inner {
        private String text = "I am Inner private";
        public void printText() {
            System.out.println(text);
        }    }}
```

```
public class Outer {
    private String text = "I am Outer private!";
    public class Inner {
        private String text = "I am Inner private";
        public void printText() {
            System.out.println(text);
            System.out.println(Outer.this.text);
        }   }}
```

  - In the above example both the **Outer** and **Inner** class contains a field named **text**.
  - When the **Inner** class refers to **text** it refers to its own field.
  - When **Outer** refers to **text** it also refers to its own field
  - Java makes it possible though, for the **Inner** class to refer to the **text** field of the **Outer** class
    - » To do so it has to prefix the text field reference with **Outer.this.** (the outer class name + .this. + field name)

Nested Classes – Non- Static nested class

- There are two kinds of inner classes
  - Local inner classes
    - An inner class declared within the body of a method
  - Anonymous inner classes
    - An inner class declared within the body of a method without naming it
- Modifiers for inner class
  - Can be used the same modifiers that are used for other members of the outer class
    - e.g., you can use the access modifiers **private**, **public**, and **protected** to restrict access to inner classes

Nested Classes

```
public class DataStructure {
   private final static int SIZE = 15;
   private int[] arrayOfInts = new int[SIZE];
   public DataStructure() {
     //fill the array with ascending integer values
      for (int i = 0; i < SIZE; i++)
        {arrayOfInts[i] = i;}}

  public void printEven() {
    //print out values of even indices of the array
    DataStructure out= new DataStructure();
    DataStructure.InnerEvenIterator iterator =
                                  out.new InnerEvenIterator();
      while (iterator.hasNext())
        {System.out.println(iterator.getNext() + " ");
      } }
```

```
//inner class implements the Iterator pattern
private class InnerEvenIterator {
 private int next = 0;
 public boolean hasNext() {
         //check if a current element is the last in the array
           return (next <= SIZE - 1); }
  public int getNext() {
       //record a value of an even index of the array
       int retValue = arrayOfInts[next];
      //get the next even element
        next += 2;
        return retValue; } }

 public static void main(String s[]) {
   //fill the array with integer values and print out only values
   of even indices
    DataStructure ds = new DataStructure();
    ds.printEven(); }
}
```

Nested Classes

- **DataStructure** class consists of:
  - **DataStructure** outer class which has:
    - An array filled with integer values
    - A method to **add** an integer onto the array
    - A method to **print** out values of even indices of the array
  - **InnerEvenIterator** inner class, which:
    - Refers directly to the **arrayOfInts** instance variable of the **DataStructure** object
    - Is like a standard Java **iterator**
- Iterators
  - Are used to step through a data structure
  - Have methods to test for the last element, retrieve the current element, and move to the next element

Reusing Classes

- Reuse code by creating new classes based on existing classes
- There are two ways to accomplish this:
  - **Inheritance**
    - Creates a new class as a **type** of an existing class
    - The form of the existing class is reused
  - **Composition**
    - Objects of an existing class are contained in a new class
      - The new class is **composed** of objects of existing classes
    - The functionality of the code is reused

Reusing Classes -Inheritance

- Is one of the main techniques of object-oriented programming
- Inheritance supports the concept of “reusability”
  - By inheritance, the fields and the methods of the parent class can be reused in the child class
- Allows to create new classes that are built upon existing classes
- Moreover, the child can add its own fields and methods in addition to the superclass fields and methods

Reusing Classes - Inheritance

- Models “is *a*” relationships which is also known as a ***parent-child*** relations
  - Is a mechanism in which one object acquires all the properties and behaviors of a parent object
  - Inheritance uses similarities and differences to model groups of related objects
- Where there’s Inheritance, there’s an Inheritance Hierarchy of classes

Reusing Classes - Inheritance

- **Concepts used in inheritance:**
  - **Generalization**: Extracts shared characteristics from two or more classes, and combining them into a generalized **superclass**
    - » Shared characteristics can be attributes, or methods
  - **Specialization**: Creates new subclasses from an existing class
    - » A subclass contains specific attributes, or methods that only apply to the objects of that class
- **Super Class:**
  - The class whose features are inherited (known as parent class or base class)
- **Sub Class:**
  - The class that inherits the other class (known as derived class, extended class, or child class)

Reusing Classes

- Inheritance is a way of:
  - Organizing information
  - Grouping similar classes
  - Modeling similarities between classes
  - Creating a taxonomy of objects

Reusing Classes

- Inheritance
  - In Java can only inherit from one superclass
  - C++ allows a subclass to inherit from multiple super-classes (error prone)
  - In Java, every class extends the **Object** class which is most generic class either directly or indirectly

Reusing Classes

- Inheritance
  - Is specified by the **extends** keyword
  - A subclass, is defined by starting with another already defined class, called superclass, and adding (and/or changing) methods, instance variables, and static variables
    - » The subclass inherits all public methods, as well as public and private instance and static variables from the superclass.
    - » The subclasses can add more instance variables, static variables, and/or methods
    - » Definitions for the inherited variables and methods do not appear in the derived class
    - » The code is reused without having to explicitly copy it, unless the creator of the subclass redefines one or more of the superclass methods

Reusing Classes - Inheritance

- Example of inheritance

```
// base class
class Bicycle {
    // the Bicycle class has two fields
    public int gear;
    public int speed;
     // the Bicycle class has one constructor
    public Bicycle(int gear, int speed)
    {
        this.gear = gear;
        this.speed = speed;
    }

   // the Bicycle class has three methods
    public void applyBrake(int decrement)
     { speed -= decrement; }

   public void speedUp(int increment)
    {speed += increment; }

    // toString() method to print info of Bicycle
    public String toString()
    {
        return ("No of gears are " + gear + "\n"
                + "speed of bicycle is " + speed);
    }
}
```

Reusing Classes - Inheritance

- Example of inheritance

```
// derived class
class MountainBike extends Bicycle {
     // the MountainBike subclass adds one more field
    public int seatHeight;
     // the MountainBike subclass has one constructor
    public MountainBike(int gear, int speed, int startHeight)
    {
        // invoking base-class(Bicycle) constructor
        super(gear, speed);
        seatHeight = startHeight;
    }
     // the MountainBike subclass adds one more method
    public void setHeight(int newValue)
    {   seatHeight = newValue;  }

 // overriding toString() method
 of Bicycle to print more info
 @Override
public String toString()
    {
        return (super.toString() + "\nseat height is "
                + seatHeight);
    }
}
```

Reusing Classes - Inheritance

- Example of inheritance

```
// driver class
public class Test {
    public static void main(String args[])
    {

        MountainBike mb = new MountainBike(3, 100, 25);
        System.out.println(mb.toString());
    }
}
```

Reusing Classes - Inheritance

- Types of inheritance in java
  - There can be three types of inheritance in java:
    - » **Single Inheritance**
      - In single inheritance, subclasses inherit the features of one superclass
      - In the image, class A serves as a base class for the derived class B
    - » **Multilevel Inheritance**
      - A derived class will be inheriting a base class and as well as the derived class also act as the base class to other class
      - In the image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C
      - In Java, a class cannot directly access the grandparent's members
    - » **Hierarchical Inheritance**
      - In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass
      - In the image, class A serves as a base class for the derived class B, C .