Classes & Objects, Encapsulation in Java
Today

- **Classes & Objects**
  - Defining Classes, Objects and Methods
  - Accessor and Mutator Methods
  - Constructors
  - Static Members
  - Wrapper Classes
  - Parameter Passing
  - Delegation

- **Encapsulation**
  - Information Hiding
  - Encapsulation
  - The public and private Modifiers
  - UML Class Diagrams
  - Overloading
  - Packages
Class and Method Definitions

- Java program consists of objects
  - Objects of class types
  - Objects that interact with one another

- Program objects can represent
  - Objects in real world
  - Abstractions
# Java Classes

- A class is a collection of fields (data) and methods (procedure or function) that operate on that data.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Attributes</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle</td>
<td>center</td>
<td>circumference()</td>
</tr>
<tr>
<td></td>
<td>radius</td>
<td>area()</td>
</tr>
</tbody>
</table>
Defining a Java Class

- Syntax:

```
class  ClassName{
    [fields declaration]
    [methods declaration]
}
```

- Bare bone class definition:

```
/* This is my first java class. 
It is not complete yet. */
class Circle {
    // fields will come here
    // methods will come here
}
```
Adding Fields to Class Circle

- Add fields

```java
class Circle {
    public double x, y;  // center coordinates
    public double r;     // radius of the circle
}
```

- The fields are also called the **instance variables**.
  - Each object, or instance of the class has its own copy of these instance variables

- Do not worry about what **public** means at the moment.
  - Access modifiers (public, private and protected will be covered later)
Adding Methods to a Class

- A class with only data fields **has no life.**
  - Objects created by such a class **cannot respond to any message.**

- **Methods** are declared inside the body of the class.

- The general form of a method declaration is:

  ```java
  type MethodName (parameter-list) {
  Method-body;
  }
  ```

- `methodName(parameter-list)` part of the declaration is also known as the method signature.
  - Method signatures in a class must be unique!
public class Circle {
    public double x, y; // center of the circle
    public double r; // radius of the circle

    // Method to return circumference
    public double circumference() {
        return 2 * 3.14 * r;
    }

    // Method to return area
    public double area() {
        return 3.14 * r * r;
    }
}

Adding Methods to Class Circle
Defining Reference Variables of a Class

- A class can be thought as a type

- A variable (reference) can be defined as of that type (class)

```java
Circle circleA, circleB;
```

- Points to nothing (Null Reference)

- Points to nothing (Null Reference)
Creating Objects of a Class

- Objects are created by using the `new` keyword

```java
Circle circleA;
circleA = new Circle();

Circle circleB = new Circle();
```

Two different `circle` objects!
Creating Objects of a Class

circleA = new Circle();
circleB = new Circle();
circleB = circleA;

This object does not have a reference anymore: inaccessible!
Garbage Collection

- The object which does not have a reference cannot be used anymore.

- Such objects become a candidate for automatic garbage collection.

- Java collects garbage periodically and releases the memory occupied by such objects to be used in the future.
Using Objects

- Object’s data is accessed by using the dot notation

```java
Circle circleA = new Circle();
circleA.x = 25.0;
circleA.y = 25.0;
circleA.r = 3.0;
```

- Object’s methods are invoked by sending messages

```java
double area = circleA.area();
```
public class Circle {
    public double x, y; // center of the circle
    public double r;   // radius of the circle

    // Methods to return circumference and area
    public double circumference() {
        return 2 * 3.14 * r;
    }
    public double area() {
        return 3.14 * r * r;
    }
    public static void main(String[] args) {
        Circle circleA = new Circle();
        circleA.x = 25.0;
        circleA.y = 25.0;
        circleA.r = 3.0;

        double area = circleA.area();
        System.out.println("Area of the circle is "+ area);
    }
}
Class Files and Separate Compilation

- Each Java class definition is usually written in a file by itself
  - File begins with the name of the class
  - Ends with .java

- Class can be compiled separately

- Helpful to keep all class files used by a program in the same directory
public class Dog {
    public String name;  // Instance variables
    public String breed;
    public int age;

    // Method that returns nothing: void method
    public void writeOutput() {
        System.out.println("Name: " + name);
        System.out.println("Breed: " + breed);
        System.out.println("Age in calendar years: " + age);
        System.out.println("Age in human years: " +
            getAgeInHumanYears());
    }

    // Method that returns a value
    public int getAgeInHumanYears() {
        int humanAge = 0;
        if (age <= 2) {
            humanAge = age * 11;
        } else {
            humanAge = 22 + ((age - 2) * 5);
        }
        return humanAge;
    }
}
public class DogDemo {
    public static void main(String[] args) {
        Dog balto = new Dog();
        balto.name = "Balto";
        balto.age = 8;
        balto.breed = "Siberian Husky";
        balto.writeOutput();

        Dog scooby = new Dog();
        scooby.name = "Scooby";
        scooby.age = 42;
        scooby.breed = "Great Dane";
        System.out.println(scooby.name + " is a " + scooby.breed + ".");
        System.out.print("He is " + scooby.age + " years old, or ");

        int humanYears = scooby.getAgeInHumanYears();
        System.out.println(humanYears + " in human years.");
    }
}

Name: Balto
Breed: Siberian Husky
Age in calendar years: 8
Age in human years: 52

Scooby is a Great Dane.
He is 42 years old, or 222 in human years.
Accessor and Mutator Methods

- A public method that returns data from a private instance variable is called an accessor method, a get method, or a getter.
  - The names of accessor methods typically begin with `get`.

- A public method that changes the data stored in one or more private instance variables is called a mutator method, a set method, or a setter.
  - The names of mutator methods typically begin with `set`. 
public class Circle {
    public double x, y;  // center of the circle
    public double r;    // radius of the circle

    public double getX() { return x; }
    public void setX(double centerX) { x = centerX; }
    public double getY() { return y; }
    public void setY(double centerY) { y = centerY; }
    public double getR() { return r; }
    public void setR(double radius) { r = radius; }

    // Methods to return circumference and area
    ...
}
Constructors

- A constructor is a special method that gets invoked “automatically” at the time of object creation.

- Constructors are normally used for initializing objects with default values unless different values are supplied.

- Constructors have the same name as the class name.

- Constructors cannot return values.

- A class can have more than one constructor as long as they have different signatures (i.e., different input arguments syntax).
public class Circle {
    public double x, y; // center of the circle
    public double r;    // radius of the circle

    // Constructor
    public Circle(double centerX, double centerY, double radius) {
        x = centerX;
        y = centerY;
        r = radius;
    }

    // Methods to return circumference and area
    ...
}

Circle aCircle = new Circle(10.0, 20.0, 5.0);
Multiple Constructors

- Sometimes we may want to initialize in a number of different ways, depending on the circumstance.

- This can be supported by having multiple constructors having different input arguments (signatures).
public class Circle {
    public double x, y; // center of the circle
    public double r;    // radius of the circle

    // Constructor
    public Circle(double centerX, double centerY, double radius) {
        x = centerX;
        y = centerY;
        r = radius;
    }

    public Circle(double radius) {
        x = 0; y = 0; r = radius;
    }

    public Circle() {
        x = 0; y = 0; r = 1.0;
    }

    // Methods to return circumference and area
    ...
}

Circle aCircle = new Circle(10.0, 20.0, 5.0);
Circle bCircle = new Circle(5.0);
Circle cCircle = new Circle();
Default and No-Argument Constructors

Every class must have at least one constructor

- If no constructors are declared, the compiler will create a default constructor
  - Takes no arguments and initializes instance variables to their initial values specified in their declaration or to their default values
    - Default values are **zero** for primitive numeric types, **false** for **boolean** values and **null** for references
Common Programming Error

- If a class has constructors, but none of the public constructors are no-argument constructors, and a program attempts to call a no-argument constructor to initialize an object of the class, a compilation error occurs.

- A constructor can be called with no arguments only if the class does not have any constructors (in which case the default constructor is called) or if the class has a public no-argument constructor.
The Keyword **this**

- The **this** keyword can be used to refer to the object itself.
- It is generally used for accessing class members (from its own methods) when they have the same name as those passed as arguments.

```java
public class Circle {
    public double x, y;    // center of the circle
    public double r;       // radius of the circle

    public double getX() { return x; }
    public void setX(double x) { this.x = x; }
    public double getY() { return y; }
    public void setY(double y) { this.y = y; }
    public double getR() { return r; }
    public void setR(double r) { this.r = r; }

    // Methods to return circumference and area
    ...
}
```
Static Variables

- Java supports definition of variables that can be accessed without creating objects of a class.
  - Such members are called Static members.

- This feature is useful when we want to create a variable common to all instances of a class.

- One of the most common example is to have a variable that could keep a count of how many objects of a class have been created.

- Java creates only one copy for a static variable which can be used even if the class is never instantiated.
Using Static Variables

- Define the variable by using the **static** keyword

```java
public class Circle {
    // Class variable, one for the Circle class.
    // To keep number of objects created.
    public static int numCircles;

    // Instance variables, one for each instance
    // of the Circle class.
    public double x, y, r;

    // Constructor
    Circle (double x, double y, double r)
        this.x = x;
        this.y = y;
        this.r = r;
        numCircles++;
}
```

```java
Circle circleA = new Circle(10, 12, 20); // numCircles = 1
Circle circleB = new Circle(5, 3, 10); // numCircles = 2
```
Instance vs. Static Variables

- **Instance variables**: One copy per object. Every object has its own instance variables.
  - e.g. $x, y, r$ (center and radius of the circle)

- **Static variables**: One copy per class.
  - e.g. `numCircles` (total number of circle objects created)
Static Methods

- A class can have methods that are defined as `static`.

- Static methods can be accessed without using objects. Also, there is **NO** need to create objects.

- Static methods are generally used to group related library functions that don’t depend on data members of its class.
  - e.g., Math library functions.
Using Static Methods

class Comparator {
    
    public static int max(int a, int b) {
        if (a > b)
            return a;
        else
            return b;
    }

    public static String max(String a, String b) {
        if (a.compareTo(b) > 0)
            return a;
        else
            return b;
    }

}

// Max methods are directly accessed using ClassName.  
// NO Objects created.
System.out.println(Comparator.max(5, 10));
System.out.println(Comparator.max("ANKARA", "SAMSUN"));
More Static Methods: The **Math** Class

- It is like including libraries in C language
- It contains standard mathematical methods
  - They are all static
  - `Java.lang.Math`

```java
Math.pow(2.0, 3.0)  // 8
Math.max(5, 6)     // 6
Math.round(6.2)    // 6
Math.sqrt(4.0)     // 2.0
```
Object Cleanup (Destructor)

- Recall: Memory deallocation is automatic in Java
  - No dangling pointers and no memory leak problem.
- Java allows to define `finalize` method, which is invoked (if defined) just before the object destruction.
- This presents an opportunity to perform record maintenance operation or clean up any special allocations made by the user.
- The finalize method will be called by the Garbage Collector, but when this will happen is not deterministic. Try to avoid finalize.

```java
protected void finalize() throws IOException {
    Circle.numCircles = Circle.numCircles--;
    System.out.println("Number of circles:" + Circle.num_circles);
}
```
Wrapper Classes

Each of Java's primitive data types has a class dedicated to it.

- **Boolean, Byte, Character, Integer, Float, Double, Long, Short**
- These are known as wrapper classes, because they "wrap" the primitive data type into an object of that class.
- They contain useful predefined constants and methods.
- The wrapper classes are part of the `java.lang` package, which is imported by default into all Java programs.
- Since Java 5.0 we have autoboxing and unboxing.

```java
// Defining objects of wrapper class
Integer x = new Integer(33);
Integer y = 33; // Autoboxing
int yInt = y; // Unboxing

// Convert string to an integer
String s = "123";
int i = Integer.parseInt(s);

//Converting from hexadecimal to decimal
Integer hex2Int = Integer.valueOf("D", 16);
```
Parameter Passing

- Method parameters which are objects are passed by reference.

- Copy of the reference to the object is passed into method, original value unchanged (e.g. circleB parameter in next slide)
public class ReferenceTest {

    public static void main (String[] args){
        Circle c1 = new Circle(5, 5, 20);
        Circle c2 = new Circle(1, 1, 10);
        System.out.println("c1 Radius = "+ c1.getRadius());
        System.out.println("c2 Radius = "+ c2.getRadius());

        parameterTester(c1, c2);

        System.out.println("c1 Radius = "+ c1.getRadius());
        System.out.println("c2 Radius = "+ c2.getRadius());
    }

    public static void parameterTester(Circle circleA, Circle circleB){
        circleA.setRadius(15);
        circleB = new Circle(0, 0, 100);

        System.out.println("circleA Radius = "+ circleA.getRadius());
        System.out.println("circleB Radius = "+ circleB.getRadius());
    }
}

c1 Radius =  20.0

c2 Radius = 10.0

circleA Radius = 15.0

circleB Radius = 100.0

c1 Radius =  15.0

c2 Radius = 10.0
Delegation

- Ability for a class to delegate its responsibilities to another class.

- A way of making an object invoking services of other objects through containership.
Using Delegation

```java
public class Point {
    private double xCoord;
    private double yCoord;

    public double getXCoord(){
        return xCoord;
    }
    public double getYCoord(){
        return yCoord;
    }
}

public class Circle {
    private Point center;
    public double getCenterX(){
        return center.getXCoord(); // Delegation
    }
    public double getCenterY(){
        return center.getYCoord(); // Delegation
    }
}
```
Information Hiding

- Programmer using a class method need **not** know details of implementation
  - Only needs to know **what** the method does

**Information hiding:**
- Designing a method so it can be used without knowing details

- Also referred to as **encapsulation**

- Method design should separate **what** from **how**
Encapsulation

**Encapsulation**: Hiding implementation details of an object from its clients.

- Encapsulation provides *abstraction*.
  - separates external view (behavior) from internal view (state)
- Encapsulation protects the integrity of an object's data.
Visibility Modifiers

- All parts of a class have visibility modifiers
  - Java keywords
  - public, protected, private
  - do not use these modifiers on local variables (syntax error)

- public means that constructor, method, or field may be accessed outside of the class.
  - part of the interface
  - constructors and methods are generally public

- private means that part of the class is hidden and inaccessible by code outside of the class
  - part of the implementation
  - data fields are generally private
The **public** and **private** Modifiers

- Type specified as **public**
  - Any other class can directly access that object by name

- Classes are generally specified as **public**

- Instance variables are usually **not** **public**
  - Instead specify as **private**
Private fields

- A field can be declared *private*.
  - No code outside the class can access or change it.

  ```java
  private type name;
  ```

- Examples:

  ```java
  private int id;
  private String name;
  ```

- Client code sees an error when accessing private fields:

  ```java
  PointMain.java:11: x has private access in Point
  System.out.println("p1 is (" + p1.x + ", " + p1.y + ")");
  ^
  ```
We can provide methods to get and/or set a field's value:

```java
// A "read-only" access to the x field ("accessor")
public int getX() {
    return x;
}

// Allows clients to change the x field ("mutator")
public void setX(int newX) {
    x = newX;
}
```

Client code will look more like this:

```java
System.out.println("p1: (" + p1.getX() + ", " + p1.getY() + ")");
p1.setX(14);
```
public class Rectangle
{
    private int width;
    private int height;
    private int area;

    public void setDimensions (int newWidth, int newHeight)
    {
        width = newWidth;
        height = newHeight;
        area = width * height;
    }
    public int getArea ()
    {
        return area;
    }
}
// A Point object represents an (x, y) location.
public class Point {
    private int x;
    private int y;

    public Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }

    public double distanceFromOrigin() {
        return Math.sqrt(x * x + y * y);
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

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

    public void translate(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
}

Point class
public class PointMain4 {
    public static void main(String[] args) {
        // create two Point objects
        Point p1 = new Point(5, 2);
        Point p2 = new Point(4, 3);

        // print each point
        System.out.println("p1: (" + p1.getX() + ", " + p1.getY() + ")");
        System.out.println("p2: (" + p2.getX() + ", " + p2.getY() + ")");

        // move p2 and then print it again
        p2.translate(2, 4);
        System.out.println("p2: (" + p2.getX() + ", " + p2.getY() + ")");
    }
}

// Output:

p1 is (5, 2)
p2 is (4, 3)
p2 is (6, 7)
Encapsulation

- Consider example of driving a car
  - We see and use break pedal, accelerator pedal, steering wheel – know what they do
  - We do not see mechanical details of how they do their jobs

- Encapsulation divides class definition into
  - Class interface
  - Class implementation
Encapsulation

- **Class interface**
  - Tells **what** the class does
  - Gives headings for public methods and comments about them

- **Class implementation**
  - Contains private variables
  - Includes definitions of public and private methods
Encapsulation

- A well encapsulated class definition

```
Class Definition

Implementation:
Private instance variables
Private constants
Private methods
Bodies of public methods

Interface:
Comments
Headings of public methods
Public named constants

Programmer who uses the class
```
Encapsulation – Best Practices

- Preface class definition with comment on how to use class.
- Declare all instance variables in the class as private.
- Provide public accessor methods to retrieve data and provide public methods to manipulate data:
  - Such methods could include public mutator methods.
- Place a comment before each public method heading that fully specifies how to use method.
- Make any helping methods private.
- Write comments within class definition to describe implementation details.
Benefits of encapsulation

- Provides **abstraction** between an object and its clients.

- Protects an object from unwanted access by clients.
  - A bank app forbids a client to change an **Account's** balance.

- Allows you to change the class implementation.
  - **Point** could be rewritten to use polar coordinates (radius $r$, angle $\theta$), but with the same methods.

- Allows you to constrain objects' state (**invariants**).
  - Example: Only allow **Points** with non-negative coordinates.
Software Development Observations

- Interfaces change less frequently than implementations.

- When an implementation changes, implementation-dependent code must change accordingly.

- Hiding the implementation reduces the possibility that other program parts will become dependent on class-implementation details.
// Fig. 8.1: Time1.java
// Time1 class declaration maintains the time in 24-hour format.

public class Time1 {

    private int hour; // 0 – 23
    private int minute; // 0 – 59
    private int second; // 0 – 59

    // set a new time value using universal time; ensure that
    // the data remains consistent by setting invalid values to zero
    public void setTime( int h, int m, int s ) {
        hour = ( ( h >= 0 && h < 24 ) ? h : 0 ); // validate hour
        minute = ( ( m >= 0 && m < 60 ) ? m : 0 ); // validate minute
        second = ( ( s >= 0 && s < 60 ) ? s : 0 ); // validate second
    } // end method setTime

} // end class Time1
// convert to String in universal-time format (HH:MM:SS)
public String toUniversalString()
{
    return String.format("%02d:%02d:%02d", hour, minute, second);
} // end method toUniversalString

// convert to String in standard-time format (H:MM:SS AM or PM)
public String toString()
{
    return String.format("%d:%02d:%02d %s",
            ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ),
            minute, second, ( hour < 12 ? "AM" : "PM" )
    );
} // end method toString

} // end class Time1
// Fig. 8.2: Time1Test.java
// Time1 object used in an application.

public class Time1Test
{
    public static void main( String args[] )
    {
        // create and initialize a Time1 object
        Time1 time = new Time1(); // invokes Time1 constructor

        // output string representations of the time
        System.out.print( "The initial universal time is: " );
        System.out.println( time.toUniversalString() );
        System.out.print( "The initial standard time is: " );
        System.out.println( time.toString() );
        System.out.println(); // output a blank line
    }
}
// change time and output updated time
   time.setTime( 13, 27, 6 );
   System.out.print( "Universal time after setTime is: " );
   System.out.println( time.toUniversalString() );
   System.out.print( "Standard time after setTime is: " );
   System.out.println( time.toString() );
   System.out.println();

   // set time with invalid values; output updated time
   time.setTime( 99, 99, 99 );
   System.out.println( "After attempting invalid settings:" );
   System.out.print( "Universal time: " );
   System.out.println( time.toUniversalString() );
   System.out.print( "Standard time: " );
   System.out.println( time.toString() );
}
} // end class Time1Test

The initial universal time is: 00:00:00
The initial standard time is: 12:00:00 AM

Universal time after setTime is: 13:27:06
Standard time after setTime is: 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
Software Development Observations & Tips

- When one object of a class has a reference to another object of the same class, the first object can access all the second object’s data and methods (including those that are private).

- When implementing a method of a class, use the class’s `set` and `get` methods to access the class’s private data. This simplifies code maintenance and reduces the likelihood of errors.

- This architecture helps hide the implementation of a class from its clients, which improves program modifiability.
**final Instance Variables**

- **final instance variables**
  - Keyword `final`
    - Specifies that a variable is not modifiable (is a constant)
  - `final` instance variables can be initialized at their declaration
    - If they are not initialized in their declarations, they must be initialized in all constructors

- If an instance variable should not be modified, declare it to be `final` to prevent any erroneous modification.
**static final Instance Variables**

- A *final* field should also be declared *static* if it is initialized in its declaration.

- Once a *final* field is initialized in its declaration, its value can never change.

- Therefore, it is not necessary to have a separate copy of the field for every object of the class.

- Making the field *static* enables all objects of the class to share the *final* field.

- Example: `public static final double PI = 3.141592;`
An automobile class outline as a UML class diagram

- fuel: double
- speed: double
- license: String

+ accelerate(double pedalPressure): void
+ decelerate(double pedalPressure): void
UML Class Diagrams

Example:

**Purchase**

class

- name: String
- groupCount: int
- GroupPrice: double
- numberBought: int

Plus signs imply public access

Minus signs imply private access
UML Class Diagrams

- Contains more than interface, less than full implementation

- Usually written *before* class is defined

- Used by the programmer defining the class
  - Contrast with the *interface* used by programmer who uses the class
Packages and Importing

- A **package** is a collection of classes grouped together into a folder

- Name of folder is name of package

- Each class
  - Placed in a separate file
  - Has this line at the beginning of the file
    ```java
    package Package_Name;
    ```

- Classes use packages by use of **import** statement
Package Names and Directories

- Package name tells compiler path name for directory containing classes of package

- Search for package begins in class path base directory
  - Package name uses dots in place of / or \n
- Name of package uses relative path name starting from any directory in class path
Package Names and Directories

- A package name

\texttt{myjavastuff}

\texttt{libraries}

\texttt{general}

\texttt{utilities}

\texttt{general.utilities is the package name.}

\texttt{Classes in the package}

\texttt{AClass.java}

\texttt{AnotherClass.java}

\texttt{\textbackslash myjavastuff\textbackslash libraries is a class path base directory (is on the class path).}
Time Class Case Study: Creating Packages

To declare a reusable class

- Declare a `public` class

- Add a `package` declaration to the source-code file
  - must be the very first executable statement in the file
    
    - Package name example: `com.deitel.jhtp6.ch08`
    
    - `package` name is part of the fully qualified class name
      
      » Distinguishes between multiple classes with the same name belonging to different packages
      
      » Prevents name conflict (also called name collision)
// Fig. 8.18: Time1.java
// Time1 class declaration maintains the time in 24-hour format.
package com.deitel.jhtp6.ch08;

public class Time1 {
    private int hour;    // 0 - 23
    private int minute;  // 0 - 59
    private int second;  // 0 - 59

    // set a new time value using universal time; perform
    // validity checks on the data; set invalid values to zero
    public void setTime(int h, int m, int s)
    {
        hour = ((h >= 0 && h < 24) ? h : 0); // validate hour
        minute = ((m >= 0 && m < 60) ? m : 0); // validate minute
        second = ((s >= 0 && s < 60) ? s : 0); // validate second
    } // end method setTime
}
Compile the class so that it is placed in the appropriate package directory structure

- Example: our package should be in the directory
  
  ```java
  com
deitel
  jhtp6
  ch08
  ```

- *javac* command-line option `-d`
  - *javac* creates appropriate directories based on the class’s package declaration
  - A period (.) after `-d` represents the current directory
Import the reusable class into a program

- Single-type-import declaration
  - Imports a single class
  - Example: `import java.util.Random;`

- Type-import-on-demand declaration
  - Imports all classes in a package
  - Example: `import java.util.*;`
Overloading Basics

- When two or more methods have same name within the same class

- Java distinguishes the methods by number and types of parameters
  - If it cannot match a call with a definition, it attempts to do type conversions

- A method's name and number and type of parameters is called the *signature*
/** This class illustrates overloading. */

class Overload {

    public static void main (String [] args) {
        double average1 = Overload.getAverage (40.0, 50.0);
        double average2 = Overload.getAverage (1.0, 2.0, 3.0);
        char average3 = Overload.getAverage ('a', 'c');
        System.out.println ("average1 = " + average1);
        System.out.println ("average2 = " + average2);
        System.out.println ("average3 = " + average3);
    }

    public static double getAverage (double first, double second) {
        return (first + second) / 2.0;
    }

    public static double getAverage (double first, double second, double third) {
        return (first + second + third) / 3.0;
    }

    public static char getAverage (char first, char second) {
        return (char) (((int) first + (int) second) / 2);
    }
}
Overloading and Type Conversion

- Overloading and automatic type conversion can conflict

- Remember the compiler attempts to overload before it does type conversion

- Use descriptive method names, avoid overloading when possible
You cannot overload a method where the only difference is the type of value returned.

```java
/**
   * Returns the weight of the pet.
   */
public double getWeight()

/**
   * Returns '+' if overweight, '-' if underweight, and '*' if weight is OK.
   */
public char getWeight()
```
Summary

- Classes, objects, and methods are the basic components used in Java programming.

- Constructors allow seamless initialization of objects.

- Classes can have static members, which serve as global members of all objects of a class.

- Objects can be passed as parameters and they can be used for exchanging messages.
Summary

- Usage of visibility modifiers for encapsulation
- Separation of interface and implementation is important
- Class designers use UML notation to describe classes
- Use packages for software reusability
- Overloading must be done with care