Inheritance

Introduction:

Inheritance is a powerful tool in object oriented design. Inheritance supports:

- Customization and enhancement of the working classes
- Code reuse and discourages the code duplication
- Defining more general class and modify to suit a particular situation

RVK................

Features of Inheritance

- Inheritance is the process of acquiring the properties of one class by another class.
- New classes derive capabilities (expressed as fields and methods) from existing classes.
- Inheritance saves development time by encouraging the reuse of proven and debugged classes.
- The `extends` keyword causes a subclass to inherit all fields and methods declared in a non-final superclass (including the superclass' superclasses).

```
class DerivedClass extends BaseClass
```

Types of inheritance

Java supports following types of inheritance

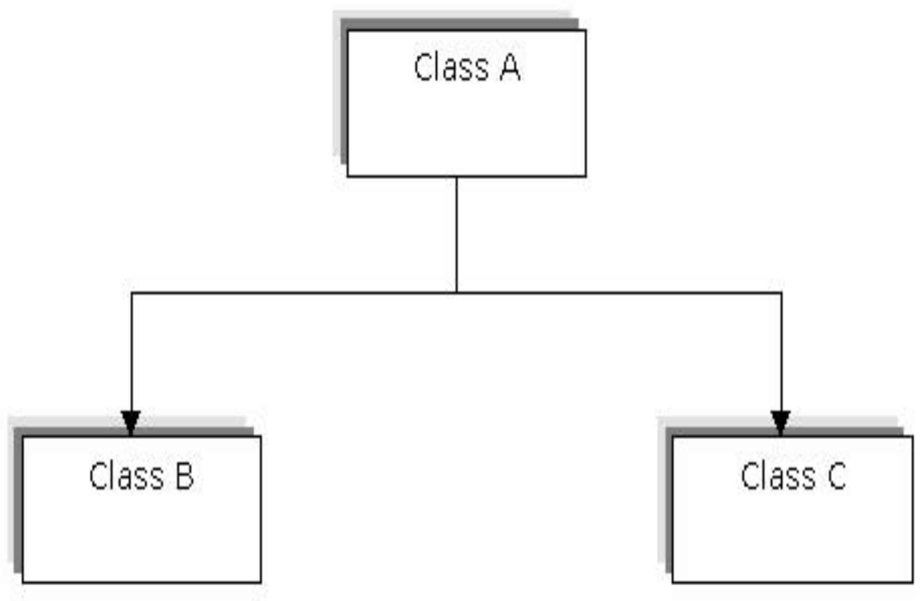
Single Inheritance

Multilevel inheritance

Hierarchial inheritance

Inheritance in Java (Contd.)

- Implementing Different Types of Inheritance
  - Single level inheritance
    - Derives a subclass from a single superclass. For example, subclasses B and C inherit the properties of a single superclass, A. The following figure shows the structure of single level inheritance:



Inheritance in Java (Contd.)

- Implementing Different Types of Inheritance (Contd.)
  - Single level inheritance
    - The following syntax shows how to implement single level inheritance:

```
class A
{
}
class B extends A
{
}
```

In the preceding syntax, the `extends` keyword is used to derive a subclass from a superclass.

Inheritance in Java (Contd.)

- Implementing Different Types of Inheritance (Contd.)
  - Multi level inheritance
    - The following syntax shows how to implement Multi level inheritance:

```
class A
{
}
class B extends A
{
}
class C extends B
{
}
```

In the preceding syntax, the `extends` keyword is used to derive a subclass from a superclass.

Basic rules of inheritance are:

– Constructors are not inherited by subclasses

– Subclasses do not inherit the superclass‘s private members

– Static members are not inherited

Method Overriding

- A subclass can provide a different implementation for a method defined in the superclass.
- Rules:

– The prototype of the method in the subclass must exactly be the same as that of the super class.

– The access modifier of the overriding method of the subclass cannot be more narrow than that allowed by superclass.

– The overriding method of the subclass cannot throw a broader exception than

that thrown by the super class unless it is a runtime exception.

– A subclass cannot override methods that are declared `final` in the superclass.

super keyword:

Keyword super is used inside the overridden child class method to call the base class method or a constructor.

Ex:

class b extends a

```
{ super.put();// calls put() of class a}
```

or super() calls default constructor of the base class

or super(x,y) calls parameterized constructor

Final specifier:

- Used to create symbolic constants.

Eg. final double pi=3.14;

- When used with methods prevents overriding
- When used on a class prevents inheritance

```
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**

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();
}}
```

```
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()`**

```
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 !

Polymorphism

Recall what is Polymorphism with an example.

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

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

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);
}}
```

```
Output:
Name :Ned
ID :1
Date of appointment 1.1.2006
Name :Sam
ID :2
```

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();
}
```

Array conversions and Polymorphism

- Polymorphic conversions happen for arrays as well. That is, an array of subclass objects can be automatically converted to array of superclass object.

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);   }
```

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);
```

Activity

- What code will you add to print HOD before HOD’s name in 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();
}
```

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.
```

Exercise

- *The Charity Collection Box contains money in different currencies - dollars-cents or pounds-pence or rupees-paise. All of these currencies have notes and coins. The note and coin numbers are counted when they are added based on their value (that is number of 5 rupee notes, or $1 dollar note).*
- *A super class representing Currency is created with different denomination for of notes and coins. Subclass Dollar, Pound and Rupee has conversion methods to rupees, print() and compute().*
- *Create class called CollectionBox that allows entry of these currencies in terms of number of notes and coins of different denomination. Create a display method that allows any of these currency types and displays the total amount collected in terms of Rupees. (Assume1 dollar= Rs. 50 and 1 pound = 1.6232 U.S. dollars)* (45 mins)

`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{~~
~~…~~
~~}}~~

$\Big\}$ error

Exercise

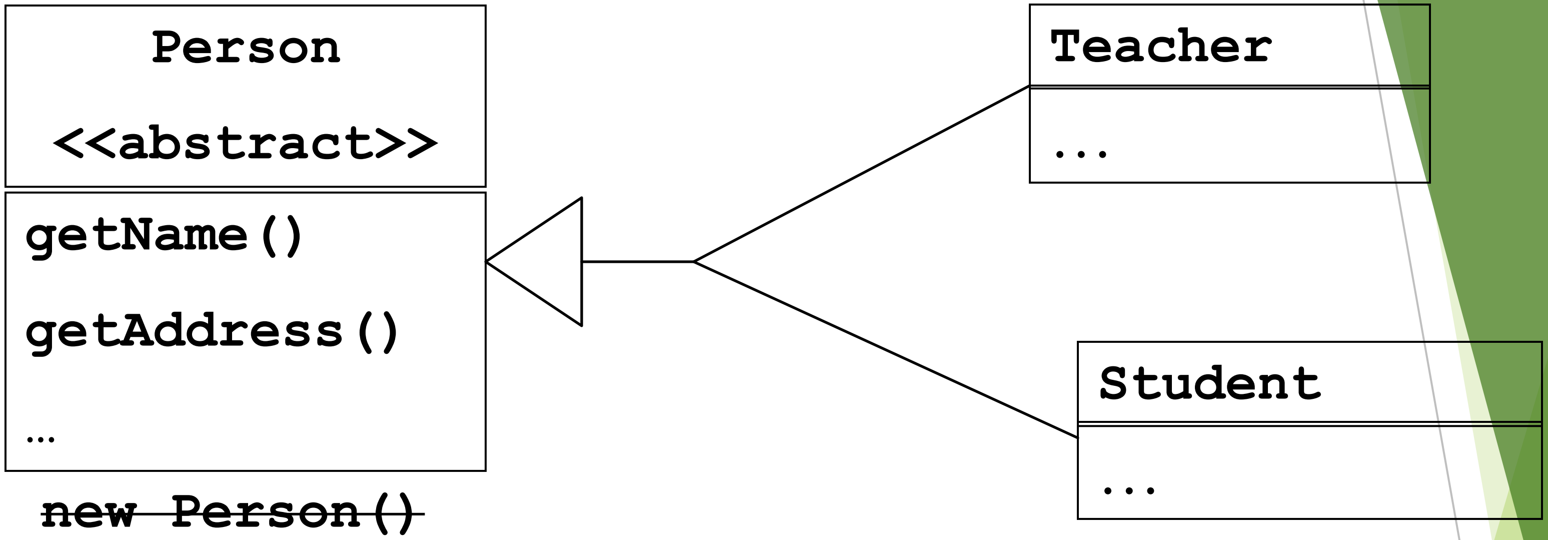
- *An application interacts with databases of different types. Every database object has user name, password and a url.*
- *Databases classes- Oracle, SQLServer and MySql are created.*
- *Only10 database objects are allowed at a time. And these are stored in an array that can represent any of the above objects. No further subclasses can be created to prevent the numbers going beyond 10.*
- *Provide a display method that will print the details of the database objects in the array.*
- *Test the application by creating 10 objects and display it and then creating 11th object.*

Hint: GetInstance() method that returns instances of the object is overridden in the individual classes. This will make sure that number of instances does not go beyond 10. (30 mins)

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`**.

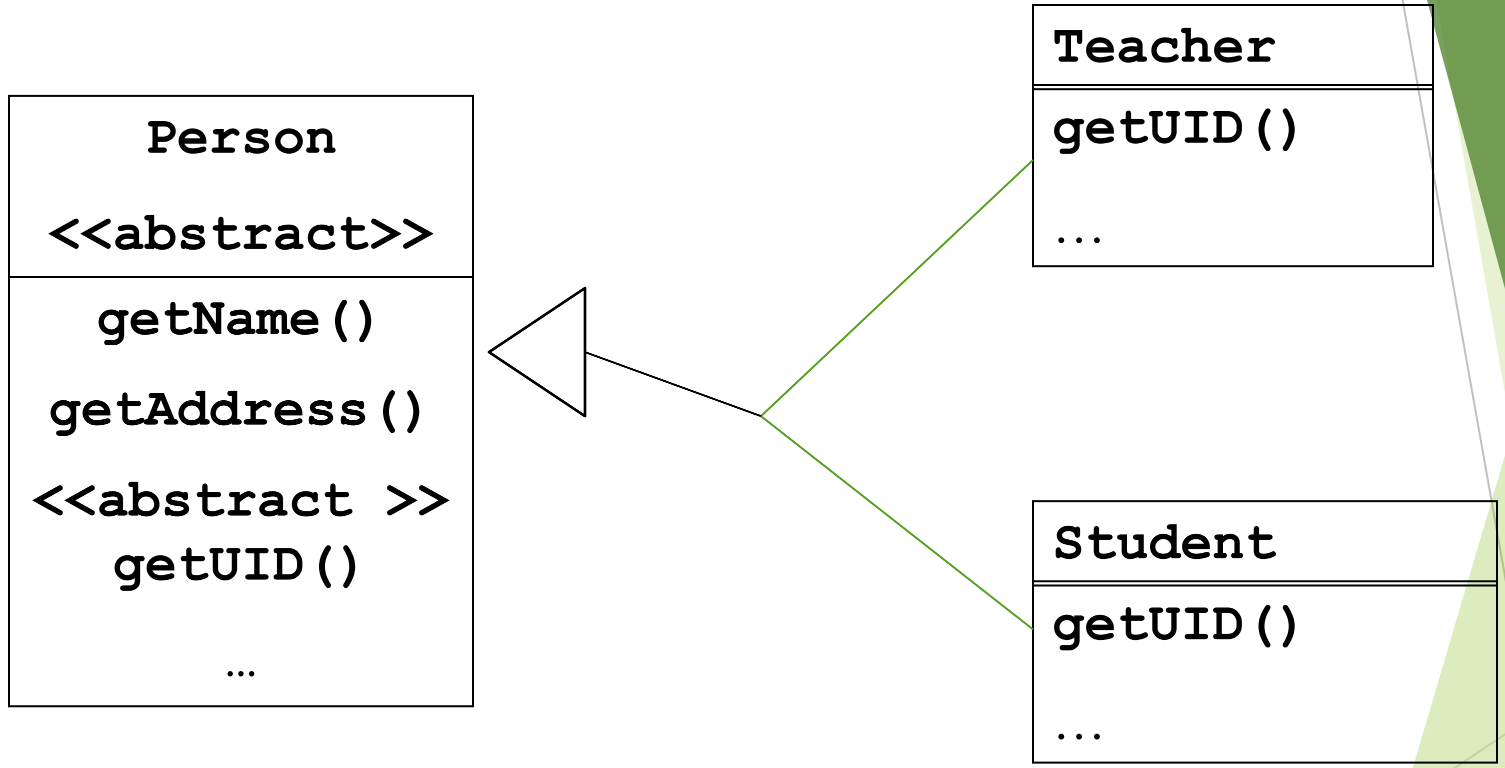
Example scenario for abstract class



```
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

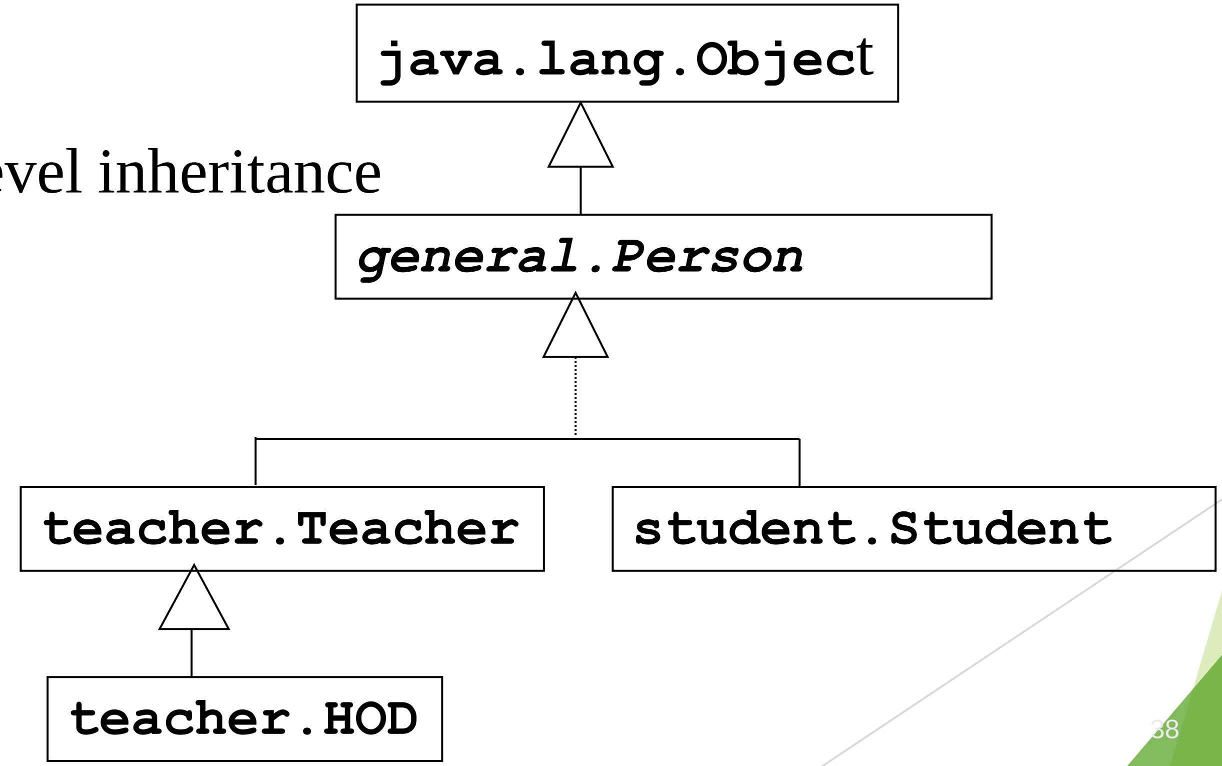
- *In the previous exercise, determine which class and method can be made abstract class or method. Test the application.*

(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.

```
package teacher;
public class Teacher{
…
@Override
public String toString(){
return getName()+" (" +factId+ ")‶; }}
public static void main(String str[]){
Teacher f=new Teacher ("Tom");
  System.out.println(f);
}}
```

`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.

```
Thread t1=new Thread("one");
Thread t2=new Thread("one");
System.out.println(t1.equals(t2));   //false

Integer i1=new Integer(10);
Integer i2=new Integer(10);
System.out.println(i1.equals(i2));  //true

String s1=new String("hello");
String s2=new String("hello");
System.out.println(s1.equals(s2));  //true
```

Justify the answers………….

```
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");
   }
} }}
```

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*

```
package student;
public class Grade{
@Override
public boolean equals(Object o){..}
@Override
 public int hashCode(){
   return g.getGrade().hashCode();
   }
}
```

finalize()

- 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{
}
}
```

- **The getClass() Method**

- The getClass() method is a final method (cannot be overriden) that returns a runtime representation of the class of this object. This method returns a Class object which you can query the Class object for various information about the class such as its name, its superclass, and the names of the interfaces that it implements. This sample method gets and displays the class name of an object:
- String s="hello";
- System.*out.println(s.getClass().getName());*
- *Returns --- java.lang.String*

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 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)

```
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;
}
```

toString() – gives the string representation of the Object.

Can be overridden in any class to print the contents of the object.

hashcode() – returns the unique identification code of the object

```
Thread t1=new Thread("one");
Thread t2=new Thread("one");
System.out.println(t1.equals(t2));

String s1=new String("hello");
String s2=new String("hello");
System.out.println(s1.equals(s2));

System.out.println(s1.hashCode());
System.out.println(s2.hashCode());
System.out.println(t1.hashCode());
System.out.println(t2.hashCode());
```