Advanced Programming - JAVA
Lecture 4
OOP Concepts in JAVA
PART II

Mahmoud El-Gayyar

elgayyar@ci.suez.edu.eg

Outline

- *Ad hoc-Polymorphism*
  - Method overloading
- *Sub-type Polymorphism*
  - Method overriding
  - Dynamic binding
  - Object methods
  - Interfaces
- *Paramedic Polymorphism*
  - Java Generics

Outline

- ***Ad hoc-Polymorphism***
  - Method overloading
- *Sub-type Polymorphism*
  - Method overriding
  - Dynamic binding
  - Object methods
  - Interfaces
- *Paramedic Polymorphism*
  - Java Generics

Polymorphism

- *Idea of polymorphism*
  - In computer science, polymorphism is the idea of allowing the same code to be used with different types, resulting in more general and abstract implementations.
- *Types of polymorphism*
  - **Ad-hoc** polymorphism (Method Overloading)
  - **Subtype (inclusion)** polymorphism(Method Overriding)
  - **Parametric** polymorphism (Java Generics)

Ad-hoc Polymorphism

- *This is called **method overloading***
  - In this case different methods within the same share the same name but have different method signatures (name + parameters)

```
public static float max(float a, float b)
public static float max(float a, float b, float c)
public static int max(int a, int b)
```

  - When a method is called, the call signature is matched to the correct method version.

Ad-hoc Polymorphism

- *If an exact signature match is not possible, the one that is **closest via "widening"***
  - "Widening" means that values of "smaller" types are cast into values of "larger" types Ex: int to long; int to float ; float to double
- *Note: This type of polymorphism is not necessarily object-oriented – can be done in non-object-oriented languages*

Ad-hoc Polymorphism

- *If two or more versions of the method are possible with the same amount of "widening", the call is ambiguous, and a compilationerror will result*

```
static public void method1(Integer integer) ;
static public void method1(String string);
method1(null);          //compiler error
```

Outline

- *Ad hoc-Polymorphism*
  - Method overloading
- ***Sub-typePolymorphism***
  - Method overriding
  - Dynamic binding
  - Object methods
  - Java Interfaces
- *Paramedic Polymorphism*
  - Java Generics

Sub-type Polymorphism

- *Sometimes called "**true polymorphism**"*
- *Consists basically of two ideas:*

1) ***Method overriding***

   - A method defined in a **superclass** is redefined in a **subclass** with an **identical method signature**
   - Since the signatures are identical, rather than overloading the method, it is instead **overriding the method**
     - *For subclass objects, the definition in the subclass replaces the version in the superclass*

Sub-type Polymorphism

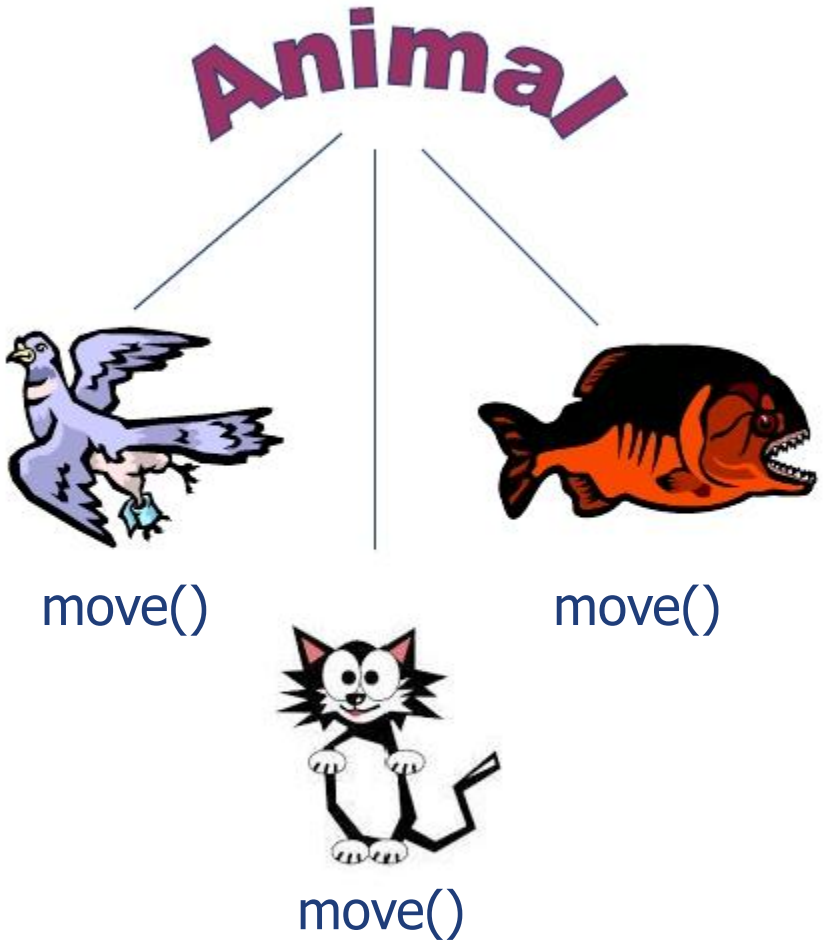
2) *Dynamic (or late) binding*

- The code for a method call is associated during **run-time**
- The actual method executed is determined by the **type of the object**, not the type of the reference
- Allows superclass and subclass objects to be accessed in a regular, consistentway
- This is very useful if we want access collections of mixed data types (ex: draw different graphical objects using the same draw() method call for each)

Dynamic/Late Binding

```
Animal [] A = new Animal[3];
A[0] = new Bird();
A[1] = new Cat();
A[2] = new Fish();
for (int i = 0; i < A.length; i++)
        A[i].move();
```

- References are all the same, but objects are not
- Method invoked is that associated with the OBJECT, NOT with the reference



Object, Method and Instance Variable Access

- *When mixing objects of difference classes, some access rules are important to know:*
  - **Superclass references** can always be used to **access subclass objects,** but NOT vice versa

    Animal A = new Bird(); // this is ok

    Bird B = new Animal(); // this is an ERROR
  - Given a ***reference R*** of ***class C***, only methods and instance variables that are defined (initially) in ***class C or ABOVE*** in the class hierarchy can be ***accessedthroughR***

Object, Method and Instance Variable Access

- *Example: Suppose class Fish contains a new instance variable waterType and a new method getWaterType()*

```
Fish F = new Fish();
Animal A = new Fish();
System.out.println(F.getWaterType()); // ok
System.out.println(A.getWaterType()); //error
```

  - *The above is NOT legal, the method is not visible from the reference's point of view (A is an Animal reference so it can only "see" the data and methods defined in class Animal)*

```
System.out.println(((Fish) A).getWaterType());
```

  - *This is ok, since we have now cast the reference to the Fish type*

`Object` Methods

- *Every class automatically inherits from **Object***
- *Class Object defines a set of methods that every class inherits*
  - ***`public String toString()`***
  - ***`public boolean equals(Object obj)`***
- *Each of these forms a contract to which all objects must adhere*
- *Object has a default implementation for each of these methods*
  - Unless our classes override them, they inherit this behavior
  - May or may not be what our classes require

toString()

- *toString() returns a string representation of the object*
- *RULE: **All classes should override this method***
- *Default implementationfrom the Object class constructs a string like:*

```
ClassName@30E50DA3
```

  - Name of the class, '@' character, followed by the ***HashCode*** for the object

equals()

- *Indicates whether two objects are logically equal to each other*
  - Ex. `string1.equals("done")`
- *Defaultimplementationfrom the Object class is equivalent to '=='*
  - For any two references, x and y:

    *`x.equals(y)` is **true** if and only if `x == y`*
  - The references must point to the same object for equals() to return **true**

Contract of equals()

- *Consider the following equals() method*

```
public boolean equals(Object o)
{
  MyClass obj= (MyClass) o;
  return (name.equals(obj.name);
}
```

- *What's wrong with this?*
  - Object o could be null, need to check

    ```
    if(o == null) return false;
    ```

  - `o may be not an instance of MyClass (Exception)`

Template for equals()

- *For any class, the general form of the equals() method should be:*

```
public class MyClass{
   public boolean equals(Object o){
       if(o == null) return false;
       if(o instanceof MyClass){
          MyClass my = (MyClass) o;
          //perform comparison
       }
     return false;
   }
}
```

Interfaces

- *Java allows only **single inheritance***
  - A new class can be a subclass of only one parent (super) class
  - However, it is sometimes useful to be able to access an object through more than one superclass reference
  - We may want to identify an object in multiple ways:
    - *One based on its inherent nature (i.e. its inheritance chain)*
      - Ex: A Person
    - *Others based on what it is capable of doing*
      - Ex: An athlete
      - Ex: a pilot

Interfaces

- *A Java **interface** is a named set of methods*
  - *However, no method bodies are given – just the headers*
  - *Static constants are allowed, but no instance variables are allowed*
  - *No static methods are allowed*
  - Any Java class (no matter what its inheritance) can implement an interface by implementing the methods defined in it
  - A given class can implement any number of interfaces

Example: Interfaces

```
public interface Laughable{
   public void laugh();
}

public interface Booable{
   public void boo();
}
```

- *Any Java class can implement Laughable by implementing the method laugh()*
- *Any Java class can implement Booable by implementing the method boo()*

Example: Interfaces

```
public class Comedian implements Laughable, Booable
{
   // various methods here (constructor, etc.)
   public void laugh()
   {
         System.out.println("Ha ha ha");
   }

   public void boo()
   {
         System.out.println("You stink!");
   }
}
```

Interface Variable

- *An interface variable can be used to reference any object that implements that interface*
  - Note that the same method name (ex: laugh() below) may in fact represent different code segments in different classes
  - Also, only the interface methods are accessible through the interface reference

```
Laughable L1, L2, L3;
L1 = new Comedian();        // implements Laughable
L2 = new SitCom();          // implements Laughable
L3 = new Clown();           // implements Laughable
L1.laugh();   L2.laugh();  L3.laugh();
```

Interfaces

- *Polymorphism and Dynamic Binding also apply to interfaces*
  - An interface variable can be used to reference any object that implements that interface
  - However, only the interface methods are accessible through the interface reference
- *Recall our previous example:*

```
Laughable [] funny = new Laughable[3];
funny[0] = new Comedian();
funny[1] = new SitCom();
funny[2] = new Clown();
for (int i = 0; i < funny.length; i++)
    funny[i].laugh();
```

"Generic" Operations

- *How does it benefit us to be able to access objects through interfaces?*
  - Sometimes we are only concerned about a given property or behavior of a class
    - *The other attributes and methods still exist, but we don't care about them for what we want to do*
  - For example: Sorting
    - *We can sort a lot of different types of objects*
      - Various numbers
      - People based on their names alphabetically
      - Movies based on their titles
      - Employees based on their salaries
    - *Each of these classes can be very different*
    - *However,something about them all allows them to be sorted*

“Generic” Operations

- *They all can be compared to each other*
  - So we need some method that invokes this comparison
- *In order to sort them, we don't need to know or access anything else about any of the classes*
  - Thus, if they all implement an interface that defines the comparison, we can sort them all with a single method that is defined in terms of that interface
- *Huh?*
  - Perhaps it will make more sense if we develop an example…but first we will need some background!

Comparable Interface

- *Consider the Comparable interface:*
  - It contains one method:

```
int compareTo(Object r);
```

  - Returns a negative number if the current object is less than r, 0 if the current object equals r and a positive number if the current object is greater than r
  - Not has restrictive as equals() – can throw `ClassCastException`
- *Consider what we need to know to sort data:*
  - is A[i] less than, equal to or greater than A[j]
- *Thus,* ***we can sort Comparable data without knowing anything else about it → very nice !!***

Using *Comparable*

- *Think of the objects we want to sort as "black boxes"*
  - We know we can compare them because they implement Comparable
  - We don't know (or need to know) anything else about them
- *Thus, a single sort method will work for an array of any Comparable class*
  - Let's write it now, altering the code we already know from our simple sort method
  - See example

Outline

- *Ad hoc-Polymorphism*
  - Method overloading
- *Sub-typePolymorphism*
  - Method overriding
  - Dynamic binding
  - Object methods
  - Java Interfaces
- ***Paramedic Polymorphism***
  - Java Generics

Generics

- *We will know later that*
  - Each collection or Iterator returns an Object (to work for any kind)
  - Requires us to cast the reference to an instance that we need

```
List list = new LinkedList();

list.add("Some Pig");

String s = (String) list.get(0); //The cast can be annoying
```

  - The list may also not really contain Strings
- *We'd like to force the List to only contain specific types*
  - We wouldn't need the cast
  - We could be sure what type of objects the List contained
- *This is where "Generics" works well*

Generics

- *We "parameterize" the instance of our List with the type of object we expect it to contain using the <> syntax*

```
List<String> list = new LinkedList<String>();
list.add("Some Pig");
String s = list.get(0);
```

  - Declares a "List of Strings" instead of a simple List
  - Compiler can now ensure only Strings are added to this particular list
  - We no longer need the casts

Generics - Motivation

```
List v = new ArrayList();

v.add("test");

Integer i = (Integer)v.get(0);    // Run time error
```

```
List<String> v = new ArrayList<String>();

v.add("test");

Integer i = v.get(0);             // (type error) Compile time error
```

Writing a Generic Class

- *Use the <> syntax in the class definition*

```
public interface List<E>{
  void add(E x);
}
```

- *This is similar to declaring parameters in a method*
  - Called **Formal Type Parameters**
- *The <E> declares that a type must be used when an instance is created*
  - The type is then used in place of anywhere the 'E' is used in the class definition
    - *e.g. add(E x);*

Example: a Generic Class

```
public class Entry<K, V> {
        private K key;
        private V value;
        public Entry(K k, V v) {
                key = k; value = v;
        }
        public K getKey() {
                return key;
        }
        public V getValue() {
                return value;
        }
        public String toString() {
                return "(" + key + ", " + value + ")";
        }
}
```

```
Entry<String, String> grade440 =
new Entry<String, String>("mike", "A");

Entry<String, Integer> marks440 =
new Entry<String, Integer>("mike", 100);
```

Summary

- *Ad hoc-Polymorphism*
  - Method overloading
- *Sub-type Polymorphism*
  - Method overriding
  - Dynamic binding
  - Object methods
  - Interfaces
- *Paramedic Polymorphism*
  - Java Generics