

# ***Advanced Programming***

## **CSE 201**

**Instructor: Sambuddho**

(Semester: Monsoon 2024)

Week 3 – Inheritance

# Basics of Inheritance (in Java)

- Closely connected to polymorphism.
- Def\_1: Referencing many related objects as one generic type.
- Def\_2: Reference of 'parent' class utilizing attributes and methods of a 'child' class, depending on which one it is referencing.

# Slide Acknowledgement

CS15, Brown University

# Basics of Inheritance (in Java)

- Closely connected to polymorphism.
- Def\_1: Referencing many related objects as one generic type.
- Def\_2: Reference of 'parent' class utilizing attributes and methods of a 'child' class, depending on which one it is referencing.

# Spot the Similarities



- What are the similarities between a convertible and a sedan?
- What are the differences?

# Convertibles vs. Sedans

## Convertible

- Top Down Roof  
(Retractable Roof)

## Sedan

- Fixed Roof

- Drive
- Brake
- Play radio
- Lock/unlock  
doors
- Turn off/on  
turn engine

# Can we model this in code?

- In some cases, objects can be very closely related to each other
  - Convertibles and sedans drive the same way
  - Flip phones and smartphones call the same way
- Imagine we have an **Convertible** and a **Sedan** class
  - Can we enumerate their similarities in one place?
  - How do we portray their relationship through code?

## Convertible

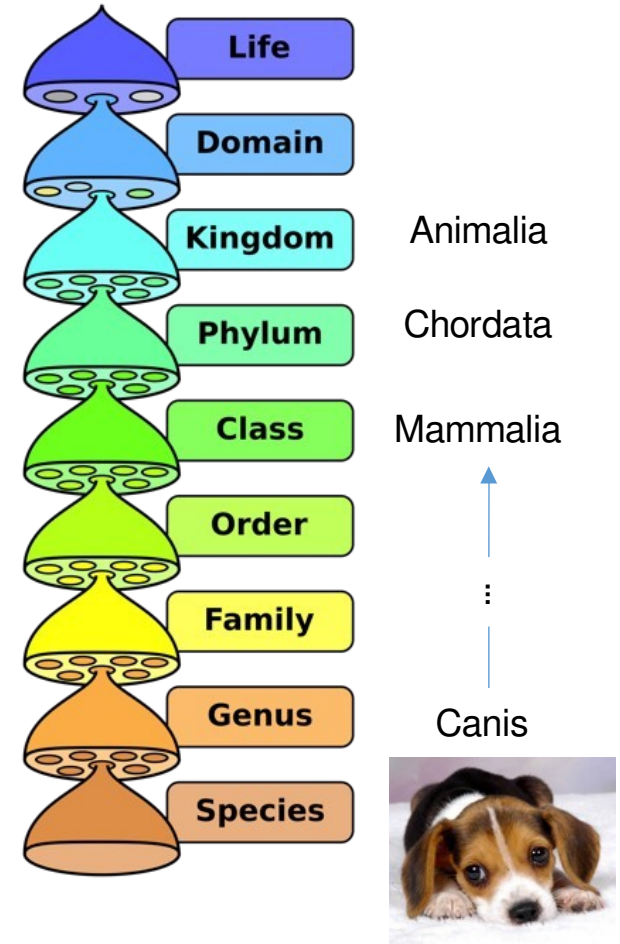
- putTopDown()
- turnOnEngine()
- turnOffEngine()
- drive()

## Sedan

- parkInCompactSpace ()
- turnOnEngine()
- turnOffEngine()
- drive()

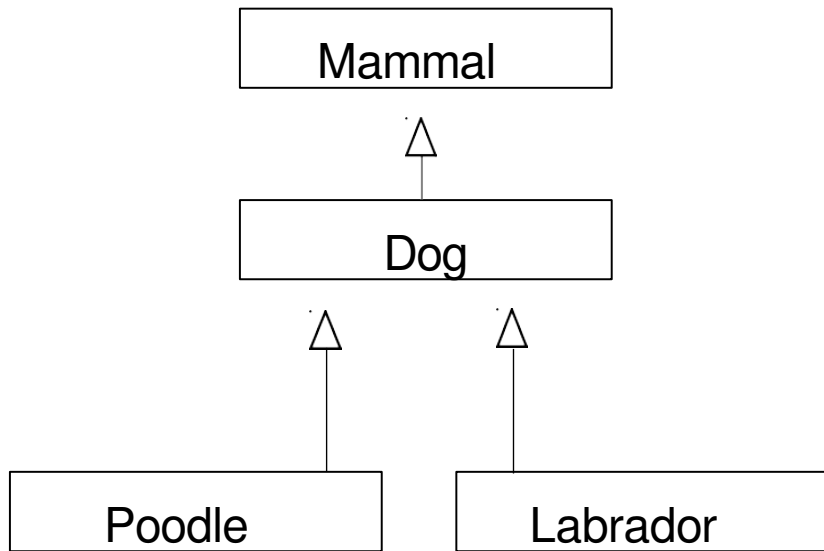
# Inheritance

- In OOP, inheritance is a way of modeling very similar classes
- **Inheritance** models an “**is-a**” relationship
  - A **sedan** “is a” **car**
  - A **dog** “is a” **mammal**
- Remember: **Interfaces** model an “**acts-as**” relationship
- You’ve probably seen inheritance before!
  - Taxonomy from biology class



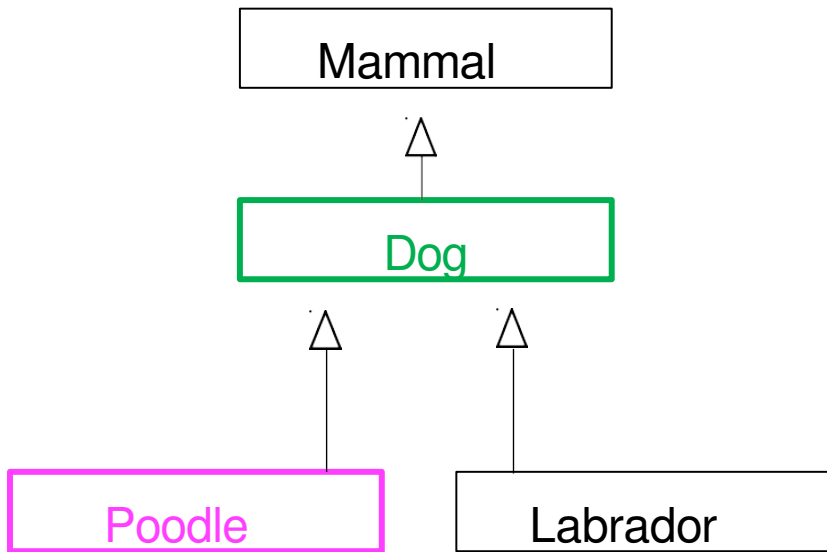


# Modeling Inheritance (1/2)



- This is an inheritance diagram
  - Each box represents a class
- A Poodle “is-a” Dog, a Dog “is-a” Mammal
  - Transitively, a Poodle is a Mammal
- “Inherits from” = “is-a”
  - Poodle inherits from Dog
  - Dog inherits from Mammal
- This relationship is not bidirectional
  - A Poodle is a Dog, but not every Dog is a Poodle (could be a Labrador, a German Shepard, etc)

# Modeling Inheritance (2/2)



- **Superclass/parent/base**: A class that is inherited from
- **Subclass/child/derived**: A class that inherits from another
- “A **Poodle** is a **Dog**”
  - **Poodle** is the **subclass**
  - **Dog** is the **superclass**
- A class can be both a **superclass** and a **subclass**
  - Ex. Dog
- In Java you can only inherit from one superclass (no multiple inheritance)
  - Other languages, like C++, allow for multiple

inheritance, but too easy to mess up

# Motivations for Inheritance

- A subclass inherits all of its parent's **public** and **protected** capabilities
  - If Car defines drive(), Convertible inherits drive() from Car and drives the same way. This holds true for all of Convertible's subclasses as well
- Inheritance and Interfaces both legislate class's behavior, although in very different ways
  - Interfaces allow the compiler to enforce method implementation
    - An implementing class will have all capabilities outlined in an interface
  - Inheritance assures the compiler that all **subclasses** of a **superclass** will have the superclass's public capabilities without having to respecify code – methods are inherited
    - A Convertible knows how to drive and drives the same way as Car because of inherited code
- Benefit of inheritance
  - Code reuse
    - If drive() is defined in Car, Convertible doesn't need to redefine it! Code is inherited Only need to
    - implement what is different, i.e. what makes Convertible special

# Superclasses vs Subclasses

- A superclass factors out commonalities among its subclasses

- describes everything that all subclasses have in common.

**NOTE: Java classes can have only one parent (super) class, unlike C++.**

- A subclass differentiates/specializes its superclass by:

- **adding new methods:**

- 

- **Overriding inherited methods:**

-

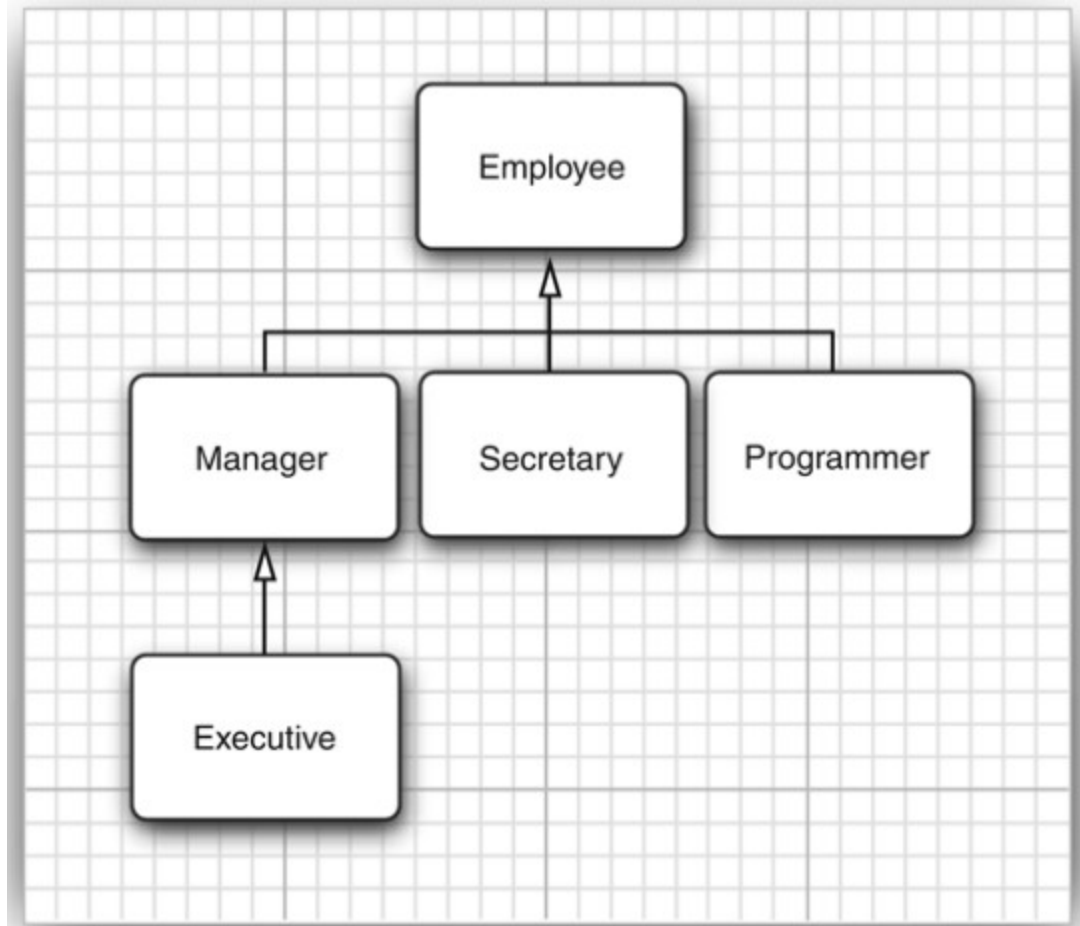
# Method Overriding

- Override parent class 'public' methods (not private).
  - Child can access 'public' methods and attributes of parent class.
  - Using parent class attributes – super.<Attib\_name>
  - Using parent class methods – super.f()
  - Calling parent class constructor

```
<subclass constructor>{  
    super(<arguments>)  
}
```

```
public class Manager extends Employee  
{  
    private double bonus;  
    . . .  
    public void setBonus(double bonus)  
    {  
        this.bonus = bonus;  
    }  
}  
  
public double getSalary()  
{  
    double baseSalary = super.getSalary();  
    return baseSalary + bonus;  
}
```

# Inheritance and Polymorphism



- Polymorphism is an incredibly powerful tool.
- Allows for generic programming.
- Treat multiple classes as their generic type while still allowing specific method implementations to be executed.
- Polymorphism+Inheritance is strong generic coding

# Inheritance and Polymorphism

```
1 package inheritance;
2
3 import java.time.*;
4
5 public class Employee
6 {
7     private String name;
8     private double salary;
9     private LocalDate hireDay;
10
11     public Employee(String name, double salary, int year, int month, int day)
12     {
13         this.name = name;
14         this.salary = salary;
```

```
15         hireDay = LocalDate.of(year, month, day);
16     }
17
18     public String getName()
19     {
20         return name;
21     }
22
23     public double getSalary()
24     {
25         return salary;
26     }
27
28     public LocalDate getHireDay()
29     {
30         return hireDay;
31     }
32
33     public void raiseSalary(double byPercent)
34     {
35         double raise = salary * byPercent / 100;
36         salary += raise;
37     }
38 }
```

# Inheritance and Polymorphism

```
1 package inheritance;
2
3 public class Manager extends Employee
4 {
5     private double bonus;
6
7     /**
8      * @param name the employee's name
9      * @param salary the salary
10     * @param year the hire year
11     * @param month the hire month
12     * @param day the hire day
13     */
14     public Manager(String name, double salary, int year, int month, int day)
15     {
16         super(name, salary, year, month, day);
17         bonus = 0;
18     }
```

```
20     public double getSalary()
21     {
22         double baseSalary = super.getSalary();
23         return baseSalary + bonus;
24     }
25
26     public void setBonus(double b)
27     {
28         bonus = b;
29     }
30 }
```

---



# Inheritance and Polymorphism

```
1 package inheritance;
2
3 import java.time.*;
4
5 public class Employee
6 {
7     private String name;
8     private double salary;
9     private LocalDate hireDay;
10
11     public Employee(String name, double salary, int year, int month, int day)
12     {
13         this.name = name;
14         this.salary = salary;
```

```
15         hireDay = LocalDate.of(year, month, day);
16     }
17
18     public String getName()
19     {
20         return name;
21     }
22
23     public double getSalary()
24     {
25         return salary;
26     }
27
28     public LocalDate getHireDay()
29     {
30         return hireDay;
31     }
32
33     public void raiseSalary(double byPercent)
34     {
35         double raise = salary * byPercent / 100;
36         salary += raise;
37     }
38 }
```

# Inheritance and Polymorphism

```
Employee e;  
e = new Employee(. . .); // Employee object expected  
e = new Manager(. . .); // OK, Manager can be used as well
```

```
Manager boss = new Manager(. . .);  
Employee[] staff = new Employee[3];  
staff[0] = boss;
```

```
boss.setBonus(5000); // OK
```

Why ?

```
staff[0].setBonus(5000); // ERROR
```

# Rules for Method Calls

- 1. Compiler looks at types of objects and method names, determines the appropriate method based on the return type. The compiler also resolves.
- 2. Argument types.
- 3. Method types – 'private' , 'static' , 'final'.
- 4. Polymorphic associations – dynamic binding at runtime.

# Preventing Inheritance

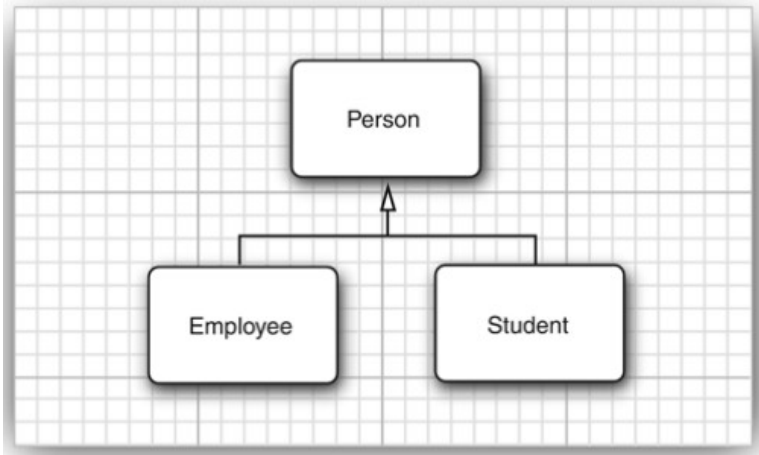
- 1. What all a child class inherits:
  - Public objects and methods.
  - Private of parent is never inherited.
  - Parent constructors (by default public, like everything else in Java).
- Preventing inheritance:

```
public final class Executive extends Manager
{
    . . .
}
```

- You could also prevent specific methods
- from being inherited *without* making them private.

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

# Abstract Classes



```
public abstract class Person
{
    . . .
    public abstract String getDescription();
}
```

```
1 package abstractClasses;
2
3 public abstract class Person
4 {
5     public abstract String getDescription();
6     private String name;
7
8     public Person(String name)
9     {
10         this.name = name;
11     }
12
13     public String getName()
14     {
15         return name;
16     }
17 }
```

---

```
1 package abstractClasses;
2
3 import java.time.*;
4
5 public class Employee extends Person
6 {
7     private double salary;
8     private LocalDate hireDay;
9
10    public Employee(String name, double salary, int year, int month, int day)
11    {
12        super(name);
13        this.salary = salary;
14        hireDay = LocalDate.of(year, month, day);
15    }
16
17    public double getSalary()
18    {
19        return salary;
20    }
21
22    public LocalDate getHireDay()
23    {
24        return hireDay;
25    }
26
27    public String getDescription()
28    {
29        return String.format("an employee with a salary of $%.2f", salary);
30    }
31
32    public void raiseSalary(double byPercent)
33    {
34        double raise = salary * byPercent / 100;
35        salary += raise;
36    }
37 }
```

# Protected Access

- 'Protected' methods and objects cannot be accessed by objects of the class much like 'private'.
- 'Protected' methods and objects/variable can be accessed by child classes.

# Object – the Cosmic Super Class

- Object – parent of all classes (implicit); not of primitive types – int, char, byte etc.

```
Object obj = new Employee("Harry Hacker", 35000);
```

- Can be: `Employee e = (Employee) obj;`
- Arrays – regardless they are of primitive types or of classes, are of type Object.

```
Employee[] staff = new Employee[10];  
obj = staff; // OK  
obj = new int[10]; // OK
```

# Object – equals() method

- Test if two objects 'equal' or same. Could mean many things – reference to same object (default), same value etc.
- Object class defines equals(). Other classes, can extend it with their own definition.

```
public class Employee
{
    . . .
    public boolean equals(Object otherObject)
    {
        // a quick test to see if the objects are identical
        if (this == otherObject) return true;

        // must return false if the explicit parameter is null
        if (otherObject == null) return false;

        // if the classes don't match, they can't be equal
        if (getClass() != otherObject.getClass())
            return false;

        // now we know otherObject is a non-null Employee
```

```
        Employee other = (Employee) otherObject;

        // test whether the fields have identical values
        return name.equals(other.name)
            && salary == other.salary
            && hireDay.equals(other.hireDay);
    }
}
```



# Object – toString() method

- Used to for printing a string equivalent information of the class. E.g. `System.out.println(s);` the return type is `String` and the name is `toString()`
- ```
class MyClass{
```
- ```
...
```
- ```
public String toString(){
```
- ```
    return "XYZ";
```
- ```
}
```
- ```
}
```

# Object Wrappers and Autoboxing

- Object corresponding to a primitive type [Integer, Long, Float, Double, Short, Byte and Boolean].
- Their objects are immutable – once a wrapper object has been created their values cannot be changed.
- They are 'final' and cannot be inherited.
- Integer[] list;
- list = new Integer[500];
- list[0] = 1;
- list[1] = 1;
- list[i]++;
- But list[i] == list[j] fails.

# Object Wrappers and Autoboxing

- `list[i].intValue() == list[j].intValue();`

## `java.lang.Integer` 1.0

- `int intValue()`  
returns the value of this Integer object as an int (overrides the `intValue` method in the `Number` class).
- `static String toString(int i)`  
returns a new `String` object representing the number `i` in base 10.
- `static String toString(int i, int radix)`  
lets you return a representation of the number `i` in the base specified by the `radix` parameter.
- `static int parseInt(String s)`
- `static int parseInt(String s, int radix)`  
returns the integer whose digits are contained in the string `s`. The string must represent an integer in base 10 (for the first method) or in the base given by the `radix` parameter (for the second method).
- `static Integer valueOf(String s)`
- `static Integer valueOf(String s, int radix)`  
returns a new `Integer` object initialized to the integer whose digits are contained in the string `s`. The string must represent an integer in base 10 (for the first method) or in the base given by the `radix` parameter (for the second method).

## `java.text.NumberFormat` 1.1

- `Number parse(String s)`  
returns the numeric value, assuming the specified `String` represents a number.

# Enum classes

- Enumerated types – alternatives to constants.
- `public enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE }`

```
public enum Size
{

    SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

    private String abbreviation;

    private Size(String abbreviation) { this.abbreviation = abbreviation; }
    public String getAbbreviation() { return abbreviation; }
}
```

```
Size s = Enum.valueOf(Size.class, "SMALL");
```

# Generic ArrayLists

- ArrayList is a *generic class* with a *type parameter*.
- An example of polymorphism.
- Provides a dynamically growing (or shrinking) array of objects.
- `ArrayList<Integer> mylist = new ArrayList<Integer>();`
- or
- `ArrayList<Integer> mylist = new ArrayList<>();`

# Generic ArrayLists

`java.util.ArrayList<E>` 1.2

- `ArrayList<E>()`  
constructs an empty array list.
- `ArrayList<E>(int initialCapacity)`  
constructs an empty array list with the specified capacity.
- `boolean add(E obj)`  
appends obj at the end of the array list. Always returns true.
- `int size()`  
returns the number of elements currently stored in the array list. (Of course, this is never larger than the array list's capacity.)
- `void ensureCapacity(int capacity)`  
ensures that the array list has the capacity to store the given number of elements without reallocating its internal storage array.
- `void trimToSize()`  
reduces the storage capacity of the array list to its current size.

- Accessing ArrayList elements
- `mylist.get(index);`
- `mylist.set(index,val);`
- `ArrayList.toString()` already defined. Prints comma separated array element values.

# Interfaces

```
public class Employee implements Comparable<Employee>
{
    private String name;
    private double salary;

    public Employee(String name, double salary)
    {
        this.name = name;
        this.salary = salary;
    }

    public String getName()
    {
        return name;
    }

    public double getSalary()
    {
        return salary;
    }

    public void raiseSalary(double byPercent)
    {
        double raise = salary * byPercent / 100;
        salary += raise;
    }

    /**
     * Compares employees by salary
     * @param other another Employee object
     * @return a negative value if this employee has a lower salary than
     *         otherObject, 0 if the salaries are the same, a positive value otherwise
     */
    public int compareTo(Employee other)
    {
        return Double.compare(salary, other.salary);
    }
}
```

# Interfaces

```
x = new Comparable(. . .); // ERROR
```

```
Comparable x; // OK
```

```
x = new Employee(. . .); // OK provided Employee implements Comparable
```



# Interfaces – Single Inheritance with Multiple Interface Implementations.

- `public interface Comparable{`
- `public int compareTo(Object otherobject);`
- `}`
  
- `class Manager extends Employee implements Comparable {`
- `....`
- `}`



**Can be as many as the programmer feels like.**

# Interfaces –Default Methods

- *Somewhat* like a default constructor, but you cannot instantiate objects of interfaces!

```
public interface Comparable<T>
{
    default int compareTo(T other) { return 0; }
    // by default, all elements are the same
}
```

```
public interface Iterator<E>
{
    boolean hasNext();
    E next();
    default void remove() { throw new UnsupportedOperationException("remove"); }
    . . .
}
```

# Popular Use Case – Callbacks.

- Callback function frameworks use interfaces.
- Event listeners – need to implement these interfaces and interface functions.

```
java.awt.event

public interface ActionListener
{
    void actionPerformed(ActionEvent event);
}

class TimePrinter implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        System.out.println("At the tone, the time is "
            + Instant.ofEpochMilli(event.getWhen()));
        Toolkit.getDefaultToolkit().beep();
    }
}
```

- `Timer t = new Timer(1000,new TimePrinter());`

# Popular Use Case – Callbacks.

```
import java.awt.*;
import java.awt.event.*;
import java.time.*;
import javax.swing.*;

public class TimerTest
{
    public static void main(String[] args)
    {
        var listener = new TimePrinter();

        // construct a timer that calls the listener
        // once every second
        var timer = new Timer(1000, listener);
        timer.start();

        // keep program running until the user selects "OK"
        JOptionPane.showMessageDialog(null, "Quit program?");
        System.exit(0);
    }
}

class TimePrinter implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        System.out.println("At the tone, the time is "
            + Instant.ofEpochMilli(event.getWhen()));
        Toolkit.getDefaultToolkit().beep();
    }
}
```

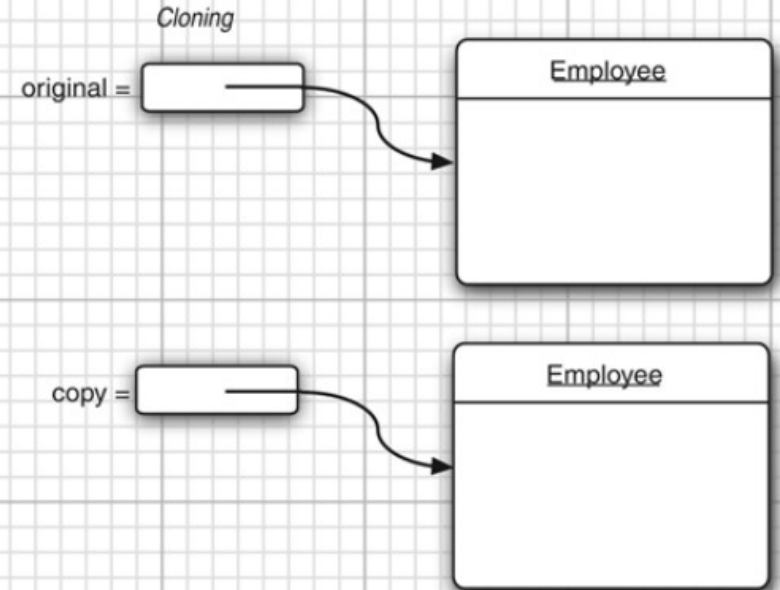
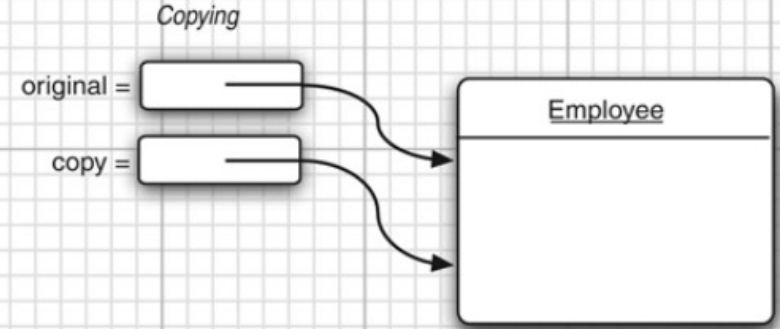
---

# Object Cloning

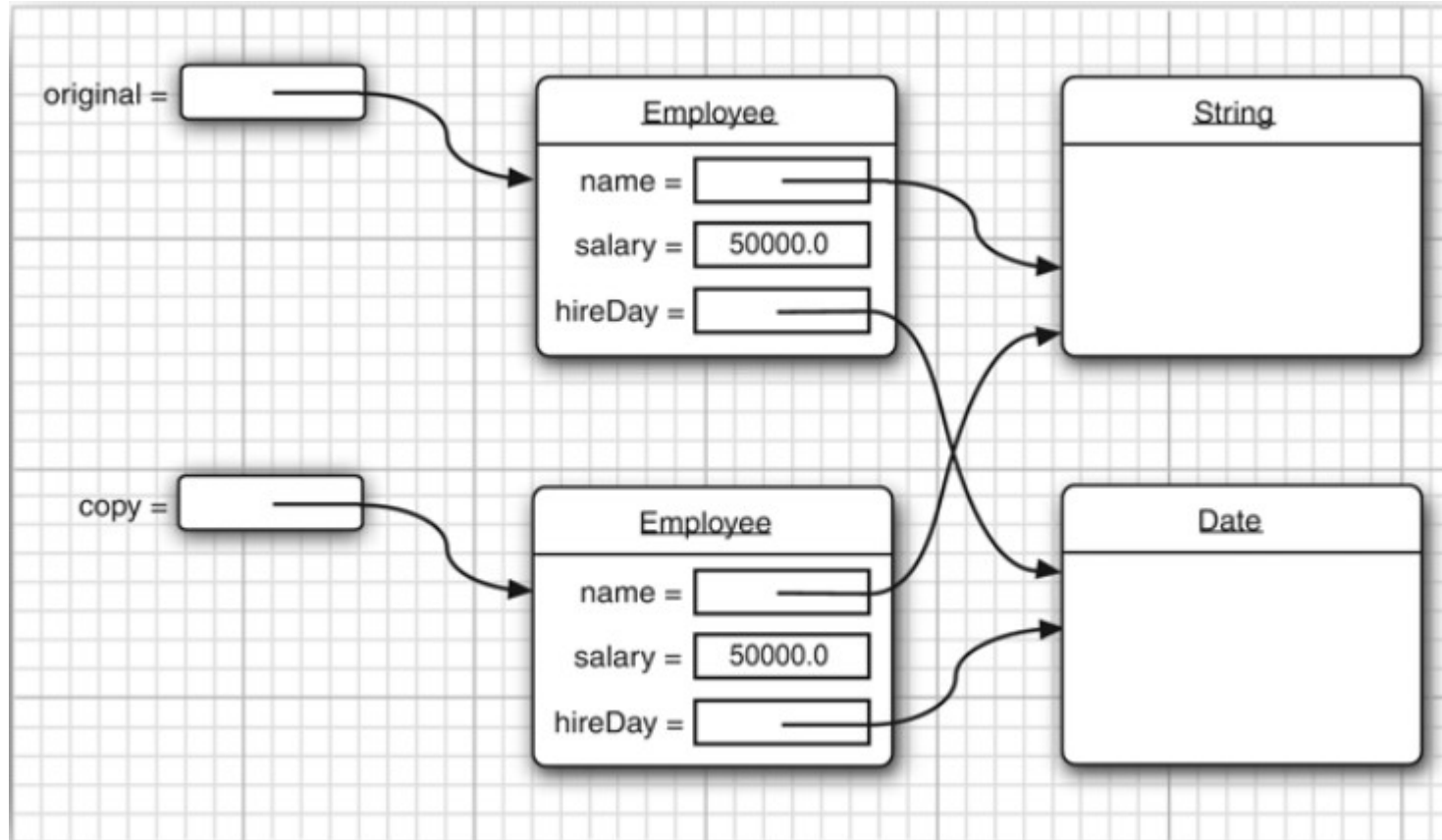
```
var original = new Employee("John Public", 50000);  
Employee copy = original;  
copy.raiseSalary(10); // oops--also changed original
```



```
Employee copy = original.clone();  
copy.raiseSalary(10); // OK--original unchanged
```



# Object Cloning – Default:Shallow Copy



# Object Cloning – Deep Copy: Implement Cloneable

```
class Employee implements Cloneable
{
    // public access, change return type
    public Employee clone() throws CloneNotSupportedException
    {
        return (Employee) super.clone();
    }
    . . .
}
```

```
class Employee implements Cloneable
{
    . . .
    public Employee clone() throws CloneNotSupportedException
    {
        // call Object.clone()
        Employee cloned = (Employee) super.clone();

        // clone mutable fields
        cloned.hireDay = (Date) hireDay.clone();

        return cloned;
    }
}
```