Chapter 3: Object Oriented Programming with Java
## Outline

Section 1: Classes ..........................................................................................................................4
- Object Oriented Programming ........................................................................................................5
- Classes ........................................................................................................................................6
- Defining a Class in Java ..................................................................................................................7
- Object Creation .............................................................................................................................8
- Referencing Instance Variables ....................................................................................................9
- Memory Management .....................................................................................................................10
- Defining Methods ..........................................................................................................................11
- Calling Methods ..............................................................................................................................12
- Passing Parameters ........................................................................................................................13
- Overloading Methods ......................................................................................................................15
- Accessors/Mutators .........................................................................................................................16
- this Keyword ................................................................................................................................17

Section 2: Constructors ..................................................................................................................18
- Constructors .................................................................................................................................19
- Default Values ...............................................................................................................................21
- Private Constructors ........................................................................................................................22
- Finalizers .......................................................................................................................................23
- Static Modifier ...............................................................................................................................24
- Driver Class ..................................................................................................................................25

Section 3: Inheritance .....................................................................................................................26
- Inheritance ....................................................................................................................................27
- Implementing a Subclass ................................................................................................................28
- Implementing Inheritance ..............................................................................................................29
- Memory Allocation in Classes .........................................................................................................30
- Overriding Methods .......................................................................................................................31
- Polymorphism .................................................................................................................................32
- Default Constructors .....................................................................................................................34
- Super Keyword .............................................................................................................................35
- Java Class Hierarchy ......................................................................................................................37

Section 4: Modifiers .........................................................................................................................38
- Modifiers .......................................................................................................................................39
- Final Modifier .................................................................................................................................41

Section 5: Abstract Classes, Interfaces and Packages ....................................................................42
- Abstract Methods and Classes .........................................................................................................43
- Interfaces in Java ............................................................................................................................44
- Interface Example ..........................................................................................................................45
- Import Statement ............................................................................................................................46
- Packages ........................................................................................................................................47
- Packages, Classes, Files and Directories .......................................................................................48
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASSPATH Environmental Variable</td>
<td>49</td>
</tr>
<tr>
<td>Packages and Scoping</td>
<td>50</td>
</tr>
<tr>
<td>Creating an Executable JAR File</td>
<td>51</td>
</tr>
<tr>
<td>Recursion</td>
<td>52</td>
</tr>
<tr>
<td>Summary</td>
<td>54</td>
</tr>
</tbody>
</table>
Section 1: Classes
Object Oriented Programming

Three instance variables have been defined: one String reference variable and two integers. An instance of the Robot class is created; memory is allocated from the heap large enough to hold these three pieces.

There are three key aspects of Object Oriented Programming (OOP):

- Encapsulation
- Inheritance
- Polymorphism

The state of an object is typically hidden from the users of the object. This is called encapsulation. Methods are normally visible and allow the user of the object to manipulate that object. There are two ways of classifying methods:

- Mutator methods - Methods that can change a state of an object
- Accessor methods - Those methods that return the state of an object

Consider a circuit board designer. When the designer is engineering a new product, he doesn’t start designing at the gate level. He designs from the integrated circuit or component level. He pulls a book off the shelf of fully tested, specified, available components. Each component states its expected input values and output values.

Internally each of these components may be very complicated, may contain other components, or require the availability of other components such as a clock pulse. The point is, the designer doesn’t have to know or understand the internal complexities of the component, he just has to know how to use it.

Object oriented programming is intended to work in a similar manner. Using the paradigms of object oriented programming; our programs will be made up of many self contained components or "objects", each with specific pre-defined interfaces. Each component is capable of receiving messages that define its behavior.

Consider a class for representing motorized vehicles. The vehicle class serves as a model for many possible vehicles. With this vehicle class, we can instantiate many objects, each with differing features. Some have four tires, others 2, some burn gasoline other diesel fuel, but all behave alike (all used for transportation) and are recognizable as vehicles.
Classes

A class is a pattern or template for creating multiple objects with similar features. It defines the instance variables and methods of the class. An object is an instance of a class. The state of an object is reflected in the value of its instance variables. The behavior of an object is determined by the methods it possesses.

Methods frequently manipulate instance variable to define the new state of an object. In a well designed class, the instance variables can only be changed by the methods. They are private to the class. These methods should fully support the intent of the class including the creation of the object. In a motorized vehicle class there may be attributes such as:

- Engine status: Stopped or running
- Number of wheels: 1, 2, 3, 4, 6, 18
- Engine size: 4, 8, 12 cylinders
- Speed: Speed of vehicle
- Fuel Capacity: Gallons

The state of an object is determined by its attributes that are defined by variables. Each instance of a class (or object) has its own set of variables, called instance variables.

Methods modify or reference the instance variables of a class. Examples include:

- Start the engine / turn off the engine
- Accelerate
- Stop
- Get fuel

An object’s behavior is defined through methods. Methods are similar to ‘C’ functions but are defined inside a class.
Defining a Class in Java

A class definition in Java consists of member variable declarations and method declarations. It begins with the class keyword. The body of the class is enclosed with brackets. The body of a class contains all instance variables and methods.

```java
class classname {
    // define instance variables
    // define methods
}
```

Variables and methods can be declared using the private or public keywords as modifiers. Using private makes that variable or method only accessible within the class. The use of the public keyword allows access to the variable or method outside of the class.

An example of a class definition for a vehicle is illustrated below. The name of the class is Vehicle and it possesses two instance variables and two methods. The first instance variable is engineOn and is a boolean variable used to indicate whether the engine is on or off. The second variable is numberCylinders and specifies the number of cylinders that the vehicle has.

```java
class Vehicle {
    private boolean engineOn;
    private int numberCylinders;

    boolean getEngineState() {
        return engineOn;
    }
}
```

Another example:

```java
class Robot {
    public String name;
    private int weight;
    private int height;

    // Methods follow
}
```
**Object Creation**

By making these instance variables private, the designer restricts access to the variables. They are then only accessible through methods. If they were made public instead, other users can directly access the variables. This would improve the efficiency of the program, but may hinder future maintenance efforts. It would be more difficult to change these variables and to enforce any sort of validation checks on changes to the variables.

The `getEngineState` method is not passed arguments and returns a boolean result.

The following occurs when a new object is created:

- The new keyword is used to create an instance of a class
- Memory is physically allocated for the new instance of the class
- A constructor is called to do initialization
- A reference to the object is returned

```java
Vehicle v1 = new Vehicle();
Vehicle v2;
```

Each instance of a class has its own set of instance variables that are independent of each other. The methods are shared among the instances of the class.

Referencing Instance Variables

To reference an instance variable, the dot notation is used. Its syntax is as illustrated below:

```java
variable = objectName. variableName;
```

The Employee class is defined below with several standard instance variables.

```java
public class Employee {
    private String name;
    private int zip;
    public Date dob;
}
```

The following illustrates possible references:

```java
Employee jim = new Employee();
jim.dob = birthDate;
Date birthDate = jim.dob;
jim.name = "Unknown" // illegal statement outside of Employee class
```
Memory Management

Java memory management is dynamic and automatic. When the new method is invoked, it automatically allocates memory. This memory is allocated dynamically on the heap. Each object has a reference count that is used to determine its lifetime.

A reference variable may de-reference an instance by

- Being reassigned to another object or
- By setting it to null

When the reference count goes to zero, automatic garbage collection is performed as needed.

In the next example, the reference variables, v1 and v2, are part of the someMethod activation record. A new object of type Vehicle is allocated from the heap. A reference to this object is assigned to v1. v2 has not been assigned a Vehicle object and should not be used.

```java
public void someMethod() {
    Vehicle v1 = new Vehicle();
    Vehicle v2;
}
```

![Memory Management Diagram](image)
Defining Methods

Methods are similar to function in C. They are defined as part of the class definition and normally follow the declaration of the instance variables. The method declaration specifies a return type. void means that the method does not return a value.

In the following examples, the first method is startEngine. If the engine is off, then it will set the engineOn variable to true. Otherwise, a message will be printed indicating that the engine is already on. The second method isRunning simply returns the state of the engine.

```java
class Vehicle {
    private boolean engineOn;
    private int numberCylinders;

    public void startEngine () {
        if (engineOn)
            System.out.println ("Already running");
        else {
            engineOn = true;
            System.out.println ("Turned on");
        }
    }

    public boolean isRunning () {
        return engineOn;
    }
}
```
Calling Methods

The syntax used for invoking methods appears similar to using instance variables. The dot notation is used with the method name. Methods take 0 or more arguments.

The following is the syntax for using a method

objectName.methodName();

The following illustrates the use of methods with the Vehicle class:

    boolean state;
    Vehicle v = new Vehicle ();
    v.startEngine();
    state = v.isRunning ();

Important: All instance methods execute against an object.
Passing Parameters

In Java only primitive data types and object references are passed to a method or constructor. When they are passed they are effectively passed by value. That is, a copy of the primitive data type or the reference is copied. Objects are not passed, only references to the objects are passed.

Consider the following example:

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

    ParameterDemo() {
        size = 10;
    }

    void setSize(int size) {
        this.size = size;
    }

    int getSize() {
        return size;
    }
}
```

```java
public class ParameterDemoDriver {
    private static void doubleSize(ParameterDemo pd) {
        pd.setSize(pd.getSize() * 2);
    }

    private static void replaceSize(ParameterDemo pd) {
        pd = new ParameterDemo();
        pd.setSize(100);
    }

    public static void main(String args[]) {
        ParameterDemo demo = new ParameterDemo();
        doubleSize(demo);
        System.out.println(demo.getSize());
        replaceSize(demo);
        System.out.println(demo.getSize());
    }
}
```

The output of the program follows:

```
20
20
```

In the method doubleSize, a reference to the demo object is passed and is modified by that method as would be expected.
In the replaceSize method, a reference to the demo object is also passed. However, in the method, a new ParameterDemo object is created and assigned to the local variable pd. The new ParameterDemo object has its size variable set to 100. When the method returns, the reference to the new ParameterDemo object is not returned and assigned to the demo reference variable. The new ParameterDemo object is lost and eventually collected by the JVM garbage collection thread. The demo object is not changed as reflected in the output.
Overloading Methods

Multiple methods with the same name are permitted in Java. This provides a convenient technique for implementing methods that differ in argument types. This technique is called overloading.

Each overloaded method has a unique signature. A signature is a combination of:

- Argument types
- Number of arguments
- Order of arguments

The following is an example of overloading the max method:

```java
class OverloadingDemo {
    public int max(int n1, int n2, int n3) {...}
    public float max(long n1, long n2, long n3) {...}
    public float max(float n1, float n2) {...}
}
```

The compiler is able to determine which method to use based on the parameters used.

```java
int num;
float result;

OverloadingDemo d = new OverloadingDemo();
num    = d.max(45, 98, 2);
result = d.max(45.0f, 0.056f);
```

Notice that the return type is not part of the signature. The following overloading example is invalid:

```java
public int max(int n1, int n2, int n3) {...} //Ambiguous overloaded method
public long max(int n4, int n5, int n6) {...} //Ambiguous overloaded method
```
Accessors/Mutators

An accessor method is one that reads, or accesses, a variable of a class. A mutator method is one that modifies a variable of a class. These methods are usually public and the variables are normally declared as private.

There is a consistent pattern that you can use that facilitates accessor and mutator methods. This is a good convention to follow as it is also used in other areas of Java such as with Java Beans.

**Accessor**

```java
public propertyType getProperty() { … }
```

**Mutator**

```java
public void setProperty(propertyType variable) { … }
```

The `Property` corresponds to the name of the variable and `propertyType` is its data type. The use of these methods provides read/write capability to a property/attribute of a class. If only the accessor method is provided, then the property is said to be read-only. If only the mutator method is provided, then the property is said to be write-only.

Accessor and mutator methods are provided for the Order class.

```java
public class Order {
    private int orderNumber;
    private int partNumber;
    private int quantity;

    public int getOrderNumber() {
        return orderNumber;
    }

    public void setOrderNumber(int newOrderNumber) {
        if (newOrderNumber > 0) {
            orderNumber = newOrderNumber;
        } else {
            System.out.println("Invalid Order Number");
        }
    }
}
```

Notice that the return type of `getOrderNumber` is the type as used by `setOrderNumber` and the private `orderNumber` instance variable.
**this Keyword**

The this keyword always refers to the current object. In the following example, the same names are used for instance variables and for parameters to the method.

```java
class Circle {
    Point center;
    float radius;

    void setCenterAndRadius (Point center, float radius) {
        center = center;
        radius = radius;
    }
}
```

This code does not have the intended consequences of modifying the respective instance variables. The scope of the instance variables is the entire class. The scope of the parameters is only the method. The parameters will have "precedence" over the instance variables. The effect is that two numbers are passed to the method and then they are assigned to themselves. The instance variables are not modified.

There are two ways of correcting this problem:

- Change the names (which makes it more readable)
- Use the this keyword (as shown below)

```java
class Circle {
    Point center;
    float radius;

    void setCenterAndRadius (Point center, float radius) {
        this.center = center;
        this.radius = radius;
    }
}
```

There are three uses for the **this** keyword:

- Accessing instance variables
- Passing the current object to a method
- Returning the current object from a method
Section 2: Constructors
Constructors

Initialization of an object’s instance variables is important. The use of an initialization method that the user must explicitly invoke is not a foolproof technique. Automatic initialization is performed through a method called a constructor.

Constructors are used to perform any initialization needed for a new object. Important characteristics of constructors include:

- Constructors have the same name as the class
- Constructor overloading is permitted
- Constructors are not methods
- Constructors do not have a return type, not even void

The following illustrates how constructors are defined:

```java
class Point {
    private double x;
    private double y;

    public Point() {
        x = 10.0;
        y = 10.0;
    }

    public Point (double newx, double newy) {
        x = newx;
        y = newy;
    }
}
```

The next illustration shows how the constructor is used. Two Point objects are created: beginning and end. beginning is created with an initial value of (0.0, 0.0) and end has the value (150.0, 250.0).

```java
Point beginning;
Point end;

beginning = new Point();
end = new Point(150.0, 250.0);
```
Java automatically provides a default constructor for each class. This constructor will essentially initialize its instance variables to 0 as explained on the next page. However, if the programmer adds a constructor to the class, then the class will no longer have a default constructor unless the programmer explicitly adds one.

**As a general rule, always add a default constructor to a class.** This is particularly important when the class is used as a base class in inheritance.
Default Values

Instance variables are automatically assigned a default value. The value assigned depends on its data type. This default value is assigned before a constructor is executed.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>0</td>
</tr>
<tr>
<td>Float</td>
<td>0.0</td>
</tr>
<tr>
<td>Double</td>
<td>0.0</td>
</tr>
<tr>
<td>Char</td>
<td>'0'</td>
</tr>
<tr>
<td>String</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>Object</td>
<td>null</td>
</tr>
</tbody>
</table>
Private Constructors

A constructor is sometimes declared as private. When declared as private it cannot be used outside of the class. If all of the constructors are declared private then it is not possible for the user of a class to create an instance of the class. However, that doesn't mean that the user can't get an instance of the class.

There are times when it is desirable to have one and only one instance of a class in an application at a time. For example, perhaps only one time or locale object may be desirable or needed in an application. This is what is known as the singleton pattern.

If more than one instance is desirable we may still want to restrict the number of objects of that type that are created. Too many objects might have a performance impact on the system.

The following class illustrates how a private constructor can be useful. Only one constructor is provided and it is declared as private. To provide access to an object a public getInstance method is provided. This method needs to be implemented as a static method (since the user is unable to create an object of that type) and can be implemented to return only one or several instances of the class. The getInstance method below implements the singleton pattern.

class Clock {
    private double seconds;
    private double minutes;
    private double hours;
    private Clock instance = null;

    private Clock() {
        ...
    }

    public static Clock getInstance() {
        if (instance == null) instance = new Clock();
        return instance
    }
}

Finalizers

A finalizer method performs any necessary action just before the garbage collector frees the object. The finalizer has the name finalize and uses no arguments. A finalizer is optional. The finalizer cannot be overloaded

protected void finalize () {
    System.out.print ("Object released");
}

Finalizers are critical to C++. In java, they are used infrequently.

There has been several attempts to come up with a way to force the execution of all finalize methods before the program terminates. None of the approaches have been very successful. There is a runFinalizerOnExit method of the System class that was introduced to force finalizer to execute. However, this method was eventually discovered to result in potential stability problems in a few rare instances. As a result, the method was deprecated.

Deprecation is a technique whereby a language can denote certain features of the language, such as classes and methods, to be obsolete. This may because of potential errors that can occur with their usage or because a better technique has been devised.

When a method is deprecated, it is not immediately removed from the language. Instead the compiler generates a warning that can be removed using the –Deprecation compiler switch. The warning informs the programmer that the method has been deprecated and that in some subsequent version of the language the method will go away. The Java help facilities documents all deprecated classes and methods and provides an alternative approach when appropriate.
Static Modifier

Sometimes it makes sense to have a single variable that can be shared and accessed by all instances of a class. The keyword `static` is used to declare a variable or method. When used with a variable, it is called a class variable and is local to the class itself. There is always a single copy of the variable.

When used with a method, the method is called a class method. Class methods can only access static data (has no this member).

The following illustrates using the static modifier. The variable, `primeRateIndex`, is declared as static double. Each instance of the class `Stock` will share this variable. Since a class variable can exist even if an instance of the class does not, then it can be accessed at any time. A reference to the class variable or method can be made through a reference variable or through the name of the class.

```java
public class Stock {
    static  private double  primeRateIndex = 0.0;
    private int numShares = 0;

    public static double getPrimeRateIndex() {
        // numShares = 10;       // illegal - no access
        return primeRateIndex;
    }

    public static void main (String args[]) {
        Stock obj = new Stock();
        obj.getPrimeRateIndex();   // valid access
        Stock.getPrimeRateIndex(); // more readable
    }
}
```

A static method may only access a static variable or another static method. Since a static method may execute against the class name and there may not be any instances of a class, it would be difficult for a static method to access an instance variable when no instance variables exist. If there were several instances of the class, then which instance variable would be accessed?

The following summarizes the rules regarding static members:

<table>
<thead>
<tr>
<th></th>
<th>Can the method access</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Static variable</td>
</tr>
<tr>
<td>Instance method</td>
<td>Yes</td>
</tr>
<tr>
<td>Static method</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Driver Class

When a class is created it is generally intended to be used by other programmers. This tends to promote reuse. A well defined class is flexible and easy to use. The user of the class does not have to create his instance of the class.

In a Java application the main method is the first method to be executed. If the application consists of multiple classes, it is not feasible that each class has its own main method. A Java application needs one and only one main method.

As a result the majority of classes do not have a main method. None of the classes in the Java library packages (such as those in the java.lang package) have main methods. When you create your own class, it will normally not have a main method.

However, one class in an application needs a main method. This class is referred to as the driver or controller class. Its purpose is to have a main method that will create and use other classes. The following is an example of a driver class that uses the Account class:

class TestAccount {
    public static void main (String args[]) {
        Account account;
        account = new Account();
        account.deposit(500);
        ...
    }
}

The Account class will not have a main method. There will normally be two source files: one for Account and a second for TestAccount. They would both be in the same directory and compiled separately. The Account class should be compiled first so that when it is referenced in the TestAccount class the compiler will be aware that it exists. Later we will discuss further the uses and configuration of classes and packages.
Section 3: Inheritance
Inheritance

Inheritance is an important concept of any object oriented programming language. Inheritance promotes reuse of code and it makes possible the concept of polymorphism. One class is said to be derived from another class. There are several terms used to denote inheritance relationships:

- Base class and derived class
- Superclass and subclass
- Parent class and child class

```
Class A
  ↓
Class B
  ↓
Class C  Class D  Class E
```

Base Class

Base Class

Derived Class

Derived Class
Implementing a Subclass

A class is implemented through the use of the extends keyword followed by the base class name. Multiple classes can be defined within the same file but only one may be public. In the following example, the Employee class is defined. The SalaryEmployee class is defined to be a subclass of Employee.

```java
class Employee {
    // Implementation of Employee class
}

class SalaryEmployee extends Employee {
    // Implementation of SalaryEmployee class
}
```

Applets are created by extending the Applet class. The applet illustrated in Chapter 1 creates a subclass, HelloWorldApplet, which extends the Applet class. HelloWorldApplet inherits all of the methods and properties of this class. In this case, the paint method is overwritten by HelloWorldApplet.

```java
import java.awt.Graphics;

public class HelloWorldApplet extends java.applet.Applet {
    public void paint (Graphics g) {
        g.drawString("Hello World!", 5, 15);
    } // paint
}
```
Implementing Inheritance

An instance variable or method declared in a base class is inherited in a derived class. This means that it can be used as if it were declared within the derived class directly.

```java
public class Employee {
    public int id;
    public float computePay() { … }
}

public class SalaryEmployee extends Employee {
    // Implementation of SalaryEmployee class
}
```

In this example, the Employee class is defined to consist of an id instance variable and a method called computePay. As shown below, an instance of SalaryEmployee can use the id or computePay method.

```java
SalaryEmployee e = new SalaryEmployee();
e.id = 12345;
System.out.println(e.computePay());
```
Memory Allocation in Classes

An important part of inheritance is how memory is allocated for the base and derived classes. Understanding how memory is allocated will give you a more in depth understanding and appreciation for inheritance and will enable you to better understand polymorphism. It will also help you understand how constructors work.

When a class is instantiated and an object is created, memory is allocated for the base class first followed by the derived class. As memory is allocated for each class, its constructor will execute.

```java
public class Employee {
    private int id;
    private float pay;
}

public class SalaryEmployee extends Employee {
    private float stock;
}
```

When an instance of Employee is created, memory is allocated for both member variables.

<table>
<thead>
<tr>
<th>id</th>
<th>pay</th>
</tr>
</thead>
</table>

When an instance of SalaryEmployee is created, memory is allocated first for its base class Employee and then for the SalaryEmployee's member variables. Notice how the base class precedes the derived class in memory.

<table>
<thead>
<tr>
<th>id</th>
<th>pay</th>
<th>stock</th>
</tr>
</thead>
</table>
**Overriding Methods**

It is possible to overload base class methods in a subclass. These methods must have the same signature as that of the method in the base class. This new method will replace the original definition. This technique is used to implement polymorphism.

```java
public class Employee {
    public int id;
    public float computePay() {return 0.0; }
}

public class HourlyEmployee extends Employee {
    public float hours;
    public float payRate;
    public float computePay() {return hours * payRate; }
}
```

In this example, two instance variables, `hours` and `payRate`, have been added to the `HourlyEmployee` class. Also the `computePay` method has been overridden.

The terms overloading and overriding are easily confused. There are three characteristics that you need to keep straight.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Overloading</th>
<th>Overriding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method name</td>
<td>Same</td>
<td>Same</td>
</tr>
<tr>
<td>Signature</td>
<td>Different</td>
<td>Same</td>
</tr>
<tr>
<td>Class</td>
<td>Same class</td>
<td>In a derived class</td>
</tr>
</tbody>
</table>
Polymorphism is an important OOP technique. It makes software easier to maintain by minimizing the amount of work required to adapt and modify an existing program.

Polymorphism is defined as the application of the same method against different objects with potentially different actual methods being executed. The method that is used is determined at run-time by the Java virtual machine.

*A method is said to be polymorphic if the behavior of the methods varies depending on the object the method runs against*

In Java a reference to base class and any of its derived classes can be assigned to a base class reference variable. This is possible because of the way memory is allocated for base and derived classes. In a derived class, the instance variables of the base class are allocated first, followed by the instance variables of the derived class. When a base class reference variable is assigned to a derived class object it sees the base class instance variables that it expects plus the "extra" derived class instance variables.

Below, assigning the base class reference variable e to either object makes sense from the perspective of the reference variable because it expects to see the instance variables id and pay.

```
Employee e;
```

If the code fragment shown below is executed base upon the previous declarations of the Employee and SalaryEmployee classes, then the SalaryEmployee’s computePay method will be executed and not that of the Employee class.

```
Employee e = new HourlyEmployee();
System.out.println(e.computePay());
```
The computePay method is said to be polymorphic in relation to the object it is running against. If the computePay method ran against an Employee object then the Employee's computePay method would execute.

Employee e = new HourlyEmployee();
System.out.println(e.computePay());  // HourlyEmployee's method
e = new Employee();
System.out.println(e.computePay());  // Employee's method
Default Constructors

If a constructor is not provided for a class, a default constructor is used. A class derived from a base class can call the constructor of the base class. The base class constructor can be invoked using the keyword super.

```java
class Point {
    private int x;
    private int y;

    public Point() {
        x = 10;
        y = 10;
    }

    public Point(int newX, int newY) {
        x = newX;
        y = newY;
    }
}

class Circle extends Point {
    private int radius;

    public Circle (int newX, int newY, int newRadius) {
        super(newX, newY);
        radius = newRadius;
    }

    public Circle (int newRadius) {
        // The default base class constructor
        radius = newRadius;
    }

    public Circle () {
        super(0,0);
        radius = 0;
    }
}
```

If the super keyword is used, it must be the first statement in the method. If super is not used then the default constructor of the base class is called. If the default constructor does not exist then an error will occur.
Super Keyword

The super keyword has two uses in Java:

- It is used to call the base class constructor
- It is used to access an overridden method in the base class

The first use was illustrated on the previous page. The second use needs to be supported when the overridden method of the derived class needs to call the base class method. For example, suppose we need to write a display method that displays information about an employee.

```java
public class Employee {
    private int id;
    private String name;

    public void display() {
        System.out.println("ID: "+ id);
        System.out.println("Name: "+ name);
    }
}
```

The HourlyEmployee class is then derived from the Employee class which needs its own display method. However, it is desirable to be able to call the base class display method because:

- We don’t wait to duplicate code
- If we change the number of base class instance variables we don't want to change the display related code in more than one place
- The base class instance variables are private and we can't access them without using accessor methods that may or may not be present

Instead we will use the super keyword to indicate that we want to call the corresponding base class method as shown below:

```java
public class HourlyEmployee extends Employee {
    private float hours;
    private float payRate;

    public void display() {
        super.display();
        System.out.println("Hours Worked: "+ hours);
        System.out.println("Pay Rate: "+ payRate);
    }
}
```
Unlike the use of the super keyword as the first statement in a constructor, the super keyword can be used anywhere in a derived class method.

```java
public void display() {
    System.out.println("Employee Base Data");
    super.display();
    System.out.println("Hourly Employee Data");
    System.out.println("Hours Worked: " + hours);
    System.out.println("Pay Rate: " + payRate);
}
```

When using the super keyword to call a base class method, it is not necessary to call the same method that the as the one the code resides in. For example:

```java
public float computePay() {
    float bonusAmount = 0.0f;

    if (super.getYearsOfService() > 20)
        bonusAmount = 250.0f;
    return super.computePay() + bonusAmount;
}
```

If the getYearsOfService method was not overridden in the derived class, then there is no reason to use the super keyword since the derived class will automatically inherit it. In the above example, it doesn't hurt to use the super keyword. Some programmers prefer to use to indicate explicitly that the method is a base class method. The only drawback is if the method is ever overridden, then it might not be appropriate to use the base class method.

It is not possible to call a base's base class method.

```java
super.super.display(); // illegal
```
Java Class Hierarchy

Java is organized into many packages of hundreds of classes. New packages and classes are being added on a regular basis making it nearly impossible to keep up with all of the capabilities of Java.

However, all classes in java has as a base class, either directly or indirectly the class Object. In the classes that you defined, if you do not explicitly extend another class, Java will automatically extend this class from the Object class.

The Object class is found in the java.lang package. It contains several methods of which only a few are of interest to us at this time. These methods should be overridden by your class.

- **equals** – This method will return true if the two classes are equal
- **toString** – This method will return a String that describes the class/object

**equals** should be overridden to allow meaningful comparisons of two objects. What it means for two objects to be equal is dependent on the class. Two Employee objects might be equal if they have the same name. Or they could be equal if they have the same employee grade and live in the same ZIP code. The definition of equality is up to the programmer. The default implementation of the **equals** method does a bit by bit comparison of the two objects which is not necessarily always the right thing to do.

The **toString** method by default returns the name of the class followed by a seemingly random character string. As we will see later in the course, this method is used in a number of places. It behooves the programmer to override this method and provide a more meaningful string representation of the object. For an employee object, it might simply return the name of the employee.
Section 4: Modifiers
Modifiers

Modifiers are used as prefixes when declaring instance and static variables and methods. Modifiers are applied in various combinations to provide specific behaviors. Modifier order is not important, but consistent style lends to readability. All modifiers are optional though there are some default modifiers.

Good Object Oriented programming suggests using as many as are applicable. Examples of modifiers include:

- **public** - visible to all methods inside and outside of its own class
- **protected** - protection between the current class and subclasses. Object is invisible outside of the class, fully visible to subclasses
- **private** - cannot be seen by any class other than the one in which it is defined (including subclasses)
- **"package"** - default protection, "package" visibility. Only classes within the package have access (public within the package)

The following sequence illustrates the use of public, private and protected modifiers:

```java
public class TestClass {
    public int publicInt;
    private int privateInt;
    protected int protectedInt;
    int defaultInt;  // default (package)
}
```

All of the variables are of type int. publicInt is a public variable. It can be seen by all methods inside and outside of this class. Int is visible only to those classes within this package. protectedInt is visible to this class and subclasses. It is not visible elsewhere. privateInt is only visible within this class.
Given the following package/class organization:

And the following class declaration:

```java
public class A{
    public int publicInt;
    private int privateInt;
    protected int protectedInt;
    int defaultInt;  // default (package)
}
```

The following table shows the visibility of each of the declaration types.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>publicInt</td>
<td>Visible</td>
<td>Visible</td>
<td>Visible</td>
<td>Visible</td>
<td>Visible</td>
</tr>
<tr>
<td>privateInt</td>
<td>Visible</td>
<td>Invisible</td>
<td>Invisible</td>
<td>Invisible</td>
<td>Invisible</td>
</tr>
<tr>
<td>protectedInt</td>
<td>Visible</td>
<td>Visible</td>
<td>Visible</td>
<td>Invisible</td>
<td>Visible</td>
</tr>
<tr>
<td>defaultInt</td>
<td>Visible</td>
<td>Visible</td>
<td>Visible</td>
<td>Invisible</td>
<td>Invisible</td>
</tr>
</tbody>
</table>
Final Modifier

The final modifier has meaning that depends on where it is used. When applied to a variable, the variable is a constant, and cannot be modified. When applied to a method, the method cannot be overridden in a subclass. When applied to a class, the class cannot be subclassed.

```java
public final class TestClass {
    public final double PI = 3.14159;

    public final double FinalMethod () {
        return PI;
    }
}
```

The keyword `final` is used to define a constant. An initial value is required when the variable is declared. If an initial value is not assigned to the variable, a compile time error will occur. An attempt to later assign a value to the variable will also generate an error. If the variable contains a reference to an object, the reference will not vary. However, the state of the object may change.

```java
final Color colorOfSun = Color.yellow;
```
Section 5: Abstract Classes, Interfaces and Packages
Abstract Methods and Classes

An abstract class is typically used to force a derived class to implement a specific set of methods. Abstract classes are useful in design of an Object Oriented inheritance hierarchy. The class and/or one or more methods of the class are declared as abstract. Abstract classes can not be instantiated. A concrete class, in contrast, must have all of the abstract methods in its hierarchy tree implemented

```java
public abstract class AbstractClass {
    private int age;
    public abstract void display();
}

class Concrete extends AbstractClass {
    public void display() {
        System.out.println("Age: "+ age);
    }
}
```

The abstract keyword is needed for both the method and the class.

An abstract class can also possess concrete methods as this second version of AbstractClass demonstrates.

```java
public abstract class AbstractClass {
    private int age;
    private String name;
    public abstract void display();
    public String getName() { return name; }
    public void setName(String newName) { name = newName; }
}
```
Interfaces in Java

An interface specifies a set of behavior that other classes must implement. The interface effectively pass on method descriptions. An interface cannot be instantiated. To instantiate a class that implements an interface, all the methods of the interface(s) must have be implemented.

```java
interface Coach {
    public void tutor ();
    public void personalInstructor ();
}

class Teacher implements Coach {
    public void tutor () { }
    public void personalInstructor () { }
}
```
Interface Example

It is possible to derived from a base class and implement from more than one interface as illustrated below:

Base Class:

Derived Class:

In the following example, Host and Coach interfaces are declared. The class Teacher extends Person and provides implementation of the Host and Coach classes. Notice that implementations are provided for GreetStudents and Tutor methods. Other methods have also been added to the Teacher class.

```java
interface Host {
    public void greetStudents ();
    public void maintainClassRoom ();
}

interface Coach {
    public void tutor ();
    public void personalInstructor ();
}

class Person {
    private String name;
    private String address;

    private Date dateOfBirth;
    public String getName() { return name;};
}

class Teacher extends Person implements Host, Coach {
    protected int yearsOfExperience;
    public void greetStudents () {} ;
    public void maintainClassRoom () {} ;
    public void tutor () {} ;
    public void personalInstructor () {} ;
}
```
Import Statement

Java assumes that all applications are developed as packages. The fully qualified name of a class reflects the directory structure that contains that class. The import statement permits the use of the class name without having to fully specify it.

// Approach 1:

insurance.auto.Claim c1 = new insurance.auto.Claim();
c1.processClaim();

// Approach 2:

import insurance.auto.Claim;
...
Claim c1 = new Claim();
c1.processClaim();

The import statement does not include code. It is used to clarify which classes to use and to reduce the amount of typing that is required. The asterisk can be used to indicate that all of the classes of a package may be used (but not the sub-packages).

import airline.Ticket;
import airline.*;

The * specifies all of the class of a given package. It does not specify all of the classes of "sub" packages. For example, given two packages employee and employee.benefits:

import employee.*;
import employee.benefits.*;

The first import imports only those classes that are in the employee package. It does not include those classes that are in the employee.benefits package. If you also need to import the class in the employee.benefits package you will need to use the second import statement also.

The package java.lang contains many commonly used classes and is included automatically in every application.
Packages

The purpose of a package is to

- Group related classes together
- Provide a mechanism that allows classes with the same name but different packages to be specified

It is only natural that classes that have similar functionality should somehow be grouped together. All of Java's IO classes are grouped together in the java.io package. All of Java's network classes are found in the java.net package. This grouping mechanism provides us with a single logical grouping that is easier to talk about and to work with.

There are hundreds of Java packages in existence. It is possible that different packages may have classes that have the same name. If we needed to use both classes in the same application we would need a mechanism to differentiate between the two classes. The package mechanism provides this capability.

In the next example two different Ticket classes are specified. t1 is an airline ticket while t2 is a bus ticket.

```java
import airline.Ticket;
import bus.Ticket;

airline.Ticket t1;
bus.Ticket t2;
```
### Packages, Classes, Files and Directories

A package is a related group of classes and interfaces. Packages are used to ensure unique names for classes. To declare that a class belongs to a specific package two times are needed.

- A package declaration is used in the class source file
- The class file needs to be moved to the appropriate package directory

The package declaration is the first statement in a Java source file. It consists of the keyword `package` followed by the name of the package and then a semicolon. The following declares the Ticket class as a member of the airline package:

```java
package airline;

public class Ticket {
    // Ticket class implementation
}
```

When the developer uses a class, the import statement is used to specify the class of interest. In the following class, the airline ticket class is used:

```java
import airline.Ticket;
```

Every class is a member of a package. If the package statement is not used when declaring a class it is automatically added to an unnamed default package. All classes in the same directory that are not assigned to a package are added to the default package.

The second requirement is to move the class file to the appropriate package directory. Somewhere on the system there must exist a directory structure that reflects the package name. For example, for the package name employee.benefits there needs to be a directory named employee that has a subdirectory named benefits. All of the class files for the employee package are placed in the employee directory. All of the class files for the employee.benefits package are placed in the benefits subdirectory.

You may also find that a package directories and classes are zipped into a JAR (Java Archive file - .jar). If you look for a specific package structure in a directory system, you may find a JAR file instead. By zipping packages into a jar file, memory on the hard drive can be minimized. If you find such files, do not unzip them because the Java compiler and JVM expects them to be in a jar file.
CLASSPATH Environmental Variable

The CLASSPATH environmental variable is used to identify the root directory of the packages. For example, if the employee directory is located at d:\development\increment1, the CLASSPATH environmental variable is set as follows:

```
c:>set CLASSPATH=d:\development\increment1;%CLASSPATH%
```

The CLASSPATH variable only needs to be set for non-standard packages.
Packages and Scoping

Assuming the following package/class arrangement:

```
class A {
    public int v1;
    private int v2;
    protected int v3;
    int v4;
}
```

The following table summarizes the scoping rules for these declarations. These rules apply to both variables and methods declared in class A. It may be necessary to declare an instance of class A in some of these classes to have access to the instances variables of A.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>public int v1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>private int v2</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>protected int v3</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>int v4</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Creating an Executable JAR File

To create a executable JAR file:

- Create a manifest file
- Execute the jar command

1) Create a manifest file

The file needs to contain the following where className is the name of the class:

```
Main-Class: className
```

There should be a CRLF at the end of the line.

2) Next, create the JAR file using the following command:

```
c>jar cmf manifestFile Application.jar ApplicationClassName.class
```

Where:

cmf – c means to create a new file
m means to modify the manifest
f means to output the archive to file

- `manifestFile` - The name of the manifest file
- `Application.jar` – The name of the JAR file to be created with the .jar extension
- `ApplicationClassName.class` – The class(es) to be included on the JAR

Executing the JAR File

The jar file can be executed using the command:

```
java -jar Application.jar
```

This command can be placed in a batch file to simplify its execution.
Recursion

Recursion is an important programming concept. Essentially a method is said to be recursive if it calls itself directly or indirectly. For example, the function foo is recursive:

```java
public int foo(int n) {
    foo(n-1);
    return n;
}
```

However, the method will never return because it keeps calling itself. If the method is called with an initial value of 5, then it will call itself with the value of 4 and then with the value of 3 and then with the value of 2, etc. This is called unbounded recursion and is not very desirable.

With unbounded recursion, the program will eventually exceed the space allocated for its program stack. When this happens the program will terminate.

If foo called another method called foo2 which called foo3 which called foo, then this is a case of indirect recursion. If it is unbounded, then the program will eventually terminate abnormally.

Bounded recursion is when the method stops calling itself. This is a desirable quality and is required of almost all useful recursive methods. The simplest way to bound recursion is to have a decision statement that either calls the function or doesn't call the function as illustrated below:

```java
public int foo(int n) {
    if (n>1) {
        foo(n-1);
        return 2;
    } else {
        return 1;
    }
}
```

In this case, foo calls itself when n is greater than one else it returns a 1. The else part of the if statement bounds the recursive method. If we didn't decrement n each time we called foo, we would still have unbounded recursion.

In the then portion of the if statement we arbitrarily returned the number two. We could create a more useful method by returning a different value. In the next example, the value returned from foo is stored in the variable result. Result is then multiplied by the value of n.
public int foo(int n) {
    if (n>1) {
        int result = foo(n-1);
        return n * result;
    } else {
        return 1;
    }
}

This is actually a recursive solution to N factorial. An equivalent, but more concise version follows:

public int factorial(int n) {
    if (n>1) {
        return factorial (n-1) * n;
    } else {
        return 1;
    }
}

It is important to remember that each time a method is invoked, the method's activation record is pushed on to the stack. With a recursive call, there will be several instance of that method's activation record on the stack at a time and each one will have its own copy of the local variables and parameters passed to it.
Summary

- Classes and object are made up of methods and variables
- All objects have distinct behaviors and attributes associated with them
- Attributes are the individual qualities each object possesses
- Behaviors represent the actions an object may take during its lifetime
- A constructor is used to initialize an instance of a class
- Overloading of a method provides alternative ways of passing information to a method
- Class variables and methods are shared among all instances of a class
- Classes are used to define templates that are used to instantiate concrete objects
- Java provides modifiers to limit visibility of methods and variables
- The final keyword is used to create constants variables, and classes that cannot be subclassed
- Package hierarchies are used to create name spaces and allow for classes with duplicate names
- Interfaces effectively replace the need for multiple inheritance