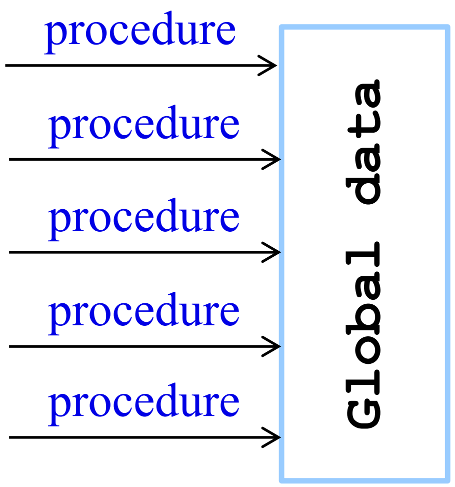
Java Objects and Classes

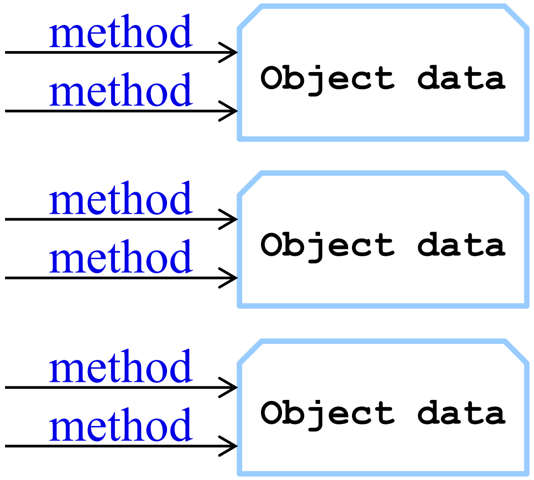
Object-Oriented Programming

- *Object-oriented* (OO) programming requires different way of thinking than *structured, procedural* programming
  - Transition is not always easy
- OO program is made of *objects*
  - Specific functionality, exposed to users
  - Hidden implementation
    - From Java platform packages
    - Custom-designed by programmers

OO vs Procedural Programming

- Procedural programming
  - A set of procedures to solve a problem
    - Algorithms
  - Ways to store data
    - Global data structures
- OO programming
  - Ways to store data
    - Object data structures
  - Methods to manipulate data
    - Algorithms





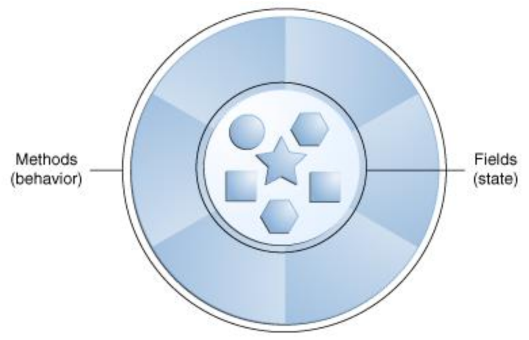
What are Objects

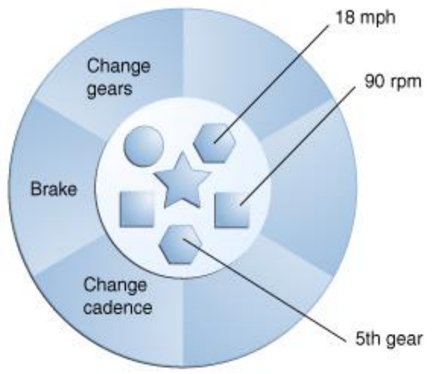
- Software Object
  - State
    - Stores data – *fields/variables*
  - Behavior
    - Performs tasks – *methods/functions*

- An **object** stores its ***state*** in ***fields*** and exposes its behaviour through ***methods***

- ***Methods*** operate on an object's internal state and serve as the primary mechanism for ***object-to-object communication***

- By ***attributing* state *and providing* methods *for changing that state***, the object remains in control of ***how the outside world is allowed to use it***





Identifying the state and behaviour for real-world objects is a great way to begin thinking in terms of object-oriented programming

A bicycle modelled as a software object

Key Characteristics of Objects

- Behaviour
  - What can you do with this object, or what methods can you apply to it?
  - Behaviour is defined by methods
- State
  - How does the object react when you apply those methods?
  - A change of the state must be a consequence of method calls
- Identity
  - How is the object distinguished from others that may have the same behaviour and state?
  - Individual objects always differ in their identity and usually differ in their state

Benefits of Software Objects

- Modularity:
  - The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.
- Information-hiding:
  - By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
- Code re-use:
  - If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.
- Pluggability and debugging ease:
  - If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace *it*, not the entire machine.

Analysis and Design

- Analysis
  - What the system needs to do, by defining a set of actors and activities through identifying domain objects, physical and conceptual
- Design
  - How the system does it, by modeling it using Unified Modeling Language (UML)
- Other phases in a software development project
  - Implementation, test and deployment

Software Development Process

Understand the problem;
Document what the system needs to do

Analyse business process in terms of data flow
Identify the system's input and output:
Modelling system's behaviours

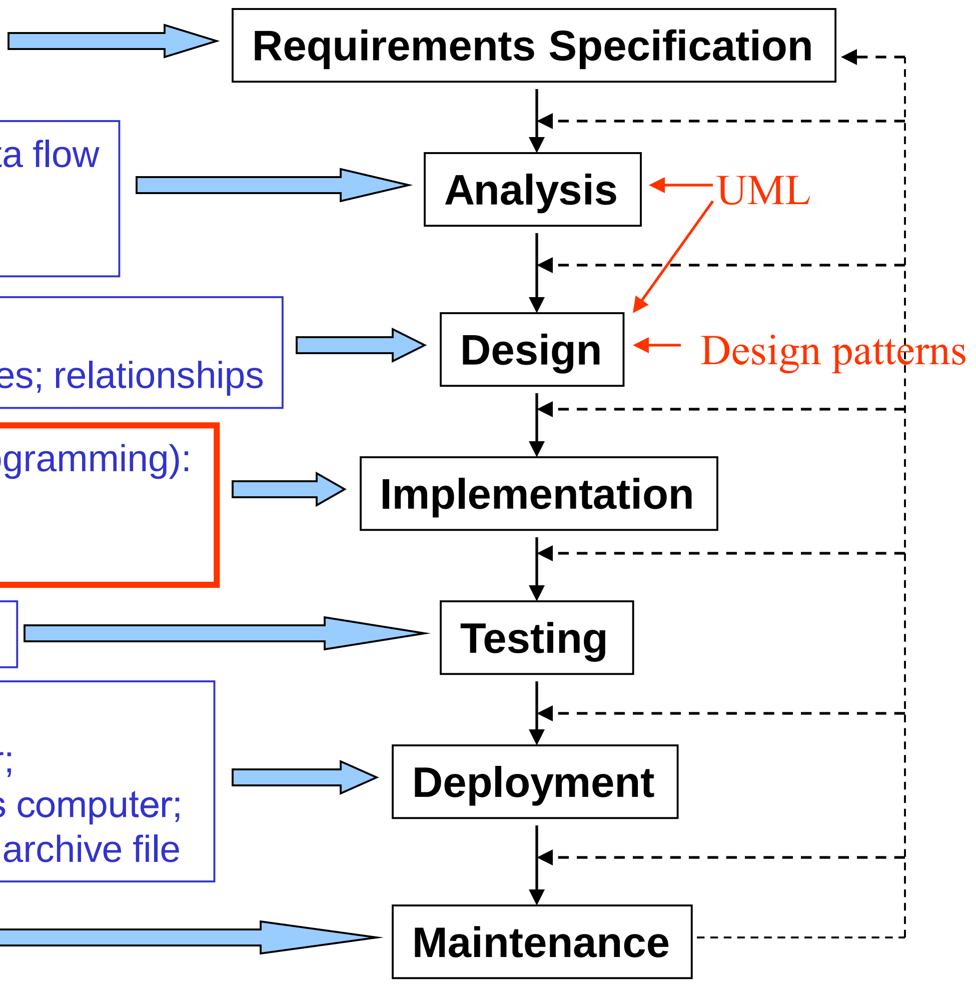
Design system's components:
levels of abstraction;classes and interfaces; relationships

Translate design into programs (Java programming):
write programs for each components;
put to work together

Ensure the code meets the requirements

Make the project available for use:
Java applet - installing it on a Web server;
Java application - installing on the client's computer;
package all classes into a Java archive file

Change and improve the product



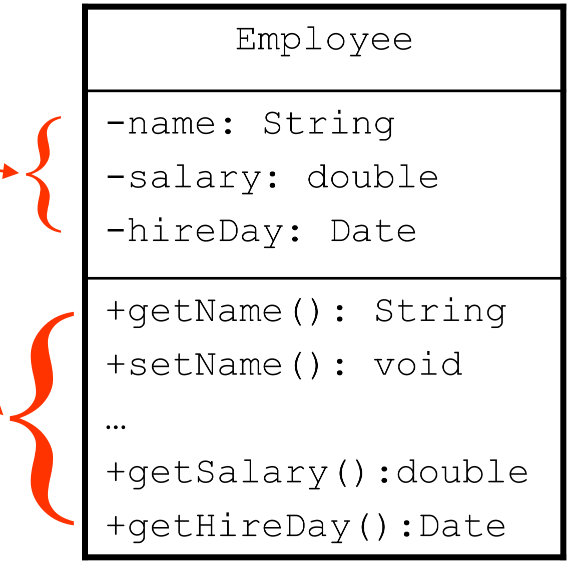
Do we need a design?

- If you build a bench, you don’t need a blueprint
- If you build a hut, most people may still don’t need a blueprint
- If you build a house, most people do need a blueprint
- If you build a skyscraper, no people do not need a blueprint

How many people have a design before writing a program?

Classes as Blueprints for Objects

- *A class is a description* of an object:
  - A class describes the ***data*** that each object includes
  - A class describes the ***behaviours*** that each object exhibits
- Many objects can be constructed from a class
  - An instance of class
  - There are many cycles (in different states – speed, gears) – similar attributes and behaviours described by a class



Java Programming Process

- General OO programming
  - Identify classes
    - group a related set of attributes and behaviors in a class
  - Add methods to each class
    - write an algorithm once to be used in many situations

- Java framework/package programming
  - ***Frameworks*** and ***APIs*** of Java standard libraries – large groups of objects that support a complex activity
    - Frameworks/packages can be used "as is" or be modified to extend the basic behavior

OO Programming

Java classes support three *key features of OO Programming*

- Encapsulation
  - Combining data and behaviour – information hiding – black box
    - Never allow direct access to instance fields by other classes
    - Objects interact through their methods
  - Hiding implementation details

- Inheritance
  - Building a class (subclass) by extending another classe (superclass)
  - The subclass has all the properties and methods of the superclass

- Polymorphism
  - Every object of a subclass is an object of the superclass
  - Object reference variable of a superclass can refer to multiple actual types of objects of subclasses and select the appropriate method at runtime (dynamic binding)

Class Definition

```
class ClassName
{
     field1
     field2
     .  .  .
     constructor1
     constructor2
     .  .  .
     method1
     method2
     .  .  .
}
```

4 components of a class

- class declaration
- attribute variable declarations and initialization (optional)
- methods (optional)
  - accessors and mutators
- comments (optional)

Declaring A Class (An Example)

Visibility modifiers

```
public class Employee {
    private String name;
    private double salary;
    private Date hireDay;

    public Employee(String n, double s, int year,
                    int month, int day){
      name = n;
      salary = s;
      …
    }
    public String getName(){
        return name;
    }
    public double getSalary()  {
        return salary;
    }
    public void raiseSalary(double byPercent){
        double raise = salary * byPercent / 100;
        salary += raise;
    }
    …
}
```

Declare fields

Declare constructor

Declare methods

Accessors

Getter method

Mutators

Setter method

Construct an Object from a Class

Example:

invoking the constructor

```
Employee aStaff = new Employee(“FirstName LastName, 8000, 1980, 8, 12);
```

Accessing Object Members - The “dot” notation:

```
<object>.<member>
```

Used to access object members including *attributes* and *methods*

Example:

A method has two parameters

implicit parameter *explicit parameter*

```
aStaff.raiseSalary(5);
aStaff.salary = 8400; //only permissible if x is public
```

Constructor

- Constructors always have the same name as the class name
- A class can have more than one constructors
  - All constructors have the same name but different parameters
  - ***Overloading*** resolution – the compiler sorts out the correct method by matching the parameter types
  - The `this(…)` as the first statement in a constructor calls another constructor
- There is at least one constructor in every class
- If no constructors are provided, the default constructor is presented automatically
  - The default constructor takes no arguments
  - The default constructor has no body
    - It sets all the instance fields to their default values
  - You cannot use default no argument constructor if you have defined any constructors
- The default constructor enables you to create object instances with `new ClassName()` without having to write a constructor

Java Source Files

- The name of the file must match the name of the **`public`** class
  - You can only have one **`public`** class in a source file
    - A public class must be defined in its own file
  - You can have any number of nonpublic classes

- Every class can have a **`main`** method
  - The bytecode interpreter starts running the code in the **`main`** method in the class that is given

```
java aClassName
```

- It is common for Java programs to have multiple source files

Encapsulation

- encapsulation is used to enclose in a class the attributes and methods that logically belong together
- The class controls access to its encased attributes by providing public methods to act on the encased variables

Advantages of Encapsulation

- Because access is provided through an interface, the internals can be kept safe from outside interference and misuse.
- As long as the interface is not modified, the internals can be changed without affecting users of the interface.
- Encapsulation facilitates reuse of objects. The user is unaffected by any changes to the internals of the object.

UNIVERSITY OF WOLLONGONG

Information Hiding: Problem

```
public class Employee {
    public String name;
    public double salary;
    public Date hireDay;

    public String getName(){
        return name;
    }
    public double getSalary() {
        return salary;
    }
    …
}
```

```
Employee me = new Employee();

//I like it very much
me.salary = 1000000;
```

Information Hiding: Solution

```
public class Employee {
    private String name;
    private double salary;
    private Date hireDay;

    public String getName(){
        return name;
    }
    public double getSalary() {
        return salary;
    }
    …
}
```

```
Employee him = new Employee();

//No one can do it !
him.salary = 10;
^ compilation error
```

Encapsulation

- Hide the implementation details of a class
- Force the user to use an interface to access data
- Make the code more maintainable

```
public class Employee {
    private String name;
    private double salary;
    private Date hireDay;

    public double getSalary() {
        if (salary < 30000) {
            /* throw an exception
             * telling that you should be
             * paid more than that :-(
             */
        }else if (salary > 1000000) {
            /* throw an exception
             * telling that it's time
             * to promote you :-)
             */
        }
        return salary;
    }
    …
}
```

The idea is that you can validate data, not necessarily throw an exception. Exceptions are only used for errors.

Java Encapsulation Mechanisms

- Package mechanism
- Class mechanism

Source File Layout

package_declaration

import_declaration

class_declaration

```
package shipping.reports;

import shipping.domain.*;
import java.util.List;
import java.io.*;

public class VehicleCapacity {
    private List vehicles;
    public void generateReport(Writer output){
        ...
    }
}
```

The `package` Statement

- Specify the package declaration
- Only one package declaration per source file
- If no package is declared, then the class belongs to the default package
- Package names must be hierarchical and separated by dots

Example:

```
package shipping.reports;
```

The `import` Statement

- Tells the compiler where to fine classes to use
- The only benefit of the import statement is convenience

eg:

```
import shipping.domain.*;
import java.util.List;
import java.io.*;
```

In "**`private List vehicles`**" where the class **`List`** is found in the package **`java.util.List;`** The class **`Writer`** in the package **`java.io.Writer`**

Directory Layout and Packages

- Packages are stored in the directory tree containing the package name

Example:

```
shipping/
    |
    |------- domain/
    |          |
    |          |------- Company.class
    |          |------- Vehicle.class
    |          |------- Truck.class
    |
    |------- GUI/
    |
    |------- reports/
               |
               |------- VehicleCapacityReport.class
```

Encapsulation: Package Mechanism

- The package mechanism also provides a layer of encapsulation
- For a class to be visible outside its package, it must be declared with the modifier `public`. By not declaring a class with the public modifier, you can use the package mechanism to hide (encapsulate) the class from all classes that do not belong to the same package

Encapsulation: Class Mechanism

1. Create a class that groups together variables and the methods that operate on the variables
2. Visibility modifiers: restrict access by other classes to the variables and methods of this class
   - `private` modifier
     - accessible only by methods that are members of the same class
   - Default access – no explicit protection modifier
     - access is permitted from any method in classes that are members of the *same* package as the target
   - `protected` modifier
     - accessible from methods in classes that are members of the same package and from any method in any subclass
   - `public` modifier
     - accessible to *any* class in *any* package

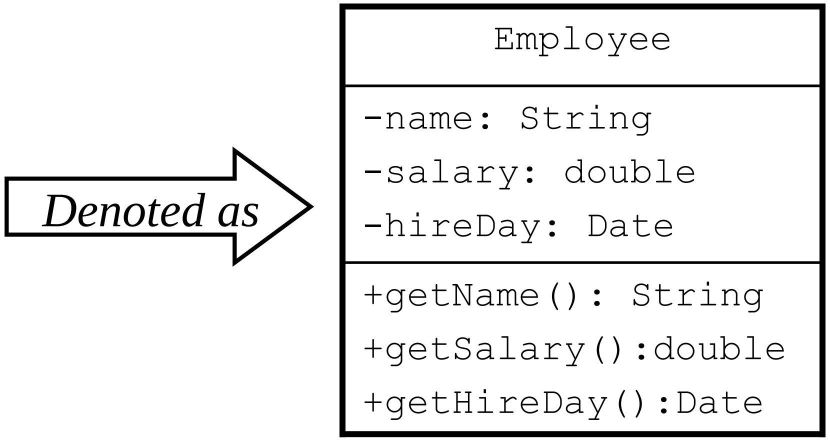
Accessibility Criteria

Modifier	Same Class	Same Package	Subclass	Universe
`private`	Yes			
default	Yes	Yes		
`protected`	Yes	Yes	Yes	
`public`	Yes	Yes	Yes	Yes

Look at the Problem #1

- You spend 1 week to build a class with 3 methods
  - `Class Employee`

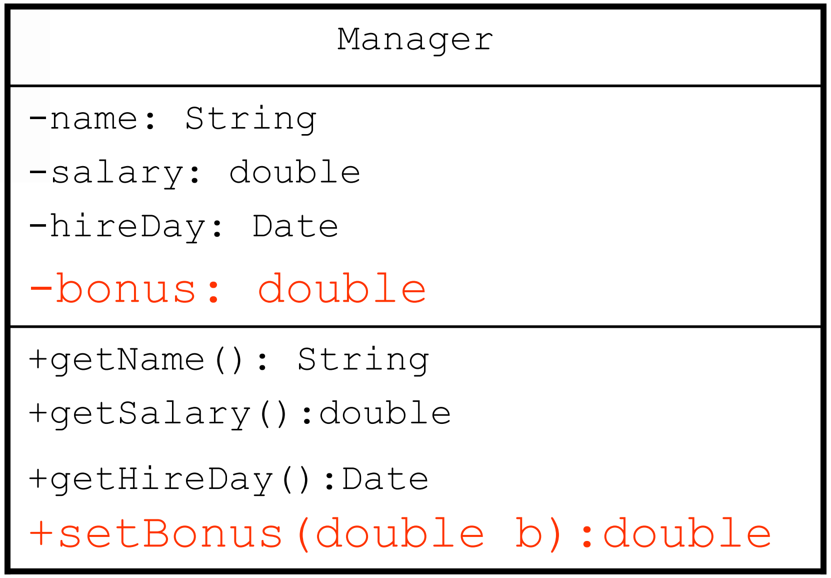
```
public class Employee {
    private String name;
    private double salary;
    private Date hireDay;

    public String getName(){
        return name;
    }
    public double getSalary() {
        return salary;
    }
    …
}
```



- Next Monday, the manager asks you build a Manager class that can set bonus – `Class Manager`

Solution #1

- Build a new class called Manager from scratch



Looking for Better Solution

- Why re-invent the wheel?
- Is there any way the new class can be built upon existing one?

Concept of Inheritance

- Inheritance is everywhere in our world
- Fundamental concept of OO programming

Idea:

create a new class from existing classes

Inheritance

- Inheritance is an object-oriented mechanism whereby a new class can be derived from an existing class
- New methods and fields are added to the new class to adapt to new situations.

Inheritance in Java

subclass

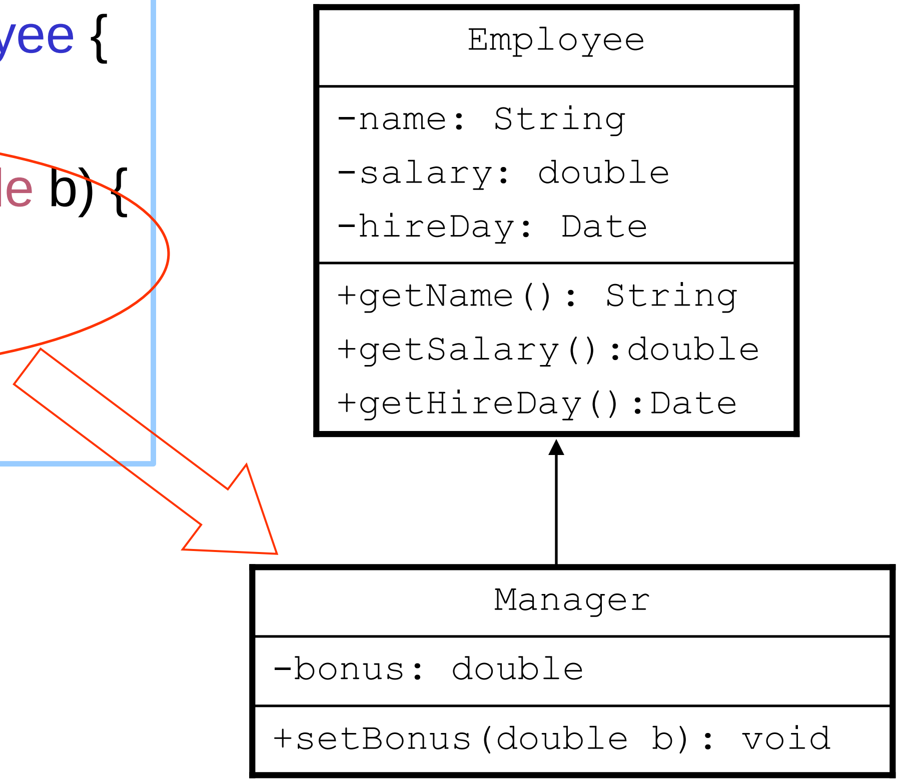
superclass

```
Class newClass extends OldClass{
    /* New methods and fields */
}
```

A class can only extend a single class, unlike C++

Solution #2

```
Class Manager extends Employee {
      private double bonus;
      public void setBonus(double b) {
            bonus =b;
      }
}
```



Inherited method

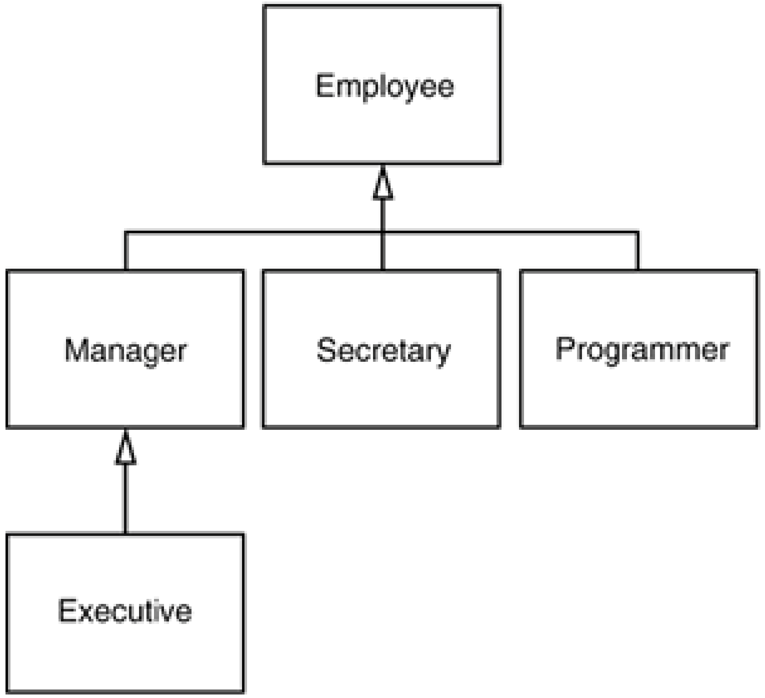
```
Manager myManager = new Manager();

myManager.getName();  //what we like to have
```

- All methods and fields are available for subclass, except private ones;
- Constructor is not inherited

Inheritance Hierarchy

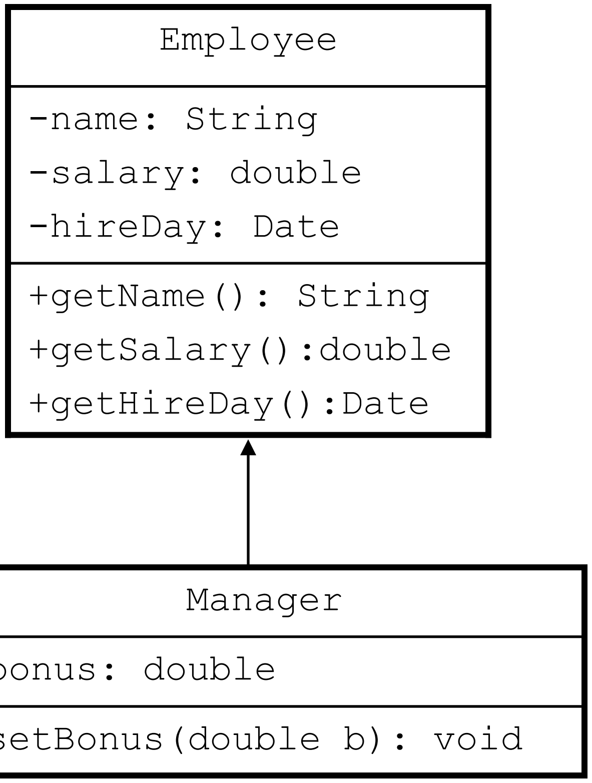
eg.

`Employee` inheritance hierarchy



Look at Problem #2

How to get Manager's salary?



```
myManager.getSalary();  //Manager not happy
```

?

Solution #1

```
Class Manager extends Employee {
    private double bonus;
    public void setBonus(double b) {
        bonus =b;
    }
    public double getSalary() {
        return salary + bonus;
    }
}
```

?

✓

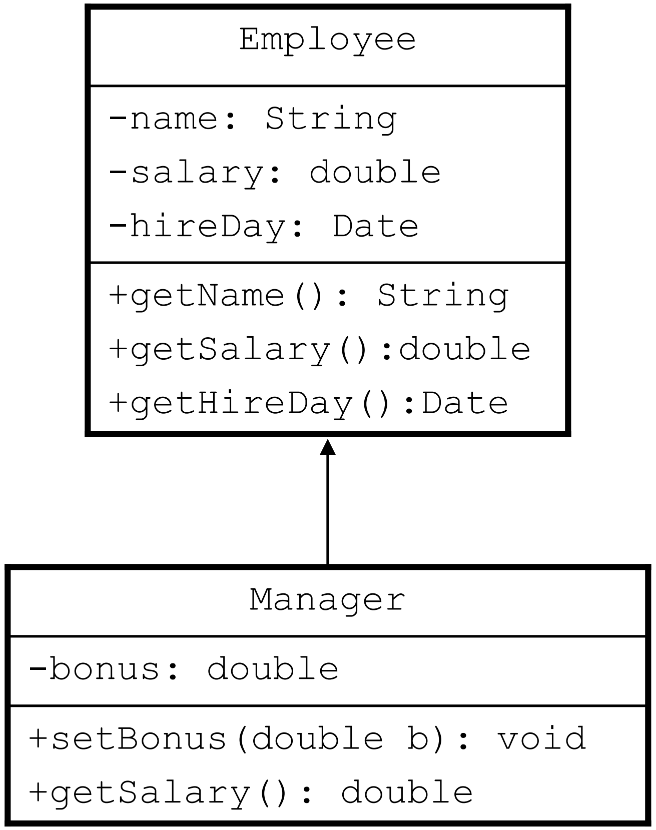
To override supperclass method

?

```
myManager.getSalary();   //computer not happy
```

Solution #2

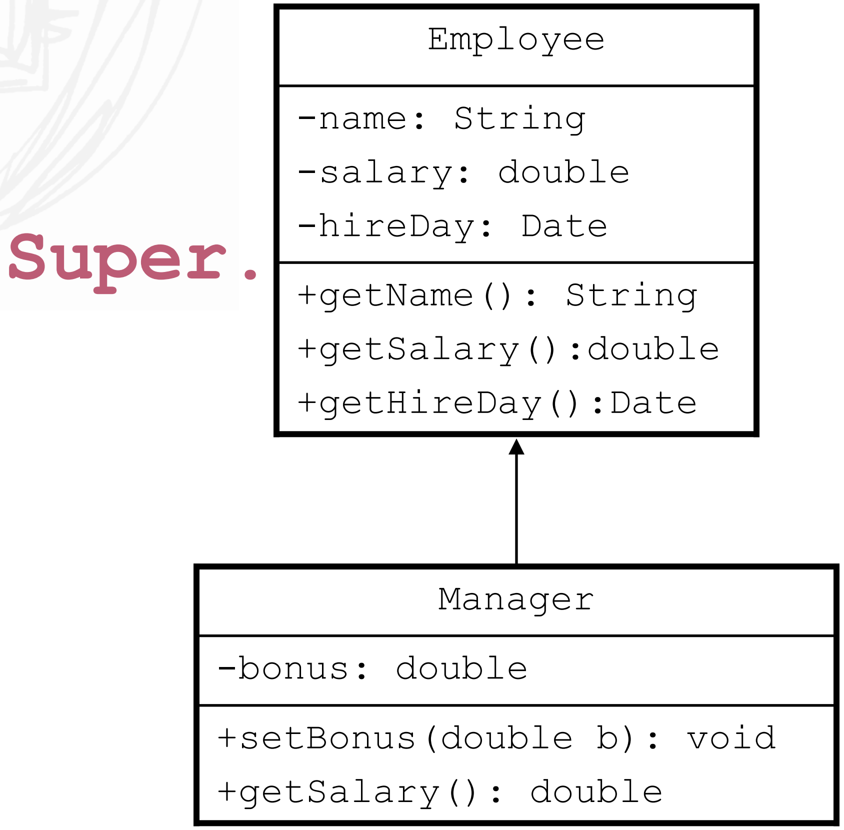
```
Class Manager extends Employee {
    private double bonus;

    public void setBonus(double b) {
        bonus =b;
    }
    public double getSalary() {
        double baseSalary = getSalary();

        return baseSalary + bonus;
    }
}
```



```
myManager.getSalary();  ?//your program will crash
```

Concept of Keyword Supper

- How to access methods of supperclass?



Solution #3

```
Class Manager extends Employee {
    private double bonus;
    public void setBonus(double b) {
         bonus =b;
    }
    public double getSalary() {
        double baseSalary = super.getSalary();

        return baseSalary + bonus;
    }
}
```

```
myManager.getSalary();   //eventually !
```

Problem #3 and its Solution

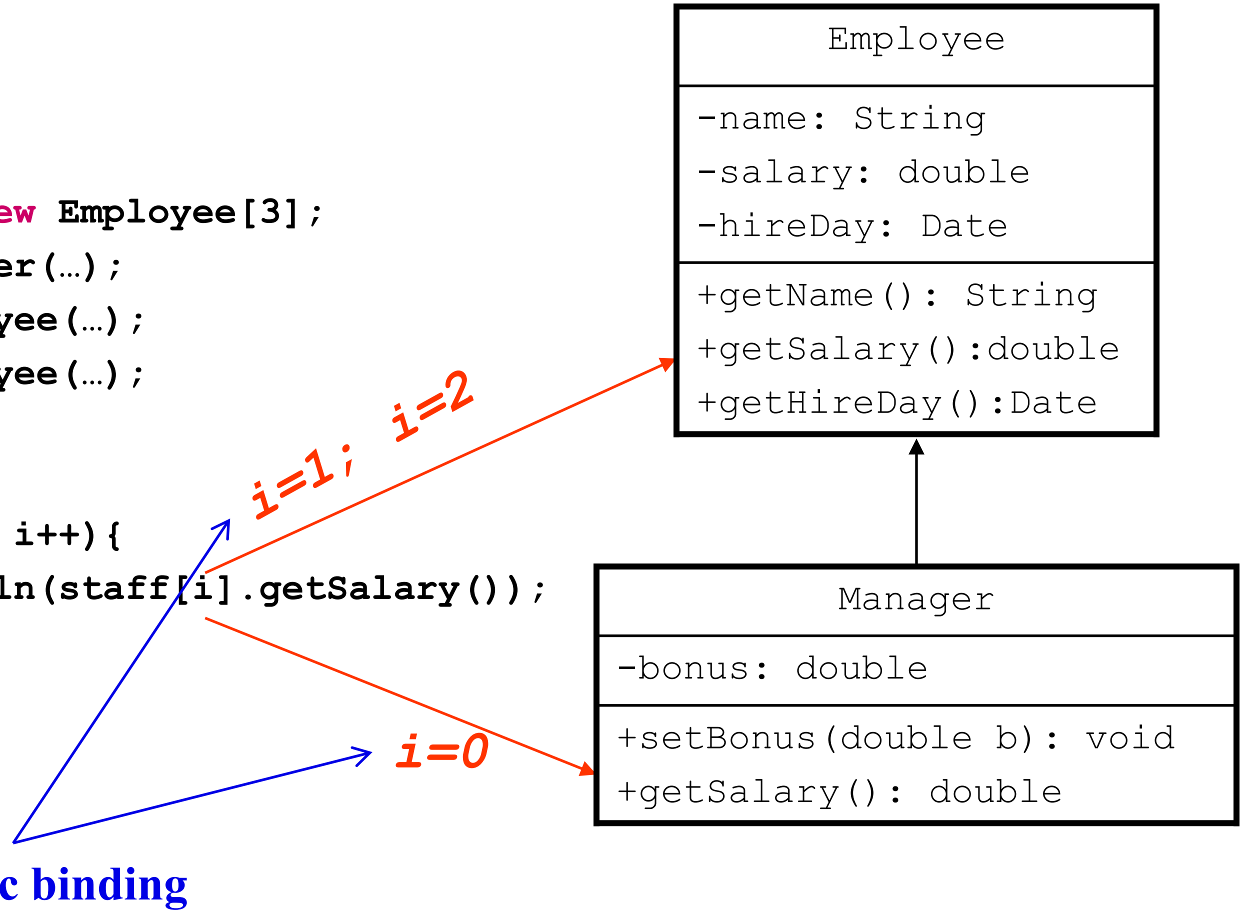
- Payroll system needs to print salary of all employees

```
Employee me = new Employee(…);
Employee you = new Employee(…);
Manager boss = new Manager(…);

//print salary
System.out.println(me.getSalary());
System.out.println(you.getSalary());
System.out.println(boss.getSalary());
```

Is there a better way to do this?

Better Solution: Polymorphism

```
Employee[] staff = new Employee[3];
staff[0] = new Manager(…);
staff[1] = new Employee(…);
staff[2] = new Employee(…);

//print salary
for (int i = 0; i<3; i++){
    System.out.println(staff[i].getSalary());
}
```



Polymorphism

- The ability for objects from separate, yet related classes to receive the same message, but act on it in their own way
- Many different (poly) forms of objects that share a common interface respond differently when a method of that interface is invoked
- Dynamic binding makes programs extensible without the need for modifying existing code

Mechanisms of Polymorphism

- The top class in the hierarchy represents a *common interface* to all classes below it in the hierarchy. This class is referred to as the *base class*.
- All classes below the base class represent a number of forms of objects.
- The user ***must*** refer to all forms of objects using a reference of the ***base class*** type.
  - You cannot assign a superclass reference to a subclass variable ?

Using `instanceof` Operator

Determine if an object is of a particular subclass by using the **`instanceof`** operator

eg:

```
public class Employee extends Object
public class Manager extends Employee
public class Contractor extends Employee

……

public void checkStatus (Employee e) {
    if (e instanceof Manager) {
       // call him `Sir'
    } else
    if (e instanceof Contractor) {
      // keep company secrets
    } else {
      // plain employee, talk freely over a drink
    }
}
```

Casting Objects

- Use **`instanceof`** to test the type of an object
- Restore full functionality of an object by *casting*
- Check for proper casting using the following guidelines:
  - Casts up hierarchy are done implicitly
  - Downward casts must be to a subclass and checked by the compiler
  - The object type is checked at runtime when runtime errors can occur

eg: **`anManager = (Manager) stuff[0];`**

The `Object` Class

The `Object` class is the root of all Java classes !

`java.lang.Object`

A class declaration with no extends clause implicitly uses "extends Object"

Example:

```
public class Employee {
        …
}
```

is equivalent to

```
public class Employee extends Object {
        …
}
```

Java primitive types are not objects

The `equals` Method

- Default implementation of Object
  - If references identical

- State-based equality
  - Two objects are considered equal when they have the same state

Example: `equals` Method

```
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);
   }
}
```

Properties of `equals` Method

As required by Java Language Specification

- It is reflexive
  - for any non-null reference **`x`**, **`x.equals(x)`** should return **`true`**.
- It is symmetric
  - for any references **`x`** and **`y`**, **`x.equals(y)`** should return **`true`** if and only if **`y.equals(x)`** returns **`true`**.
- It is transitive
  - for any references **`x`**, **`y`**, and **`z`**, if **`x.equals(y)`** returns true and **`y.equals(z)`** returns **`true`**, then **`x.equals(z)`** should return **`true`**.
- It is consistent
  - If the objects to which **`x`** and **`y`** refer haven't changed, then repeated calls to **`x.equals(y)`** return the same value.
- For any non-null reference **`x`**, **`x.equals(null)`** should return **`false`**.

toString Method

- Convert an object to a String
- Used during string concatenation
- Override this method to provide information about a user-defined object in readable format

eg:

```
        Date now = new Date();
        System.out.println(now);

    Is roughly equivalent to

        System.out.println(now.toString());
```

Wrapper Class

Primitive Data Type	Wrapper Class
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

eg:

```
int pInt = 500;
Integer wInt = new Integer(pInt);
int p2 = wInt.intValue();

String str = “123”;
int x = Integer.parseInt(str);
```

Autoboxing

- All primitive types have class counterparts (*wrappers*)
  - You may need to convert a primitive type like `int` to an object

```
eg:
    ArrayList<Integer> list = new ArrayList<>()
```

~~ArrayList<int>~~ `list = …`

  - Wrapper classes are *immutable*
  - Wrapper classes are `final`
- The compiler does the boxing and unboxing, not the VM

eg:

```
Integer n =3;
n++;
```

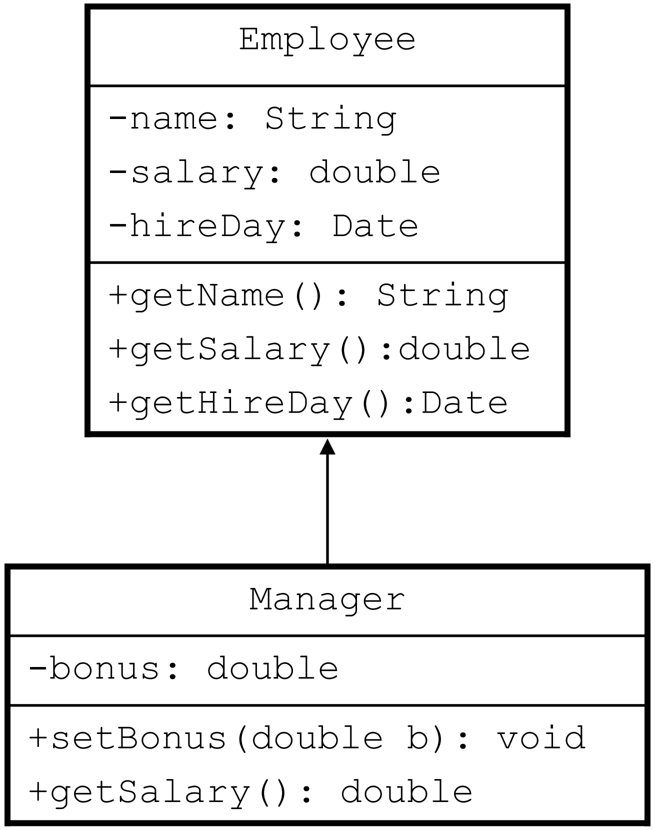
$\leftarrow$ Automatic boxing

$\leftarrow$ Automatic unboxing

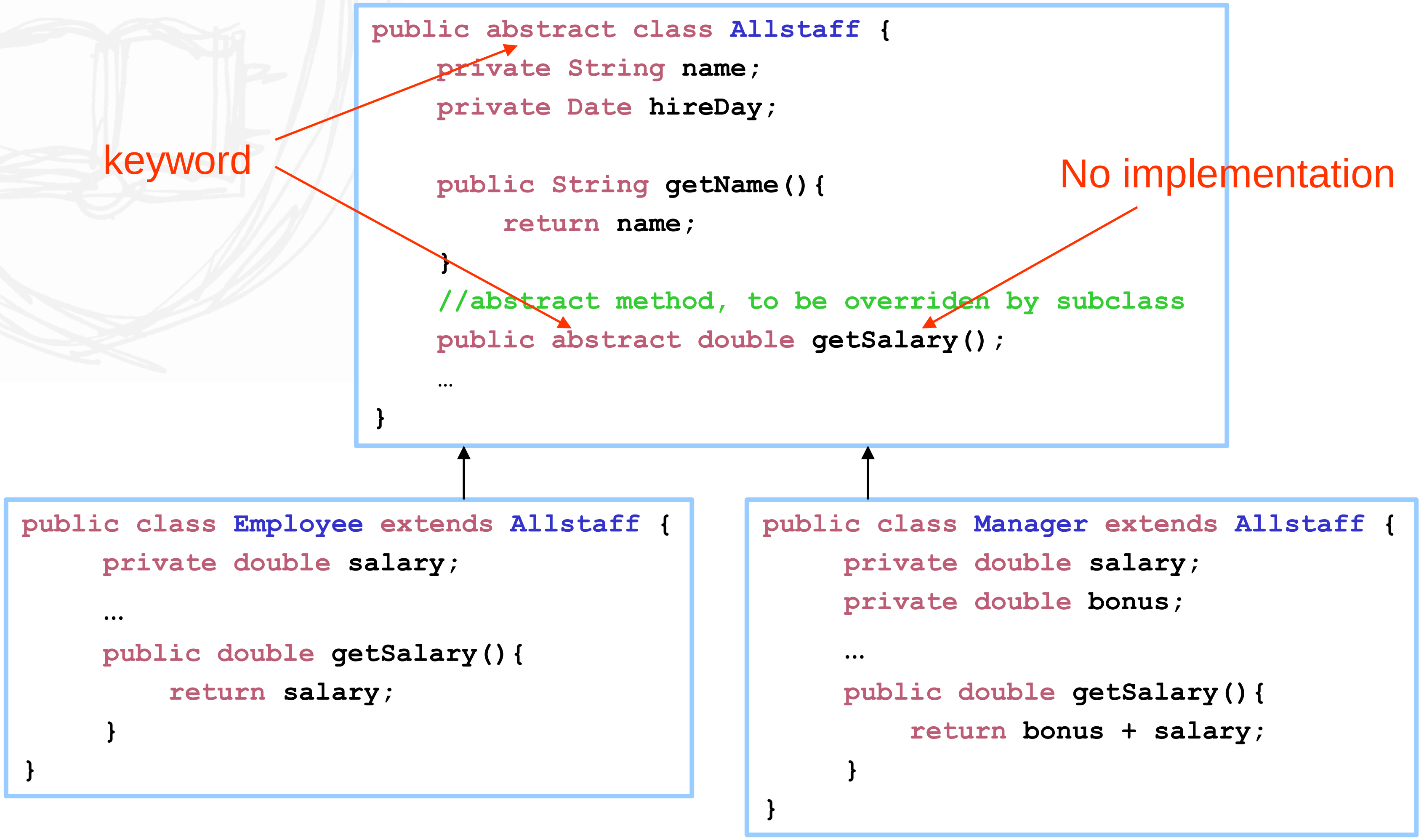
Look at The Problem Again

How to get Manager's salary?

- Used method overriding
  - An instance method in a subclass with the same *signature* (name, the number and type of its parameters) and *return type* as an instance method in the superclass



Another Solution: Abstract Class



```java
public abstract class Allstaff {
    private String name;
    private Date hireDay;

    public String getName(){
        return name;
    }
    //abstract method, to be overriden by subclass
    public abstract double getSalary();
    …
}
```

```java
public class Employee extends Allstaff {
    private double salary;
    …
    public double getSalary(){
        return salary;
    }
}
```

```java
public class Manager extends Allstaff {
    private double salary;
    private double bonus;
    …
    public double getSalary(){
        return bonus + salary;
    }
}
```

Abstract Class

- A superclass declares a method that does not supply an implementation
- The implementation is supplied by the subclass
  - Abstract methods
  - Abstract class

One More Solution: Interface

No modifier

All are public

```
public interface Allstaff {
        String getName()
        double getSalary();
        …
}
```

```
public class Employee implements Allstaff {
      private double salary;
      …
      public double getSalary(){
            return salary;
      }
}
```

```
public class Manager implements Allstaff {
      private double salary;
      private double bonus;
      …
      public double getSalary(){
            return bonus + salary;
      }
}
```

Implementation of Interface

```
public interface aInterface {
    …
}
```

```
public class myClass
    – implements aInterface
    – extends otherClass implements aInterface
    – extends otherClass implements aInterface,
      anotherInterface
```

Interfaces

- A contract between client code and the class that implements the interface
  - Interface is not a class
    - You can never use the `new` operator to instantiate an interface
    - You can `extend` interfaces
    - An interface variable must refer to an object of a class that implements the interface
  - A set of requirements for the classes that want to conform to the interface
- All methods without implementation
- Many classes can implement the same interface
- A class can implement many interfaces
- Program to the interfaces, where possible
  - Take advantage of polymorphism

Uses of Interfaces

- Declaring methods that one or more classes are expected to implement
- Determining an object’s programming interface without revealing the actual body of the class
- Capturing similarities between unrelated classes without forcing a class relationship
- Simulating multiple inheritance by declaring a class that implements several interfaces

The static Keyword

- The **`static`** key word is used as a modifier on variables, methods, and inner classes
- The **`static`** keyword declares the attribute or method is associated with the class as a whole rather than any particular instance of that class
- Static members are called "*class members*", such as "*class attributes*" or "*class methods*"

Class Attributes

- Shared among all instances of a class

```
public class Count{
    private int serialNumber;
    public static int counter = 0;

    public Count() {
        count++;
        serialNumber = counter;
   }
}
```

Class Methods

- Access program code when there is no instance of a particular object available
- Using `static` keyword to mark a class method
- A `static` method cannot access any variables apart from the local variable, `static` attributes and its parameters

Class Methods

```
public class Count{
    private int serialNumber;
    private static int counter = 0;

    public static int getTotalCount(){
       return counter;
    }
    public Count() {
       count++;
       serialNumber = counter;
   }
}
```

Output of TestCounter program:

```
Number of counter is 0
Number of counter is 1
```

```
public class TestCounter{
    public static void main(String[] args){
       System.out.println("Number of counter is "
                             + Count.getTotalCount());
       Count count1 = new Count();
       System.out.println("Number of counter is "
                             + Count.getTotalCount());
    }
}
```

UNIVERSITY OF WOLLONGONG

Invoking Class Attribute and Methods

- Class methods are not part of every object instance and should not be invoked using an object reference variable.
- Use the class name:

```
Classname.attribute;
Classname.method();
```

eg:

```
Math.PI;
Math.sqrt(x);
Integer.parseInt(args[0]);
```

Static Block

- A static block that does not exist within a method body
- Static block code executes only once when the class is loaded
- Used to initialize static attributes

```
public class Count{
   public static int counter;

   static{
      count=Integer.getInteger("myApp.Count.counter").intValue();
   }
}
```

```
public class TestStaticInit{
   public static void main(String[] args){
      System.out.println("counter = "+ Counter.counter);
   }
}
```

```
>java -DmyApp.Count.counter=47 TestStaticInit
Counter = 47
```

Class Attributes and Methods

- Consider declaring a method or attribute `static`, if:
  - Performing the operation on an individual object or associating the attribute with a specific object is not important
  - Accessing the attribute or method before instantiating an object is important
  - The method or attribute does not logically belong to an object

The **`main`** method does not operate on any objects; when a program starts, there are not any objects yet. The **`static`** **`main`** method executes and constructs the objects that the program needs.

The `final` Keyword

Preventing inheritance

- Cannot subclass a **`final`** *class*
- Cannot override a **`final`** *method*
- A **`final`** *variable* is a constant
  - A final variable can be set only once
    - A final variable can be set after declaration, called “blank final variable”
    - A blank final instance attribute must be set in every constructor
    - A blank final method variable must be set in the method body

Inner Classes

- An ***inner class*** is a class declared inside another class.
- An inner class can be declared inside a class but outside a method, or inside a method of a class
  - Allow a class definition inside another class definition
  - Group classes that logically belong together
  - Have access to their enclosing class's scope

Example: Inner Class

```java
public class Outer {
    private int size;

    /* Declare an inner class called "Inner" */
    public class Inner{
        private int size;
        public void doStuff(int size) {
            size++;               //local
            this.size++;          //Inner attribute
            Outer.this.size++;    //Outer attribute
        }
    }

    public void testTheInner() {
        Inner i = new Inner();
        i.doStuff(20);
    }
}
```

```java
Outer outer = new Outer()
Outer.Inner inner = outer.new Inner();
```

Class Design Hints

- Always keep data private
  - Doing anything else violates encapsulation
- Always initialize data
  - Don’t rely on the defaults; supplying defaults or setting defaults in all constructors
- Don’t use too many basic types in a class

```
private String street;
private String city;
private String state;
private int postcode;
```

$\longrightarrow$ `class Address`

- Not all fields need individual field accessors and mutators

Class Design Hints

- Use a standard form for class definitions
  - Java coding style: fields first then methods
  - Others: public, package scope, private
  - No universal agreement; important to be consistent
- Break up classes that have too many responsibilities

```
public class CardDeck //bad design
{
  private int[] value;
  private int[] suit;

  public CardDeck() {…}
  public void shuftle() {…}
  public int getTopValue() {…}
  public int getTopSuit() {…}
  public void draw() {…}
}
```

$\rightarrow$

```
public class CardDeck
{
  private Card[] cards;

  public CardDeck() {…}
  public void shufle() {…}
  public Card getTop() {…}
  public void draw() {…}
}

public class Card
{
  private int[] value;
  private int[] suit;

  public Card(int aValue, int aSuit) {…}
  public int getValue() {…}
  public int getSuit() {…}
}
```

- Make the names of your classes and methods reflect their responsibilities

Design Hints for Inheritance

- Place common operations and fields in the superclass
- Don’t use protected fields
  - Anyone can form a subclass of your class
- Use inheritance to model the “is-a” relationship
- Don’t use inheritance unless all inherited methods make sense
- Don’t change the expected behavior when you override a method
- Use polymorphism, not type information