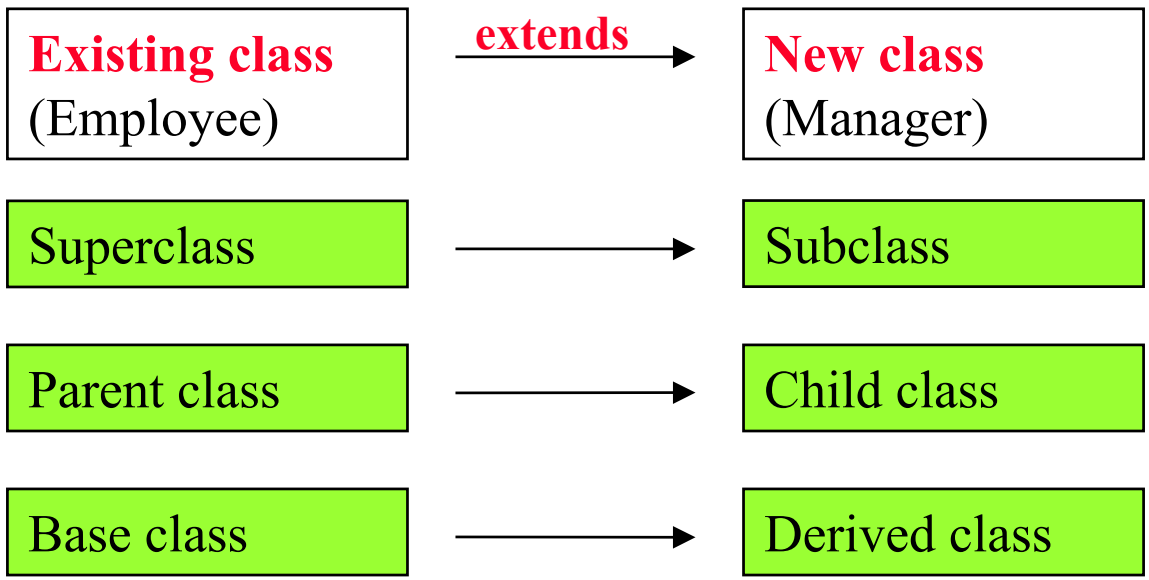
Chapter 5

- **First Steps with Inheritence**
- **Casting**
- **Abstract Classes**
- **More on `Object`: The Cosmic Superclass**
- **The Class `Class` (Run-time Type Identification)**
- **Interfaces**
- **Protected Access**
- **Design Hints for Inheritance**

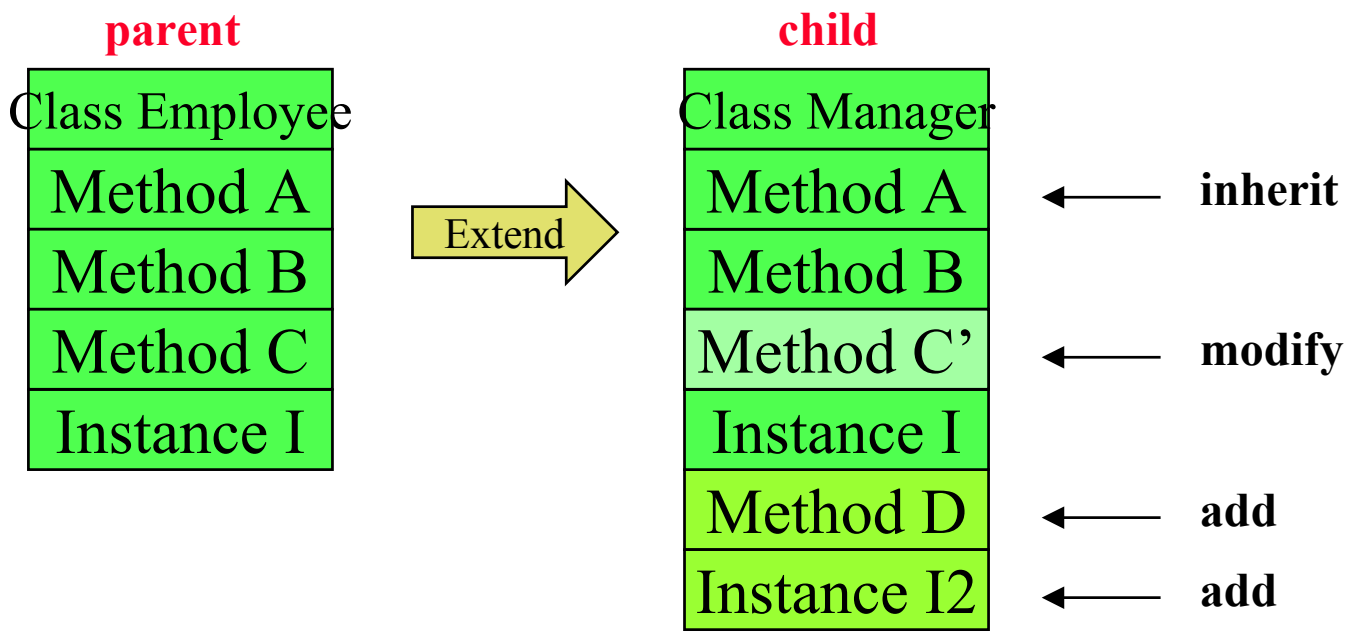
First Steps with Inheritence

- Let us modify the **`Employee`** class whereby managers are treated in a special way:
  - they have secretary
  - they earn extra 1/2% bonus for every year of service
- We define a **new class, `Manager`**, and add functionality, but we can retain some methods and **all** the instance fields of the **`Employee`** class.
- There is an obvious **„is-a"relationship** between **`Manager`** and **`Employee.`** Every manager **is an** employee: this is the hallmark of inheritance.



Inheritance

- Classes can be built on other classes. We say that a class that builts on another class `extends` it.
- The general concept of extending a base class is called `inheritance`.
- When you extend a base class, the new class initially has all the properties and functions of its `parent`. You can choose whether you want to modify any function of the parent. You can also supply new functions that apply to the `child class` only.
- **Instance fields can only be added in inheritance.**



```
class Manager extends Employee
{  public Manager(String n, double s, Day d)
   {    super(n, s, d);
        secretaryName = "";
   }
   public void raiseSalary(double byPercent)
   {    // add 1/2% bonus for every year of service
        Day today = new Day;
        double bonus = 0.5 * (today.getYear() - hireYear());
        super.raiseSalary(byPercent + bonus);
   }
   public string getSecretaryName()
   {    return secretaryName;
   }
   public void setSecretaryName(String name)
   {    secretaryName = name;
   }
   private String secretaryName;
}
```

Analyzing the `Manager` Class

- The **header** for the `Manager` class is a little different:

```
class Manager extends Employee
```

  The keyword **`extends`** indicates that you create a **subclass**.

- Next, notice the **constructor** for the `Manager` class:

```
public Manager(String n, double s, Day d)
{       super(n, s, d);
        secretaryName = "";
}
```

  The keyword **`super`** refers to the superclass. So the line

```
super(n, s, d);
```

  is a shorthand for "call the constructor of the Employee class". The reason is that **every constructor of a subclass is also responsible for constructing the data fields of the superclass**.

- The call to **`super`** must be the first line in the constructor of the subclass.
- **A subclass can have more instance fields than its superclass (but not less). The instance fields of the superclass should not be written in the subclass:**

```
private String secretaryName;
```

Analyzing the `Manager` Class (Cont.)

- **Many of the methods are not repeated**, they are **inherited** from the superclass (for example: `getName`). You need to indicate only the **differences** between the subclass and superclass. For example, we are adding accessor and mutator methods to handle the name of the secretary:

```
public string getSecretaryName()
{       return secretaryName;
}
public void setSecretaryName(String name)
{       secretaryName = name;
}
```

- Many times you have to **redefine** methods. For example:

```
public void raiseSalary(double byPercent)
{       // add 1/2% bonus for every year of service
        Day today = new Day;
        double bonus = 0.5 * (today.getYear() - hireYear());
        super.raiseSalary(byPercent + bonus);
}
```

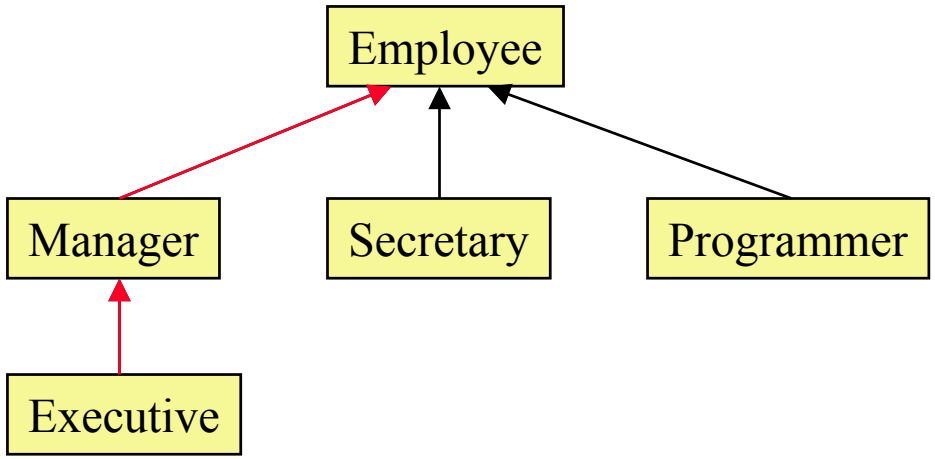
Use of the Employee and Manager Classes

- **Exercise: study the `ManagerTest` program.**

```
import java.util.*;
import corejava.*;
public class ManagerTest
{  public static void main(String[] args)
   {  Employee[] staff = new Employee[3];
      staff[0] = new Employee("Harry Hacker", 35000,
         new Day(1989,10,1));
      staff[1] = new Manager("Carl Cracker", 75000,
         new Day(1987,12,15));
      staff[2] = new Employee("Tony Tester", 38000,
         new Day(1990,3,15));
      int i;
      for (i = 0; i < 3; i++) staff[i].raiseSalary(5);
      for (i = 0; i < 3; i++) staff[i].print();
   }
}
```

Employee Inheritance Hierarchy

- Inheritance need not stop at deriving one layer of classes. We could have an **`Executive`** class that is derived from **`Manager`**. The collection of all classes extending from a common parent is called an **inheritance hierarchy** (see figure).



- The path from a particular class to its ancestors in the inheritance hierarchy is an **inheritance chain**.

Working with subclasses

- **There are two rules of using subclasses:**
  1. Any object of a subclass must be usable in place of its superclass' objects. Put in another way: **Subclass objects are usable in any code that uses superclass.**

     If this is not true, do not use inheritance!
  2. **Subclass objects have at least as many instance fields as superclass objects have** because **instance fields can only be added in inheritance**.

- For example, **you can assign a subclass object to a superclass variable (rule 1):**

```
Employee[] staff = new Employee[3];
Manager boss = new Manager("Carl Cracker", 75000,
  new Day(1987,12,15));
staff[0] = boss;
```

  In this case, the variables **`staff[0]`** and **`boss`** refer to the same area of memory. However, **`staff[0]`** is only considered to be an **`Employee`** object by the compiler.

- **A subclass object can be passed as an argument to any method that expects a superclass parameter.** The converse is false in general because of rule 2. For example:

```
boss = staff[0];        // error, but casting is a remedy
```

Casting

- You occasionally need **to convert an object from a parent class to a child class**. This is done by **casting.**
- **Syntax:** surround the target type with brackets and place it before the object:

```
Manager boss = (Manager)staff[0];
```

- There is **only one reason for casting**: to use an object in its full capacity after its actual type has been downplayed. For example, the `staff` array had to be an array of `Employee` objects since **some** of its entries were regular employees. We would need to cast the managerial elements of the array back to Manager in order to access any of its new fields.
- **Warning: performing a cast is not a good programming practice.**

  In our case the only reason to perform the cast is to use a method that is unique to managers, such as `getSecretaryName`. You'd better redesign the parent class and add a `getSecretaryName` method, which simply returns an empty string.
- **Warning: one bad cast will terminate your program.**

Thumb rules for casting

- The type of an object variable describes the kind of object the variable refers to and what it can do.
- If you assign a superclass object to a subclass variable, you are promising more, and **you must confirm** that you mean what you say to the compiler **by the `(Subclass)` cast notation**:

```
Manager boss = (Manager)staff[0];
```

- **In case of incorrect casting**, Java notices the broken promise, generates an exception and the **program will die**.
- It is a **good programing practice** to check whether or not your object is an instance of another object before doing a cast. This is accomplished by the **`instanceof`** operator:

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

- The compiler will not let you make a cast if there is no chance for the cast to succeed. For instance:

```
Window w = (Window)staff[0];
```

will not succeed because `Window` is not a subclass of `Employee`.

- **Summary: You can only cast within an inheritance hierarchy. Use `instanceof` to check a hierarchy before casting from a parent to a child class.**

Polimorphism

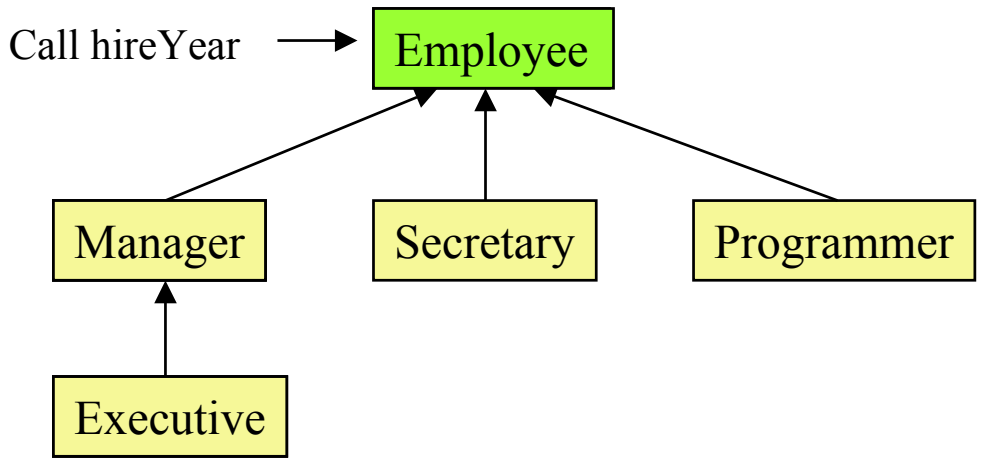
- An object's ability to decide what method to apply to itself, depending on where it is in the inheritance hierarchy, is called **polimorphism**.
- **Implementation of polimorphism:**

  When you send a message that asks a subclass to apply a method using certain parameters, the following happens:

  – The subclass checks whether or not it has a method with that name and with exactly the same parameters. If so, it uses it.

  – If not, Java moves to the parent class and looks there for a method with that name and those parameters. If finds, it calls that method. If not, Java moves up in the parent chain.

  If Java cannot find a matching method in the whole inheritance chain, you get a **compile-time error**.



Polimorphism (cont.)

- The implementation of polimorphism leads to one of the fundamental rules of inheritance :

  A method defined in a subclass with the same name and parameter list as a method in one of its ancestor classes **hides the method of the ancestor class from the subclass.**

  For example, the `raiseSalary` method of the `Manager` class is called instead of the `raiseSalary` method of the `Employee` class when you send a `raiseSalary` message to a `Manager` object.
- The name and parameter list of a method is called the method's **signature**.
- In Java, having **methods in a superclass and a subclass with the same signature** can lead to ***polimorphism*.**
- The key to making polimorphism work is called **late binding** (or **dynamic binding**).

  This means that the compiler does not generate the code to call a method at compile-time. Instead, the compiler generates code to calculate which method to call, using type information from the object.
- The regular function call mechanism is called **static binding.**
- **Static binding --** depends on the method alone**;**
- **Dynamic binding --** depends on the type of the object variable *and* the position of the actual object in the inheritance hierarchy.

Polimorphism (cont.)

- In Java, having **methods in a superclass and a subclass with the same signature but differing return types** will give you a **compile-time error**.

- For example, you cannot have a method

  `void raiseSalary(double)` in the `Employee` class and a method

  `int raiseSalary(double)` in the `Manager` class.

Comparison of polymorphism and name overloading

- Polimorphism in an inheritance hierarchy is called **true polimorphism**, to distinguish it from the more limited kind of **name overloading**.

- **True polimorphism**
  - name and parameter list (signature) must be the same
  - scope is the inheritance hierarchy
  - resolved dynamically

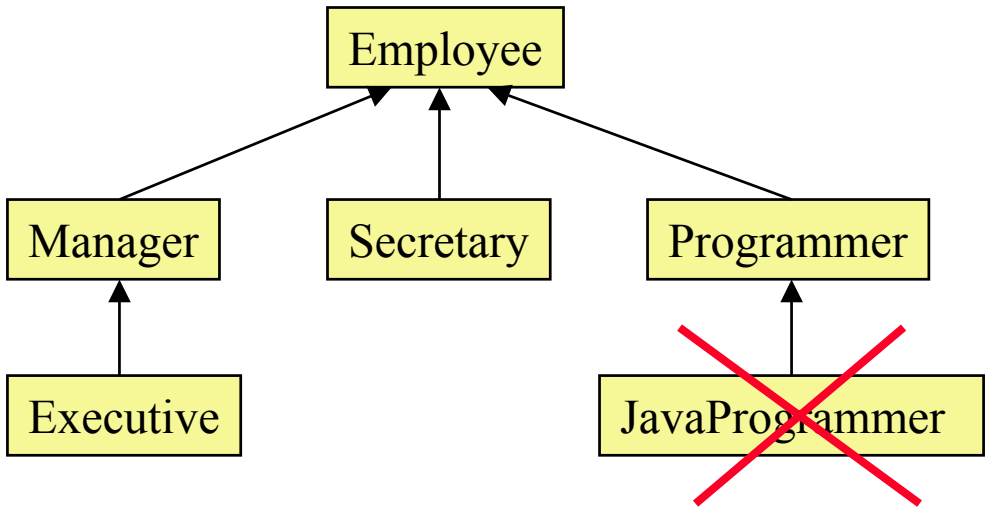
- **Name overloading**
  - name must be the same but parameter list can be different
  - scope is the same class
  - resolved statically

Preventing Inheritance: `Final` Classes and Methods

- Occasionally, you want to prevent someone from deriving a class from one of your classes. These classes are called **final classes** and you use the **`final` modifier in the definition of the class** to indicate this. For example, in case of

```
final class Programmer;
```

The `JavaProgrammer` subclass cannot be defined.



Preventing Inheritance: `Final` Classes and Methods (cont.)

- You can also make any **method final** in a normal class.

- There are two reasons to make a **class or method** final:

  1. Efficiency

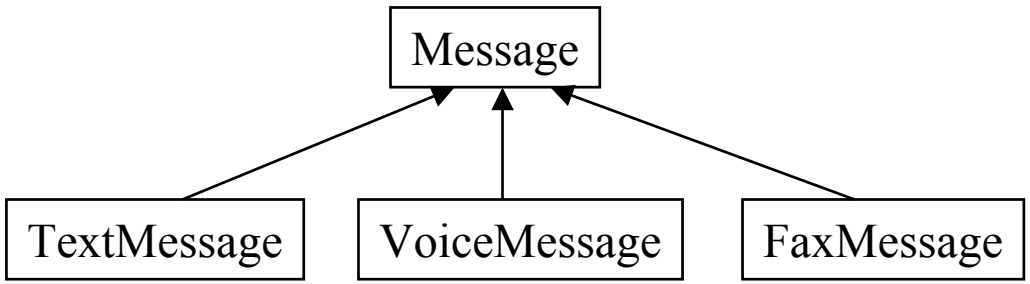
  Dynamic binding has more overhead than static binding - thus, virtual methods run slower.

  2. Safety

  The flexibility of the dynamic binding mechanism means that you have no control over what happens when you call a method.

Abstract Classes

- As you move up the inheritance hierarchy, classes become more general and more abstract and look like a **framework** for other classes.
- Consider, for example, an electronic messaging system that integrates your e-mail, faxes, and voice mail. Following the OOP principles, the program will need 3 types of classes and a common parent class `Message` as shown in the figure.



- All messages have a common method, called `play()`. It is easy to figure out how to play a voice message, text message or fax message. But how do you implement `play()` in the parent class `Message`?
- You can't. In Java, you use the **`abstract` keyword** to indicate that **a method cannot yet be specified** in the parent class but it also means a **promise that all nonabstract descendants of this abstract class will implement that abstract method**.

Abstract Classes (cont.)

- **A class with one or more abstract methods must itself be declared abstract. An abstract class has at least one abstract method.**
- **Abstract classes can have concrete data and methods.** For example, the `Message` class can store the sender of the message and have a concrete method that returns the sender's name:

```
abstract class Message
{  public Message(String from)
         { sender = from; }
   public abstract void play();
   public String getSender()
         { return sender; }
   private String sender;
}
```

- **Abstract methods act as placeholder methods that are implemented in the subclasses.**

MailboxTest.java

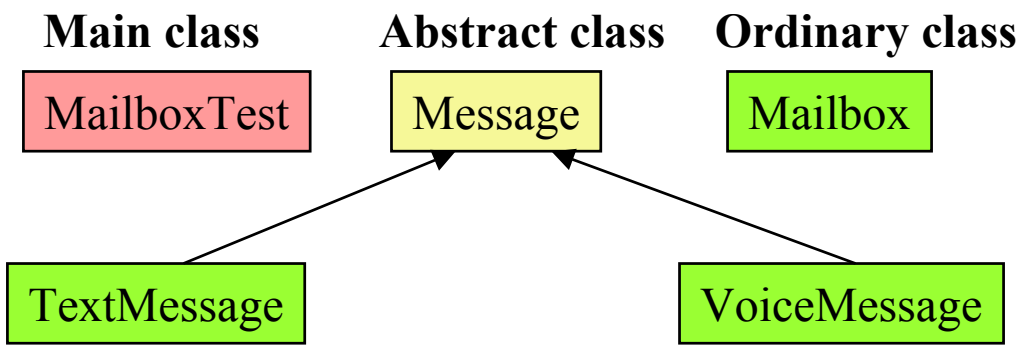
- **Exercise:**
  1. **Draw the inheritance hierarchy diagram of the program**
  2. **Analyse the OOP aspects of the code of MailboxTest.java**
  3. **Explain the data structure of the mailbox**
  4. **Run the code of MailboxTest.java**

- **Explanation:**
  - **The user interface is kept simple and ugly to allow you to focus on the OOP aspects instead of being distracted by GUI.**
  - **Do not worry too much about the code**
    - **for playing the wave file**
    - **the undocumented feature of Java that lets you play audio clips from within an application**
  - **When you run the program, you are supplied with two sample audio files, or you can use your own. They must be in .au format.**

Mailbox Test.java

- **Exercise:**

  2. Draw the inheritance hierarchy diagram of the program



- **Notice that no instance (object) is generated from class `MailboxTest` since its main method is static.** Accordingly,

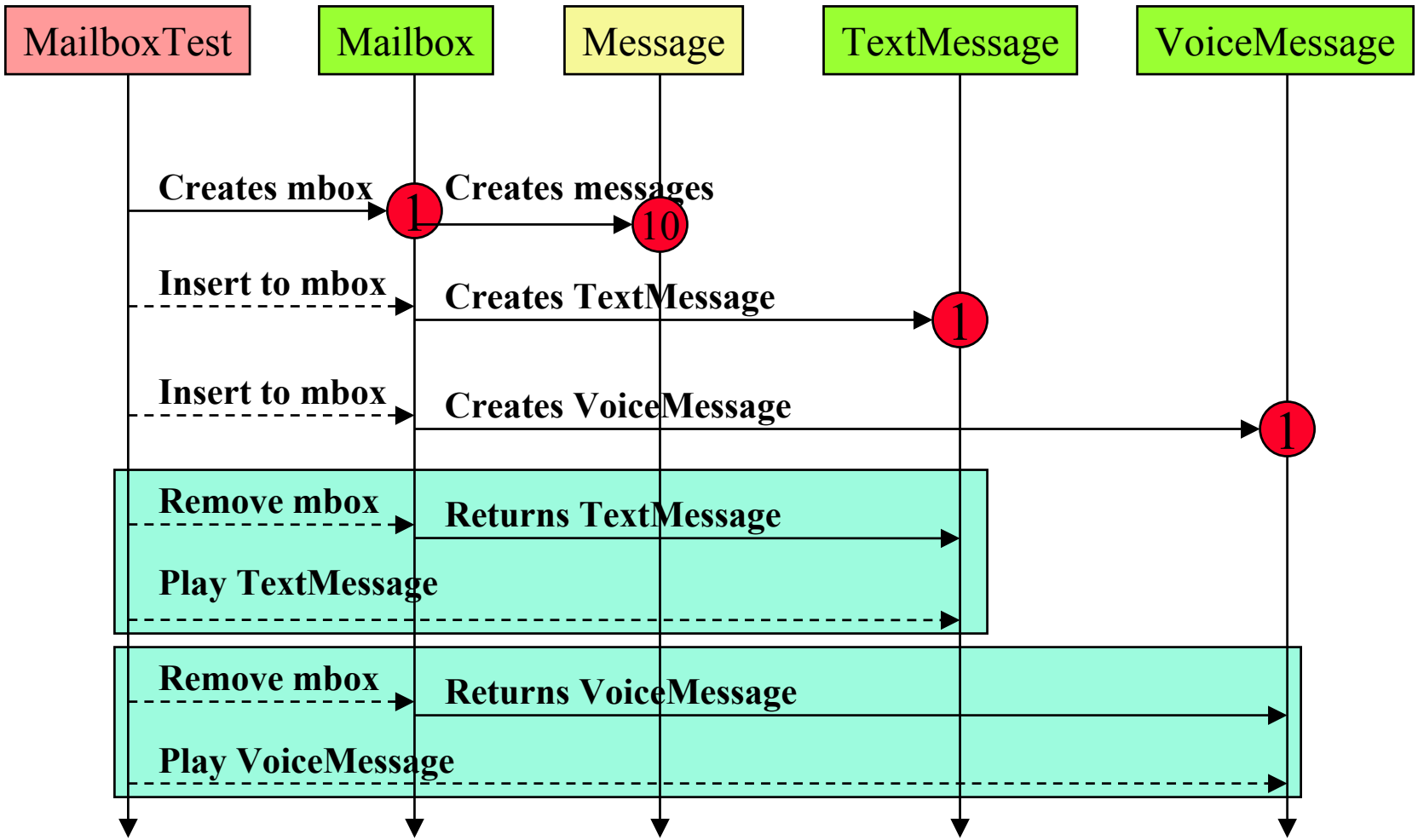
```
System.out.println("Current object: " + this);
```

cannot be used within the method `main` (its usage will result in compiler error).

Analysis of MailboxTest.java

Objects: 10

Classes:



Mailbox Test.java

- **Exercise**:

  3. Explain the data structure of the mailbox

- **Explanation**:

```
private final int MAXMSG = 10;
private int in = 0;
private int out = 0;
private int nmsg = 0;
private Message[] messages = new Message[MAXMSG];
```

  - It can contain up to 10 messages in the array `messages` which are the instants of the abstract class `Message` which has one instance field called `sender`
  - When the main program calls

    `mbox.insert(new TextMessage(from, msg));`

    the abstract instance field becomes a concrete one with the structure:

    `String sender, text;`

Catching Exceptions

- **When an error occurs at run time, a Java program can "throw exception".** Throwing an exception is less violent than terminating the program, because it provides the option of "catching" the exception and dealing with it.
- **If an exception is not caught anywhere, the program will terminate**, and a message will be printed to the console giving the type of the exception.
- **Syntax**: To run code that might throw an exception, you have to place it inside a **"try" block**. Then you have to provide an emergency action to deal with the exception:

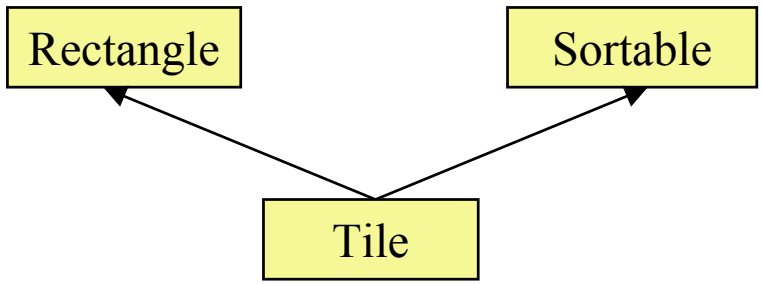
```
try
{  code that might throw exceptions  }
catch(ExceptionType e)  {  emergency action  }
```

- This mechanism is used in the code that plays an audio clip.

```
AudioPlayer ap = AudioPlayer.player;
try
{  AudioStream as
        = new AudioStream(new FileInputStream(filename));
   ap.start(as);
}
catch(IOException e) {}        // ignore the error and do not
                               // play the clip
```

Multiple Inheritance

- **Multiple inheritance** means that a class can have more than one superclass.



- The **tile class** models tiled windows on a screen desktop. Tiled windows are **rectangles** plus a „z-order”. Windows with a larger z-order are displayed in front of those with a smaller z-order. That’s why tiles should be sorted and inherit the sort method from class **Sortable**.

Interfaces
(Multiple Inheritance)

- **Java does not support multiple inheritance.**
- **Instead, Java introduces the notion of interfaces** to recover much of the functionality provided by multiple inheritance.
- Reason: **multiple inheritance makes compilers either very complex** (see C++) **or very inefficient** (see Eiffel).
- An interface is a promise that your class will implement certain methods with certain signatures. You even use the keyword **`implements`** to indicate that your class will keep the promise.
- For example, you want to create an interface called **`Sortable`** that could be used by any class that will sort. The code might look like this:

```
public interface Sortable
{  public int compare(Sortable b);
}
```

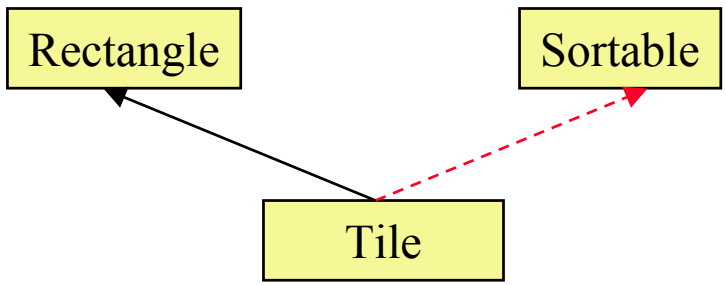
- **This code promises that any class that implements the `Sortable` interface will have a `compare` method that will take a `Sortable` object. A `Sortable` object is any instance of a class that implements `Sortable`** (i.e. it has a `compare` method).

Interfaces (cont.)

- **To tell Java that your class implements `Sortable`, you have the class header:**

```
class Tile extends Rectangle Implements Sortable
```

where Tile is a class that extends the class `Rectangle` and models tiled windows on a screen desktop.



Interfaces (cont.)

- Then all you need to do is implement a `compare` method inside the class.

```
class Tile extends Rectangle implements Sortable
{       . . .
        public int compare(Sortable b)
        {  Tile tb = (Tile)b;
           return z - tb.z;
        }
        . . .
        private int z;
}
```

- **Exercise:**

1. Study the code of TileTest.java (Note that we needed to put the static shell_sort method in a separate class, Sort. **You cannot put static methods into interface classes**.)

2. Draw the inheritance hierarchy of the **TileTest.java** program.

Properties of Interfaces

- **Interfaces are not instantiated with `new`.**
- They have certain properties similar to ordinary classes:
  - Once you set up an interface **you can declare that an object variable will be of that interface type** with the same notation used in case of ordinary classes:

```
Sortable x = new Tile(. . .);
Tile y = new Tile(. . .);

if (x.compare(y) < 0) . . .
```

  - **You can extend one interface in order to create another.**

```
public interface Moveable
{  public void move(double x, double y);
}
public interface Powered extends Moveable
{  public String PowerSource();
}
```

Properties of Interfaces (cont.)

- **You cannot put instance fields in an interface.**
- **You can supply constants in an interface.**

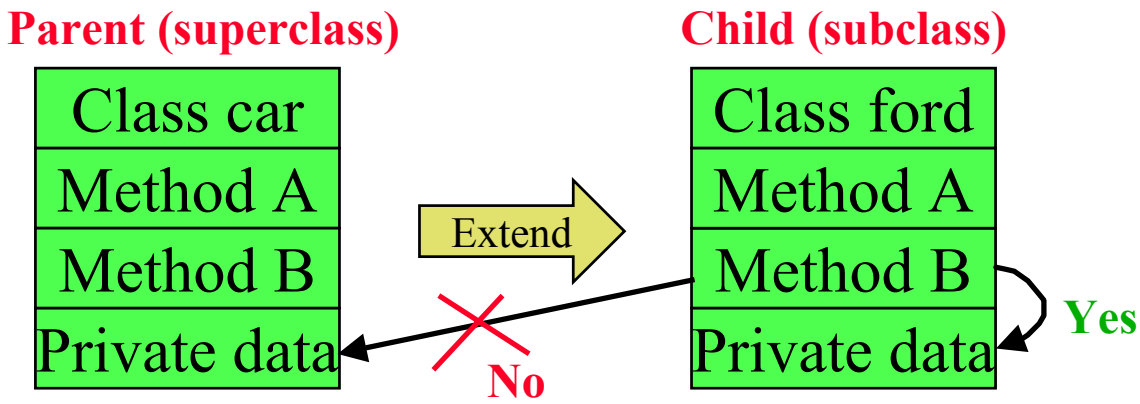
```
public interface Powered extends Moveable
{  public String PowerSource(PoweredVehicle);
   public final int speedLimit = 95;
}
```

- **Classes can implement multiple interfaces.**

```
class Tile extends Rectangle implements Cloneable, Sortable
```

Pivate Methods

- **A subclass cannot access the private data members of its superclass!**



- **Explain, why?**
- Because maybe there is no object created from the parent class and hence the private data of parent class does not exist when the child object runs.

Access Modifiers in Java

1. **`private`:** visible to the class only (their access is denied even for subclasses)
2. **`public`:** visible to the world
3. **`protected`:** visible to the package and all subclasses
4. **`--`:** visible to the package (the default, no modifier needed)
5. **`abstract`:** Abstract methods act as placeholder methods that are implemented in the subclasses
6. **`static`:**
   - Static methods can be used without creating an instance of a class
   - Static fields do not change from one instance of a class to another
7. **`final`:** prevents inheritance of the class or polimorphism of the method

Design Hints for Inheritance

- **Common operations and instance fields should be placed in the superclass.**
- **Use inheritance to model the ”is-a” relationship**.
  - The subclass should be a special case of the superclass!
- **Don’t use inheritance unless all inherited methods make sense**.

  Suppose we want to write a `Holiday` class. Surely every holiday is a day, so we can use inheritance.

  ```
  class Holiday extends Day { . . . }
  ```

  There is a problem with the `advance` method of the `Day` class. It can turn holidays into non- holidays, so it is not an appropriate operation for holidays.

Design Hints for Inheritance (cont.)

- **Use polymorhism, not type information**

  Whenever you find the code of the form

```
if (x is of type1)
        action1(x);
else if (x is of type2)
        action2(x);
```

  Think polymorhism.

  If `action1` and `action2` represent a common concept, make the concept a method of a common parent class or interface of both types. Then you can simply call:

```
x.action();
```

  Code with polymorhic methods or interface implementations is much easier to maintain and extend than code with type tests.

Thank You ...

Object Wrappers

- Occasionally, you need **to convert a basic type like `int` to an object**. **All basic types have class counterparts**. For example, there is a class `Integer` corresponding to the basic type `int`. These kinds of classes are called **object wrappers** and they are **`final`**.
- The major reason for which wrappers were invented is **generic programming**. The **container classes** (see Chapter 9) can store arbitrary objects but no numbers.
- **Example** to illustrate the concept: Suppose you want to find the index of an element in an array. This is a generic situation and you can reuse the code for employees, dates, etc.

```
Static int find(Object[] a, Object key)
{  int i;
   for (i = 0; i < a.length; i++)
      if (a[i].equals(key)) return i;
   return -1;     }       // not found
```

- For example:

```
Employee[]       staff;
Employee         harry;
. . .
int n = find(staff, harry);
```

- If you want to find a number in an array of integers, **you must use `Integer` objects instead of `int` variables** and then you can take advantage of the generic code.
- **Exercise:** write the necessary code for integers.

Reading a Page in the HTML Documents

- **Organization of the API documentation pages:**
  - **The name of the class (or interface);**
  - **The inheritance chain for this class (starting from `java.lang.Object`);**
  - **The name of the class along with the access modifiers such as public or final, the classes it extends, and the interfaces it implements.**
  - **A discussion of the class**
  - **A list of all the variables, constructors and methods in the class;**
  - **A more detailed discussion of the variables, constructors and methods .**
- **Exercise:**

  Study the class `java.lang.Integer` API description.

  (file:///C|/java/api/java.lang.Integer.html#_top_)

More on `Object`: The Cosmic Superclass

- The **`Object` class** is the **ultimate ancestor - every class extends** `Object`**.** You don't have to say:

  ```
  class Employee extends Objects
  ```

- Here are versions of the API descriptions of the basic parts of the `Object` class from **java.lang.Object** (file:///C|/java/api/java.lang.Object.html#_top_)

  ```
  Class getclass()
  ```

  Returns the **`Class`** that contains information about the object.

  ```
  boolean equals(Object obj)
  ```

  Compares two objects for equality; returns true if the objects point to the same area of memory (i.e. they are equal), and false otherwise. (Other classes in the Java hierarchy are free to override `equals` for a more meaningful comparison.)

  ```
  Object clone()
  ```

  Creates a clone of the object. Java allocates memory for the new instance and copies the memory of the current object to the memory allocated for the clone.

  ```
  String toString()
  ```

  Returns a String that represents the value of this Object.

The Class `Class` (Run-time Type Identification)

- While your program is running, Java always maintains what is called **run-time type identification** (RTTI) on all objects. It keeps track of the class to which each object belongs. This is used by Java to select the correct methods at run time.
- **You can also access this information**. The class that holds this information is called **`Class`**. The **`getClass()`** method in the `Object` class returns an instance of this class type. The **`getName()`** method of `getClass` returns the name of the class. For example:

  ```
  System.out.println(e.getClass().getName() + " " + e.getName());
  ```

  prints:

  ```
  Employee Harry Hacker
  ```

  if `e` is an employee, and the code prints:

  ```
  Manager Harry Hacker
  ```

  if `e` is a manager.
- The **`newInstance()`** method can create a new instance of the class on the fly and calls the constructor to initialize the newly created object. For example:

  ```
  e.getClass().newInstance();
  ```

  would create a new instance of the same class type as `e`.

The Class `Class` (cont.)

- The combination of **`newInstance()`** and **`forName()`** methods lets you **create an object from a class name stored in a string**. For example:

```
String s = "Manager";
Manager m = (Manager) Class.forName(s).newInstance();
```

- **Exercise:**

  1. Study the class `java.lang.Object` API description.

  (file:///C|/java/api/java.lang.Object.html#_top_)

  2. Study the class `java.lang.Class` API description.

  (file:///C|/java/api/java.lang.Class.html#_top_)

Exercise

- **Where is the error in the code below?**

```
class Manager extends Employee
{  public Manager(String n, double s, Day d)
   {    super(n, s, d);
        secretaryName = "";
   }
   public void raiseSalary(double byPercent)
   {    // add 1/2% bonus for every year of service
        Day today = new Day;
        double bonus = 0.5 * (today.getYear() - hireDay.year);
        super.raiseSalary(byPercent + bonus);
   }
   private String secretaryName;
}
```

Exercise Solutions

- **Where is the error in the code below?**

```
class Manager extends Employee
{  public Manager(String n, double s, Day d)
   . . .
   public void raiseSalary(double byPercent)
   {    // add 1/2% bonus for every year of service
        Day today = new Day;
        double bonus = 0.5 * (today.getYear() - hireDay.year);
        super.raiseSalary(byPercent + bonus);
   }
}
```

Answer: `hireDay` **is a private instance field in the superclass** `Employee` **and hence, it cannot be accessed directly from the** `Manager` **subclass. Instead of**

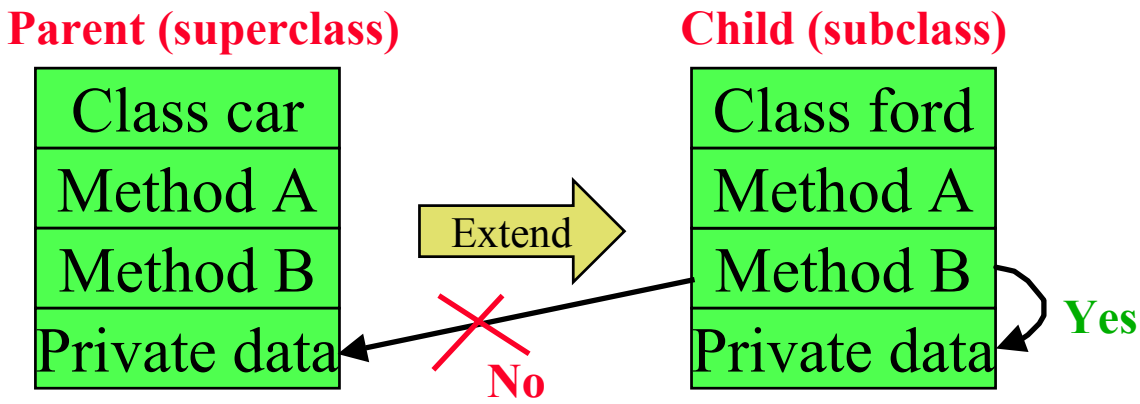
```
hireDay.year
```

you should use the `hireYear()` **public method of** `Employee`**:**

```
double bonus = 0.5 * (today.getYear() - hireYear());
```

Pivate Methods

- **A subclass cannot access the private data members of its superclass!**



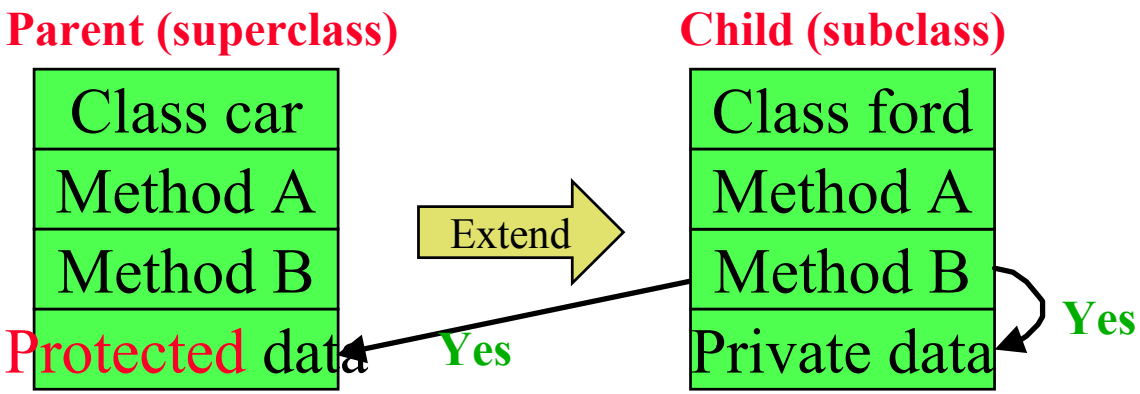
- **Explain, why?**
- Because maybe there is no object created from the parent class and hence the private data of parent class does not exist when the child object runs.

Protected Access

- There are cases when **you want a subclass to access a method or data of the superclass.**
- In such case you declare the method or data as **protected** instead of private.
- For example, the line

```
double bonus = 0.5 * (today.getYear() - hireDay.year);
```

is correct if the **`hireDay`** object is declared as **protected.**



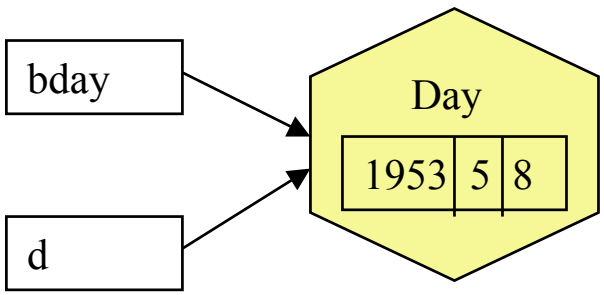
- **Be careful!** If you use **protected instance fields**, you can no longer change the implementation of your class without upsetting the other programmers using your class.
- **Protected methods** make more sense. This indicates that the subclasses (which presumably, know their ancestors well) can be trusted to use the method correctly, but other classes cannot.
- A good example is the **`clone` method** in the **class `Object`**.

Copying and Cloning

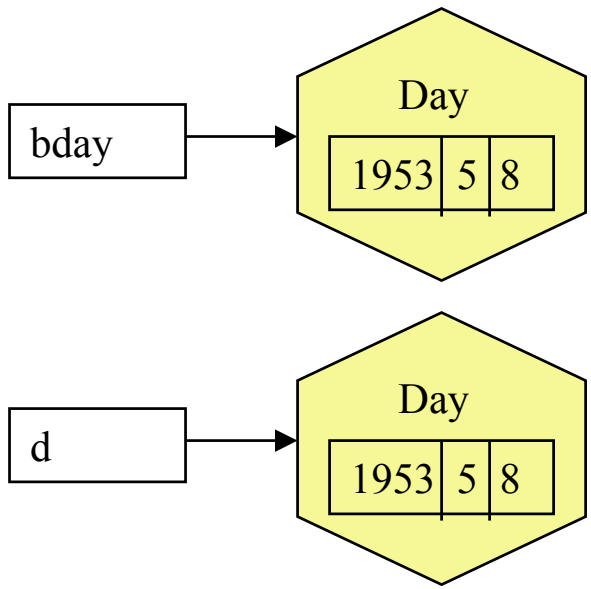
- **Copying**: when you make a copy of a variable, the original and the copy are references to the same object. => **A change to either variable also effects the other.**

```
Day bday = new Day(1953, 5, 8);
Day d = bday;
d.advance(100);          // oops -- also changed bday
```

Copying



Cloning



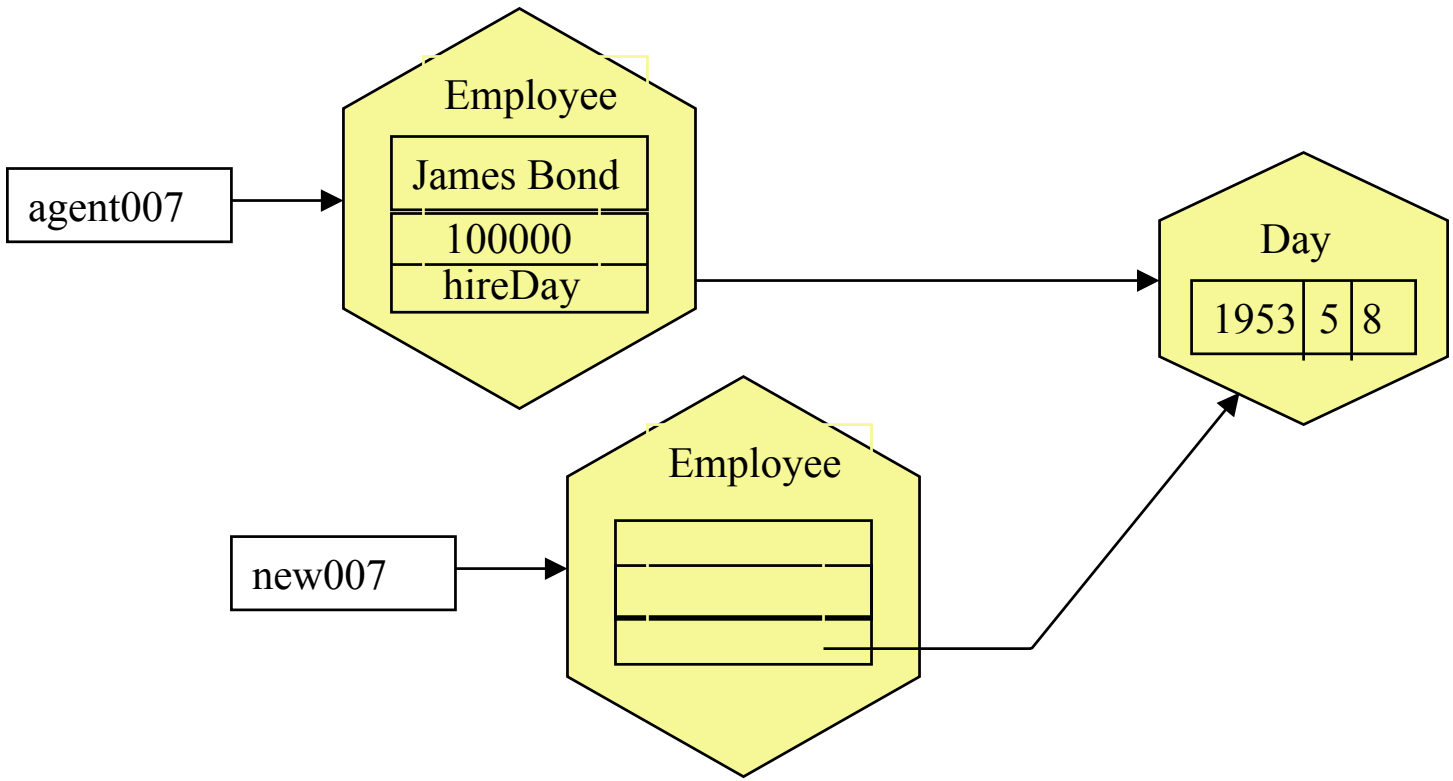
- **Cloning**: when you create a new object of a variable, that begins as identical with the original variable. => **A change to either variable does not effect the other.**

Cloning

- To create a clone of bday see the example:

```
Day bday = new Day(1953, 5, 8);
Day d = (Day)bday.clone();
                    // must cast -- clone returns an object
d.advance(100);  // ok -- bday unchanged
```

- However, cloning is not so simple, **if the object contains other objects** (i.e. pointers). In such case the **bitwise copy results in the sharing of the contained object**:



Cloning in case of nested objects

- If we want to clone employees, we have to call the `Object` clone method to make a bitwise copy, then clone the `Day` object:

```
public class Employee implements Cloneable
{ . . .
     Public Object clone()
     {  try
        {    Employee e = (Employee)super.clone();        // ?
             e.hireDay = hireDay.clone();
             return e;
        }    catch(CloneNotSupportedException e)
             // this shouldn't happen, since we are cloneable
              return null;
        }
     }
}
```

Thank You ...

Properties of Interfaces (cont.)

- **You cannot put instance fields in an interface.**
- **You can supply constants in an interface.**

```
public interface Powered extends Moveable
{  public String PowerSource(PoweredVehicle);
   public final int speedLimit = 95;
}
```

- **Classes can implement multiple interfaces.**

```
class Tile extends Rectangle implements Cloneable, Sortable
```

(Java has an important built-in interface called `Cloneable`. If your class implements `Cloneable`, the `clone` method in the `Object` class will make a bitwise copy of your class's objects.)