AJOU UNIVERSITY

Dynamic Tomorrow Ajou

Principle of Inheritance in OOP

Rajib Paul (Ph.D.)
Department of Software and Computer Engineering
Ajou University

5.1. Class, Superclass, and Subclass

- we can create a new class from an existing class.
- The new class *inherits* features from an existing class.
- **Example**: **Managers** are in many aspects like employees.
- However, in other aspects they are different **because**
- Managers gets a ***bonus***
- Every manager is an employee, but not every employee is a manager.
- The Manager class is a *subclass* of an existing employee class.
- The **Employee** class is a *superclass*.

Example of Inheritance : Listing 5.1 inheritance/ManagerTest.java(1/3)

```
package inheritance;
public class ManagerTest
{
 public static void main(String[] args)
 {
  Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15); // create  object
   boss.setBonus(5000);
   Employee[] staff = new Employee[3];
   staff[0] = boss; // fill the staff array with Manager and Employee objects
   staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
   staff[2] = new Employee("Tommy Tester", 40000, 1990, 3, 15);
   for (Employee e : staff) // print out information about all Employee objects
    System.out.println("name=" + e.getName() + ",salary=" + e.getSalary());
 }
}
```

Example of Inheritance : Listing 5.2 inheritance/EmployeeTest.java(2/3)

```
package inheritance;
import java.time.*;
public class Employee
{
 private String name;
 private double salary;
 private LocalDate hireDay;
public Employee(String name, double
salary,  int year, int month, int day)
{
 this.name = name;
this.salary = salary;
```

```
public String getName()
{
    return name;
}
public double getSalary()
 {
   return salary;
 }
public LocalDate getHireDay()
{
  return hireDay;
 }
public void raiseSalary(double byPercent)
{
 double raise = salary * byPercent / 100;
  salary += raise;
 }
```

Example of Inheritance : Listing 5.3 inheritance/Manager.java(3/3)

```
package inheritance;
public class Manager extends Employee
{
  private double bonus;
  public Manager(String name, double salary, int year, int month, int day) {
   super(name, salary, year, month, day);
     bonus = 0;
  }
  public double getSalary() {
   double baseSalary = super.getSalary();
  return baseSalary + bonus;
  }
  public void setBonus(double b) {
     bonus = b;
  }
} // end of manager
```

Defining Subclasses

- Step 1: Use the "extends" keyword to define subclasses

```
public class Manager extends Employee
{
// added methods and fields unique to managers class
}
```

- **Step 2: Add fields and methods:**

```
public class Manager extends Employee {
  private double bonus;
  . . . ……;
  public void setBonus(double bonus)
    {
      this.bonus = bonus;
    }
}
```

- **Manager inherits** methods from **Employee** superclass:
  - getName,getHireday,
  - getSalary, raiseSalary
- **Manager** inherits fields from Employee superclass:
  - **salary** is present in all Manager objects.

Override methods and provide constructors in subclasses

- When an inherited method is not appropriate, we *override* it in the subclass

```
//First attempt to override
public class Manager extends Employee
{
  . . .
  public double getSalary() // overriding getSalary() method of Employee
  {
    return salary + bonus; // won't work
  }
}
```

- **Note: Subclass methods cannot access private superclass fields.**

Override methods and provide constructors in subclasses

```
// Second  attempt to override
public double getSalary()
  {
    return getSalary() + bonus;  // still won't work due to recursive call
  }
// third  attempt : Use "super" keyword  to avoid recursive call
public double getSalary()
{
   return super.getSalary() + bonus;
}
```

Constructor of Subclass

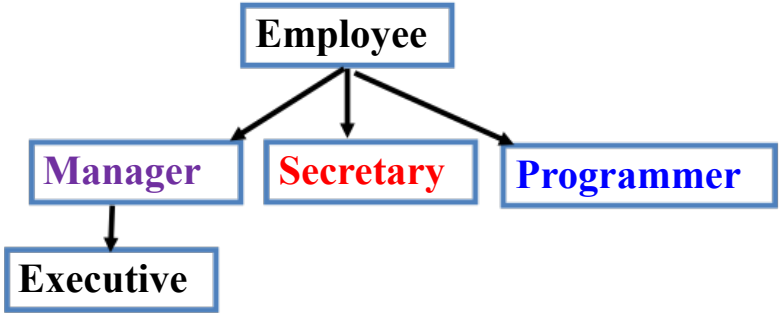
- Subclass constructor can invoke superclass constructor

```
public Manager(String name, double salary, int year, int month, int day, double bonus)
{
  super(name, salary, year, month, day); // call constructor of Employee Class
  this.bonus  = bonus;  //  name, salary, year, month, day are private to Employee class
}
```

- In **subclass** constructor, call using super must be the first statement.
- If there is **no** explicit call to superclass constructor, **no-argument** constructor of superclass is invoked using **super**();
- If **superclass** does not have a **no-arg** constructor, the compiler reports an error

5.1.1 Inheritance Hierarchies

*



Note 1: Inheritance is **"IS-A"** relationship between two classes. This relation is true from **bottom** to **top**, but not from **Top** to **bottom**

Note 2: Inheritance hierarchy has one or more layers

Note 3: The path from a subclass to its root superclass is called the **inheritance chain** of the subclass

Note 3: **Secretary** class has nothing to do with **Manager** **or** **Programmer** class

5.1.2 Polymorphism

a) **Static polymorphism**

- **one method name** refers to a method with many types of argument types
- It is related to method overloading
- It is related to **static binding**
- **The appropriate method is selected during compile time**

b) Dynamic Polymorphism

- **One object variable** like variable e (in **stack area**) can refer to many actual types (in heap area).
- **It** is related to **dynamic binding:**
- **The appropriate method is selected during runtime and it is not determined during compile time**

5.1.2 Polymorphism

- **What happens when a method is applied to an object (heap area) ?**

A) Static binding:

- **Assume we have class C with method f**
- **Assume method f() is overloaded method such as f(int),f(double), f(String).**
- **Assume x is reference variable of the class C as follows.**

  C x = new C( ); // x is an implicit parameter.
- **Assume the following method call.**

  x. f(double); // Compiler resolve name conflict by using the argument type.
- **Compiler resolves method overloading conflict(static binding)**
- **compiler enumerates all methods of class C whose name is f .**
- **Hence, compiler knows all possible candidates for the method to be invoked.**
- Note 1 : the name and parameter type list for a method is called the method's *Signature*
- Note 2: **return type** of a method is not part of a signature of a method.

5.1.2 Polymorphism

- **What happens when a method is applied to an object (heap area) ?**

B) Dynamic Binding

- **Assume we have class C with method f**
- **Assume method f() is overloaded method such as f(int),f(double), f(String).**
- **Assume x is reference variable of the class C as follows.**

  C x = new C( ); // x is an implicit parameter.
- **Assume class D is subclass of class C.**
- **Assume the actual object type in heap is type D.**

Case 1: If method f() is also defined in class D(method overriding), then the method f() in class D is invoked

Case 2: if method f() is not defined in class D(no method overriding), then the method f() in the supper class is searched and invoked

Comparison

- Static binding(Compiler)
- **Looks** at the **declared type** of the object and **method** name
- Selects method f which has correct match.
- Example, for x.f("Bye"), complier chooses f(String), not f(int).
- Hence, the **method call x.f() is** not depend on the **actual object type** of the implicit parameter x (in heap area)
- **Two Overloaded methods** should have identical signature
- **Two Overloaded methods can the** same or different return type

- dynamic binding(JVM)
- When the program runs, JVM selects the method that matches the actual type of the object to which an implicit parameter x refers
- **JVM resolves method overriding conflict(dynamic binding**
- The **override** methods in subclass should have identical signature the supper class method and its return type can be supper class type.
- Access modifier of overriding

Example of Dynamic Binding

- **Dynamic Polymorphism is related to the substitution rule of "IS-A" relationship**
- **This rule stats that we can use a subclass object when the program expect a super class object**

Example 1:

```
Employee e ; // e is super class variable
e = new Employee(…..);  // employ object expected
e = new Manager (…..);  // employ object expected, but manager object is assigned.
```

Note : we cannot assign a supper class reference to a subclass variable

Example :

```
Manager m = e ; // error because all employees are not managers
```

Example of Dymanic Polymorphism

- **Consider a mix of employees and managers:**

```
// construct a manager object
Employee[] staff = new Employee[3];
staff[0] = new Manager("Carl Cracker", 80000, 1987, 12, 5, 5000 );
staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
```

- **Print out everyone's salaries:**

```
for (Employee e : staff)
  System.out.println(e.getName() + " " + e.getSalary());
```

- **Which getName method gets called?**
- **There is only one: Employee.getName**
- **Question: Which getSalary method gets called?**
  Is Employee.getSalary or Manager.getSalary?
- **Answer: It depends on the actual type of e (reference variable e)**

Output:

Carl Cracker 85000.0
Harry Hacker 50000.0
Tony Tester 40000.0

Note : the declared type of e (stack area) is Employee . But the actual object(heap area) referenced by e is Employee or Manager . Hence, JVM decide the correct object at run time.

- Employee Table

- getName: Employee.getName();
- getSalary: Employee.getSalary();
- getHireDay: Employee.getHireDay();
- rasieSalaryName: Employee.raiseSalar()

Remarks:

Manager class **inherits 3** method from **Employee** class **without** override

b) Manager class inherits one method from **Employee** by overriding

c) **Manager** class add its own class

- Manager Table

- getName: Employee.getName();
- getSalary: **Manager**.getSalary();
- getHireDay: Employee.getHireDay();
- rasieSalaryName: Employee.raiseSalar()
- SetBonus: **Manager**.setbonus();

a) getName(),getHireDate(),raiseSalary()

b) getSalary();

c) setbonus();

Q. **Assume Employee e =new Employee();**

How to resolve the call: e.getsalary() ?

Example: Signature of Overriding method

- Caution: Argument types of **overriding** method must **match** *exactly*:

```
class Employee
{
    public void setBoss(Employee boss) {……}
     ..........
}
class Manager
{
  public void setBoss(Manager boss) {.....} //Error: different argument
   ..........
}
```

Example: return type of Overriding

Note 1: Use @Override annotation to make the compiler check:

@Override public void setBoss(Employee boss)

Note 2: Return type can be *covariant*:

```
class Employee
{
    public Employee getBoss() { ... }
  ..................................................................
}
class Manager
{
  public Manager getBoss() { ... }   // ok, covariant return type type
  ..................................................................
```

5.1.4. Preventing Inheritance: Final Classes and Methods

- **To if you want prevent another programmer from creating a new subclass from your class, declare your class with final key word.**

```
public final class Executive extends Manager
{
   . . .
}  // If a class is final, there is no dynamic polymorphism(dynamic binding)
```

- **We can also prevent method overriding using final key word**

```
public class Employee
{
  . . .
  public final String getName() { return name; }
```

5.1.5. Down Casting

```
Employee[] staff = new Employee[3];
staff[0] = new Manager("Carl Cracker", 80000, 1987, 12, 5, 5000 );
staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
```

Question 2: which of the following is correct to call the **setBonus(...) method of a Manager subclass ?**

a) With down casting(CORRECT)

```
Manager boss = (Manager) staff[0] ;
boss.setBonus(...); // setBonus method is defined only in the subclass
```

b) Without down casting(ERROr)

```
staff[0]. setBonus(...); // setBonus method is defined only in the subclass
```

5.1.5. Down Casting

```
Employee[] staff = new Employee[3];
staff[0] = new Manager("Carl Cracker", 80000, 1987, 12, 5, 5000 );
staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
```

Question 1: Can we call the **getSalary(...) method of a Manager subclass without down casting as follows ? (yes)**

```
staff[0]. getSalary(...) ;  // getSalary() is defined in both subclass &
                            //  superclass
```

5.1.5. "Instance of" Operator before Down Casting

- Manager boss=(manager) staff[0]; //CORRECT.
- Manager boss=(manager) staff[1]; // ERROR due to a ClassCastException .
- To avoid this use, test it using " **instanceOf"** operator as follows:

```
if (staff[1] instanceof Manager )
{
  boss = (Manager) staff[1];
  . . .
}
```

Note 1: Down casting is possible from superclass to subclass only.

Example:

```
String c = (String) staff[1]; // Compile-time error because String class is not subclass
                                 of Employee class
```

Note 2: There is no down casting between two subclasses .

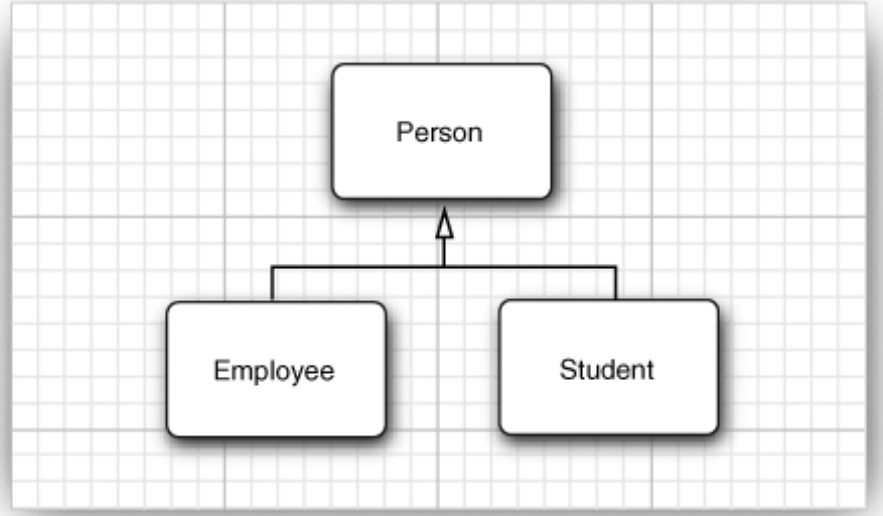
5.1.6. Abstract classes and abstract Methods

- When we move up in the **inheritance hierarchy**, classes become more general (abstract)
- **Example**: we can modify Employee class hierarchy by adding a Person as a **superclass** of an **Employee** class
- **Why we need a high level of abstraction**?
- To factor out common attributes such as name and common methods like **getName**().
- It is also used to **factor out common abstract** methods like **getDescription()** to return a **brief description** of a person, such as

a) an employee with a salary of $50,000.00

b) a student majoring in computer science

5.1.6. Abstract classes and abstract



5.1.6. Abstract classes and abstract Methods

- However, when factoring **out a common** class, it is difficult to implement

a common method in the super class ;

- Example: The **Person** class knows **nothing** about the salary of a person and the major of a person. It only knows the name of the person.
- **Approach 1: The method returns empty String**

```
String Person.getDescription():
  {
     String s = " ";
     return s ;
  }
```

Approach 2: by declaring abstract class without providing an implementation:

Syntax:

```
public abstract String getDescription();
```

5.1.6. Abstract classes and abstract Methods

Note: abstract classes to have fields, constructors, and concrete methods;

```
public abstract class Person
{
 private String name; // field
 public Person(String n) // constructor
 {
   name = n;
  }
 public String getName()
 {
 return name;  // concrete method
 }
  public abstract String getDescription(); //no implementation
}
```

Note 1: abstract methods are place holders
Note 2: In addition to abstract methods,
abstract classes to have , fields, constructors, and concrete methods
Note 3: We **cannot** create an instance of an abstract class in heap area.
Note 4: we can create object variable of abstract class on stack area

```
Person p1 = new Person(« kim"); // Error( note 3)
Person p2 = new Student(« kim", « maths"); // Ok(note 4)
```

Note: Abstract class must have direct or indirect concrete subclass

Example: Listing 5.4 AbstractClasses/PersonTest.java(1/4)

```
package abstractClasses;
public class PersonTest
{
  public static void main(String[] args)
  {
    Person[] people = new Person[2];
    // fill the people array with Student and Employee objects
    people[0] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
    people[1] = new Student("Maria Morris", "computer science");
    // print out names and descriptions of all Person objects
    for (Person p : people)
      System.out.println(p.getName() + ", " + p.getDescription());
  }
}
```

Example: Listing 5.5 AbstractClasses/Person.java(2/4)

```
package abstractClasses;
public abstract class Person
{
  public abstract String getDescription();
  private String name;
  public Person(String name)
  {
     this.name = name;
  }
  public String getName()
  {
     return name;
  }
```

Example: Listing 5.6 AbstractClasses/Employee.java(3/4)

```
package abstractClasses;
import java.time.*;
public class Employee extends Person
{
  private double salary;
  private LocalDate hireDay;
  public Employee(String name, double salary,
   int year, int month, int day)
  {
    super(name);
    this.salary = salary;
    hireDay = LocalDate.of(year, month, day);
  }
  public double getSalary()
  {
   return salary;
  }
  public LocalDate getHireDay()
  {
   return hireDay;
  }
  public String getDescription()
  {
   return String.format
   ("an employee with a salary of $%.2f", salary);
  }
  public void raiseSalary(double byPercent)
  {
  double raise = salary * byPercent / 100;
   salary += raise;
  }

} // End of employee
```

Example: Listing 5.7 AbstractClasses/student.java(4/4)

```
package abstractClasses;
public class Student extends Person
{
  private String major;
  public Student(String name, String major)
  {
    // pass n to superclass constructor
    super(name);
    this.major = major;
  }
  public String getDescription()
  {
    return "a student majoring in " + major;
  }
}
```

5.1.7. Protected Access Modifier

- **Note 1: fields in a class are best tagged as private**
- **Note 2**: Methods in a class are usually tagged as public.
- **Note 3**: Any **field or method** declared private **will not be visible** to other classes
- **Note 4**: A subclass cannot access the **private fields** of its superclass.
- **Note 5** : If we use **protected** modifier, subclass methods access a superclass field.
- **For example**, if the **superclass Employee** declares the **hireDay** field as **protected** instead of **private**, then the **Manager** methods can access it directly.

5.1.7. Protected Access Modifier

- **Example:**

```
public class Employee
{
  protected double salary;
}
```

- A **Manager**(subclass) method can access the salary field of Employee class
- But only inside the Manager instances(objects) , and **not inside** other **Employee** objects(instances)
- **Caution 1:** Protected **features** are visible to all subclasses and to all other classes in the same package .
- **Caution 2:** Anyone can extend a class with protected modifier.
- Hence, **protected modifier** is against the spirit of data encapsulation

5.1.7. Protected Access Modifier

- A **summary** of the four access modifiers in Java that control visibility :

1. Visible to the class only (private).
2. Visible to the world (public).
3. Visible to the package and all subclasses (protected).
4. Visible to the package—the default(no modifiers are needed).

5.2. "Object" class : The Root of class hierarchy in Java

- The "**Object**" class is a superclass of all Java classes (**java.lang.Object**).
- **Every class** has **Object** as a **superclass directly or indirectly by default**.

// Object is a super class explicitly

- public **class** Employee **extends** Object ;

// Object is a super class implicitly

public **class** Employee ;

// Object is a super class indirectly

- public **class Employee** **public class Manger extends Employee**;

5.2. "Object" class : The Root of class hierarchy in Java

- We can use a variable of type "**Object**" class to refer to objects of any class type (**principle of polymorphism**)

Example:

```
Object obj1 = new Employee("Harry Hacker", 35000);
Object obj2 = new int[10];
```

Note 1: in Java, only primitive types(int, double, etc.) **are not** objects.

Note 2: All array types are class types that extend the "**Object"** class.

Note 3 : Since a variable of type **Object** is used as a generic holder, to do specific operation , we need to casting

Example: Employee **e** = (Employee) obj1 ;

5.2. "Object" class : The Root of class hierarchy in Java

- The "**Object**" class has **no** field to be inherited by all other classes
- However, it has **11 methods** that are **inherited** or **overridden** by **other classes.**
- Public final Class<?> getClass(); // **1**
- public int hashCode(); // **2**
- public boolean equals(Object obj) // 3
- public String toString() // 4
- protected Object clone() throws CloneNotSupportedException // **5**
- protected void finalize() throws Throwable // 6
- **5** methods related to Mutitreded programming ();
- **Totally 11 methods**.

5.2.1. The equals() Method of "Object" class

- equals() method in "**Object**" class tests whether two **object references** are identical.
- We have to override equals() method to compare the same **state** of two objects.
- **Example: Consider two Employee objects by comparing their fields as follows.**

```
Class Employee  // this class extends Object class implicitly
{
public boolean equals(Object otherObject) // overriding
{
L1.  if (this == otherObject) return true;
L2.   if (otherObject == null) return false;
L3:  if (this.getClass() != otherObject.getClass() ) return false;
L4:   Employee other = (Employee) otherObject;  // downcasting
L5:  return  name. equals(other.name)&& salary == other.salary && hireDay.equals(other.hireDay);
}
}
```

5.2.1. The equals() Method of "Object" class..

- If **name** or hireDay are **null**, How to compare them?
- Solution: invoke "Objects.equals(a,b) " method.
- This method returns true if both arguments a and b are null;
- It **returns** false if only one is null.
- Otherwise, modify the **Line 5** of the previous code as follows:

```
return Objects.equals(name, other.name)
&& salary == other.salary
 && Object.equals(hireDay, other.hireDay);
```

How to override equals() method in a Subclass

- **First,** invoke equals() method on supper class: super.equals()
- If it **returns true**, then compare instance fields of a subclass.

```
public class Manager extends Employee
{
  . . ……
  public boolean equals(Object otherObject)
  {   // to check  that "this" and other belong to the same class
    if ( ! super.equals(otherObject) ) return false;
    Manager other = (Manager) otherObject;
    return this.bonus == other.bonus; // compare fields
  }
}
```

5.2.2 Equality Testing and Inheritance

Q1. If **implicit** and **explicit** parameters belong to the subclass and supper class, how should the equals() method behave ?

Q2. Should an Employee equal to a Manager ?

- In Java, the **equals()** method has the following properties

1) **It is Reflexive**: **x.equals(x)** return true for any **non-null reference** x;

2) **It is symmetric : x.equals(y)** return true iff **y.equals(x)** return true

3) **transitive** : if **x.equals(y)** return true, **y.equals(z)** return true, then **x.equals(z)** return true

- **Note** : It is **hard** to do with mixed types like **Manager**(m) and **Employee**(e)
- By symmetry, **m.equals(e)** and **e.equals(m)** must **return** the same value.
- Hence, the meaning of **equals** must be **fixed** in the superclass to avoid this problem
- Note: **The standard Java library contains over 150** implementations of equals() method

Example: Equality Testing and Inheritance continued…

```
public class Person  // Steps  to override  equals() method of " Object" root class.
{
1.  public final boolean equals(Object otherObject)  // explicit parameter
    {
2.    if (this == otherObject) return true; // fcompare implicit and explicit parameters
1.          if (otherObject == null ) return false ; // check whether explicit parameter is null
4.    if ( ! (otherObject instanceOf  Person) ) return false; // compare  parent and child
5.    Person other = (Person) otherObject; // cast to  a variable our type
6.    return this.id == other.id;  // compare each filed
    }
}
// Note:  if line 1 is replaced by  the foolowing line, it is wrong
  public boolean equals (Employee other);  // Error  signature is different from parent class
```

4.2.3. The hashCode Method

- A **Hash code** is an integer derived from an **object**.
- Hash codes should be scrambled : If x and y are distinct objects,

then there is a high probability that x.hashCode () and **y.hashCode**() are different.

- The **hashCode()** method is implemented in the "**Object"** class using **object's memory address**
- **Hence**, every object has a default hash code **by inheritance**

Example: String class overrides **Hashcode()** using the following algorithm:

```
int hash = 0;
for (int i = 0; i < length(); i++)
    hash = 31 * hash + charAt(i);  // public char charAt (int index);
        // Sample out put
```

String	HashCode(derived from content of the string)
"Hello".hashCode()	69609650,
"Harry".hashCode()	69496448.

Note: StringBuilder class(java.lang.StringBuilder) did not override the hashcode() of **Object Class**

Example 2: The hashCode() Method of Employee class.

```
L1.  public class Employee
L2   {
     . . ……..
L3.  public int hashCode()
L4   {
L5.    return Objects.hash(name, salary, hireDay); //combine hash codes of the  fields(Java 7)
     }
} //Note: line 5 can be replaced by the following codes
public int hashCode()
{
 returun Objects.hashCode(name) + new Double(salary).hashCode()+ Objects.hashCode(hireDay);
 // return  name.hashCode+ new Double(salary).hashCode()+ hireDay.hashCode();  // before java 7
}
```

4.2.3. The hashCode Method Con'd

Rules of hashCode

a) Hash codes must be consistent. If x and y are equal objects , then their hash codes must be equal.

b) **Object.hashCode()** is derived from the memory location of the object on heap memory

c) Our definitions of equality and hashCode must be compatible:

If **x.equals(y)** is true, then **x.hashCode**() must **return** the same value as **y.hashCode().**

- For example, if we define **Employee.equals** to compare employee **IDs**, then the hashCode() method needs to hash the IDs, not employee names.
- Hence, if we **override equals**() method , we must also **override hashCode()** method.

d) Combine the **hash codes** of the fields that the equals method compares

5.2.4. The toString() Method

- public String toString();
- The toString() **method returns** a string representation of an object in heap memory.
- toString method() is ubiquitous because when we concatenate a string and an object, java invoke toString() method on the object automatically as shown below.

a) "Center: " + p ; // compiler call p.toString() automatically

b) " "+ P ; // this is similar to p.toString() ;

c) **System.out.println**(p); // this is similar to System.out.println(p.toString ());

Note: The "Object" class defines the toString() to print the class name and the hash code of the object (**Object.toString**();)

For example, the call System.out.println(System.out), display the following:

java.io.PrintStream@2f6684 **because PrintStream class did not override it.**

Note: we must Override toString() to get meaningful meaning for our own class like

Example 1: The toString Method

- Note: we must **override** **toString()** to get meaningful display for our own class like
- **Example**: to display the out: **java.awt.Point[x=10,y=20]** , write the following code

```
public class Point // write the toString() method of Point class.
{
 ………………………………………………………………………………………
   public String toString()  // override it
   {
      return  " java.awt.Point[x =" + x + ", y= " + y + "] ";
   }
}   output : java.awt.Point[x=10,y=20]
```

Example 2: Inheritance and the toString() Method

a) In Employee class, we can override toString() as follows.

```
public String toString()
{
  return getClass().getName()
     + "[name=" + name + ",salary=" + salary + ",hireDay=" + hireDay + "]";
}
```

b) In Manager subclass:

```
 public String toString()
{
  return super.toString() + "[bonus=" + bonus + "]";
}
```

c) Output format of manager class:

```
    Manager[name=...,salary=...,hireDay=...][bonus=...]
```

Example: Listing 5.8 Equals/EqualsTest.java(1)

```
package equals;
public class EqualsTest
{
public static void main(String[] args)
{
Employee alice1 = new Employee
("Alice Adams", 75000, 1987, 12, 15);
Employee alice2 = alice1;
Employee alice3 = new Employee
("Alice Adams", 75000, 1987, 12, 15);
Employee bob = new Employee
 ("Bob Brandson", 50000, 1989, 10, 1);
```

```
System.out.println("alice1 == alice2:"
+ (alice1 == alice2));

System.out.println("alice1 == alice3:"
+ (alice1 == alice3));

System.out.println("alice1.equals(alice3):"
+ alice1.equals(alice3));

System.out.println("alice1.equals(bob):"
+ alice1.equals(bob));

 System.out.println("bob.toString(): " + bob);
```

Example: Listing 5.8

Equals/EqualsTest.java(2)

```
Manager carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
boss.setBonus(5000);
System.out.println("boss.toString(): " + boss);
System.out.println("carl.equals(boss): " + carl.equals(boss));
 System.out.println("alice1.hashCode(): " + alice1.hashCode());
System.out.println("alice3.hashCode(): " + alice3.hashCode());
System.out.println("bob.hashCode(): " + bob.hashCode());
System.out.println("carl.hashCode(): " + carl.hashCode());

} // end of main()
}//end of EqualsTest class
```

Example: Listing 5.9

Equals/Employee.java(1)

```
package equals;
import java.time.*;
import java.util.Objects;
public class Employee
{
private String name;
private double salary;
private LocalDate hireDay;

public Employee(String name, double salary, int year, int month, int day)
{
this.name = name;
this.salary = salary;
hireDay = LocalDate.of(year, month, day);
}
```

```
public String getName()
{
 return name;
}
 public double getSalary()
 {
   return salary;
 }
 public LocalDate getHireDay()
 {
   return hireDay;
 }
public void raiseSalary(double byPercent)
{
 double raise = salary * byPercent / 100;
  salary += raise;
}
```

Example: Listing 5.9

Equals/Employee.java(2)

```
public boolean equals(Object otherObject)
{
 // a quick test to
 if (this == otherObject) return true;
 if (otherObject == null) return false;
 // to test class match
 if (getClass() != otherObject.getClass())
 return false;
 // now otherObject is a non-null Employee
 Employee other = (Employee) otherObject;      // test  field by field
 return Objects.equals(name, other.name)
 && salary == other.salary
 && Objects.equals(hireDay, other.hireDay);
}
```

```
public int hashCode()
{
 return Objects.hash
 (name, salary, hireDay);
}
public String toString()
{
return getClass().getName()
 + "[name="
 + name
 + ",salary="
 + salary
 + ",hireDay="
 + hireDay
 + "]";
```

Example: Listing 5.10

Equals/Manager.java

```
package equals;
public class Manager extends Employee
{
private double bonus;
 public Manager(String name, double salary, int year,   int month, int
day)
  {
    super(name, salary, year, month, day);
    bonus = 0;
  }
public double getSalary()
```

```
public boolean equals(Object otherObject)
{
 if (!super.equals(otherObject)) return false;
   Manager other = (Manager) otherObject;
// super.equals compare class of  this and other
  return bonus == other.bonus;
}
public int hashCode()
{
    return super.hashCode() + 17 *
      new Double(bonus).hashCode();
}
public String toString()
{
 return super.toString()
```