Basics of the Object-Oriented Programming

Associate Professor Viorica Chifu

Interfaces – Default Method

- Example of default method declared in an interface (this feature in available starting with Java 8.0)

```
public interface Interface1{
      void method1(String str);
      default void log(String str)
          { System.out.println("I1 logging::"+str); }
}
```

  - log(String str) is the **default method** in the Interface1
  - when a class will implement Interface1, it is not mandatory to provide implementation for **default** methods of interface

Interfaces - Default Methods

- Let’s consider that we have another interface with following methods:

```
public interface Interface2 {
   void method2();
   default void log(String str)
     { System.out.println("I2 logging::"+str); }
}
```

  - If we have a class that implementing both **Interface1** and **Interface2** and doesn’t implement the common default method, compiler can’t decide which one to chose
  - In this case it’s made mandatory to provide implementation for common **default** method (i.e., log() method) of interfaces, otherwise compiler will throw compile time error

Interfaces - Default Methods

- Example of implementing the default method

```
public class MyClass implements Interface1, Interface2 {
        @Override
        public void method2() {}

        @Override
        public void method1(String str) {}

        @Override
        public void log(String str)
            { System.out.println("MyClass logging::"+str); }
}
```

Interfaces - Default Methods

- Important points about java interface default methods:
  - Default methods has bridge down the differences between **interfaces** and **abstract classes**
  - Default methods will help us in removing the implementation from the base classes
    - We can provide default implementation in the interface and the classes that implements interface can chose which one of the default method to override
  - One of the major reason for introducing default methods in interfaces is to enhance the Collections API in Java 8 to support lambda expressions

Interfaces - Static Methods

- **Static** method in a Java Interface
  - Is a method defined in the interface with the keyword **static**
  - Unlike other methods in **interface**, a static method contain the complete definition of the function
  - A **static method** cannot be **overridden** or **changed** in the implementation class
  - To use a static method in a class, you need to precede the name of the method with the name of the interface

Interfaces – Static Methods

- Example 1 of using static methods:
  - In this example a static method is defined in an interface and is called in class **InterfaceDemo** which implements the interface:

```
interface NewInterface {
  static void hello()
    {System.out.println("Hello, New Static Method Here"); }

    void overrideMethod(String str);
}
```

```
public class InterfaceDemo implements NewInterface {
   public static void main(String[] args)
    {
       InterfaceDemo interfaceDemo = new InterfaceDemo();
        // Calling the static method of interface
        NewInterface.hello();
        // Calling the abstract method of interface
        interfaceDemo.overrideMethod("Hello, Override Method
                                                       here");
     }

     @Override
    public void overrideMethod(String str)
      { System.out.println(str); }
    }
```

Interfaces – Static Methods

- Example 2 of using static methods:
  - In this example the same name method is implemented in the class that implements the interface
    - In this case the method becomes a static member of that class

```
interface PrintDemo {
  static void hello()
    {System.out.println("Called from Interface PrintDemo");}
}
```

```
public class InterfaceDemo implements PrintDemo
  {
     public static void main(String[] args)
      {
          //Call Interface method as Interface name is
             preceeding with method
          PrintDemo.hello();
          //Call Class static method
           hello();
      }
      // Class Static method is defined
       static void hello()
          {System.out.println("Called from Class"); }
}
```

Interfaces - Default Methods

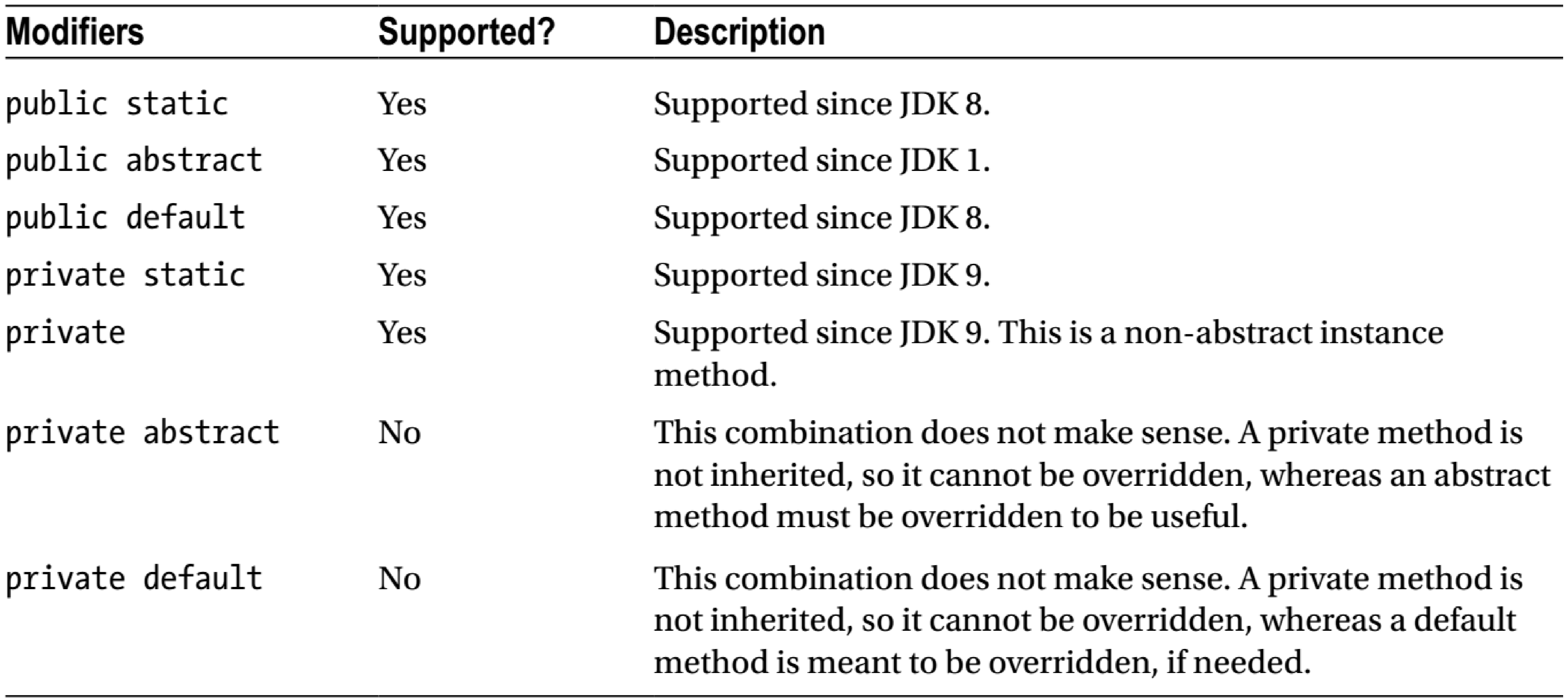
- **Private methods**
  - Java 9 introduced private methods
  - **Private methods can be implemented *static* or non-*static***
  - These methods are only accessible within that interface only and cannot be accessed or inherited from an interface to another interface or class

```
public interface AnInterface {
    default int m1(… parameters …) {
        // … m1 specific code
        // … common code m1 and m2
    }

   default int m2(… parameters …) {
        // … m2 specific code
        // … common code m1 and m2
    }
}
```

```
public interface AnInterface {
    default int m1(… parameters …) {
        // … m1 specific code
       // call to pm1m2
    }
   default int m2(… parameters …) {
        // … m2 specific code
        // call to pm1m2
    }
    private int pm1m2( … parameters …) {
        …
       return val;
    }}
```

Interfaces

- **Supported Modifiers in Method Declarations**



Modifiers	Supported?	Description
`public static`	Yes	Supported since JDK 8.
`public abstract`	Yes	Supported since JDK 1.
`public default`	Yes	Supported since JDK 8.
`private static`	Yes	Supported since JDK 9.
`private`	Yes	Supported since JDK 9. This is a non-abstract instance method.
`private abstract`	No	This combination does not make sense. A private method is not inherited, so it cannot be overridden, whereas an abstract method must be overridden to be useful.
`private default`	No	This combination does not make sense. A private method is not inherited, so it cannot be overridden, whereas a default method is meant to be overridden, if needed.

Interfaces – Marker Interface

- **Nested interfaces**
  - Declared in the body of another class or interface
  - Are static by default and any members declared inside it, including methods, inherit this static modifier.
  - The nested interface must be referred to by the outer interface. It can't be accessed directly.

```
interface Showable{
  void show();
  interface Message{
   void msg(); }
}
class TestNestedInterface1 implements Showable.Message{
 public void msg(){System.out.println(“Nested interface");}
 public static void main(String args[]){
  Showable.Message message=new TestNestedInterface1();
  message.msg();
 }  }
```

- **Group related interfaces so that they can be easy to maintain**

Interfaces – Marker Interface

- A Marker *interface* (also called a *taginterface*) is simply an interface with no methods
- Some examples of tag interfaces from the JDK:
  - Serializable
  - EventListener
  - Clonable
- Why define an interface with no methods?
  - Marker interfaces are used to indicate that a class has certain characteristics or should be treated in a particular way.
  - Marker interfaces are often used in Java to provide run-time type information about objects

Interfaces –Tag Interface

- Marker interfaces have become less popular in recent years because they can be seen as a code smell
  - Because marker interfaces do not declare any methods, they do not provide any behavior or functionality
    - This can make it difficult to understand what a marker interface does or how it is used
  - In addition, marker interfaces can be limiting because a class can only implement a limited number of them
- As an alternative to marker interfaces, many developers now prefer to use annotations in Java
  - Annotations provide similar functionality to marker interfaces, but they can declare attributes and provide more detailed information about the class
  - Annotations can also be used more flexibly than marker interfaces, because a class can have an unlimited number of annotations

Interfaces - Example of Java predefined interface

- **Comparable** Interface
  - Is defined in **java.lang** package
  - Is used to order the objects of the user-defined class
    - It provides a single sorting sequence only, i.e., you can sort the elements on the basis of single data member only
  - Has only one method that must be implemented

public int *compareT*o(Object other);

- **Collections** class
  - Provides static methods for sorting the elements of a collection
  - Method of Collections class for sorting List elements
    - **sort**
      - Is used to sort the elements of **List** by the given **Comparator**

Interfaces - Example of Java predefined interface

- Example of the Comparable interface that sorts the list elements on the basis of age

```
class Student implements Comparable<Student>
{
  String name;
  int age;
  Student(String name,int age)
     {  this.name=name;
        this.age=age;  }
  public int compareTo(Student st)
     {
      if(age==st.age)
            return 0;
      else if(age>st.age)
            return 1;
      else
           return -1;
     }
}
```

```
import java.util.*;
public class TestSort {
  public static void main(String args[]){
           ArrayList<Student> al=new ArrayList<Student>();
           al.add(new Student(“Ana",23));
           al.add(new Student(“Victor",27));
           al.add(new Student(“Dan",21));
           Collections.sort(al);
           for(Student st:al){
            System.out.println(“ name: "+st.name+“, age: "+st.age);
   }
   }
}
```

```
Output >>
 Dan 21
 Ana 23
 Victor 27
```

Interfaces - Example of Java predefined interface

- **Comparator** interface
  - Is defined in **java.lang** package
  - Is used to order the objects of a user-defined class
    - It provides multiple sorting sequences, i.e., you can sort the elements of type Students for name or age or anything else
  - Contains the method compare
    - compare(Object obj1,Object obj2)

Interfaces - Example of Java predefined interface

- Example of using **Comparator** interface to sort the list elements on the basis of age and name

```
class Student {
 String name;
  int age;
  Student(String name,int age)
    { this.name=name;
      this.age=age; }
   }
```

```
import java.util.*;
class AgeComparator implements Comparator<Student>
 {
 public int compare(Student s1, Student s2){
    if(s1.age==s2.age)
      return 0;
   else
         if(s1.age>s2.age)
               return 1;
        else
              return -1;  }  }
```

```
import java.util.*;
class NameComparator implements Comparator<Student>
 {
   public int compare(Student s1, Student s2)
    {        return s1.name.compareTo(s2.name);      }
 }
```

Interfaces - Example of Java predefined interface

- Example of using Comparator interface to sort the list elements on the basis of age and name

```
class Main {
public static void main(String args[]){
           ArrayList<Student> al=new ArrayList<Student>();
           al.add(new Student(“Ana",23));
           al.add(new Student(“Victor",27));
           al.add(new Student(“Dan",21));
           for(Student st:al)
              System.out.println(“ name: "+st.name+“, age: "+st.age);
          Collections.sort(al, new AgeComparator());
          for(Student st:al){
             System.out.println(“ name: "+st.name+“, age: "+st.age);
          Collections.sort(al, new NameComparator());
          for(Student st:al){
           System.out.println(“ name: "+st.name+“, age: "+st.age);    }}
```

Output>>

```
Ana 23
Victor 27
Dan 21

Dan 21
Ana 23
Victor 27

Ana 23
Dan 21
Victor 27
```

Interfaces - Example of Java predefined interface

- **AgeComparator.java**
  - This class defines comparison logic based on the age
  - If the age of the first object is greater than the second, we are returning a positive value
  - If the age of the first object is less than the second object, we are returning a negative value
  - If the age of both objects is equal, we are returning 0
- **NameComparator.java**
  - This class provides comparison logic based on the name
  - In such case, we are using the compareTo() method of String class, which internally provides the comparison logic

Polymorphism

- Is the ability of an object to take on many forms
- Type of polymorphism
  - Ad-hoc polymorphism
  - Inclusion polymorphism
  - Coersion Polymorphism
  - Parametric Polymorphism

Polymorphism

- Ad-hoc polymorphism
  - Refers to methods that all have the same name, but are distinguished by the number and/or type of parameters
  - In a java programming language, ad-hoc polymorphism carried out with a method **overloading concept**
- Inclusion polymorphism
  - It refers to the ability of a subclass to inherit and use the methods and properties of its superclass
    - In other words, an object of a subclass can be treated as an object of its superclass and can be used wherever an object of the superclass is expected
  - In a java programming language, inclusion polymorphism carried out with a **method overriding concept**

Polymorphism

- Coercion Polymorphism
  - Occurs when an object or primitive is cast into some other type
    - » It could be either implicit or explicit
- Parametric Polymorphism
  - Is a way to make a language more expressive
  - Using parametric polymorphism, a method/a data type can be written generically so that it can handle values identically without depending on their type
    - » Such methods/ data types are called generic methods/ datatypes

Polymorphism

- The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object (**inclusion polymorphism**)
- We consider the following example:

```
Person p= new Student(“Ana”, 2854)
```

  - In this example object of type Person refer to an object of type Student

Polymorphism –Ad –hoc Polymorphism (Example)

```
import java.util.Arrays;
public class AdHocPolymorphismExample {
        void sorting(int[] list) {
                Arrays.sort(list);
                System.out.println("Integers after sort: " + Arrays.toString(list) );
        }
        void sorting(String[] names) {
                Arrays.sort(names);
                System.out.println("Names after sort: " + Arrays.toString(names) );
        }
        public static void main(String[] args) {
                AdHocPolymorphismExample obj = new AdHocPolymorphismExample();
                int list[] = {2, 3, 1, 5, 4};
                obj.sorting(list);
                String[] names = {"ana", "dana", "george", "gicu"};
                obj.sorting(names);
        }}
```

- ✓ In ad hoc polymorphism the method binding happens at the time of compilation

- ✓ Ad hoc polymorphism is also known as compile-time polymorphism

Polymorphism – Inclusion polymorphism (Example)

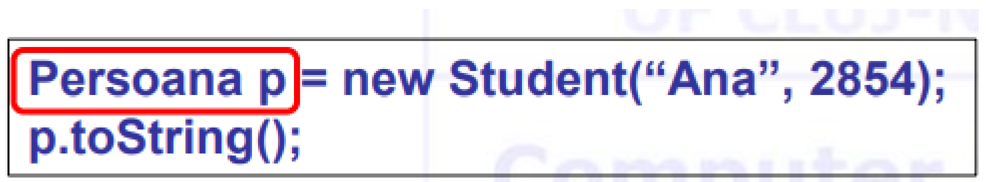
```
public class Person{
  private String name;
  Person(String name) { this.name= name;}
  protected String getName() { return name;}
  public String toString() { return “name:”+ name;}

public class Student extends Person{
  private int id;
  Student(String n , int i)
  {  super(n);
     this.id=i;   }
 public int getID() { return id; }
 public String toString()
  {return super.toString()+”, id:”+id; }
}
```

```
public class Teacher extends Person{
  private String course;
  Teacher(String n , String c)
    {  super(n);
       this.course=c;  }
 public String getCourse() { return course;}
 public String toString()
       { return super.toString()+”, course:”+ course;}
}
public class App{
 public static void main(String args[]){
 Person p[] = new Person[3];
   p[0] = new Person("Ion");
   p[1] = new Student("Ana", 2854);
   p[2] = new Teacher("Mara", “PT”);
   for(int i = 0; i < p.length; i++)
     { System.out.println( p[i].toString() ); }
   }
}
```

```
>>output:
name: Ion
name: Ana, id:1234
Name: Mara, course: PT
```

Polymorphism – Inclusion polymorphism

- **Decisions taken at compilation vs. during execution**
  - Rules for compilation
    - The compiler only knows the reference type of the object
    - Search in the class of the reference type if there is a method to be called
    - And return the signature of the method
  - Rules for execution
    - The type of the object actually created at the time of execution will be followed
    - The signature returned at the compilation time need to match with the method from the current class
    - If the method is not found in the current class, search above in the class hierarchy



```
Persoana p= new Student(“Ana”, 2854);
p.toString();
```

Polymorphism – Inclusion polymorphism

- Decisions taken at compilation vs. during execution
  - What happens when we run the following code?

```
Person p = new Student(“Ana”, 2854);
p.getID();
```

» Answer: Compilation Error

» The solution:

```
((Student) p).getID();
```

▯ to avoid execution errors, use:

```
if( p instanceof Student ) {
    // runs only if p "is a" Student at execution
   ( (Student)s ).getID(); }
```

  - What happens when we run the following code?

```
Student s = new Person(“Ion”);
```

» Answer: Compilation Error

» Solution: -don’t exist

Polymorphism – Inclusion polymorphism

- Example pf polymorphic call of a method

```
public class Person {
  public void method1()
   {  System.out.print("Person 1 "); }

  public void method2()
     { System.out.print("Person 2 "); }
}

class Student extends Person {
  public void method1() {
            System.out.print("Student 1 ");
            super.method1();
            method2();   }

 public void method2()
  { System.out.print("Student 2 "); }
}
```

```
class Undergrad extends Student {
        public void method2() {
           System.out.print("Undergrad 2 ");
    }
}
```

Polymorphism – Inclusion polymorphism

```
public class Person {
    public void method1() {
      System.out.print("Person 1 ");
      }
  public void method2() {
      System.out.print("Person 2 ");
}}

class Student extends Person {
    public void method1() {
           System.out.print("Student 1 ");
           super.method1();
           method2();  //this.method2();
         }
 public void method2() {
       System.out.print("Student 2 ");
}}
class Undergrad extends Student {
   public void method2() {
       System.out.print("Undergrad 2 ");
}}
```

What is the result when execute the following sequence of code:

```
Person p = new Undergrad();
p.method1();
```

The displayed results is:
Student 1
Person 1
Undergrad 2

Polymorphism – Inclusion polymorphism

- Discussions for the considered example:
  - First **method1**() from the **Student** class is called;
    - This because in the **Undergrad** class there is no method with this signature, so it is executed the first method that is found by going up in the class hierarchy
    - The displayed results is: “Student 1”
  - Then is called **method1**() from **Person** class (indicated by **super**, which at the compilation time determines that the call must be made to **method1**() from the **Person** class)
    - The displayed results is “Person 1”
  - Then **method2**() from the **Undergrad** class is called, because the compiler interprets the call "**method2**();" as "**this.method2**()", where **this** refers to the object from which the call is made, namely the concrete object created at the time of execution which is of type **Undergrad**

Polymorphism – Inclusion polymorphism

- Rules for calling methods using **this** and **super** operators:
  - When we call a method with **super** (ex: super.method1()), binding is done at compilation
    - Then it is check which is parent's class
  - When we call a method with **this** (e.g., **this.method2**(), or simply **method2**()), the binding is done at the time of execution, depending on the concrete type of the created object
    - It has the name of dynamic binding
- Dynamic binding
  - Appears when the decision on which method to be execution can only be taken at the execution time
  - It takes it when
    - The variable is declared to have the superclass type and
    - There is more than one polymorphic method that can be executed between the type of the variable and its subclasses

Polymorphism – Inclusion polymorphism

- How to decide which method to execute?
  1. If there is a concrete method in the current class, it runs
  2. Otherwise, check in the direct superclass if there is a method there; if yes, it is executed
  3. Repeat step 2, checking up the hierarchy until a concrete method is found and it is executed
  4. If no method has been found, then Java signals a compilation error

Reusing Classes - Composition

- Indicates that one class **contains** objects of another class
  - **has-a** can be used to describe the relationship between the two objects
- For example, a *Circle* **has-a** *Center*
  - *Center* (typically a point (x, y)) would be a field defined in the *Circle* class

Reusing Classes - Composition (Example 1)

```
public class Address {
  private String street;
  private String city;
  private String state;
  private String zip;
  public Address(String street, String city,
String state, String zip) {
    this.street = street;
    this.city = city;
    this.state = state;
    this.zip = zip;
  }
  // getters and setters
}
```

```
public class Person {
  private String name;
  private Address address;
  public Person(String name, Address address) {
    this.name = name;
    this.address = address;
  }
  // getters and setters
}
```

```
public class Main {
  public static void main(String[] args) {
    Address address = new Address("123 Main St", "Anytown", "CA",
"12345");
    Person person = new Person("John Doe", address);
    System.out.println(person.getName()); // John Doe
    System.out.println(person.getAddress().getStreet()); // 123 Main St
  }}
```

Reusing Classes - Composition(Example 2)

```
class Engine {
  public void start() {}
  public void rev() {}
  public void stop() {}
  }
class Wheel {
    public void inflate(int psi) {}
   }
class Window {
   public void rollup() {}
   public void rolldown() {}
}
class Door {
  public Window window = new Window();
  public void open() {}
  public void close() {}
}
```

```
public class Car {
  public Engine engine = new Engine();
  public Wheel [] wheel = new Wheel[4];
  public Door left = new Door(), right = new Door();
  public Car() {
     for(int i = 0; i < 4; i++)
           wheel[i] = new Wheel();
  }
public static void main(String[] args)
  {
     Car car = new Car();
     car.left.window.rollup();
     car.wheel[0].inflate(72);
   }
}
```

Reusing Classes - Composition

- Fields Initialization
  - Primitives that are fields in a class are automatically initialized to zero
  - Object references are initialized to **null**
    - If you try to call methods for any of them, you’ll get an exception

Reusing Classes – Composition

- Reference fields initialization
  - – Can be done:
    - » At the point the objects are defined
      - ▯ They’ll always be initialized before to use
    - » In the constructor of the class
    - » Right before you actually need to use the object (i.e., *lazy initialization)*
      - ▯ It can reduce overhead in situations where object creation is expensive, and the object doesn’t need to be created every time

Reusing Classes – Composition

- Example of reference fields initialization

```
class Soap {
  private String s;
    Soap()  {
         System.out.println("Soap()");
         s = new String("Constructed");
     }
    public String toString()
       { return s; }
  }
```

```
public class Bath {
     //Initializing at point of definition:
     private String s1 = new String("Happy"), s2 = "Happy", s3, s4;
     private Soap c;
     private int i;
     private float toy;
     public Bath() { System.out.println("Inside Bath()");
        s3 = new String("Joy");
        i = 47;
        toy = 3.14f;
        c = new Soap();}
     public String toString() {
        if(s4 == null) // Delayed initialization:
             s4 = new String("Joy");
        return  "s1 = " + s1 + "\n" + " s2 = " + s2 + "\n" + " s3 = " + s3
     + "\n"  + "s4 = " + s4 + "\n" + "i = " + i + "\n" + "toy = " + toy +
     "\n" +  "c = " + c; }
     public static void main(String[] args) {
         Bath b = new Bath();
         System.out.println(b.toString());  }  }
```

- ✓ When initialization isn't done at the point of definition, there's no guarantee that it will be done before sending a message to that object

- ✓ When **toString**() method is called it initializes s_4 so that all the fields are properly initialized by the time they are used

Reusing Classes - Choosing composition versus inheritance

- Both composition and inheritance allow to place sub-objects inside of new class
  - Composition explicitly does this
  - Inheritance make this implicitly

Reusing Classes - Combining composition and inheritance (Example)

```
class Engine {
  public void start()
   {  System.out.println("Engine started."); }
}

class Wheel {
  public void rotate()
  {  System.out.println("Wheel rotating.");  }
}

class Door {
  public void open()
   { System.out.println("Door opening."); }
}
```

```
class Car {
  private Engine engine = new Engine();
  private Wheel[] wheels = new Wheel[4];
  private Door[] doors = new Door[4];

public Car() {
    for(int i = 0; i < 4; i++) {
      wheels[i] = new Wheel();
      doors[i] = new Door();  } }

  public void start() {
    engine.start();
    System.out.println("Car started."); }

 public void drive() {
    for(int i = 0; i < 4; i++)
       wheels[i].rotate();
     System.out.println("Car driving.");
    }
 public void openDoor(int i) {
    doors[i].open();  }}
```

Reusing Classes - Combining composition and inheritance (Example 1)

```
class SportsCar extends Car {
  private boolean racingMode = false;

  public void setRacingMode(boolean racingMode) {
    this.racingMode = racingMode;
  }

  public void race() {
    if(racingMode) {
      System.out.println("Sports car racing.");
    }
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    SportsCar sportsCar = new SportsCar();
    sportsCar.start();
    sportsCar.drive();
    sportsCar.openDoor(0);
    sportsCar.setRacingMode(true);
    sportsCar.race();
  }
}
```

Reusing Classes - Choosing composition versus inheritance

- Composition is used when features of an existing class need to be used inside of a new class, but not its interface
- Inheritance is used when super-classes must significantly interact with their sub-classes
  - The extended class conforms to the base class, but has special properties
  - If the phrase "is-a" cannot logically be used to describe the relationship between two classes, the inheritance does not apply
    - For example, a **Car** "is-a" **Vehicle** and a **Square** "is-a" **Rectangle**

Object Class

- Is defined in the **java.lang** package
- Is the parent class of all the classes in java by default
  - It sits at the top of the class hierarchy tree
- Every class is a descendant, direct or indirect, of the **Object** class
- Every class you use or write inherits the instance methods of **Object**
- You need not use any of these methods, but, if you choose to do so, you may need to override them with code that is specific to your class

Object Class

- Some methods inherited from **Object** are:
  - protected Object **clone**() throws CloneNotSupportedException
    - Creates and returns a copy of this object
  - public boolean **equals(Object** obj)
    - Indicates whether some other object is "equal to" this one
  - protected void **finalize**() throws Throwable
    - Called by the garbage collector on an object when garbage collection determines that there are no more references to the object
  - public final Class **getClass**()
    - Returns the runtime class of an object
  - public int **hashCode**()
    - Returns a hash code value for the object

Object Class

- Some methods inherited from **Object** are:
  - public String **toString**()
    - Returns a string representation of the object
- **notify**, **notifyAll**, and **wait** methods of Object all play a part in synchronizing the activities of independently running threads in a program

Object Class

- **toString**() method
  - Return a String representation of the Object
  - The default toString() method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@’, and the unsigned hexadecimal representation of the hash code of the object
  - It is always recommended to override **toString()** method to get our own String representation of Object
  - You can use toString() along with System.out.println() to display a text representation of an object

Object Class

- **toString**() method - Example of overriding toString() method in a Book class

```
class Book{
  private String ISBN;
  private String name;
  public Book(String i, String n){
    this.ISBN=i;
    this.name=n;
}
  public String toString(){
     return “ISBN:”+ISBN+ “; “+name;
 }
 public static void main(String args[]){
  Book b= new Book(0201914670, “The Swing
Tutorial - A Guide to Constructing GUIs”);
  System.out.println(b.toString());
  }
}
```

Object Class

- **equals**() method
  - Compares two objects for equality and returns true if they are equal
  - The **equals**() method provided in the **Object** class uses the identity operator (==) to determine whether two objects are equal
    - For primitive data types, this gives the correct result
    - For objects, however, it does not
      - The equals() method provided by Object class tests whether the object references are the same
- To test whether two objects are equal in the sense of equivalency (containing the same information), you must override the equals() method

Object Class

- **equals** method - Example of overriding equals method in a Book class

```
public class Book {
       private String ISBN;
       private String name;
       Book(String i, Strinng n){
         this.ISBN=i;
         this.name=n;
     }
     String  getISBN(){
        retunr ISBN;
      }
    public boolean equals(Object obj) {
        if (obj instanceof Book)
            Book obj1= (Book) obj;
            return
ISBN.equals(obj1.getISBN());
        else
            return false; }}
```

✓ Consider the following code that tests two instances of the **Book** class for equality:

```
Book firstBook  = new Book("0201914670");
Book secondBook = new Book("0201914670");
if (firstBook.equals(secondBook)) {
     System.out.println("objects are equal");
 } else {
    System.out.println("objects are not equal");
  }
```

✓ This code displays objects are equal even though firstBook and secondBook reference two distinct objects.

✓ They are considered equal because the objects compared contain the same ISBN number