OOP WITH JAVA

Lecture 04

Sadaf Anwar

Revision Topic(s)

- Object Oriented Concepts
  - Classes, Objects
- Static fields and methods
- Set and get methods

Today

- This reference
- Final Instance Variable
- Composition
- Inheritance
- Polymorphism

This Reference

- When a non-static method is called for a particular object, the method's body implicitly uses keyword this to refer to the object's instance variables and other methods

  - enables the class code to know which object should be manipulated

When to use This reference?

- If a method contains a local variable with the *same* name as a field

```
public void n(int m){
        this.m=m;}
```

- In above example field of the class is referenced. If this keyword is not used then local variable of method ‘n’ is referenced

When to use This reference?

- To reuse initialization code provided by another of the class's constructors rather than defining similar code in the constructor's body

```
public class c{
 int temp, temp1;
 public c(){
   this(0, 0);
                }
public  c(int a, int b){
  temp=a;
   temp1=b;
                }}
```

- This makes the class easy to maintain and modify

This reference in static methods

- this reference cannot be used in a static method

- The this reference must refer to a specific object of the class, and when a static method is called, there might not be any objects of its class in memory

Final Instance Variables

- Keyword final specifies that a variable is not modifiable (i.e., it's a constant) and that any attempt to modify it is an error

```
private final int i;
private static final int tmp;
```

- Final variables can be initialized when they're declared.

Final Instance Variables

- Final variable can also be initialized in constructor of the class
  - Initializing constants in constructors enables each object of the class to have a different value for the constant
- If a final variable is not initialized in its declaration or in every constructor, a compilation error occurs.

Composition

- Composition is when class has a reference to object of other classes as members

```
class a{ }
  class b {
        private int tmp;
        private String tmp1;
        private a tmp2;
        private a tmp3;
          }
```

Composition

- Composition demonstrates that class can have as instance variables references to objects of other classes
- has-a-relationship

Example- Composition

```
class date
 {
 int month;
 int day;
 int year;
 private int checkmonth(int m){ }
 private int checkday(int d){ }
 private int checkyear(int y){ }
 }
```

```
class employee
{
private String F_name;
private String L_name;
private date birthdate;
private date hiredate;
public employee(String f, String l, date
bd, date hd)
{
Code
}
}
```

```
class test{
main()
{
date birth=new date();
date hire=new date();
employee e=new employee
("Lance", "Stephen",birth,
hire);
} }
```

Inheritance

- Inheritance is when a child class inherits members of its parent class and embellishing them with new or modified capabilities
- Is-a-relationship
- Parent class is also called super class or base class.
- Child class is also called sub class or derived class
- Every subclass object is an object of its superclass

Inheritance

- Inheritance is sometimes referred to as **specialization**
  - The subclass exhibits the behaviors of its superclass and can modify those behaviors so that they operate appropriately for the subclass

Inheritance

- The **direct superclass** is the superclass from which the subclass explicitly inherits

- An **indirect superclass** is any class above the direct superclass in the **class hierarchy**

- **Java** supports only single inheritance. It does not support multiple inheritance
  - Each class is derived from exactly one direct super class

- In Java, keyword “extends” indicate inheritance

Access Specifiers in Java

- Access specifiers control access to classes, methods, and fields

- Java supports four access specifiers
  - Public
  - Private
  - Default
  - Protected

Access Specifiers in Java

- Public
  - public class, methods, and fields can be accessed from everywhere
  - Java source code must contain only one public class (name and filename must match)

Access Specifiers in Java

- Private
  - Methods and fields only accessible within class to which they belong
  - private methods and fields are not visible within subclasses and are not inherited by subclasses

Access Specifiers in Java

- Default (no specifier)
  - Accessible inside the same package to which class, method, or field belongs
  - Not accessible outside package

Access Specifiers in Java

- Protected
  - intermediate level of access between public and private
  - methods and fields are accessible within same class
  - methods and fields are accessible in subclasses
  - accessible by classes in the same package

Access Specifiers in Java

- All public and protected superclass members retain their original access modifier when they become members of the subclass
  - -Public members of the superclass become public members of the subclass
  - -Protected members of the superclass become protected members of the subclass
  - -Superclass's private members are not accessible outside the class itself
  - - Subclass methods can refer to public and protected members inherited from the superclass simply by using the member names.

Example- Access Specifiers in Java

```
class abc{
protected int a;
}

class cde extends abc{
void method1(){
a=10;
System.out.println(a);
}
}
```

Example- Access Specifiers in Java

- A subclass can change the state of private superclass instance variables only through non-private methods provided in the superclass and inherited by the subclass

```
class abc{
private int b;
protected void set(int k)
  {
  b=k;
  }
        }
```

```
class cde extends abc
{
void method1()
{
set(10);
}
}
```

Relationship between Superclasses and Subclasses- Constructors

- Constructors are not inherited in subclasses
  - The subclass implicitly or explicitly calls the superclass's constructor
  - The first task of subclass constructor is to implicitly or explicitly call the superclass's constructor
  - Even if a class does not have constructors, the default constructor that the compiler implicitly declares for the class will call the superclass's default or no-argument constructor.

Relationship between Superclasses and Subclasses- Constructors

- Each subclass constructor must implicitly or explicitly call its superclass constructor to initialize the instance variables inherited from the superclass

Example - Implicit call to superclass constructor

```
class abc{
}

class cde extends abc{
abc(String a, int b, float c)
//implicit call to abc constructor occurs here
}
```

Example1 - Explicit call to superclass constructor

```
class abc{
abc(Sting d, int e, float f);
}

class cde extends abc{
cde(String a, int b, float c){
// explicit call to superclass abc constructor
super( a,b,c );}
}
```

Example2 - Explicit call to superclass constructor

```
class abc{
abc(Sting d, int e, float f);
}

class cde extends abc{
cde(){
// explicit call to superclass abc constructor
super( “BSE-VI”,4,3.5 );
}
}
```

Relationship between Superclasses and Subclasses- Methods Overriding

- A subclass method can override the superclass method

- To override a superclass method, a subclass must declare a method with the same signature (method name, number of parameters, parameter types and order of parameter types) as the superclass method

Example- Methods Overriding

```
class abc{
void method1(){}
}

class cde extends abc{
void method1()
{
int a=10, b=20;
int c=a+b;
super.method1();          }    To call super class method
}
}
```

Polymorphism

- “Do the right thing” (i.e., do what is appropriate for that type of object) in response to the same method call

- The polymorphism occurs when a program invokes a method through a superclass variable at execution time, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable

- Late binding

Polymorphism

- To achieve extensibility
  - New classes can be added with little or no modification to existing program
  - The only parts of a program that must be altered are those that require direct knowledge of the new classes that we add to the hierarchy

Polymorphic Behavior

- Superclass variables refers to superclass objects
- Subclass variables refers to subclass objects
- Invoking a method on a subclass object via a superclass reference invokes the *subclass* functionality
  - The type of the *referenced object*, not the type of the *variable*, determines which method is called
  - This is allowed because each subclass object *is an* object of its superclass

Polymorphic Behavior

- A superclass object is *not* an object of any of its subclasses
  - Cannot assign the reference of a superclass object to a subclass variable

Example-Polymorphic Behavior

```
class a{ method1(){} }
class b extends a{ method1(){} }
class c{
main(){
a obj1=new a();
obj1.method1();
b obj2=new b();
obj2.method1();
a obj3=obj2;
obj3.method1();
}
}
```

Invoke method on superclass object using superclas variable

Invoke method on subclass object using subclass variable

Invoke a method on a subclass object using superclass variabe

Abstract Classes and Methods

- Abstract classes
  - which cannot be instantiated
  - It is to provide an appropriate superclass from which other classes can inherit and thus share a common design
- Classes that can be used to instantiate objects are called concrete classes
- Abstract superclasses are too general to create objects; they specify only what is common among subclasses

Abstract Classes and Methods

- Use keyword “abstract” to declare classes and methods as abstract
- Abstract class normally contains one or more abstract methods

```
public abstract class abc {
   public abstract void a();
      public void a1(){}
                                    }
```

Abstract Classes and Methods

- Abstract methods are implemented by all subclasses of a superclass
- Subclasses override abstract method(s)

Example- Abstract Classes and Methods

```
public abstract class abs {
   public abstract void a();
   public void b(){}
                                    }

class abs1 extends abs{
   public void a() { }
   public void k(){}}
class abs2 extends abs
      {
public void a() {}
      }
```

Polymorphism Example1

```
class t {
main(){
abs1 a1=new abs1();
abs2 a2=new abs2();
abs[] temp=new abs[2];
temp[0]=  a1;
temp[1]= a2;
   }}
```

Polymorphism Example2

```
class t {
main(){
abs1 a1=new abs1();
abs2 a2=new abs2();
abs[] temp=new abs[2];
temp[0]=  a1;
temp[1]= a2;
for(abs var:temp)
{    var.a();
    }}}
```

Polymorphism Example3

```
class t {
main(){
abs1 a1=new abs1();
abs2 a2=new abs2();
abs[] temp=new abs[2];
temp[0]=  a1;
temp[1]= a2;
for(abs var:temp)
{    var.a();
    if(var instanceof  abs1) {System.out.println("abs1");
}}}
```

Questions?