Method

A **method** is a collection of statements that are grouped together to perform an operation.
For example: This Shape class contains a method to compute area of a circle and display the result.

```
public class Shape {
  // method to compute area of a circle
  public void computeAreaOfCircle(double radius) {
    double area = Math.PI * Math.pow(radius, 2);
    System.out.printf("Area of a circle with radius %.1f is %.1f\n", radius, area);
  }
}
```

Notice that there is no main method in the Shape class because the class is supposed to be created for reusability of tasks or methods defined in it. The purpose of main method is to execute a Java program, therefore; it should not be included here. Using a method will be explained later in this material.

To create a method, ***Method signature or Method header*** must be constructed and includes the followings.

- Access Modifier
- Return Type
- Name of Method
- Parameter list (optional)

1. **Access Modifier** is used to define a level of access for class, method, object and variable. There are four types of access modifiers in Java.
   a. public → Class, method, object or variable with public keyword can be accessed by any class from anywhere.
   b. private → Class, method, object or variable with private keyword can be accessed by its own class.
   c. protected → Class, method, object or variable with protected keyword can be accessed by any class in the same package.
   d. default→ Class, method, object or variable with none of the three above can be accessed by any class in the same environment.

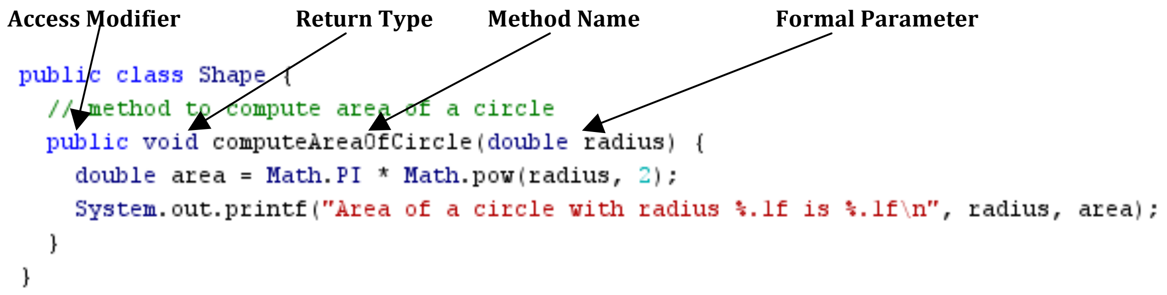
2. **Return type** is used to define a return value when a method is performed. There are two types of return type.
   a. void → Method with void keyword returns nothing to a method's caller at the end of method's performance.
   b. return → Method can include any type of object or variable to indicate a return value to a method's caller at the end of method's performance.

3. **Name of method** normally begins with verb and object in English language to define its action based on Java conventional way of naming. There must not contain any space in the name. Parenthesizes at the end of name specifies that this is a method not a class or object or variable.

   For instances: computeAreaOfCircle(), computeAreaOfRectangle(), computeArea(), getArea(), i.e.,

4. **Parameter** is used to define a list of input enclosed in parenthesizes in the method header or method caller. There are two types of parameters.
   a. Formal parameter → This is a list of variable with its type and name defined in method signature or method header as method's requirement that method's caller must pass.
   b. Actual parameter → This is a list of actual value included in method's caller passed to a method being called so it can be processed for the task.

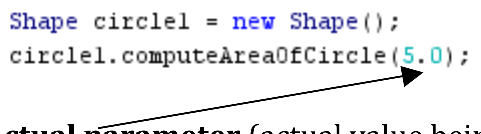
   Example of Method's Creation in Shape class



```
public class Shape {
  // method to compute area of a circle
  public void computeAreaOfCircle(double radius) {
    double area = Math.PI * Math.pow(radius, 2);
    System.out.printf("Area of a circle with radius %.1f is %.1f\n", radius, area);
  }
}
```

   Example of Method's Caller inside another Java Class

   - Shape object named circle1 must be created as a Shape class's reference
   - The object named circle1 is used with dot notation to call a method



```
Shape circle1 = new Shape();
circle1.computeAreaOfCircle(5.0);
```

   Actual parameter (actual value being passed to formal parameter as an input to the method) The decimal number 5.0 is passed to computeAreaOfCircle method as a radius for computation.

Example of Shape and ShapeExample classes
Shape.java contains four methods to perform four distinctive tasks

```java
public class Shape {
  // method to compute area of a circle
  public void computeAreaOfCircle(double radius) {
    double area = Math.PI * Math.pow(radius, 2);
    System.out.printf("Area of a circle with radius %.1f is %.1f\n", radius, area);
  }

  // method to compute area of a rectangle
  public void computeAreaOfRectangle(int length, int width) {
    int area = length * width;
    System.out.printf("Area of a rectangle with length %d and width %d is %d\n", length, width, area);
  }

  // method to compute area of a circle and return the result to the caller
  public double getArea(double radius) {
    double area = Math.PI * Math.pow(radius, 2);
    return area;
  }

  // method to compute area of a rectangle and return the result to the caller
  public int getArea(int length, int width) {
    int area = length * width;
    return area;
  }
}
```

ShapeExample.java includes main method and multiple callers to perform individual tasks.

```java
public class ShapeExample {
  public static void main(String [] args) {
     // call computeAreaOfCircle method without result returned
     Shape circle1 = new Shape();
     circle1.computeAreaOfCircle(5.0);

      // call computeAreaOfRectangle method without result returned
     Shape rectangle1 = new Shape();
     rectangle1.computeAreaOfRectangle(40, 100);

      // call getArea method one actual paramters for area of circle and recieve result back
     Shape circle2 = new Shape();
     double areaCircle = circle2.getArea(7.0);
     System.out.printf("Area of Circle 2 is %.1f\n", areaCircle);

     // call getArea method with two actual paramters for area of rectangle and recieve result back
     Shape rectangle2 = new Shape();
     int areaOfRectangle = rectangle2.getArea(100, 50);
     System.out.printf("Area of Rectangle 2 is %d\n", areaOfRectangle);
  }
}
```

Run ShapeExample

```
> run ShapeExample
Area of a circle with radius 5.0 is 78.5
Area of a rectangle with length 40 and width 100 is 4000
Area of Circle 2 is 153.9
Area of Rectangle 2 is 5000
>
```

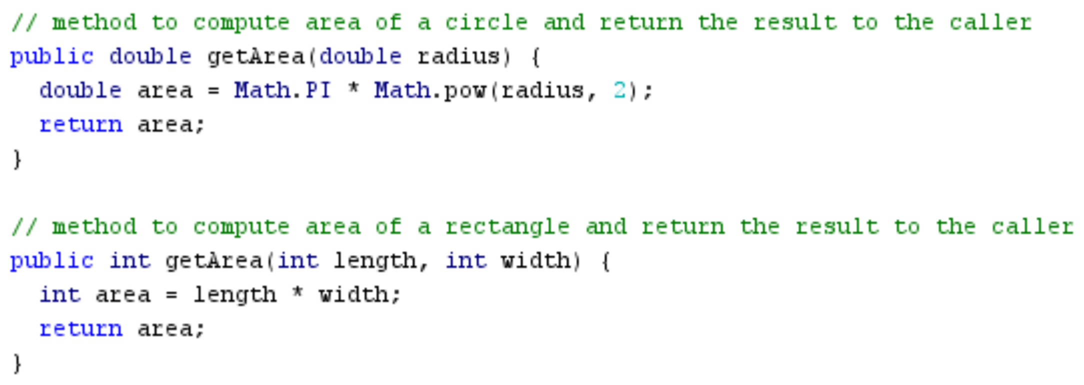
Three principles of Object Oriented Programming Languages

1. **Encapsulation** refers to information hiding which means only restricting modification but not usability. For instance, we know why and how to use a Scanner but not how it is created.

2. **Polymorphism** refers to overloading and overriding method performance.
    a. **Overloading** means multiple methods in a same class can have a same method's name but different parameter list. They all perform different tasks.

    Example of Polymorphism (Overloading)

    These two methods are both in Shape class with the same name, getArea. However, they have different parameter list, return type and perform different tasks.

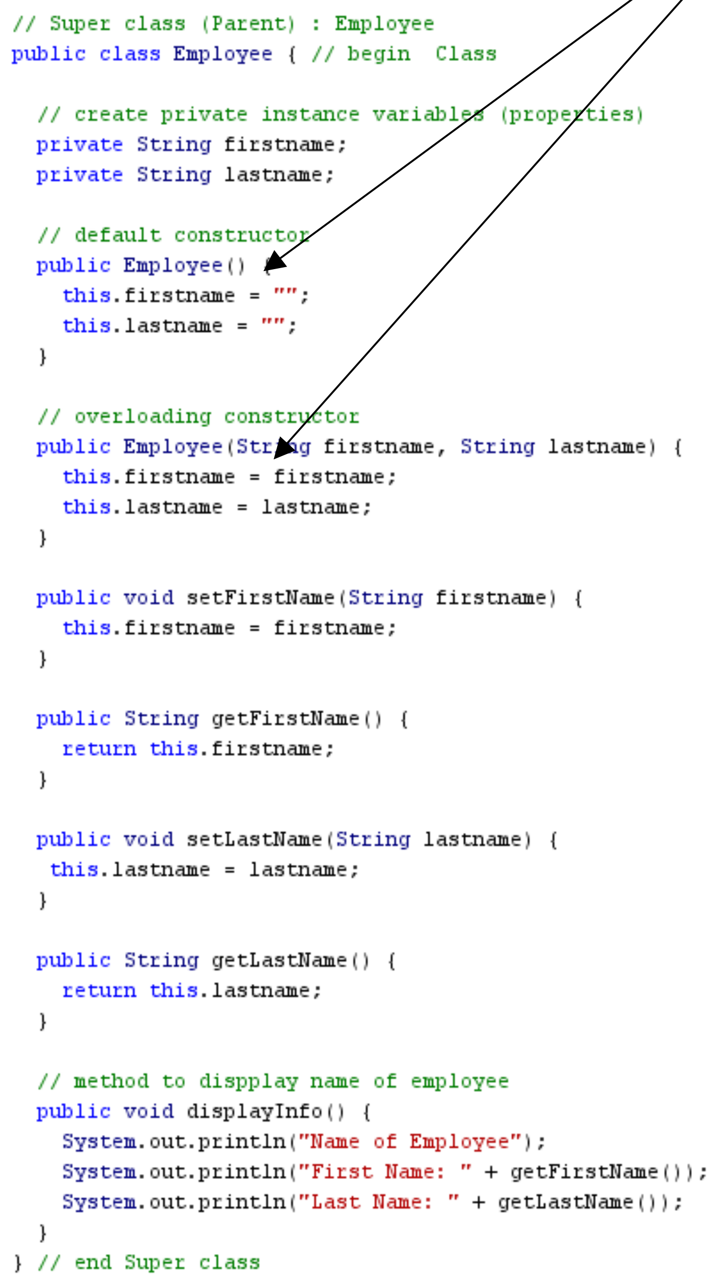


```java
// method to compute area of a circle and return the result to the caller
public double getArea(double radius) {
  double area = Math.PI * Math.pow(radius, 2);
  return area;
}

// method to compute area of a rectangle and return the result to the caller
public int getArea(int length, int width) {
  int area = length * width;
  return area;
}
```

    b. **Overriding** means multiple methods in two classes with relationship can have a same method's name. They perform different tasks. Example of overriding can be found in the next example.

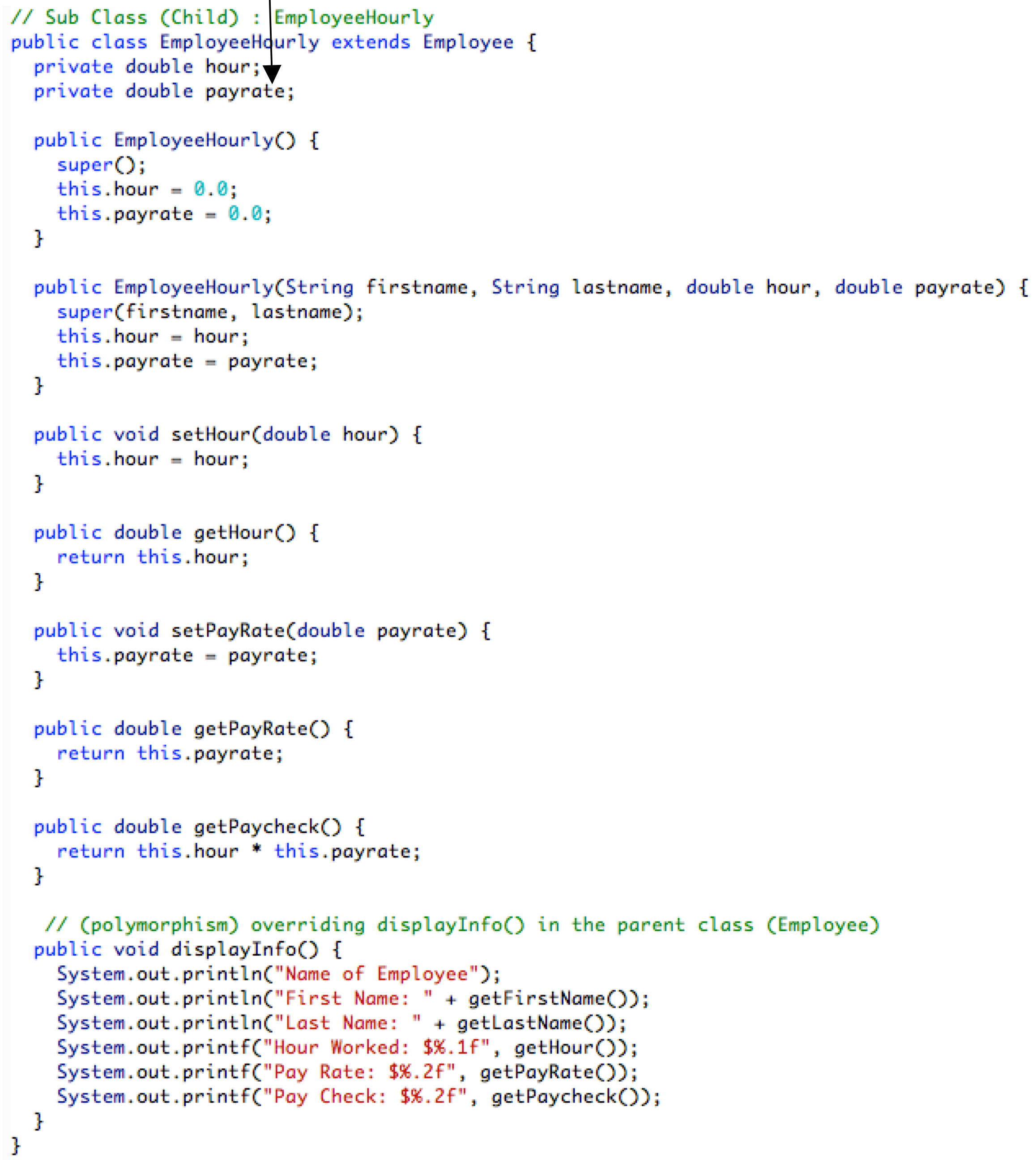
3. **Inheritance** refers to super (parent) class and sub (child) class where sub class can inherit access to public method, object and variable from its super class.

Example of Inheritance
This is a Super class (Parent), **Employee.java**, with private variables or attributes for first name and last name but four public methods which can be accessed by its sub classes. There is also overloading **Constructor** (A special kind of method that has no return type (not even void) and has the same name as its own class).



```
// Super class (Parent) : Employee
public class Employee { // begin  Class

  // create private instance variables (properties)
  private String firstname;
  private String lastname;

  // default constructor
  public Employee() {
    this.firstname = "";
    this.lastname = "";
  }

  // overloading constructor
  public Employee(String firstname, String lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }

  public void setFirstName(String firstname) {
    this.firstname = firstname;
  }

  public String getFirstName() {
    return this.firstname;
  }

  public void setLastName(String lastname) {
   this.lastname = lastname;
  }

  public String getLastName() {
    return this.lastname;
  }

  // method to dispplay name of employee
  public void displayInfo() {
    System.out.println("Name of Employee");
    System.out.println("First Name: " + getFirstName());
    System.out.println("Last Name: " + getLastName());
  }
} // end Super class
```

This is a Sub Class (Child), **EmployeeHourly.java**, with private variables or attributes for hour and rate. The keyword “**extends**” makes it a sub class of Employee. It can then inherit access to public methods from its super class. Therefore, the sub class now owns eight public methods (4 public methods of the parent (Employee) and 4 public methods in its class (EmployeeHourly).

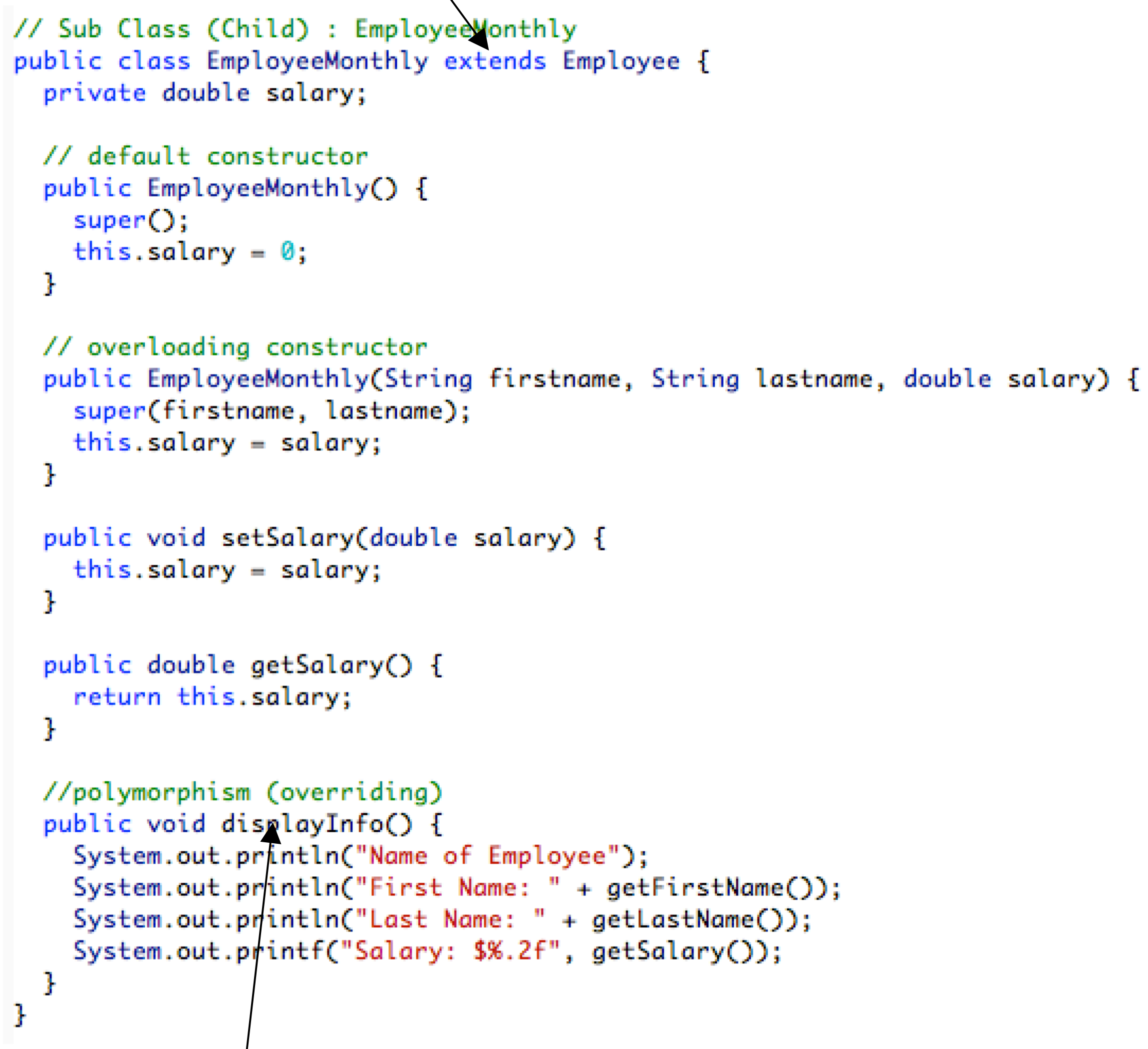


```java
// Sub Class (Child) : EmployeeHourly
public class EmployeeHourly extends Employee {
  private double hour;
  private double payrate;

  public EmployeeHourly() {
    super();
    this.hour = 0.0;
    this.payrate = 0.0;
  }

  public EmployeeHourly(String firstname, String lastname, double hour, double payrate) {
    super(firstname, lastname);
    this.hour = hour;
    this.payrate = payrate;
  }

  public void setHour(double hour) {
    this.hour = hour;
  }

  public double getHour() {
    return this.hour;
  }

  public void setPayRate(double payrate) {
    this.payrate = payrate;
  }

  public double getPayRate() {
    return this.payrate;
  }

  public double getPaycheck() {
    return this.hour * this.payrate;
  }

   // (polymorphism) overriding displayInfo() in the parent class (Employee)
  public void displayInfo() {
    System.out.println("Name of Employee");
    System.out.println("First Name: " + getFirstName());
    System.out.println("Last Name: " + getLastName());
    System.out.printf("Hour Worked: $%.1f", getHour());
    System.out.printf("Pay Rate: $%.2f", getPayRate());
    System.out.printf("Pay Check: $%.2f", getPaycheck());
  }
}
```

Example of another Polymorphism (Overriding)

The method displayInfo() in this child class is the same name as the one in its parent, Emplyee.java. However, it performs different task, which also prints hour, pay rate and paycheck.

This is a Sub Class (Child), **EmployeMonthly.java**, with private variables or attributes for hour and rate. The keyword “**extends**” makes it a sub class of Employee. It can then inherit access to public methods from its super class. Therefore, the sub class now owns six public methods (4 public methods of the parent (Employee) and 2 public methods in its class (EmployeeMonthly).



```java
// Sub Class (Child) : EmployeeMonthly
public class EmployeeMonthly extends Employee {
  private double salary;

  // default constructor
  public EmployeeMonthly() {
    super();
    this.salary = 0;
  }

  // overloading constructor
  public EmployeeMonthly(String firstname, String lastname, double salary) {
    super(firstname, lastname);
    this.salary = salary;
  }

  public void setSalary(double salary) {
    this.salary = salary;
  }

  public double getSalary() {
    return this.salary;
  }

  //polymorphism (overriding)
  public void displayInfo() {
    System.out.println("Name of Employee");
    System.out.println("First Name:  " + getFirstName());
    System.out.println("Last Name:  " + getLastName());
    System.out.printf("Salary: $%.2f", getSalary());
  }
}
```

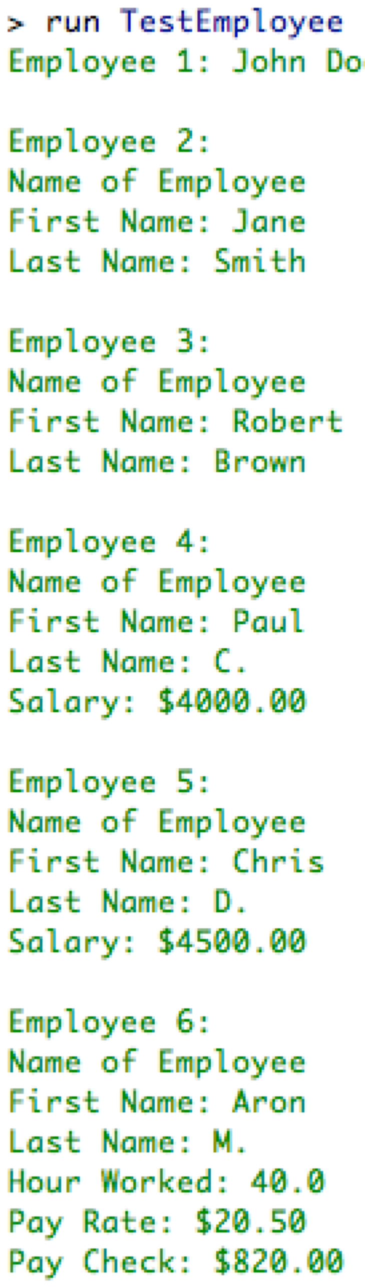
Example of another Polymorphism (Overriding)

This method displayInfo() is the same name as the one in its parent, Emplyee.java. However, However, it performs different task, which also prints salary.

TestEmployee.java includes main method to be executed.

```
public class TestEmployee {
  public static void main(String [] args) {
    Employee emp1 = new Employee(); // create emp1 with default constructor
    emp1.setFirstName("John"); // call the set method to assign the first name
    emp1.setLastName("Doe"); // call the set method to assign the last name
    // display the information by calling the get methods
    System.out.println("Employee 1: " + emp1.getFirstName() + " " + emp1.getLastName());

    Employee emp2 = new Employee();
    emp2.setFirstName("Jane");
    emp2.setLastName("Smith");
    System.out.println("\nEmployee 2: ");
    // display the information by calling the displayInfo method
    emp2.displayInfo();

    // create emp3 with overloading constructor
    Employee emp3 = new Employee("Robert", "Brown");
    System.out.println("\nEmployee 3: ");
    emp3.displayInfo();

    // Mixed examples of object creations and method calls
    EmployeeMonthly emp4 = new EmployeeMonthly();
    emp4.setFirstName("Paul");
    emp4.setLastName("C.");
    emp4.setSalary(4000.00);
    System.out.println("\nEmployee 4: ");
    emp4.displayInfo();

    EmployeeMonthly emp5 = new EmployeeMonthly("Chris", "D.", 4500.00);
    System.out.println("\nEmployee 5: ");
    emp5.displayInfo();

    EmployeeHourly emp6 = new EmployeeHourly("Aron", "M.", 40, 20.50);
    System.out.println("\nEmployee 6: ");
    emp6.displayInfo();
  }
}
```

Run TestEmployee Example



```
> run TestEmployee
Employee 1: John Doe

Employee 2:
Name of Employee
First Name: Jane
Last Name: Smith

Employee 3:
Name of Employee
First Name: Robert
Last Name: Brown

Employee 4:
Name of Employee
First Name: Paul
Last Name: C.
Salary: $4000.00

Employee 5:
Name of Employee
First Name: Chris
Last Name: D.
Salary: $4500.00

Employee 6:
Name of Employee
First Name: Aron
Last Name: M.
Hour Worked: 40.0
Pay Rate: $20.50
Pay Check: $820.00
```