Basics of the Object-Oriented Programming
Classes and Inheritance, Interfaces

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

Inheritance - Advantage of inheritance

- Supports incremental development
  - New code can be introduced without causing bugs in existing code
  - If a bug happens, it will be in the new code
    - » Is much shorter and easier to read the new code inside of modifying the body of existing code

Inheritance - protected keyword

- Is used in the context of inheritance
  - The **protected** fields/methods of a class are available in all classes that inherits from that class or anyone else in the same package
- The best approach in inheritance is to:
  - Declare the fields (i.e., attributes) of the superclass as **private** and controlled access to the **private fields** of the class through **protected** methods

Inheritance - **protected** keyword

- Example of using **protected** keyword in the context of inheritance:

```
import java.util.*;
   class Person{
     private String name;
     protected void set(String nm)
          { name = nm;  }
     public Person(String name)
          {this.name = name;}
     public String toString()
       { return "I'm a person and my name is " +name; }
      }
```

```
import java.util.*;
public class Child extends Person {
  private int age;
  public Child(String name, int age) {
        super(name);
        this.age = age; }
  public void change(String name, int age) {
           set(name); // Available because it's protected
           this.age = age;}
   public String toString() {return "I'm a person and my name is
   " +name; +“and I have " + age + “ age”;}

     public static void main(String[] args) {
           Child ch = new Child(“Anna”, 12);
           System.out.println(ch.toString());
          ch.change(“Bob”, 11);
           System.out.println(ch.toString());  } }
```

Inheritance - **final** keyword

- Means “This cannot be changed”
- Can be used in field declaration, method declaration or class declaration

Inheritance - Final Fields

- Written in the declaration of primitive field (i.e., attribute), **final** makes the value a constant
- Written in the declaration of an object reference, **final** makes the reference a constant
  - Once the reference is initialized to an object, it can never be changed to point to another object
  - However, the object itself can be modified

Inheritance – Final Fields

- Example of using **final** keyword in the declaration of a field
  - Any attempt to reassign *radius*, PI will meet with a compile error

```
public class Circle {
   // final variables
    public static final double PI = 3.141592653589793;
    public final double radius=1;
    public double xpos;
    public double ypos;
    Circle(double x, double y, double z, double r) {
         radius = r;
         xpos = x;
         ypos = y; }
 }
```

Inheritance – Final Fields

- Blank final fields - Java allows the creation of **blank final fields**
  - Are fields that are declared as **final** but are not given an initialization value
  - Can only be assigned once and it must be unassigned when an assignment occurs
  - Assignments to **finals** fields must be performed either with an expression at the point of definition of the field or in every constructor
    - » That way it’s guaranteed that the **final** field is always initialized before use

Inheritance – Final Fields

- Example of Blank final fields :

```
class First
  {  private int i;
      First(int a) {i = a;}
  }
public class BlankFinal {
   private final int i = 0; // Initialized final field
   private final int j; // Blank final field
   private final First  p; // Blank final reference field
   // Blank finals MUST be initialized in the constructor:
   public BlankFinal() {
       j = 1;  //Initialize blank final
      p = new First(1);} //Initialize blank final reference
public static void main(String[] args) {
      new BlankFinal();}
      }
```

Inheritance - Final methods

- Are method declared as final
  - A method declared as **final** can not be overridden in its subclasses
  - **final** and **private**
    - Any **private** methods in a class are implicitly **final**
      - Because you can’t access a private method from outside the class, you can’t override it

Inheritance - Final classes

- A class declared **final** means that it can not be inherited
- The fields of a **final** class can be **final** or not
- A method in a **final** class can be declared as **final**, but it doesn't add any meaning
  - This because the methods from a final class cannot be override

Inheritance - Final classes

- Example of Final Class:

```
// Making an entire class final
 class SmallBrain {}

final class Dinosaur {
        int i = 7;
        int j = 1;
        SmallBrain x = new SmallBrain();
        void f() {}
   }
```

```
 //! class F extends Dinosaur {}
 // error: Cannot extend final class 'Dinosaur'

 public class Jurassic {
   public static void main(String[] args) {
    Dinosaur n = new Dinosaur();
    n.f();
    n.i = 40;
    n.j++;
  }
}
```

Inheritance - Overriding method

- Overriding method
  - Appears in the context of class inheritance (i.e., subclass relation)
  - Refers to the specialization of a method from superclass in its subclasses
  - The method that is override in subclass has a different body

Inheritance - Overriding method

- Rules for method overriding:

  - ✓ The **argument list** (i.e., parameters) should be the same as that of the overridden method
  - ✓ The **return type** should be the same or a subtype of the return type declared in the original overridden method in the superclass
  - ✓ The access level cannot be more restrictive than the overridden method's access level
    - ✓ i.e., If the superclass method is declared **public** then the overriding method in the sub class cannot be either **private** or **protected**
  - ✓ A method declared **final** cannot be overridden
  - ✓ A method declared **static** cannot be overridden but can be re-declared
  - ✓ A subclass within the same package as the superclass can override any superclass method that is not declared **private** or **final**
  - ✓ A subclass in a different package can only override the **non-final methods** declared **public** or **protected**
  - ✓ Constructors cannot be overridden

Inheritance - Overriding method

- Example of overriding a method

```
public class Point{
 int x, y;
 Point( int x, int y){
      this.x=x;
      this.y=y;}
public String toString(){
    return “ coordinates x=“+ x + ”and y=“+y; }
}
public class Shape {
    protected Point org;
    public Shape (Point org) {
        this.org = org;
    }
  public double perimeter() {return 0.0;}
  public double area() {return 0.0;}
 }
```

```
public class Circle extends Shape{
  protected double radius;
  public Circle(Point pc, double radius) {
    super(pc);
    this.radius = radius; }
  public double perimeter()
      {return 2.0 * Math.PI * radius;}
  public double area()
      {return  Math.PI * radius * radius;}
   // String conversion
   public String toString()
    {
     return "Circle with center of " + org.toString() + " and radius " +  radius;
   }
}
```

Inheritance - Overriding method

- Example of overriding a method (cont.’)

```
import java.awt.Color;
public class ColoredCircle  extends Circle {
  private Color color;
  public ColoredCircle(Point  pc, double radius, Color color)
       {
          super(pc, radius);
          this.color = color;
       }
public String toString()
      {
   return  super.toString() + “ and color:" + color;
      }
}
```

Inheritance – super keyword

- The **super** keyword
  - Is similar to **this** keyword
  - It is used to:
    - » Differentiate the members of superclass from the members of subclass, if they have same names
    - » To invoke the superclass constructor from subclass
- Differentiating the Members
  - If a class inherits another class and if the members of the superclass have the names same as the members of sub class, to differentiate between them we use **super** keyword:
    - » super.variable;
    - » super.method();

Inheritance –super keyword

- Example of calling a method defined in the parent class from the child class
  - In the **ColoredCircle** class we can have access to the **toString**() method from the **Circle** class as follow:

```
public String toString()
   {  return  super.toString() + “ and color:" + color;  }
```

Inheritance – super keyword

- You cannot use multiple **supers**
  - It is only valid to use **super** to invoke a method from a direct parent
  - Repeating **super** will not invoke a method from some other ancestor class
  - For example, if the **Circle** class inherit the class **Shape**, and the **ColoredCircle** class inherit the **Circle** class, it would not be possible to invoke **toString**() method of the **Shape** class within a method of the **ColoredCircle** class as follows:

```
super.super.toString() // ILLEGAL!
```

Inheritance - super keyword

- Invoking the constructor from the super class in its subclass
  - We call the constructor of the parent class by using **super** keyword
  - The call of the constructor of the parent class that must be **the first statement** in the body of the constructor of the subclass:
- Example of invoking the constructor from superclass in its subclass:
  - **super**(pc, radius) calls the constructor from **Circle** class which in turn calls the constructor from **Shape** class

```
public ColoredCircle(Point  pc, double radius, Color color)
   {  super(pc, radius);
       this.color = color; }
public Circle(Point pc, double radius)
   { super(pc);
      this.radius = radius; }
```

Inheritance – this() and super()

- The difference between **this**() and **super**() in the context of constructors call in class hierarchies
  - Let’s modify the **ColoredCircle** class as follows:

```
public class ColoredCircle  extends Circle
 {
      private int color;
      public ColoredCircle(Point  pc, double radius, Color color)
        {  super(pc, radius);
           this.color = color;  }
      public ColoredCircle(Point  pc, double radius)
        { this(pc, radius, Color.green); }
      public String toString()
        { return  super.toString() + “ and color:" + color; }
  }
```

Inheritance – this() and super()

- The difference between **this**() and **super**() in the context of constructors call in class hierarchies
  - **super**()
    - Represents the call of the constructor from the superclass (i.e., parent class)
    - Must be first statement in the body of the constructor of the subclass
    - Example:
      - In **ColoredCircle** class, **super**(pc, radius) invokes the constructor from the superclass that matches the given signature - **super**(Point, *double*) (e.g., **public Circle**(**Point** *pc*, **double** *radius*))
  - **this**()
    - Invokes a constructor of the current class that matches the given signature
    - Must be the first statement in the constructor
    - Example
      - In **ColoredCircle** class, **this**(pc, radius, **Color.green**) invokes the constructor **ColoredCircle(Point, double, Color)**

Inheritance - Special rules in the context of constructors call in class hierarchies

- If no constructor is defined in the subclass
  - The default constructor (i.e., constructor with no arguments) is provided by default
    - The default constructor from subclass invokes the default constructor from superclass
- If a constructor with parameters is defined in the superclass, the default constructor is not implicitly provided in the superclass
  - In this case, when the default constructor from subclass try to call the default constructor from superclass, a compilation error occur

```
public class SuperC
   {…}
public class SubC extends SuperC {
  public SubC() {super();}
  // … fields and methods
}
```

```
public class SuperC
   {
     int field;
     SuperC(int f){field=f;}
     }
public class SubC extends SuperC {
  public SubC() {super();} //COMPILATION ERROR
  // … fields and methods
}
```

Inheritance - Name hiding

- Overloading a method from superclass in subclass does not hide the method from the superclass

All the overloaded methods of superclass are available in subclass, even though subclass introduces a new overloaded method

Inheritance - Name hiding

- Example of name hiding:

```
class SuperClass {
    char test(char c)
      {  System.out.println(“test(char)");
           return ‘d‘;  }
    float test(float f)
     {  System.out.println(“test(float)");
        return 1.0f;  }
   }
 class Example
    {… }
```

```
class SubClass extends SuperClass {
    void test(Example m)
     { System.out.println(“test(Example)"); }
   }

public class Main {
  public static void main(String[] args) {
     Subclass b = new Subclass();
     b.test('x');
     b.test(1.0f);
     b.test(new Example());
   }
 }
```

Inheritance - Hierarchy of classes

- A hierarchy of classes composed of super-classes and subclasses

```
public class Shape {…}
public class Circle extends Shape {…}
public class ColoredCircle extends Circle {...}
```

Inheritance - Hierarchy of classes

```
 public class Shape{
          protected Point org;
          public Shape(Point org)
      { this.org = org; }
           public double perimeter()
      { return 0.0;}
           public double area()
      {return 0.0;} }
```

```
public class Circle extends Shape{
   protected double radius;
   public Circle(Point pc, double radius)
     {    super(pc);
          this.radius = radius;
      }
     public double perimeter()
          {return 2.0 * Math.PI * radius;}
     public double area()
          {return  Math.PI * radius * radius;}
      //String conversion
    public String toString()
        {return "Circle with " + org.toString() + " and radius " + radius; }}
```

```
public class ColoredCircle extends Circle{
 private int color;
 public ColoredCircle(Point pc, double radius, int color)
        { super(pc, radius);
          this.color = color;
          }
 public String toString()
         { return  super.toString() + "color:" + color;}
}
```

Inheritance – Subtype relation

- An object of a subclass has the **type** of the subclass, and it also has the **type** of the superclass
- More generally, an object of a subclass has the **type** of every one of its superclass
  - **Therefore, an object of a subclass can be assigned to a variable of any superclass**
- An object of a subclass can be plugged in as a parameter in place of any of its superclasses
  - In fact, an object of a subclass can be used anyplace that an object of any of its superclasses can be used
- However, that this relationship does not go the other way
  - An object of superclass type can never be used in place of one of its derived types

Inheritance - Substitutability of subtypes

- Conversion of reference types
  - Governed by subtype relation
- Up casting
  - The conversion of a **subtype** (i.e., subclass) to one of its **super-types** (i.e., superclass)
  - Is carried out implicitly whenever necessary
- Downcasting
  - The conversion of a super-type (i.e., superclass) to one of its subtypes (i.e., subclass)
  - Requires explicit casts

Inheritance – Up casting Example

```
import java.util.*;
 class Instrument{
    public void play() {}
    static void playSymphony (Instrument i)
       { i.play(); }
     }
     //Wind objects are instruments because they have the same
     interface:
  public class Wind extends Instrument{
    public static void main(String[] args) {
      Wind flute = new Wind();
      Instrument. playSymphony(flute);  //Upcasting}
    }
```

- playSymphony() method
  - Accepts an **Instrument** reference
  - Inside **playSymphony** the code works for **Instrument,** and anything derived from **Instrument**
- In main() from Wind
  - The **playSymphony** method is called by giving it a **Wind** reference
- Upcasting
  - The act of converting a **Wind** reference into an **Instrument** reference

Inheritance - Up casting

- Is always safe because you’re going from a more specific type to a more general type
  - The subclass is a superset of the superclass
    - » It might contain more methods than the superclass, but it must contain **at least** the methods in the superclass
    - » The only thing that can occur to the class interface during the up casting is that it can lose methods, not gain them (this is why the compiler allows up casting without any explicit casts or other special notation)

Inheritance - Downcasting

- **Downcasting** is casting a supertype to a subtype
- Example:

```
Animal anim = new Cat();
Cat cat = (Cat) anim;
```

  - Here, we cast the Animal type to the Cat type
  - As **Cat** is subclass of **Animal**, this casting is called **downcasting**
- Unlike upcasting, downcasting can fail if the actual object type is not the target object type
  - For example:

```
Animal anim = new Cat();
Dog dog = (Dog) anim;
```

- This will throw a **ClassCastException** because the actual object type is Cat, and a Cat is not a Dog so we cannot cast it to a Dog

Inheritance - Downcasting

- The Java language provides the **instanceof** keyword to check type of an object before casting
- It is recommended to use the **instanceof** operator to check the type before casting because this eliminates the risk of a *ClassCastException* thrown

```
if (anim instanceof Cat)
 {  Cat cat = (Cat) anim;
    cat.meow();  }
else if (anim instanceof Dog)
 {  Dog dog = (Dog) anim;
    dog.bark(); }
```

Inheritance - Downcasting

- Use downcasting when we want to access specific behaviors of a subtype
- Example:

```
public class AnimalTrainer {
    public void teach(Animal anim)
      {
        anim.move();
        anim.eat();
        // if there's a dog, tell it barks
        if(anim instanceof Dog)
          {  Dog dog = (Dog) anim;
             dog.bark(); }
        }
    }
```

  – Here, in the teach() method, we check if there is an instance of a Dog object passed in, downcast it to the *Dog* type and invoke its specific method, *bark*()

Inheritance – Conclusions on Downcasting and Upcasting

- Casting does not change the actual object type
  - Only the reference type gets changed
- Upcasting is always safe and never fails
- Downcasting can risk throwing a *ClassCastException*, so the **instanceof** operator is used to check type before casting

Inheritance - Abstract classes

- An **abstract class** is declared with the **abstract** keyword
- **No object can be instantiated from an abstract class**
- Abstract classes save time, since we do not have to write “useless” code that would never be executed

Inheritance - Abstract classes

- ✓ Are classes that has at least one abstract method
  - ✓ Abstract method is a method that has no definition body
  - ✓ Abstract methods must be defined in the subclasses
- ✓ Abstract classes could not be instantiated into objects
  - ✓ References to abstract classes could be passed as parameters in methods
- ✓ Abstract class contains
  - ✓ Instance and class (static) variables
  - ✓ Abstract and implemented methods
  - ✓ Constructors
- ✓ Are usefulness in the generalization process
  - ✓ Every subclass of an abstract class that will be used to instantiate objects must provide implementations for all abstract methods in the superclass
- ✓ An abstract class can inherit abstract methods
  - ▪ From an interface, or
  - ▪ From a class

Inheritance - Abstract classes

- Consider the system of classes **Shape**, **Circle**, and **ColoredCircle**
  - Instances of **Shape class** are not very useful
  - Class **Shape** is better to be declared as an abstract class

```
public abstract class Shape {
    private static int counter;
    public Shape (Point org)
      { this.org = org; }
  public abstract double perimeter();
  public abstract double area();
  public int getCount ()
      { return counter; }
   }
```

Inheritance - Abstract classes

```
public class Circle extends Shape{
  protected double radius;
  public Circle(Point pc, double radius)
     {  super(pc);
        this.radius = radius; }
public double perimeter()
      {return 2.0 * Math.PI * radius;}
  public double area()
      {return  Math.PI * radius * radius;}
   // String conversion
   public String toString()
     { return "Circle with radius " + radius; }
 }
```

```
public class Rectangle extends Shape{
 private double width;
 private double height;
 public Rectangle()
   { this(0, 0);  }
  public Rectangle(double lg, double lat)
     { width = lg;
       height = lat; }
  public double area() { return width * height; }
  public double perimeter(){ return 2*(width*height) }
  public String toString()
    { return “Rectangle with width= " + width+ " and height = " + height; }
}
```

Selective inheritance - Adapter classes

- Abstract methods
  - All methods are inherited
  - All methods should be approached somehow
- Adapter classes in the inheritance context
  - Alternative to abstract classes
  - Contain methods that are implemented with a default behavior (common: empty implementation body)
  - Are a solution to selective inheritance

Selective inheritance – Example of Adapter classes

```
public class Adapter {
        public int m1(…) { }
        public double m2(…) { }
        public String m3(…) { }
        // … other class resources }

 //Alpha overrides m1 and m3
public class Alpha extends Adapter {
        public int m1(…)
              {// … m1 specific implementation}
        public String m3(…)
              {// … m3 specific implementation }
               // … other class resources }

 //Beta overrides m1 and m2
 public class Beta extends Adapter {
        public int m1(…)
                { // … m1 specific implementation}
        double m2(…)
               {// … m2 specific implementation}
                // … other class resources}
```

Interfaces

- An interface is a reference type in Java similar to class
- It is a collection of abstract methods
- A class implements an interface, thereby inheriting the abstract methods of the interface
- Along with abstract methods, an interface may also contain:
  - Constants
  - Default methods
  - Static methods
  - Nested types
- Method bodies exist only for **default methods** and **static methods**
- Writing an interface is similar to writing a class, but a class describes the state and behaviors of an object, while an interface contains behaviors that a class implements

Interfaces

- An interface is different from a class in several ways:
  - You **cannot instantiate** an interface
  - An interface does **not contain any constructors**
  - An interface **cannot contain instance fields**
    - The only fields that can appear in an interface must be declared both **static and final**
  - An interface is **not extended by a** class; it is **implemented** by a class
  - An **interface** can **extend multiple interfaces**

Interfaces

- Interface declaration
  - Use **interface** keyword to create an interface
  - Add the **public** keyword before the **interface** keyword or leave it off to give package access

```
public interface NameOfInterface {
   // Any number of final, static fields
   // Any number of abstract method declarations
  //default and static methods starting from java 8
}
```

Interfaces

- Interfaces have the following properties:
  - An interface is implicitly **abstract**
    - You do not need to use the **abstract** keyword while declaring an interface
  - Each method in an interface is also implicitly **abstract**, so the abstract keyword is not needed
  - Methods in an interface are implicitly **public**

```
/* File name : Animal.java */
interface Animal {
   public void eat();
   public void move();
}
```

Interfaces

- Implementing interfaces
  - When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface
  - All the methods of the interface need to be defined in the class that implements the interface
  - If a class does not perform all the behaviors of the interface (i.e., does not implements all the methods from the interface), the class must declare itself as **abstract**

Interfaces - Examples

```
 interface Animal {
    public void eat();
    public void move();
 }

public class Mammalian implements Animal {
   public void eat()
      { System.out.println("Mammal eats"); }
   public void move()
     { System.out.println("Mammal moves");  }
   public int noOfLegs() {  return 0;   }
   public static void main(String args[])
    { Mammalian m = new Mammalian ();
      m.eat();
      m.move(); }
                                    }
```

Interfaces

- When overriding methods defined in interfaces, there are several rules to be followed:
  - For each implemented method, the declared exceptions, should be the same as the ones declare by the interface method or subclasses of those exceptions declared by the interface method
  - The implemented method should have the same signature (i.e., name and parameters) as the interface method and should return the same type or subtype of the return type of the interface method
  - An implementation class itself can be abstract and if so, interface methods need not be implemented
- When implementation interfaces, there are several rules
  - A class can **implement more than one** interface at a time
  - A class can **extend** only one class, but **implement many interfaces**

Interfaces

- Extending Interfaces
  - An interface can **extend** another interface in the same way that a class can extend another class
  - The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface

Interfaces - Example of extending interfaces

```
// Filename: Sports.java
public interface Sports {
   public void setHomeTeam(String name);
   public void setVisitingTeam(String name); }

// Filename: Football.java
public interface Football extends Sports {
   public void homeTeamScored(int points);
   public void visitingTeamScored(int points);
   public void endOfQuarter(int quarter); }

// Filename: Hockey.java
public interface Hockey extends Sports {
   public void homeGoalScored();
   public void visitingGoalScored();
   public void endOfPeriod(int period);
   public void overtimePeriod(int ot); }
```

- ✓ The **Hockey** interface has four methods, but it inherits two from **Sports**
- ✓ Thus, a class that implements **Hockey** needs to implement all six methods
- ✓ Similarly, a class that implements **Football** needs to define the three methods from **Football** and the two methods from **Sports**
- ✓ An interface can **extend** more than one parent interface
  - The **extends** keyword is used once, and the parent interfaces are declared in a comma-separated list

public interface Hockey **extends** Sports, Event

Interfaces - Multiple inheritance in Java

- Hero class combines the class **Action** with the interfaces **CanSwim**, and **CanFly**

```
interface CanSwim
  { void swim(); }

interface CanFly
 { void fly(); }

class Action
  { public void fight() {} }

class Hero extends Action implements CanSwim, CanFly
   {  public void swim() {}
      public void fly() {}
   }
```