

Inheritance

Objectives

- Apply inheritance appropriately to write Java programs,
- Analyse when to use Object class

Table of Content

Example scenarios for inheritance	Array conversions and Polymorphism
Syntax and symbols	instanceof
Members that are not accessible	Hiding .Vs. Overriding
super	final class
Conversion and casting	Abstract class and methods
protected access	Test your understanding
Overriding	Object
Covariant Returns	Object class - important methods
@Override	toString()
Polymorphism	equals()
Static and dynamic binding	hashCode()
Polymorphism in action	finalize()

Recall

- What are the important features of object-oriented languages?

Definition

Inheritance defines relationship among classes, wherein one class shares structure or behavior defined in one or more classes.

-Grady Booch

- Approach:
 - Common properties of related classes can be defined in a generic class.
 - This generic class can then be used by more specific classes through inheritance.
 - Each of these specific classes can add things that are unique to it.
- As a result of inheritance, a hierarchy of classes is formed.
- The class that is inherited is called **super class** and the class that is inheriting is called a **subclass**.
- Java does not support inheritance from more than one class. That is, you can **extend** only from one class.

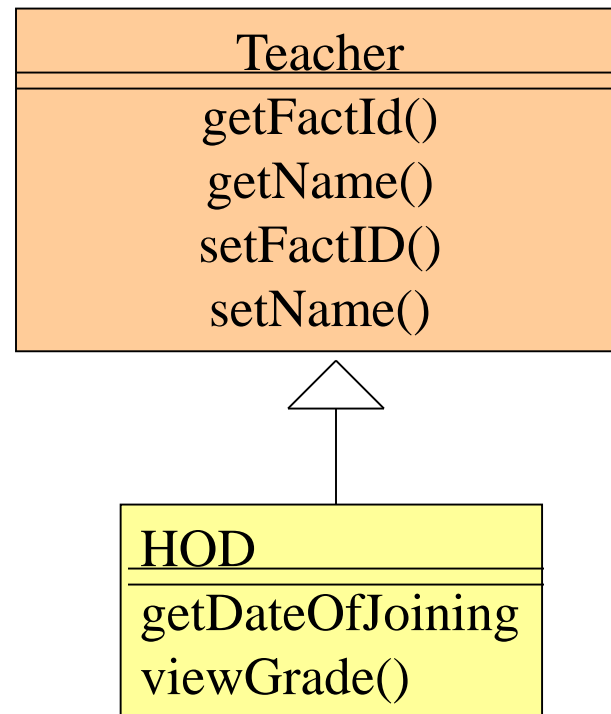
Example scenarios for inheritance

- **Student and Teacher are Person. Person** can be a super class and **Student and Teacher** can be subclass of **Person** class.
Student is-a Person, Teacher is-a Person
- **HOD is-a Teacher. Since Teacher is-a Person, HOD is also a Person**
- **Written_Test and Viva are Test**
- **Theory and Lab are ClassroomSession**
- **SeminarHall is a Classroom**

Syntax and symbols

- **extends** keyword is used to indicate inheritance relationship.
- Syntax:
 - *Class-name2 extends Class-name1*

```
public class HOD extends Teacher{  
/*Features of Teacher class automatically  
available.  
add additional members */  
}
```



Members that are not accessible

- Super class members that cannot be accessed from subclass are
 - **private** members
 - Default members (if subclasses are in the different packages).
- For example, **HOD** class cannot access **factId**, **name** or the static variable **id** of the **Teacher** class since they are **private** but can access all the other methods since they are **public**

super

- Keyword **super()** or **super(<parameters>)** is used to call the super class constructor.
- **super()** can be called only from the constructor. It must be the first statement of the constructor.
- Compiler inserts a **super()** statement in all constructor if subclass constructor does not explicitly call some form of **super()** that calls super class constructor.
- This is to ensure initialization of super class members because they are also part of subclass.

```
public HOD(String nm, String dt){  
    //super(nm) ;  
    dateOfAppointment=dt; }  
...}
```

← **super()** ;
Compiler inserts

Compilation error: cannot resolve symbol constructor Teacher()

Calling super class methods

- The **super** keyword can also be used to invoke super class methods from the subclass method.

super.getName()

- This becomes necessary only when the subclass has redefined the method in super class.

Exercise

- *Create a class called Vehicle. Create subclasses Truck, Bus and Car. Add common methods in the base class and specific methods in the corresponding subclasses. Create a class called Road and create objects for Truck, Bus and Car and display the appropriate messages. Also, in the Vehicle class constructor, initialize the variables color, no of wheels and model. Give appropriate values for these variables from the invoking subclass.*

(20 Mins)

Order of initializations

```
class Order {  
    int i;  
    static {  
        System.out.println("Order class static block ");  
    }  
    Order() {  
        i=10;  
        System.out.println("Order class constructor,i=  
" + i);  
    }  
    {  
        System.out.println("Order class instance  
block,i= " + i);  
    }  
}
```

```
class SubOrder extends Order{
int j=9;
static {
System.out.println("SubOrder class static block");
}
SubOrder () {
    j=15;
    System.out.println("SubOrder class constructor,j= "+
j);
}
{
    System.out.println("SubOrder class instance block,j=
" + j);
}
public static void main(String str[]){
new SubOrder();    }}
```

And this is the result of executing the code...

Order class static block

SubOrder class static block

Order class instance block,i= 0

Order class constructor,i= 10

SubOrder class instance block,j= 9

SubOrder class constructor,j= 15

Conversion and casting

- A subclass object reference can be converted to super class object reference automatically. But for the reverse, casting is required.
- Automatic conversion example
 - Only members of **Teacher** class are accessible

```
Teacher f= new HOD();  
f.getFactId(); //ok  
f.viewGrade(); //error
```
- Casting conversion example
 - We cast it back to **HOD**

```
HOD h=(HOD) f;  
h.viewGrade(); //ok
```
- Dangers of casting: if the original object is not of subclass type then a runtime exception will be thrown on accessing subclass methods.

```
HOD h= (HOD) new Teacher("x");  
  
// Runtime error-ClassCastException
```

protected access

- **protected** access specifier for a member of class A allows access to the members of the classes in the same package as class A (**same as default access specifier**)

+

allows access to the members of the **child classes** of class A which are in different package.

Subclass outside the package

According to [Java Language Specification](#)

6.6.2 Details on protected Access

A protected member or constructor of an object may be accessed from outside the package in which it is declared *only by code that is responsible for the implementation of that object.*

Implied -

We cannot access `protected` member through instance of `super` class.

```
package a;  
public class Order {  
    protected int i;}}
```



A class can access protected members that it has inherited, either through its own reference or its subclass references.

```
package b;  
public class SubOrder extends a.Order{  
    void f(){  
        i=10;  
        System.out.println(i);  
System.out.println((a.Order)this.i);  
        c.SubSubOrder sub=new c.SubSubOrder();  
        sub.i=20;  
        System.out.println(sub.i);}}
```



It cannot access protected members through its super class reference.

```
package c;  
public class SubSubOrder extends b.SubOrder {}
```

```

package b;
class SubSubOrder extends SubOrder{
void g(){
i=10;
System.out.println(i);
System.out.println((a.Order)this).i);
System.out.println((SubOrder)this).i);
}

```

Attempting to access i through super class reference

```

package b;
public class TestClass{
    public static void main(String
str[]){
        SubOrder s1=new SubOrder();
        s1.f();
        new SubSubOrder().g();
        s1.i=20;
    }
}

```

protected member that is inherited cannot be accessed by the classes in the same package as that of the subclass.

Overriding

- Redefinition of an inherited method declared in the super class by the subclass is called Overriding.
- Rules
 1. The signature of the method(method name + argument list) must exactly match.
 2. The return type must be same or a subtype of the return type of super class method (covariant returns)
 3. The access can be same or be increased.
 4. (List of access specifiers in order of their increasing accessibility: **private**→**default**→**protected**→**public**)
 5. Instance methods can be overridden only if they are inherited/visible by the subclass.
 6. Exception thrown cannot be new exceptions or parent class exception.

Covariant Returns

- The overridden method's return type can also be a subtype of the original method return class' subtype.

- For example1:

```
public class Teacher{  
    public Teacher getInstance()  
    {  
        return new Teacher();  
    }  
    public class HOD extends Teacher{  
        public HOD getInstance()  
        {  
            return new HOD();  
        }  
    }  
}
```

Access specifier rule

```
package teacher;

public class Teacher{

    ..

    protected void display(){

        System.out.println(

            "Name "+getName());

        System.out.println("ID

            :"+factId)    ;

    }

}
```

```
package admin;

public class HOD extends

Teacher{

    ..

    public void display(){

        super.display();

        System.out.println(

            "Date of appointment

            "+dateOfAppointment);

    }

}    Cannot have private or

    default access specifier for

    display()
```

Visibility rule

```
package teacher;

public class Teacher{
    ..
    void display() {
        System.out.println(
            "Name "+getName());
        System.out.println("ID
            :"+factId)    ;
    }
}
```

```
package admin;

public class HOD extends
Teacher{
    ..
    private void display() {
        System.out.println(
            "Name "+getName());
        System.out.println(
            "Date of appointment
            "+dateOfAppointment);
    }
}
```

Can have any access specifier here.
Since **private** and **default**
methods are not inherited/visible !

@Override

- When overriding a method, **@Override** annotation could be used
- This tells the compiler that you intend to override a method in the superclass.
- If, for some reason, the compiler detects that the method does not exist in one of the superclasses, it will generate an error.

@Override

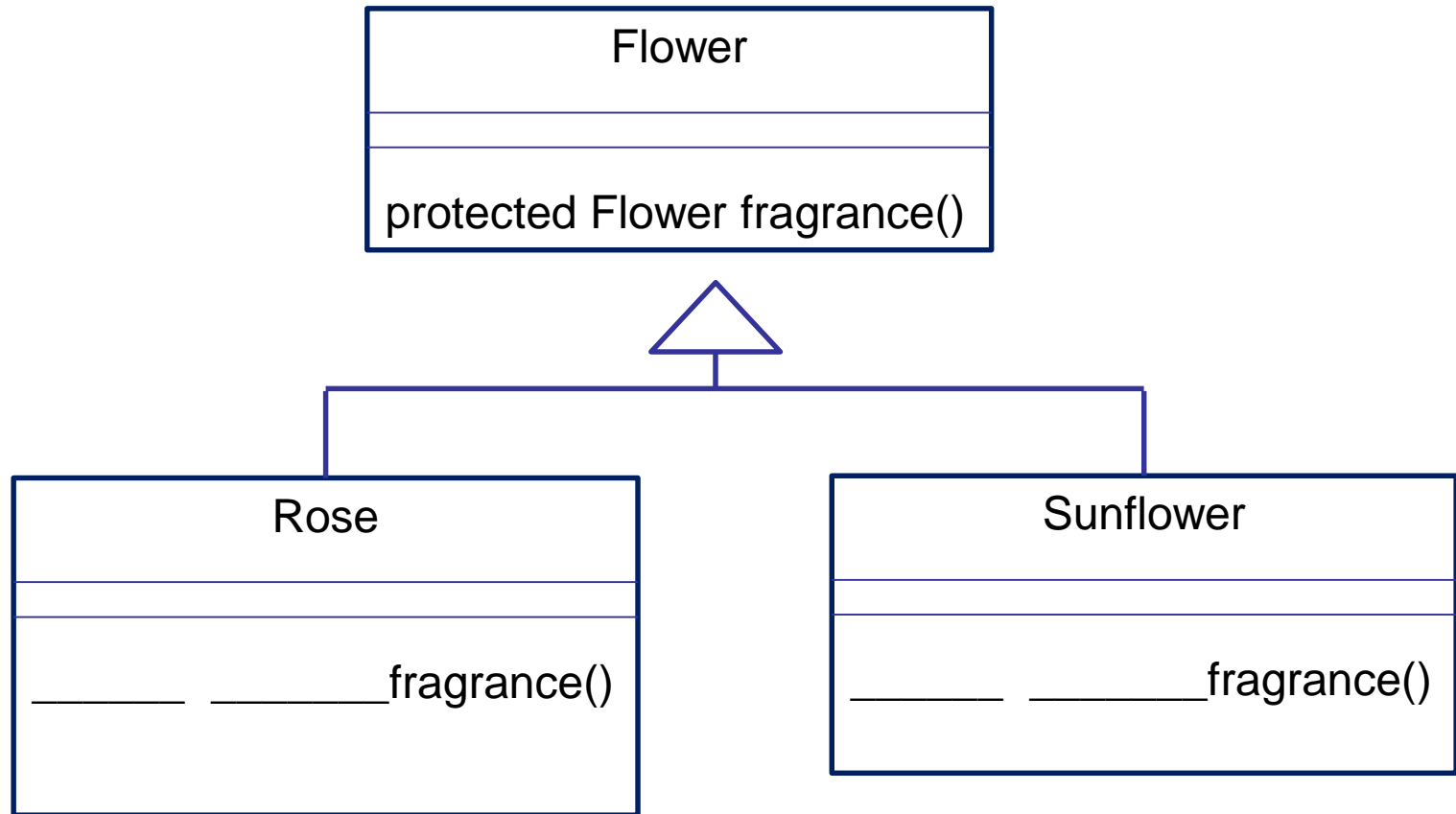
```
public void display() {  
    System.out.println("Name " + getName());  
    System.out.println("Date of appointment  
        " + dateOfAppointment);  
}
```


Polymorphism



Recall what is Polymorphism with an example.

Activity



Fill the possible valid access specifies and the return types.

Static and dynamic binding

- Compiler resolves methods called on an object using
 - Static binding/Early binding: Compiler resolves the call at the compile time
 - `HOD h = new HOD ();`
`h.viewGrade ();`
`h.getName ();`
 - Overloading is resolved at compile-time. Sometimes this is also referred to as compile-time polymorphism.
 - Dynamic binding: Compiler resolves the call at the runtime.
 - Overriding uses run-time polymorphism or simply polymorphism.

Polymorphism in action

- Where is polymorphism used in real life applications?
- Polymorphism allows us to write methods in a generic way.
- In case we want to print a list of teachers who can evaluate answer sheets.
- List of teachers could include HOD as well, because HOD is also a teacher.

```
public static void print(Teacher f[])  
{  
    for(int j=0;j<f.length;j++)  
        f[j].display();  
}
```

instanceof

- Usage:
object-ref instanceof class-name/interface-name
returns a boolean value.

Example:

```
HOD s1= new HOD("Bobby","10.7.2002");  
  
System.out.println(s1 instanceof HOD); // true  
  
System.out.println(s1 instanceof Teacher); // true  
  
System.out.println(null instanceof Student); // false  
  
Object o=s1;  
  
System.out.println(o instanceof Student); // false  
  
The statement below cause compilation error: inconvertible types  
  
System.out.println(s1 instanceof Student);  
  
System.out.println(s1 instanceof College);
```

Exercise

- *Create a class called Worker. Write classes DailyWorker and SalariedWorker that inherit from Worker. Every worker has a name and a salary. Write method pay() to compute the week pay of every worker. A Daily worker is paid on the basis of the number of days she/he works. The salaried worker gets paid the wage for 40 hours a week no matter what the actual worked hours are. Create a few different types of workers and print all the details of the workers(name, salary and D/W (indicating the type of worker)) in sorted order of the salary.*

(45 mins)

Activity

- What happens if you re-declare a private method in your subclass and call it using super class reference?
- Can we still expect polymorphic behavior?

Hiding .Vs. Overriding

- A method is said to be overridden only if it can take the advantage of runtime polymorphism!.
- Otherwise it is only hidden.
- The static methods of the super class are hidden when they are redefined in the sub class.
- The static methods of the super class cannot be redefined as non-static method in subclass or vice-versa.

```
class Classroom{  
    static int capacity=50;  
    public static void printCapacity() {  
        System.out.println("Class Room seating capacity "+  
            capacity); }  
}
```



```
class SeminarHall extends Classroom{
static int capacity =500;
public static void printCapacity(){
System.out.println("Seminar Hall seating capacity "+
capacity); }
```

```
public static void main(String str[]){
ClassRoom examHall= new SeminarHall ();
examHall.printCapacity();
}
}
```

```
//Prints :Class Room seating capacity 50.
```

Member variable re-declaration

```
package a;
public class A{
...
    public String name;
}
}

package b;
import a.*;
public class B extends A{
...
    public String name;
```

final method

- A method that is declared as **final** prevents an inheriting class from overriding that method.

```
package student;

public class Grade{

public final String getGrade() {...}

}

public class MyGrade extends Grade{

public final String getGrade() {...}    // error

}
```

final class

- A class that is declared as **final** prevents other classes from inheriting from it.
- **System**, **Scanner**, **String** class and all the wrapper classes (**Boolean**, **Double**, **Integer** etc.) are **final**.

```
package student;  
public final class Grade{  
...  
}}
```

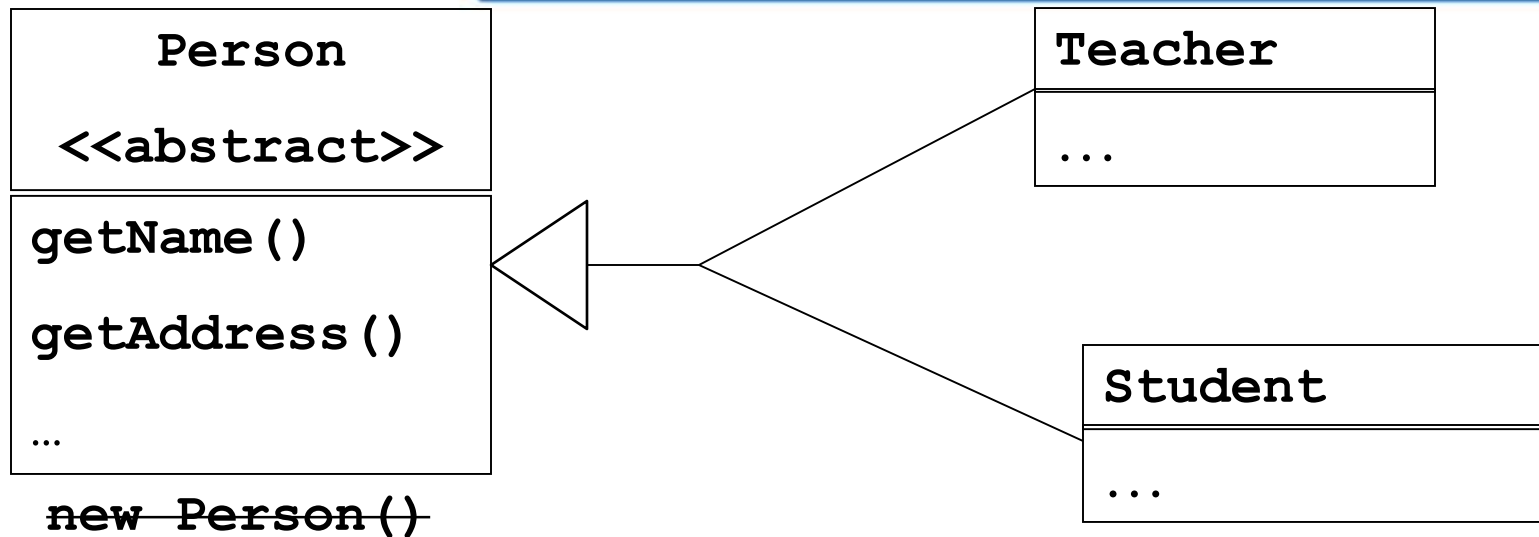
```
public class MyGrade extends Grade{  
...  
}}
```

} error

Abstract class and methods

- Declaring a class as an **abstract** class prevents you from creating instances of that class.
- Abstract methods are the methods that don't have the method body. They are just declarations
- While an **abstract** class can have abstract methods, it could also NOT have any **abstract** methods.
- The whole class must be declared **abstract**, even if a single method is **abstract**.
- An **abstract** method must not be **static**.
- A class can inherit from abstract class either by complete or partial Implementation. In the case of partial implementation, the class should be marked **abstract**.

Example scenario for abstract class



~~`new Person()`~~

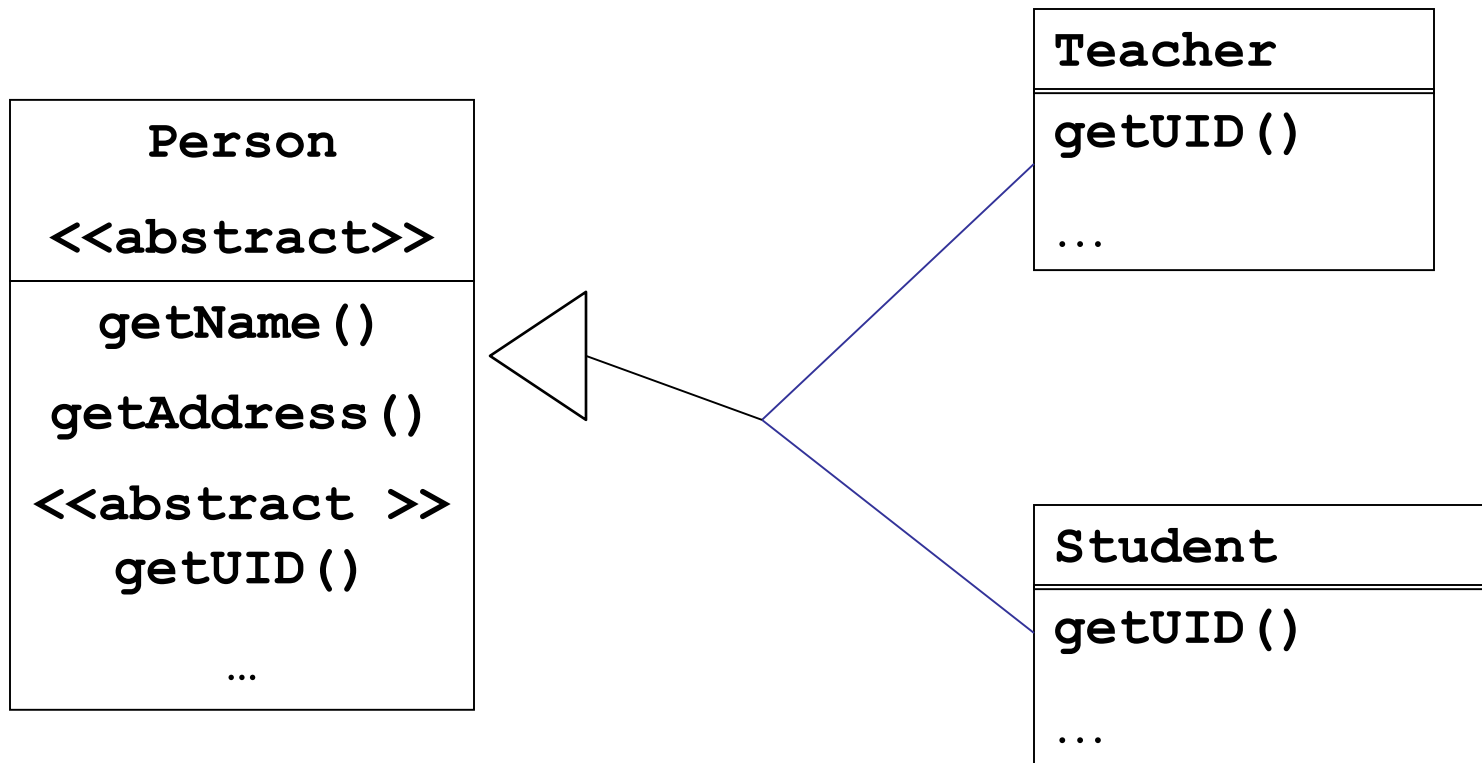
```
package general;
public abstract class Person{
//getters and setters
}
```

```
package student;
```

```
public class Student extends
general.Person{
...}
```

Example scenario for abstract method

Each person must have a unique id.



Since the implementation of id is dependent on the individual classes, we make the `getUID()` method abstract.

Test your understanding

- What will this code print?

```
Person s1= new Student("Mary");  
System.out.println(s1 instanceof Teacher );
```


Exercise

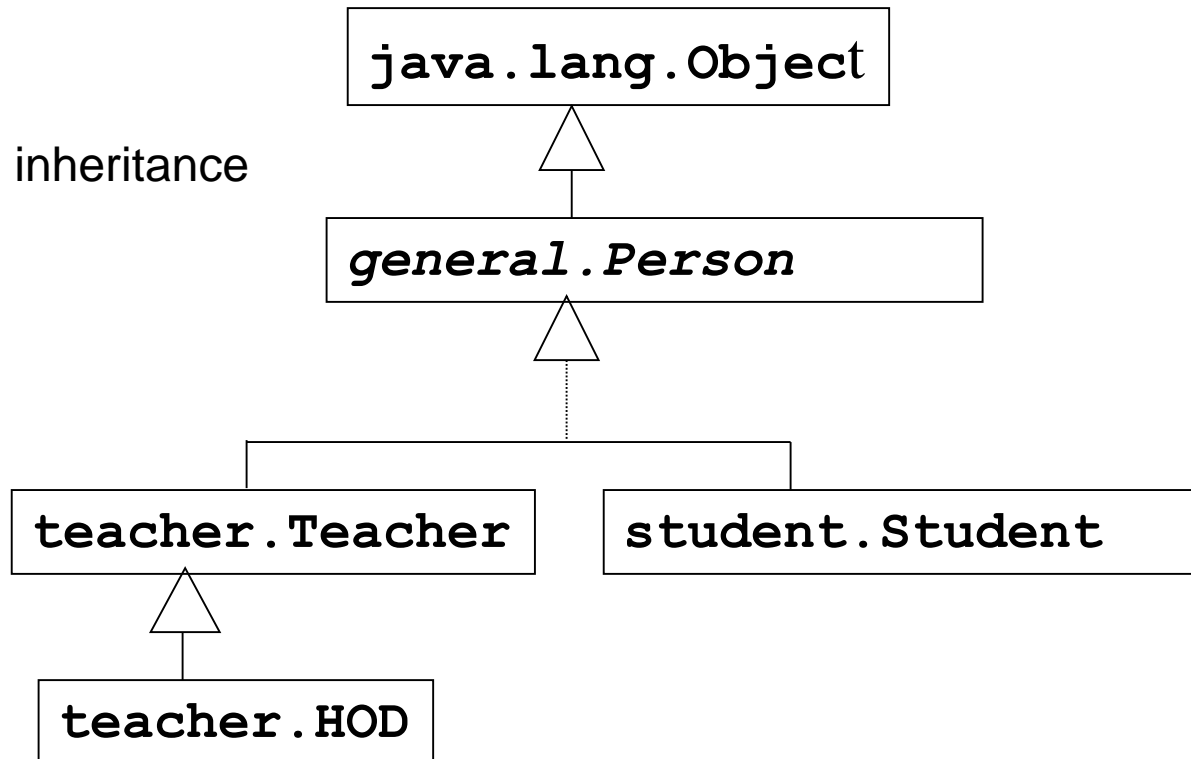
- *In the previous exercise, determine which class and method can be made abstract class or method. Test the application.*

(15 mins)

Object

- All classes in java, by default, inherit from a predefined java class called **Object**.
- **Object** class is defined in **java.lang** package.
- This class is the root of the class hierarchy.
- **Object** class is a concrete class and has a no-argument constructor.

Multi-level inheritance



Object class - important methods

- `public String toString()`
 - `public boolean equals(Object obj)`
 - `public int hashCode()`
 - `protected void finalize() throws Throwable`
- `// ignore throws Throwable for now`

toString()

- `toString()` method of the `Object` class prints class name and the unique hashcode of the object. (Hashcode is an integer value that is associated with an object.)
- If this is not desirable, then we should override `toString()` method.

`equals()`

- `==` compares the addresses
- `equals()` method was added to compare if two objects are equal based on some or all of their attribute values.
- Object class defines an `equals()` method but the implementation compares two references using `==` operator.
- Therefore, invariably, this is overridden by the classes that are interested in providing correct implementation of equals.
- Usually `equals()` implementation should not throw any exception in case classes being compared are not of same type. In such case `false` is returned.
- Also note that syntactically `==` requires the objects of same type to be compared while `equals()` can be used to compare any two objects.

Overriding equals()

```
package student;

public class Grade{

@Override

public boolean equals(Object o){

if(o instanceof Grade){

    Grade g=(Grade)o;


    return g.getGrade().equals(getGrade())

}

else return false;

}

...}
```



getGrade() returns string and since String class has overridden the equals method so we get the desired result here

hashCode ()

- This method returns a hash code value for the object. The implementation in Object class returns unique identifier for each object.
- If you override **equals**, you must override **hashCode**.
- hashCode values for equal objects must be same.
- **equals** and **hashCode** must use the same set of fields.
- Collection classes like **Hashtable**. **HashSet** etc depend on this method heavily.



*Look for Object class in Java API. Locate hashCode() methods and read **hashCode ()** contract*

Overriding hashCode ()

```
package student;

public class Grade{

@Override

public boolean equals(Object o){..}

@Override

    public int hashCode(){

        return g.getGrade().hashCode();

    }

}
```


finalize()

- This method is called just before the object is going to get garbage collector.
- A subclass will have to override the finalize method to dispose of system resources or to perform other cleanup.
- The finalize method is never invoked more than once by a Java virtual machine for any given object.

```
public class Test{  
  
    public void finalize() throws Throwable{  
  
    }  
  
}
```

Exercise

- *In the worker exercise, instead of printing individual attributes like name, salary and so on, if the object is printed automatically the details must be printed. Also two workers are same if their names are same. Therefore before printing salary report, a check needs to be made to see if duplicate workers have been entered. If so, the duplicates must be removed from the list.*

(30 mins)

Exercise

- *Overriding `hashCode()` and `equals()` method for `Student` such that all the ids that are prime and even are goes in one bucket, all the ids that are prime and odd are in another and rest in yet another.*
- *Make sure that `hashCode()` and `equals()` method follow the contract specified in the documentation*

(30 mins)

Exercise

- *A class Connection maintains attributes database url, user name and password. Class needs to maintain the number of count of Connection class. Every time a connection object is created the count must be incremented and every time it is set to null count must be decremented and object must be garbage collected explicitly by the code. At any point, there must be only 10 connection object in the memory. Write a java class to achieve this.*

(30 mins)

Summary

- Inheritance defines relationship among classes, wherein one class shares structure or behavior defined in one or more classes.
- extends keyword is used to indicate inheritance relationship.
- A subclass object reference can be converted to super class object reference automatically. But for the reverse, casting is required.
- Redefinition of an inherited method declared in the super class by the subclass is called Overriding.
- Overloading is compile-time polymorphism and Overriding is run-time polymorphism.
- A final method cannot be overridden. A final class cannot be inherited.
- An abstract class cannot be instantiated.
- All classes in java, by default, inherit from a predefined class called Object.