Inheritance

Naveen Kumar K S
Adith.naveen@gmail.com
http://naveenks.com

Recall

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

- Inheritance is one of the important features of object-oriented languages

Activity: Reuse in real life

- Suppose want to play music in the car. What will you do?
  - You will install a stereo system in the car or you will bring a music player in the car.
  - So, now, the Car ***contains*** a stereo system .
  - Suppose want a bigger car – now, some things like engine, steering wheel will remain the same, but the primary difference is that there should be more room inside the car. What will you do?
  - Go and buy a new car – the new car is likely to have all the features of the old car such as engine, steering wheel etc but will have more room inside
  - The New Car ***is a*** Old Car with more features.

Reuse in Object Oriented Language

- Object Oriented Languages also implements reuse in the same way that we do in real life.
- Using
  - has-a
  - is-a
- Has-a or composition relationship is implemented by having a class having another class as its member, or rather an object having another object as its member.
  - **`class Car{ Stereo s; …}`**
  - **`class College { Teacher[] ts; Student ss[]; … }`**
- Is-a is implemented through what we call ***inheritance*** relationship

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 class can add things that are unique to it.

- As a result of inheritance hierarchical is formed.
- The class that is inherited is called super class and the class that is inheriting is called a subclass.
- Other names of super class – parent class, base class
- Other names of subclass – child class, derived 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**

Activity: match the following

Super class

- Fruit
- Library
- Cat
- Bird

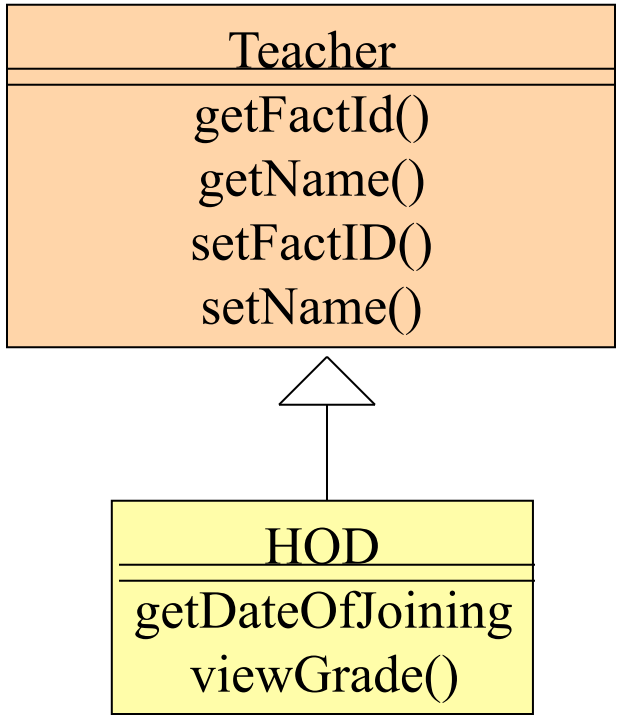
Subclass

- Parrot
- Music
- Tiger
- Books
- Apple
- Mango

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

- All the features that are there in super class are there in subclass as well. But there are some restrictions with respect to what super class features (members) can be accessed from subclass.
- Super class members that cannot be accessed from subclass
  - **`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 most of the methods since they are **`public`**

Example: code for inheritance

- HOD has dateofAppointment and viewGrade as additional features.

```
package teacher;

public class HOD extends Teacher{

private String dateOfAppointment;

public HOD(String nm, String dt){

            super(nm);

            dateOfAppointment=dt;   }

public void viewGrade(Grade[] grades){

System.out.println(getName()+ " HOD viewing
appraisal ");

for(Grade g:grades){

System.out.println("Faculty :"+
g.getFaculty().getName());
```

Calling super class constructor

Calling super class method

```
System.out.println("Student ID : "+
g.getStudent().getRegNo());

System.out.println("Grade: "+ g.getGrade());

}

}

public static void main(String str[]){

student.Student s1= new student.Student("Ravi");

student.Student s2= new student.Student(“Kumar");

HOD h=new HOD("Maverick","1.1.2006");

Teacher f= new Teacher("Sam");
```

```
System.out.println("HOD Name "+ h.getName());
Grade g[]=new Grade[2];
g[0]=new Grade(f,s1,"CS-001", Grade.A);
g[1]=new Grade(f,s2,"CS-001", Grade.B);
h.viewGrade(g);
 }
}
```

Calling super class method

super

- Keyword **super() or super(<parameters>)** is used to call the super class constructor.

```
public HOD(String nm, String dt){
   super(nm); // calls Teacher(String name)
   }
```

- **super()** (like **this()**) can be called only from the constructor.
- It must be the first statement of the constructor.
- Which also implies that **super()** and **this()** cannot be used together.
- 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.

Example : Compiler insertion of `super`

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

On compilation

```
Compilation error: cannot resolve symbol constructor
Teacher()
```

- If the super class does not have a no-argument constructor, a compile-time error occurs.
- Therefore, for classes where super class does not have no-argument constructor, all subclass constructors must explicitly call appropriate super class constructor

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

Compiler inserts

Attempting to call

`Teacher()`

- Since Teacher class does not have no-argument constructor this error occurs.
- Therefore calling super(nm) becomes compulsory from constructor of HOD class since only one constructor is defined in Teacher class which takes one argument.

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 subclass has redefined the method in super class.

More on this in overriding session

Order of initializations when subclass instance is created

1. Static initializations of super class: Static variables are initialized and static blocks are executed in the order of their appearance in the code.
2. Static initializations of subclass class : Static variables are initialized and static blocks are executed in the order of their appearance in the code.
3. Instance initializations of super class : Instance variables are initialized and instance blocks are executed in the order of their appearance in the code.
7. Super class constructor is executed.
8. Instance initializations of subclass class : Instance variables are initialized and instance blocks are executed in the order of their appearance in the code.
9. Subclass class constructor is executed.

Only after the super class part, sub class part happens.

Example: 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 vice versa, casting is required.
- Need to convert will be clear in polymorphism.
- 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

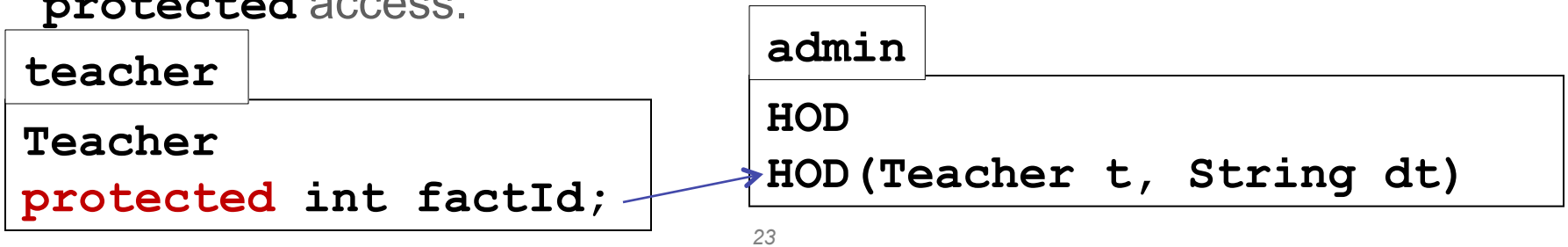
- **`protected`** access specifier for a member of class A allows access to the members of the classes in the same package as class A (same as default access specifier)

  +

  allows access to the members of the child classes of class A which are in different package.
- Therefore for classes inside the same package, **`protected`** access specifier is same as that of default access specifier.
- **`protected`** access specifier makes sense only for subclasses which are in different package from that of super class.
- It is more complicated than this for other package subclasses → *coming up*

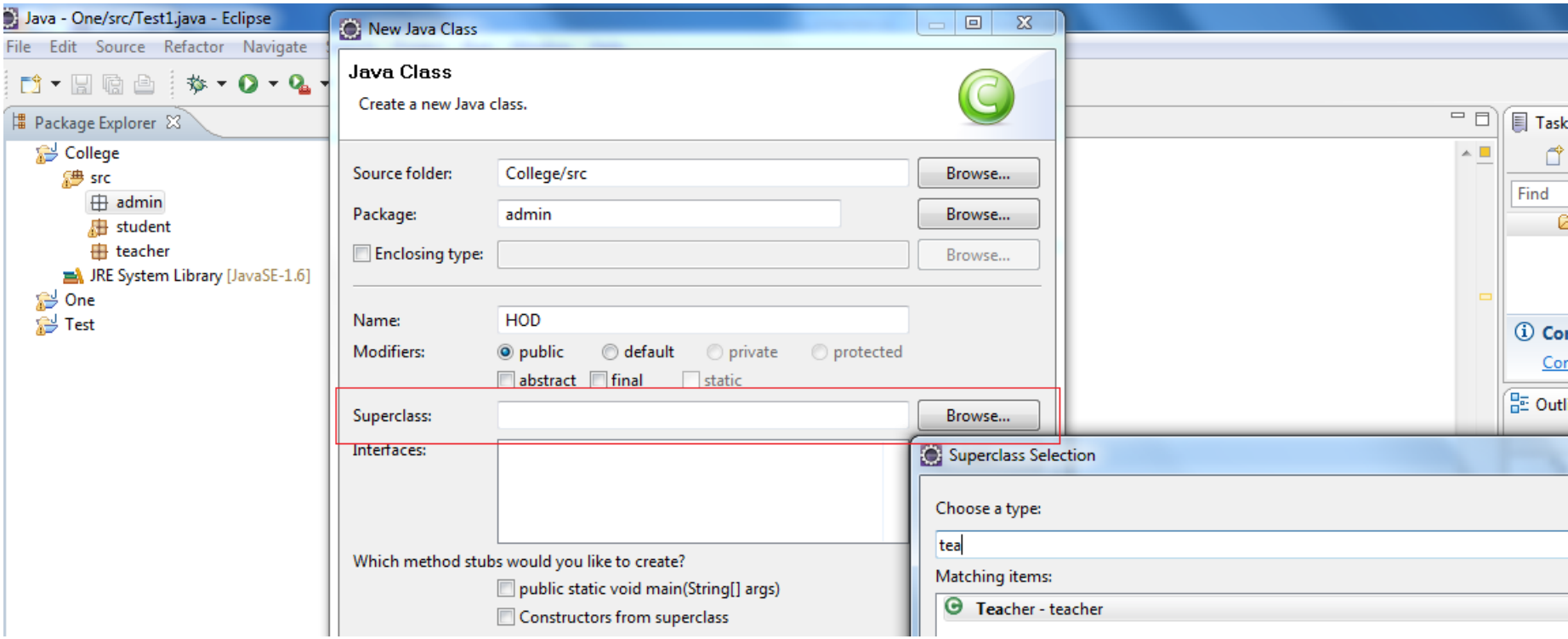
Example scenario for protected

- Let us move the HOD class to a new package called **admin**.
- If an existing teacher is made HOD, then our **HOD** class misfires. Why?
- Because **HOD** constructor creates a new **factId** every time we call **HOD** constructor , but our scenario **factId** for existing teacher must be assigned.
- To get around it, we create another constructor in **HOD** that takes **factId** and **dateOfAppointment**.
- We need to set **factId**. Since **factId** is **private** we cannot access it from **HOD**.
- None of the **public**, **private** and **default** access specifiers will be appropriate here. But **protected** access specifier for **factId** is appropriate here. We need to **name** also **protected** .
- We will also add an no argument constructor in **Teacher** class which has **protected** access.



Activity : protected and super in eclipse

- Create a new package called admin in the College project that is created.
- Create HOD class and make sure you select super class as teacher.Teacher as shown in the figure.



- Copy the following HOD code.

```
package admin;

import teacher.*;
import student.*;
import static teacher.Grade.*;
public class HOD extends Teacher {

private String dateOfAppointment;
public HOD(String nm, String dt){
super(nm);
dateOfAppointment=dt;}

public HOD(Teacher t, String dt){
this.name=t.getName();
this.factId=t.getFactId();
dateOfAppointment=dt;
}
```

```
public void viewGrade(teacher.Grade[] grades){
System.out.println(getName()+ " HOD viewing appraisal ");
for(teacher.Grade g:grades){
System.out.println("Faculty :"+ g.getFaculty().getName());
System.out.println("Student ID : "+
g.getStudent().getRegNo());
System.out.println("Grade: "+ g.getGrade());
}}
public static void main(String str[]){
Student s1= new Student("Ravi");
Student s2= new Student("Kumar");
HOD h=new HOD("Maverick","1.1.2006");
Teacher f= new Teacher("Sam");
teacher.Grade g[]=new teacher.Grade[2];
g[0]=new teacher.Grade(f,s1,"CS-001", A);
g[1]=new teacher.Grade(f,s2,"CS-001", B);
h.viewGrade(g);}}
```

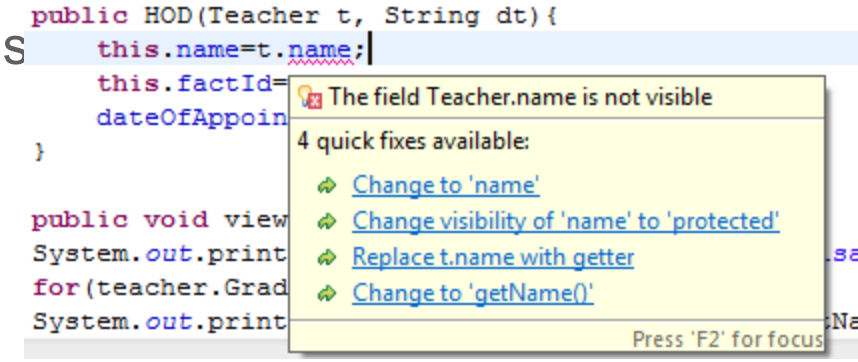
- Correct the errors by making changes to Teacher code.

- List out the corrections you made and make sure your Teacher class looks like this.

```
package teacher;
public class Teacher{
protected int factId;
protected  String name;
private static int id;
protected Teacher(){}
public int getFactId(){return factId;}
public String getName(){return name;}
public void setName(String name){this.name=name;}
private int generateId()
{id++;
return id;}
public static int getId(){return id;}
public Teacher(String name){this.name=name;
factId=generateId();}}
```

Tell me why?

- Why can't we access **`protected`** member through instance of **`super`** class? **`HOD`** class can access **`name`** through **`this`** but cannot access name through **`Teacher`** class ins



  - This is different from private members. A class can access private members even through its objects.

  - There are rules laid out by the java language specification about this.
  - The java language specification says ….

Subclass outside the package

(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.*

6.6.2.1 Access to a protected Member

Let C be the class in which a protected member m is declared. Access is permitted only within the body of a subclass S of C. In addition, if Id denotes an instance field or instance method, then: If the access is by a qualified name Q.Id, where Q is an ExpressionName, then the access is permitted if and only if the type of the expression Q is S or a subclass of S. If the access is by a field access expression E.Id, where E is a Primary expression, or by a method invocation expression E.Id(. . .), where E is a Primary expression, then the access is permitted if and only if the type of E is S or a subclass of S.

Let's understand this by example

```
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.

Test your understanding

Match the folllowing

Access specifiers

- `public`
- `protected`
- `<default>`
- `private`

Accessible…

- Within the class
- In classes of the same package
- In classes of different package
- In subclasses of the same package
- In subclasses of different package

Overriding

- Many times we may have to redefine some of the existing features of a class in a subclass.
- For example, a big car inheriting from a small car may retain its features like steering wheel but needs to have interiors like seats, seat cover etc of bigger size.
- Redefinition of an inherited method declared in the super class by the subclass is called Overriding.
- When a method is redefined there is some flexibility in terms of not having exactly same method declaration as the super class method.
- They are defined by a set of 5 rules.

Rules

- The signature of the method(method name + argument list) must exactly match.
- The return type must be same or a subtype of the return type of super class method (covariant returns)
- The access can be same or be increased.

  (List of access specifiers in order of their increasing accessibility: **`private→default→protected→public`**)
- Instance methods can be overridden only if they are inherited/visible by the subclass.
- Exception thrown cannot be new exceptions or parent class exception.

More on this in exceptions session

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

- Polymorphism is an object-oriented language feature.
- Polymorphism refers to an object's ability to use a single method name to invoke one of different methods at run time – depending on where it is in the inheritance hierarchy.
- It exists only when there is inheritance and the compiler uses dynamic binding to implement it.
- Compiler resolves methods called on a object using
  - Static binding/Early binding: Compiler resolves the call at the compile time
  - Dynamic binding: Compiler resolves the call at the runtime
- Overloading is resolved at compile-time. Sometimes this is also referred to as compile-time polymorphism.
- Overriding uses run-time polymorphism or simply polymorphism.

Static Binding Example

- ```
  HOD h = new HOD();
  h.viewGrade();
  h.getName();
  ```

- ```
  Teacher t= new Teacher();
  t.getName();
  ```

- Since the type of object is known at compile time, to be either of type HOD in the first example or of type Teacher in the 2nd example, compiler knows which method to call and so can statically bind the method.

Polymorphism by example

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

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

We defined display method in Teacher class and we have overridden this method in HOD class.

```
package teacher;
public class Test{
public static void print(Teacher f){
for(int j=0;j<f.length;j++)
f.display();}
```

- What type of arguments can you pass in the call to print method?

```
public static void main(String str[]){
admin.HOD h =new admin.HOD("Ned","1.1.2006");
print(h);
Teacher f= new Teacher("Sam");
print(f);}}
```

- Code prints HOD's **display()** when HOD reference is passed and Teacher's display() when Teacher reference is passes.
- Even though the print method calls Teacher's **display()**, based on the object type that is passed at runtime which display method to be called is decided.

HOD's display()

Output:

```
Name :Ned
ID :1
Date of appointment 1.1.2006
```

```
Name :Sam
ID :2
```

Teacher's display()

Polymorphism in action

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

How will you define such a method that prints list of teachers?

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

Array conversions and Polymorphism

Ways to call the print method:

```
public static void main(String str[]){
Teacher f[]=new Teacher[2];
f[0]=new HOD(“Ned”,”1.1.2006”);
f[1]=new Teacher(“Sam”);
print(f);

HOD f1[]=new HOD[2];
f1[0]=new HOD("Nedy","1.1.2006");
f1[1]=new HOD("Samy","1.1.2008");
print(f1);  }
```

- First print method resolution happens by exact match
- Second print method resolution happens due to automatic conversion.
- Polymorphic conversions happen at for the arrays as well. That is, an array of subclass objects can be automatically converted to array of superclass object.

instanceof

- Usage:

  object-ref **instanceof** ***class-name/interface-name***

  returns a **boolean** value.

later

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);
```

Activity

- What code will you add to print an HOD before HOD's name in the the generic print method listed below if the actual instance type is HOD?

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

private method re-declaration and polymorphism

- What happens if you re-declare a private method method in your subclass and call it using super class reference?
- Can we still expect a polymorphic behavior?
- Since the private and static methods are not inherited, they are not overridden.
- As they are not overridden, there is no polymorphic behavior.

Test your understanding

```
public class A{
void f(){
System.out.println("in
A:f()");
}
public static void
main(String str[]){
A c= new B();
c.f();
}
 }
class B extends A{
public void f(){
System.out.println("in
B:f()");
}
```

```
public class A{
private void f(){
System.out.println("in
A:f()");
}
public static void
main(String str[]){
A c= new B();
c.f();
}
 }
class B extends A{
public void f(){
System.out.println("in
B:f()");
}
```

What will the code on the left print?
What will the code on the right print?

Hiding .Vs. Overriding

- Are static methods overridden?
- Answer is No!
- A method is said to be overridden only if it can take the advantage of runtime polymorphism!.
- Otherwise it is only hidden.
- So 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.
- While instance methods are overridden, class methods are only hidden
- Also instance method cannot be overridden to be a class method and vice versa.

Example: Hiding

```
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

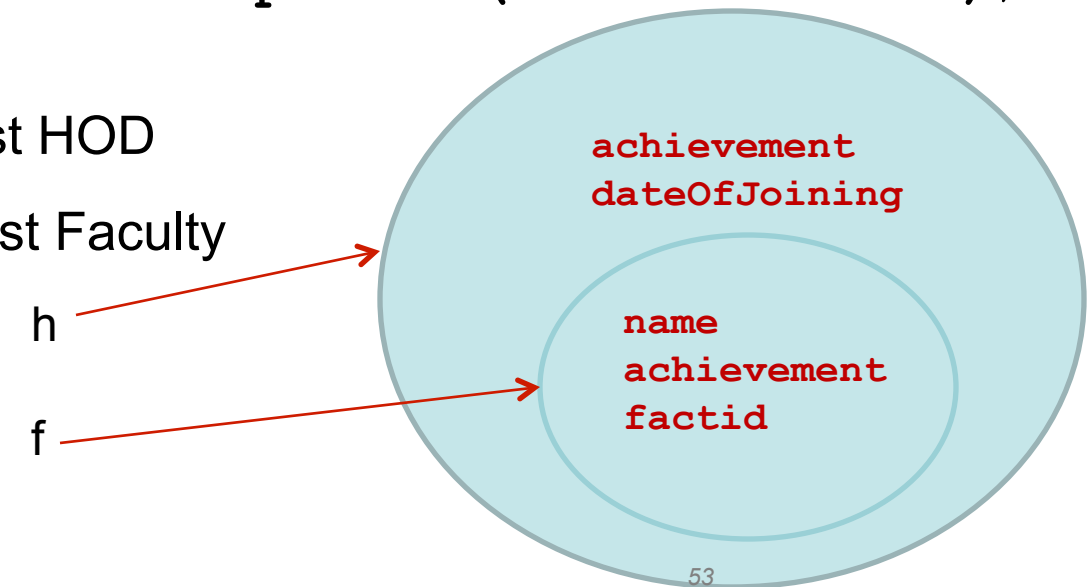
- Can we have the same member variable name in super and subclass?
- Yes, in such cases, subclass will have 2 members with same name – one that comes from its super class and one that it defines.
- Member inherited from the super class can be accessed through super reference by converting subclass reference into super class reference.

```
package teacher;
public class Teacher{
…
 public String achievements;
}}

package admin;

import teacher.*;

public class HOD extends Teacher{

…
public String achievements;
```

```
public static void main(String str[]){
        HOD h=new HOD("Maverick","1.1.2006");
        h.achievements="Best HOD";
        Teacher f=h;
        f.achievements="Best Faculty";
        System.out.println(h.achievements);
        System.out.println(f.achievements);
}}
```

Output: Best HOD

Best Faculty



`final` Revisited

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

Example: `final` method

```
package student;
public class Grade{
public final String getGrade(){
…
}}
```

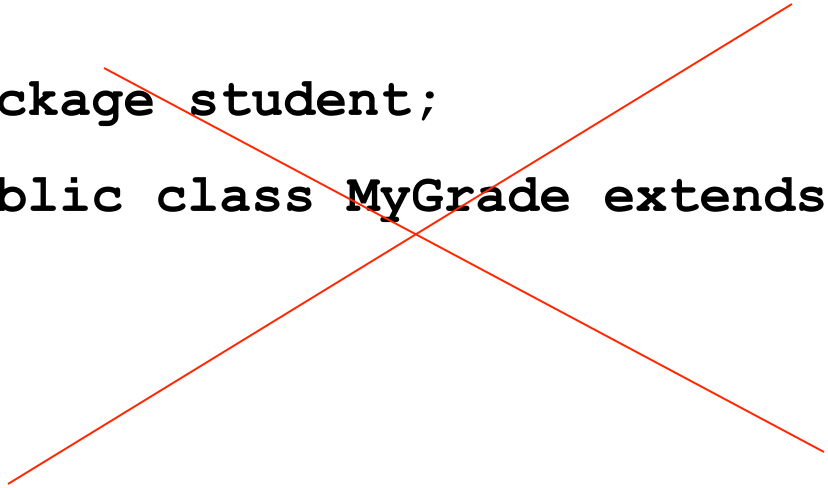
```
public class MyGrade extends Grade{
public final String getGrade(){
…
}}
```

Compilation error

Example: `final` class

```
package student;

public final class Grade{

…

}}
```

```
package student;

public class MyGrade extends Grade{

…

}}
```



Compilation error

Abstract class

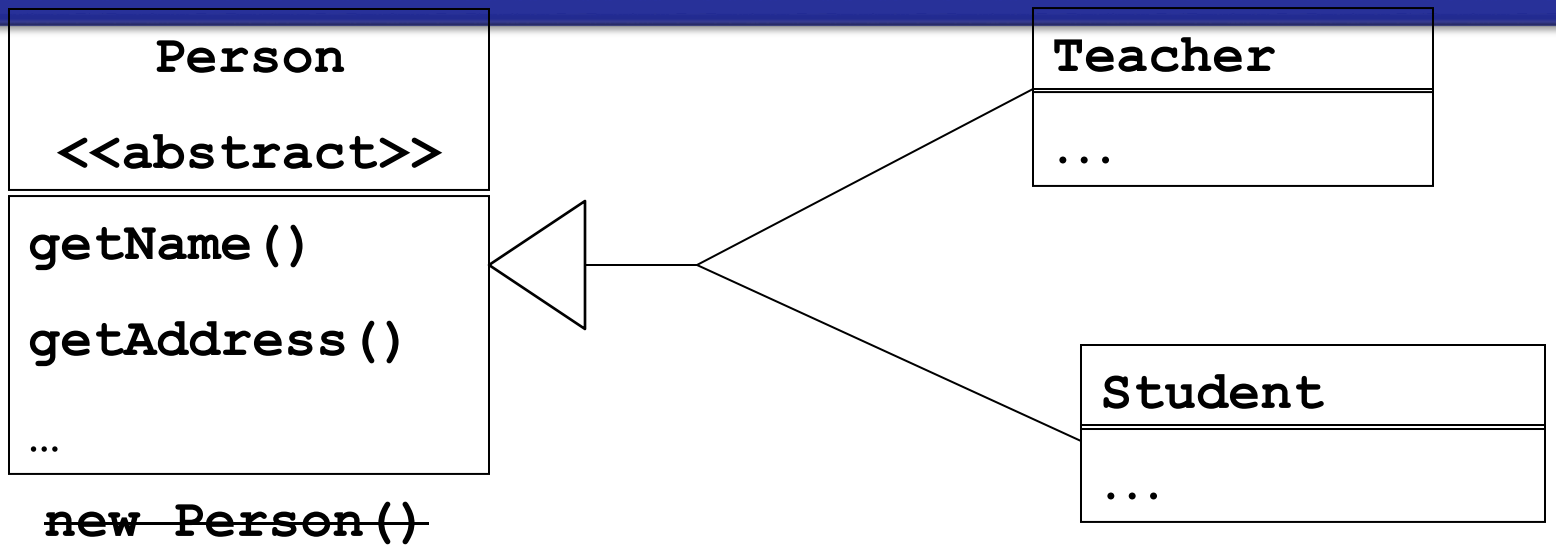
- Declaring a class as an **`abstract`** class prevents somebody from creating instances of that class.
- Abstract methods are the methods that don't have the method body. They are just declarations
- While an **`abstract`** class can have abstract methods, it could also NOT have any **`abstract`** methods.
- Bear in mind that even if a single method is **`abstract`**, the whole class must be declared **`abstract`**.
- Please note that the classes that we have created so far are called concrete classes (classes from which you could create instances.
- An **`abstract`** method must not be **`static`**.

Inheriting abstract class

- A class can inherit from abstract class in two ways
  - By Complete Implementation
    - A class that inherits from the `abstract` class and override all the abstract methods by providing implementations specific to that class.
    - This results in a concrete class
  - By Partial Implementation
    - A class that inherits from the abstract class and does NOT override one or more `abstract` method.
    - Such class must be marked as an `abstract` class.

Example scenario for abstract class

- College needs to maintain information a student or a faculty or a principal or a HOD.
- If we notice carefully, most of the common information about all these entities are name, address, date of birth, unique id.
- Imagine writing getters and setter for all these in all the classes (even if it were cut paste, it is not easy !)
- Instead a generic class called Person with all common attributes can be created and have Teacher class, Student class and others inherit from Person.
- But we don't want anybody to create Person objects. In other words, a college needs not just a person but Teachers, Students etc.
- To prevent instance creation we can mark Person as abstract.



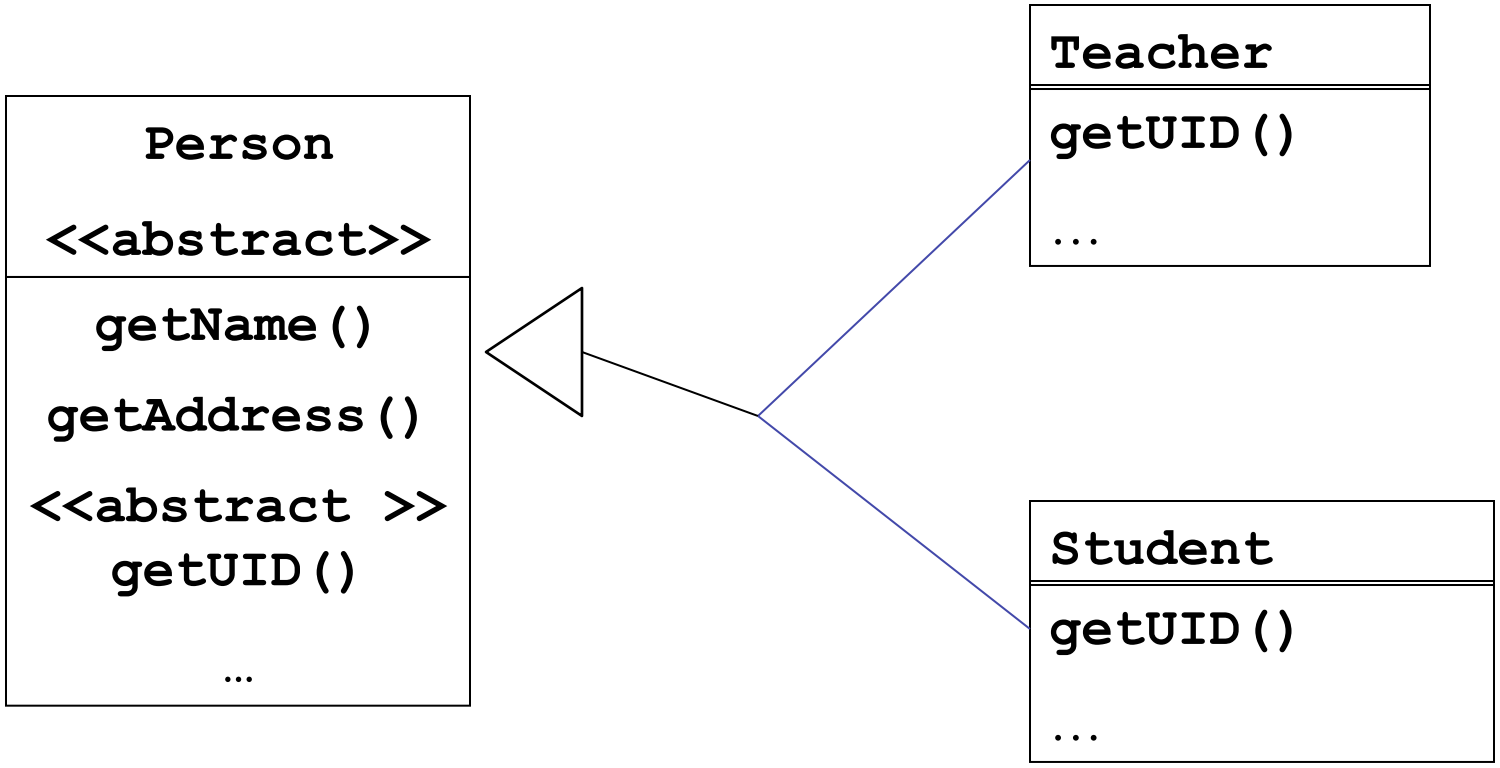
~~new Person()~~

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

package student;

public class Student extends
general.Person{

…}
```

Example scenario for abstract method

- How to make sure each person must have a unique id?
- Since every class has its own way of generating ID, the logic for this cannot be written in Person.
- Yet we need to ensure that some method to generate ID is there in all classes that inherit from Person
- Solution to this is to declare an abstract method in the Person class without any implementation. We will call this **`getUID()`**
- Now any class that inherits from Person class must provide implementation for **`getUID()`** for it to be a concrete class.

Each person must have a unique id.



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

Example: Abstract class code

```
package general;
public abstract class Person{

private String name;
private String address;
protected Person(){}

public void setName(String name){
      if(name==null)
      System.out.println("Invalid name");
      else
      this.name=name;
}

public String getName(){
return name;}
```

```
public void setAddress(String address){
      if(address==null)
      System.out.println("Invalid name");
      else
      this.address=address;}

public String getAddress(){return address;}

public abstract int getUID();

public Person(String name, String address){
setName(name); setAddress(address);}

public Person(String name){
setName(name);
}
}
```

no implementation

Two constructors added here for member initialization.

Example: inheriting from abstract class

```
package student;
public class Student extends general.Person{
//remove getters and setter for name attribute
public int getUID(){return regNo;}
public Student(String nm, String d){
super(nm);
…}}
```

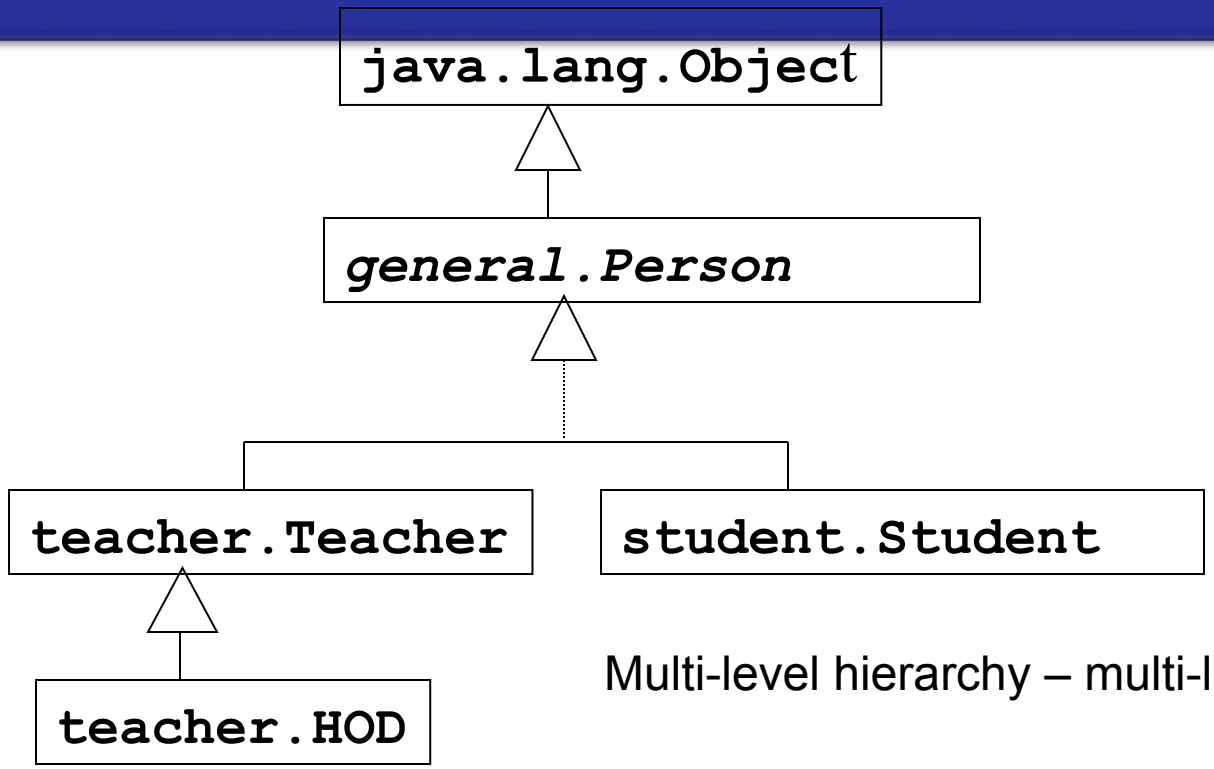
Abstract method implemented

```
  package teacher;
  public class Teacher extends general.Person{
  //remove getters and setter for name attribute
  public int getUID(){return factId;}
  etc…}
```

Object

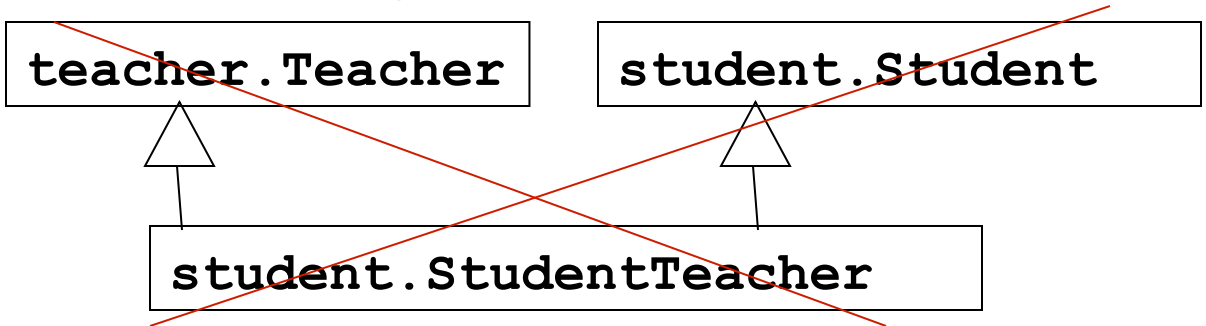
- All classes in java, by default, inherit from a predefined java class called **`Object`**.
- **`Object`** class is defined in **`java.lang`** package.
- This class is the root of the class hierarchy.
- All objects, including arrays, directly or indirectly inherit the methods of this class.
- Even if we do not explicitly write code to inherit from **`Object`**, compiler inserts **`extends Object`** to our class if it finds no **`extends`** clause specified in our class definition.
- **`Object`** class is a concrete class and has a no-argument constructor.

Application Hierarchy



Multi-level hierarchy – multi-level inheritance

Java does not support inheritance from more than one class. That is, you can **extend** only from one class.



Test your understanding

- What will this code print?

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

Object class - important methods

Two important methods (for now)

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

`// ignore throws Throwable for now`

toString()

- What will happen when we try to print objects like primitives?

```
Teacher f=new Teacher ("Tom");
System.out.println(f);
```

It prints: teacher.Teacher@f4a24a

- This is because **print** methods (and many more methods) call **toString()** method on the **Object** to get the string representation of the object. It displays whatever the **toString()** method returns.

- **toString()** method that is inherited from 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 we are not happy with this result, then we should override **toString()** method.

Overriding `toString()`

```
package teacher;

public class Teacher{

…

@Override

public String toString(){

return getName()+" (" +factId+ ")“; }}

import teacher.*;

public class Test{

public static void main(String str[]){

Teacher f=new Teacher ("Tom");

   System.out.println(f);

}}
```


Prints : Tom (1)

equals()

- When will you say that two objects are same? When they point to the same location or if some or all of the data members have same values?
- The answer depends on the kind of object and business logic of our application. So let us take **Grade** object.
- We will call two **Grade** objects same if their **grade** attributes match.
- Since **==** compares the addresses, we need **equals() method –** the kind we had in **String** class.
- Object class already defines an **equals()** method but the implementation compares two references using **==** operator.
- So all we need to do is override this method. 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

```
import student.*;
class Test{
public static void main(String str[]){
Grade g=new Grade(new Student("Raja"),new int[]{ 80,85,
91,86, 82});

Grade g1=new Grade(new Student("Rani"),new int[]{ 90,70,
89,78, 92});
if(g.equals(g1))
      System.out.println("Same");
 else
   System.out.println("Differnt");
       }
} }}
```

Activity: `equals()`

- When will you say two **`Teachers`** objects are same?

- When the **`factId`** of the two teachers object are same. So it is enough to test just the **`factId`**.
- Write **`equals()`** method to do this.

`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 class like **`Hashtable. HashSet`** etc depend on this method heavily.

`hashCode()` contract by javadoc

- Whenever it is invoked on the same object more than once during an execution of a Java application, the **`hashCode`** method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are **`equal`** according to the **`equals(Object`**) method, then calling the **`hashCode`** method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the equals(**`Object`**) method, then calling the **`hashCode`** method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

Overriding `hashcode ()`

```
package student;

public class Grade{

@Override

public boolean equals(Object o){..}

public int hashCode(){

return g.getGrade().hashCode();

}

}

else return false

…}
```

finilize()

- This methods is called just before the object is going to get garbage collector.
- A subclass will have to overrides 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{

}

}
```

Question