CPS209: COMPUTER SCIENCE II

Inheritance

Inheritance Summary

- **Inherit** methods and variables from an existing class
- **Extend** classes by adding new methods and variables
- Example: a savings account "**is a**" bank account with interest

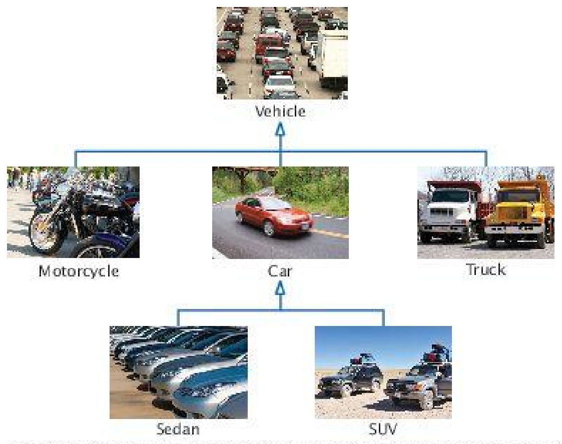
```
class SavingsAccount extends BankAccount
{
   // automatically "inherit" existing instance variables
   // automatically "inherit" existing methods
   // add new methods
   // add new instance variables
   // override existing methods
}
```

Inheritance

- Inheritance: is-a relationship
  - Java **Interface** (next lecture): use-a relationship
  - Objects are containers of variables: has-a relationship
    - e.g. a bank account object **has a** (contains a) *balance* variable and an *accountNumber* variable
- One advantage of inheritance is *code reuse*
- *Program organization* is another advantage

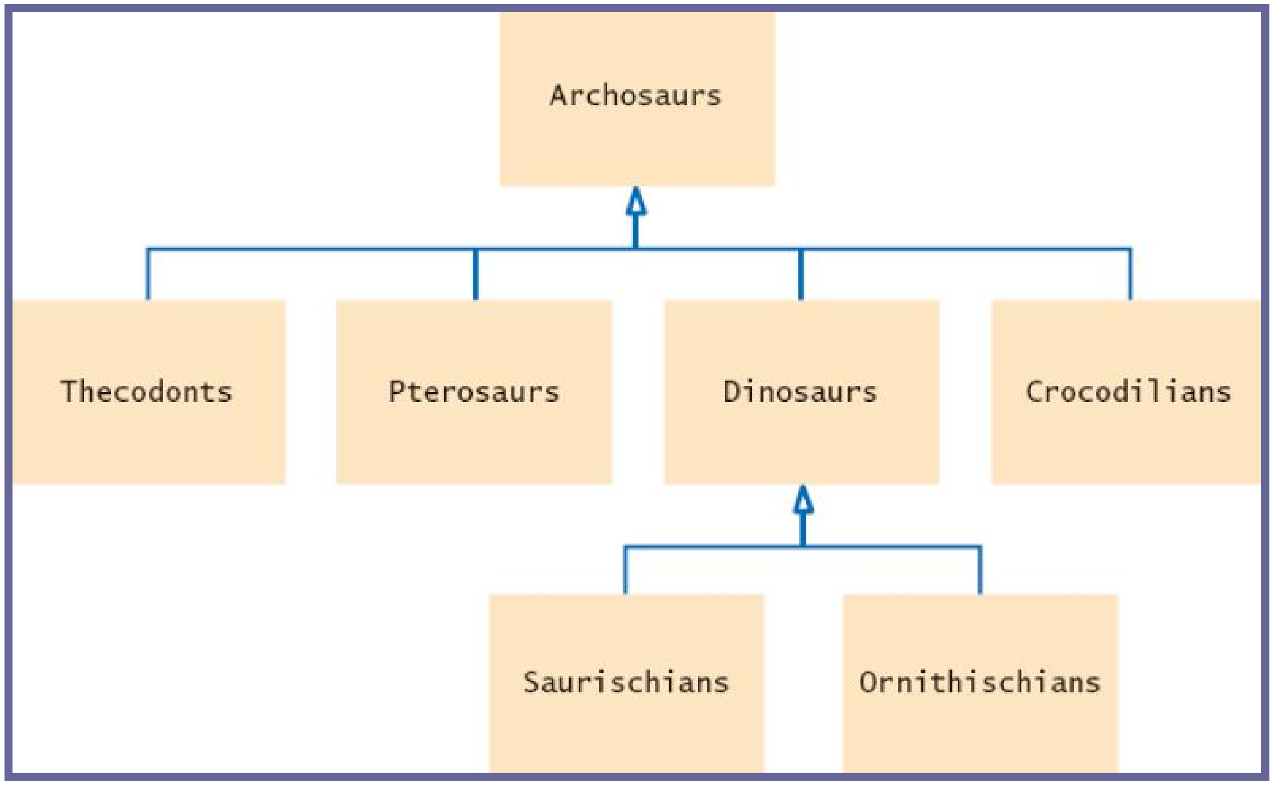
Inheritance Hierarchies

- Inheritance: the relationship between a more *general* class (superclass) and a more *specialized* class (subclass).
  - Subclass inherits data (variables) and behavior (methods) from the superclass.
- Cars share the common traits of all vehicles
  - Example: the ability to transport people from one place to another

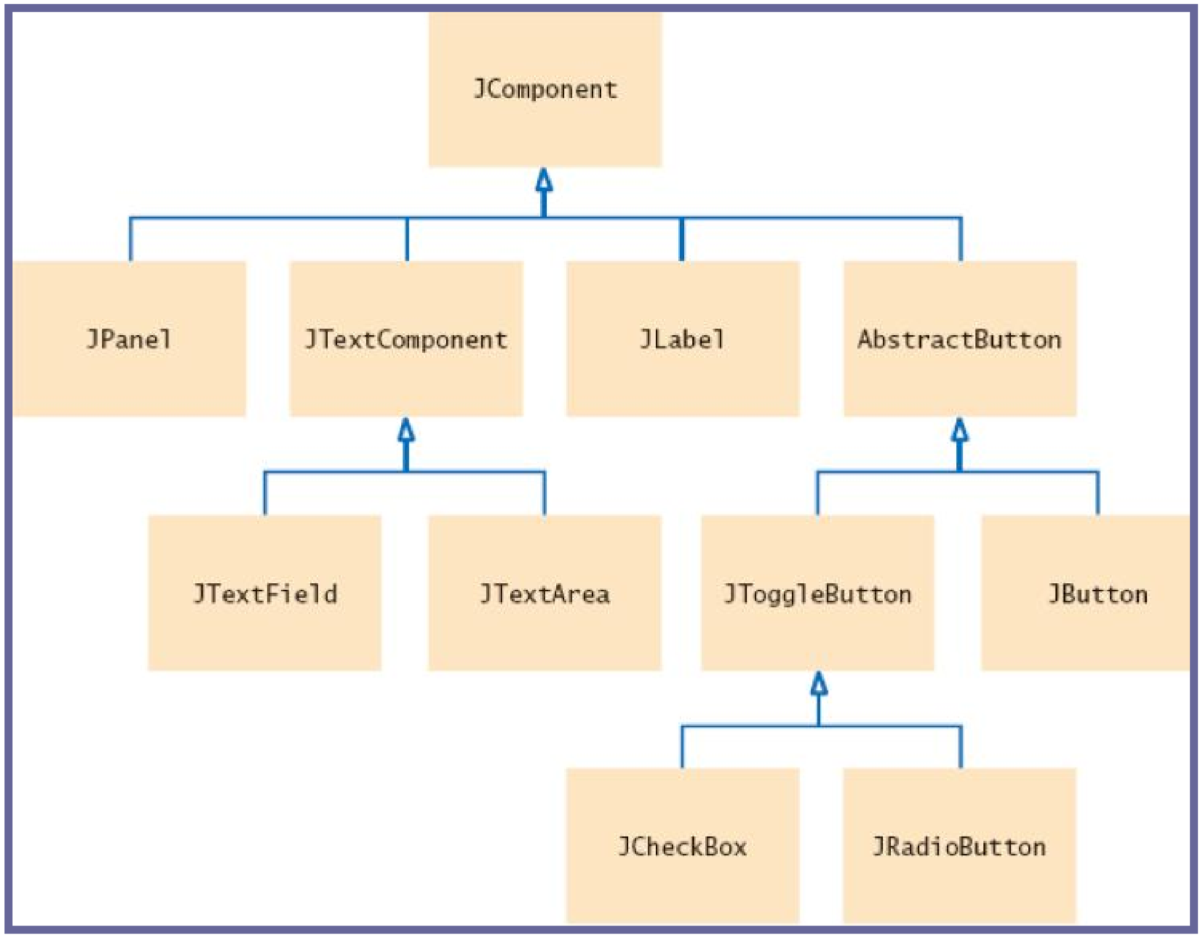


Inheritance Hierarchies

- Sets of classes can form complex inheritance hierarchies
- Example:

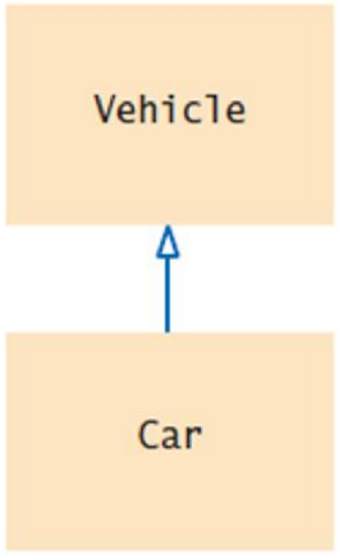


Inheritance Hierarchies: Swing Hierarchy



Inheritance Diagram

- The class `Car` inherits from the class `Vehicle`
- The `Vehicle` class is the superclass
- The `Car` class is the subclass



Inheritance Hierarchies

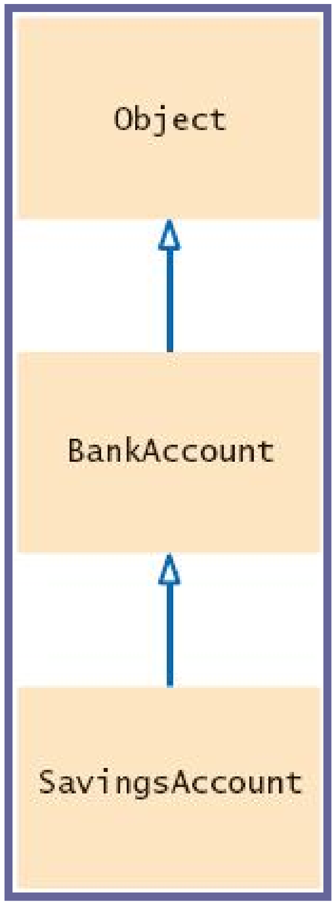
- Inheritance lets you **reuse** code instead of duplicating it.

- Two types of reuse:
  1. A subclass automatically inherits the methods of the superclass
  2. Because a car **is a** special kind of vehicle, we can use a `Car` object in algorithms that manipulate `Vehicle` objects
     - This will be explained later

- The substitution principle:
  - You can always use a subclass object when a superclass object is expected as a parameter to a method (More on this later).
    - e.g. a method that processes Vehicle objects can handle any kind of subclass of vehicle

Inheritance Diagram

- Every class automatically extends the built-in **Object** class
- Superclass **Object** (weird name, I know) has several methods but no variables
- More on this later



Inheritance

- In a subclass you:
  1. *Inherit* all instance variables of superclass
  2. *Inherit* all methods of superclass

- In a <u>subclass</u>, you can choose to:

  1. add new instance variables (fields),
  2. add new methods,
  3. *override* the inherited methods

Bank Account - example

```
public class BankAccount
{
  private long accountNumber;
  private double balance;

  public BankAccount(long accountNumber)
  {
    this.accountNumber = accountNumber;
    balance = 0;
  }

  public double getBalance()
  {
    return balance;
  }
  // continued in the next slide...
```

Bank Account - example

```
   public void deposit(double amount)
   {
      balance += amount;
   }

   public void withdraw(double amount)
   {
      balance -= amount;
   }

   public void transfer(double amount, BankAccount targetAccount)
   {
      this.withdraw(amount);
      targetAccount.deposit(amount);
   }
}
```

Bank Account – usage example

```
class BankAccountTester
{
  public static void main( String[] args)
  {
    BankAccount aliceAcc = new BankAccount(1398723);
    BankAccount bobAcc   = new BankAccount(1978394);

    aliceAcc.deposit(900);
    aliceAcc.transfer(700,bobAcc);
    // Alice’s balance = 200 ; Bob’s balance = 700
  }
}
```

Inheritance: Simple Example – Adding a new variable and method

```
public class SavingsAccount extends BankAccount
{
  private double interestRate;

  public SavingsAccount(long account, double rate)
  {
    super(account);
    interestRate = rate;
  }

  public void addInterest()
  {
     double interest = getBalance() * interestRate / 100;
     deposit(interest);
  }
}
```

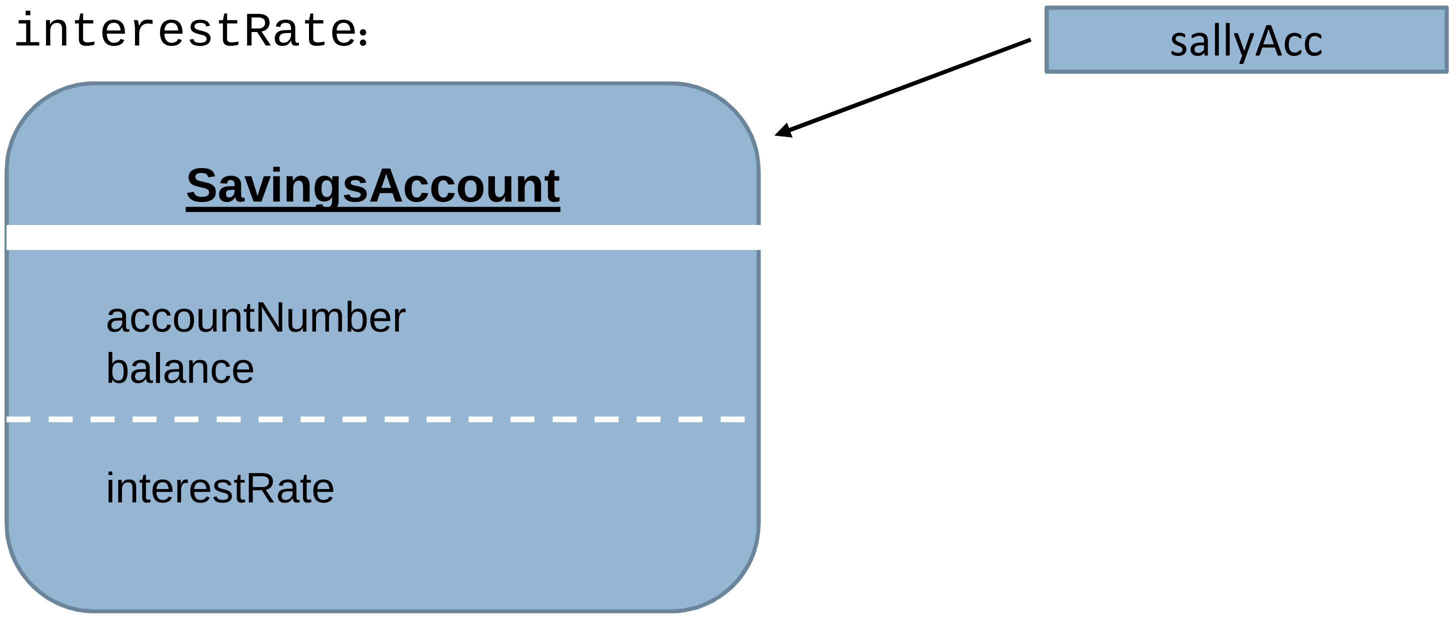
Inheritance: Simple Example

```
class SavingsAccountTester
{
  public static void main( String[] args)
  {
    BankAccount aliceAcc     = new BankAccount(1398723);
    SavingsAccount sallyAcc = new SavingsAccount(1978394,5.5);

    sallyAcc.deposit(300);
    aliceAcc.deposit(900);

    sallyAcc.addInterest();
    aliceAcc.transfer(700,sallyAcc);
    System.out.println("Sally Balance = " + sallyAcc.getBalance());
  }
}
```

Inheritance: Simple Example

- `SavingsAccount` object **automatically** inherits the `balance` and `accountNumber` instance variables from `BankAccount`,

- and adds one additional instance variable: `interestRate`:



Inheritance: Simple Example

- *Encapsulation*: `addInterest` method calls `getBalance` rather than directly updating the `balance` field of the superclass (balance field is `private` in superclass)

- Note also that `addInterest` can call `getBalance` without specifying a "this" implicit parameter (the calls apply to the same object)

Self-Check

1. What instance variables does an object of class `SavingsAccount` have?

2. Name four methods that you can apply to `SavingsAccount` objects

Answers

1. Three instance fields: `balance`, `accountNumber` and `interestRate`.
2. `deposit()`, `withdraw()`, `getBalance()`, and `addInterest()`.

Inheriting Methods

- 1) Inherit method:
  - Don't supply a new implementation of a method that exists in superclass - just use the method in superclass **as if it was part of subclass!**
  - i.e. Superclass method can be applied to the subclass objects
    - e.g. getBalance() method

Inheriting Methods

- 2) Add new method:
  - Supply a new method that doesn't exist in the superclass
  - New method can be applied <u>only to subclass objects</u>

- See Example SavingsAccount addInterest() method

Override a Method

- 3) Override a method:
  - Write a different implementation of a method that already exists in the superclass
  - Must have same signature (same name and same parameter types)
  - If a method is applied to an object of the subclass type, the overriding method is executed (not the superclass method)

- See Example: SavingsAccount deposit() method

Inheriting Instance Fields (Variables)

- **Can't (i.e. should never!) override superclass variables**
  - **Creates a "shadow" variable (bad!)**

- Inherit a field (instance variable): All variables from the superclass are **automatically** inherited

- Add new variable: Supply a new variable that doesn't exist in the superclass

Inheriting Instance Fields (Variables)

- What if you define a new field with the same name as a superclass field (i.e. shadow field)?
  - Each object would have two instance fields of the same name
  - Fields can hold different values
  - **Legal but extremely undesirable!!! Don't ever do it!!**

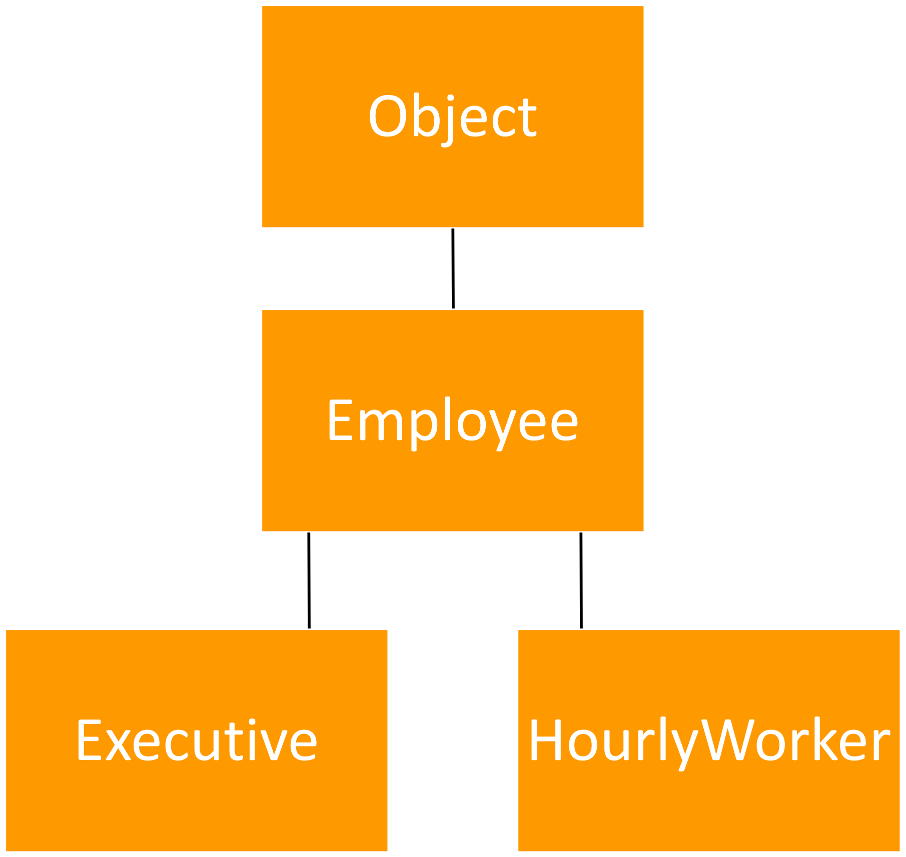
Inheritance: Complete Example

- Example: Executive is a Employee, HourlyWorker is a Employee

- class Executive extends Employee
```
{
   new methods
   new instance fields
}
```

- All methods of Employee are automatically inherited
- All instance variables of Employee are automatically inherited
- Ok to call getName(), getTotalHours(), computePay() etc. from Executive object
- Extended class = superclass, extending class = subclass

Complete Example: Employee



Class Employee

```
public class Employee
{
  static public final int STANDARD_HOURS_PER_WEEK = 35;

  public static int numEmployees = 0;

  private String name;
  private double payRate;

  public Employee()
  {
    name    = "";
    payRate = 0;
  }

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

Class Employee

```
public void setName(String name)
{
   this.name = name ;
}

public String getName()
{
   return name ;
}

public void setPayRate( double newRate)
{
   payRate = newRate ;
}

public double getPayRate()
{
   return payRate ;
}
```

Class Employee

```
    // Compute and return an employee's **weekly** pay

    public double computePay()
    {
      return payRate * STANDARD_HOURS_PER_WEEK;
    }
}
```

Adding Variables and Methods

```
public class Executive extends Employee
{
  private double  bonus;
  private boolean payBonus;

  public void setBonus(double bonus)
  {
    this.bonus = bonus;
  }

  public double getBonus()
  {
    return bonus;
  }

  public void setPayBonus()
  {
    payBonus = true;
  }
```

Overriding Methods

```
// class Executive continued...
// Override the computePay() method
// This version is ok but not as good as version on slide 34

public double computePay()
{
  if (payBonus)
  {
    double pay = STANDARD_HOURS_PER_WEEK*getPayRate()+ bonus;
    payBonus = false;
    return pay;
  }
  else
    return STANDARD_HOURS_PER_WEEK * getPayRate();
}
```

Encapsulation: inherited fields are private

- Consider computePay() method of Executive
- Can't just use/access inherited “payRate” variable directly inside computePay()
  - payRate is declared a *private* field of the superclass
- Subclass must use public interface of superclass
- Can get around this by declaring variables *public* or *protected* in superclass (but don’t!)

Overriding Methods

- Let’s look at alternate version of computePay() in class Executive:
- We want to execute the computePay() method of the superclass Employee
- This is common as in our overridden method we just want to add some extra code then call the normal computePay() method
- If we just try to call it as below, we will end up have computePay() call itself endlessly

```
public double computePay()
{
  if (payBonus)
  {
    double pay = computePay() + bonus; // ERROR!!
    payBonus = false;
    return pay;
  }
  else
    return computePay(); // ERROR!!
}
```

Overriding Methods

- What we need to do is to use super.

```
public class Executive extends Employee
{

……….

  public double computePay()
  {
    if (payBonus)
    {
      double pay = super.computePay() + bonus;
      payBonus = false;
      return pay;
    }
    else
      return super.computePay();
  }
```

Invoking a superclass method

- Can't just call **computePay()** inside **computePay()** method of class Executive
- That is the same as this.computePay()
- Calls the same method (infinite recursion!!!)
- Instead, invoke *superclass method* **super.computePay()**
- Now correctly calls computePay() method of superclass Employee

Creating subclass objects: example

```
public class ExecutiveTester
{
  public static void main(String[] args)
  {
    Executive exec = new Executive();

    exec.setName(“bossman”);
    exec.setPayRate(150.0);

    // Calls class Executive computePay
    double weeklyPay = exec.computePay();

    System.out.println(exec.getName() + “makes “ + weeklyPay);
  }
}
```

Superclass Construction

```
public class Executive extends Employee
{
 private double bonus;
 private boolean payBonus;

  public Executive(String name, double payRate, double bonus)
  {
     // initialize inherited variables by calling superclass
     // constructor method
     super(name, payRate);

     // initialize new variables
     this.bonus = bonus;
     payBonus   = false;
  }
        ...
}
```

Pass parameters to superclass constructor

- Must be the *first* statement in subclass constructor

Converting from subclasses to superclasses

- Ok to convert subclass reference to superclass reference (not other way around though!)

-
  ```
  Executive exec  = new Executive();
  Employee  empl  = exec;
  Object    o     = exec;
  ```

- However, Superclass references don't know the full story:

```
o.setBonus(1000.0);     // ERROR!
empl.setBonus(1000.0); // ERROR!
```

Subclass HourlyWorker

```
public class HourlyWorker extends Employee
{
   private double hoursPerWeek;

   public HourlyWorker(String name, double payrate, double hoursPerWeek)
   {
      super(name,payrate);
      this.hoursPerWeek = hoursPerWeek;
   }

   public double getHoursPerWeek()
   {
      return hoursPerWeek;
   }
   public void setHoursPerWeek(double hours)
   {
      hoursPerWeek = hours;
   }
}
```

Polymorphism

```
Employee     employee;
Executive    exec   = new Executive(“bossman”,150.0);
HourlyWorker worker = new HourlyWorker(“joe”,10.0,20);

employee        = exec;
double weeklyPay = employee.computePay();

employee  = worker;
weeklyPay = employee.computePay();
```

- JVM looks at **type of object the reference variable is pointing to (i.e at run time) and executes the correct method for that object**
- If reference variable employee is pointing to an Executive object, execute the computePay() method of Executive
- If reference variable employee is pointing to an HourlyWorker object, execute the computePay() method of HourlyWorker

Polymorphism

- Remember: ok to convert subclass reference to superclass reference but not other way around

```
Executive exec  = new Executive();
Employee  empl  = exec;
```

- If you want to convert superclass ref. variable to subclass, you must cast:
  - Must make sure superclass ref. variable is referring to a subclass object!!

```
Executive exec2;
if (empl instanceof Executive)
  exec2 = (Executive)employee;
```

Access control level

- public
- private
- protected (accessible by subclasses and package)
- package access (the default, no modifier)

Recommended Access Levels

- Fields: Always private
  - exception: public static final constants
- Methods: public or private
- Classes: public or package
- Don't use protected
- Beware of accidental package access (forgetting public or private)

Object Superclass

Object: The superclass of all classes

- All classes extend Object
- Most useful methods in class Object:
- String toString()
- boolean equals(Object otherObject)
- Object clone()

Object: toString()

Object: toString()

- Returns a string representation of the object
- Useful for debugging
- Example: toString() in class Rectangle returns something like:
  - java.awt.Rectangle[x=5,y=10,width=20,height=30]

- toString() used by concatenation operator:
  - BankAccount b = new BankAccount(9876543,300);
  - String test = “xyz” + b; // means String test = “xyz” + b.toString();

- toString() in class Object returns class name and object address:
  - Employee@d2460bf

Object: The superclass of all classes

- **It is common to override toString():**

```
public class Employee
{
  public String toString()
  {
    return "Employee[name=" + name + " " + "payRate=" +  payRate + "]";
  }
  . . .
}

public class Employee
{
  public String toString()
  {
    return getClass() + "[name=" + name + " " + "payRate=" +  payRate + "]";
  }
  . . .
}
```

Object: The superclass of all classes

- **It is common to override toString():**

```
public class Employee
{
   public String toString()
   {
      return "Employee[name=" + name + “ " + "payRate=” +  payRate + "]";
   }
   . . .
}

public class Employee
{
   public String toString()
   {
      return getClass() + “[name=" + name + “ " + "payRate=” +  payRate + "]";
   }
   . . .
}
```

Object: equals(Object other)

- equals() tests for equal contents

- Examples:
  - BankAccount => compare balance
  - Coin => compare value
  - Employee => compare name and/or payRate

- == tests for equal memory locations of objects!!

- Must cast the "Object other" parameter to subclass

Object: equals(Object other)

```
public class Employee
{
  public boolean equals(Object other)
  {
     Employee otherEmpl = (Employee) other;

     return name.equals(otherEmpl.name) && payRate == otherEmpl.payRate;
  }
}
```

- **NOTE**: Strings are objects so must use equals method in class String to compare two strings!!! **Cannot just write**:
  `return name == otherEmpl.name && payRate == otherEmpl.payRate;`

Object: equals(Object other)

```
public class ExecutiveTester
{
  public static void main(String[] args)
  {
      Executive exec1 = new Executive();
      exec1.setName(“bossman”);
      exec1.setPayRate(150.0);

      Executive exec2 = new Executive();
      exec2.setName(“bigger bossman”);
      exec2.setPayRate(250.0);

      if (exec1 == exec2)
        System.out.println(“two references to same object”);

      if (exec1.equals(exec2))
        System.out.println(“two objects with same name and payrate”);
  }
}
```

Abstract Methods and Classes

- When you extend a class, you have choice to override a method or not.
- Sometimes want to force the subclass creator to override a method
  - method might be common to all subclasses so should define it in superclass but there is no good default implementation
  - Example: area() method of superclass Shape and subclasses (Circle,Rectangle)
- You can make the method abstract (just signature, no body):
  - abstract public double area();
- Now subclass programmer must implement method area()

Abstract Methods and Classes

- Once a class has at least one abstract method, whole class must be made abstract. Example:
  - **abstract public class BankAccount**
- Can't create objects of abstract classes - they are used to define a common interface and behavior for all subclasses
- Abstract classes can have instance variables, abstract methods, concrete methods
- A subclass inherits instance variables and methods. Must implement abstract methods. If subclass does not implement abstract method, it too becomes abstract

Special Topic 9.4

- Final Methods and Classes
  - You can also ***prevent*** programmers from creating subclasses and override methods using `final`.
  - The String class in the Java library is an example:

```
public final class String { . . . }
```

  - Example of a method that cannot be overridden:

```
public class SecureAccount extends BankAccount
{
   . . .
   public final boolean checkPassword(String password)
   {
      . . .
   }
}
```