Contents

- Overview
- Super-classes & Sub-classes
- Overriding
- Polymorphism
- `equals` method
- `ArrayList` class
- Modifiers
Overview
4 Principles of OOP

- Encapsulation
  - Hiding code
  - All non-static members are private
  - Accessor and mutator methods to utilize data
Overview
4 Principles of OOP

- Abstraction
  - Creating boundaries so that an object ONLY contains relevant properties
  - Using multiple java files to solve a problem
  - Logic organization for multiple java files
Overview
4 Principles of OOP

- Inheritance
- Reuse code
- Creating relationships with objects
Overview

4 Principles of OOP

- Polymorphism
  - Having multiple methods with the same name, but slightly different functionality
Super-classes & Sub-classes

- Inheritance enables programmers to define a base class (super-class) and later extend it to more specialized classes (sub-classes).

- is-a relationship
  - A car is a vehicle
    - vehicle is a base-class and car is a sub-class
  - A student is a person
    - person is a base-class and student is a sub-class

- extends
  - subclass has access to all non-private properties and behaviors of super-class.
Vehicle

_constructors
Vehicle()

_getters
getSpeed: double

_setters
setSpeed(double)

Car

_constructors
Car()

 accelerate(double)

decelerate(double)
public class Vehicle
{
    private double _speed;
    public Vehicle()
    {
        _speed = 0;
    }
    public Vehicle(double speed)
    {
        _speed = speed;
    }
    public void setSpeed(double speed)
    {
        _speed = speed;
    }
    public double getSpeed()
    {
        return _speed;
    }
}

public class Car extends Vehicle
{
    Car()
    {
    }
    Car(int initialSpeed)
    {
        setSpeed(initialSpeed);
    }
    public void accelerate()
    {
        setSpeed(getSpeed()+1);
    }
    public void decelerate()
    {
        setSpeed(getSpeed()-1);
    }
}
public class MainCar
{
    public static void main(String [] args)
    {
        Car c = new Car(10d);
        for(int i=0; i<10; i++)
        {
            c.accelerate();
        }
        System.out.println("Speed =" + c.getSpeed());
        c.decelerate();
        System.out.println("Speed =" + c.getSpeed());
    }
}
**super keyword**

- Can be used to refer to the super-class

```java
public class Vehicle {
    private double _speed;
    public Vehicle() {
        _speed = 0;
    }
    public Vehicle(double speed) {
        _speed = speed;
    }
    public void setSpeed(double speed) {
        _speed = speed;
    }
    public double getSpeed() {
        return _speed;
    }
}

public class Car extends Vehicle {
    Car() {
        super();
    }
    Car(int initialSpeed) {
        super(initialSpeed);
    }
    public void accelerate() {
        super.setSpeed(getSpeed()+1);
    }
    public void decelerate() {
        super.setSpeed(getSpeed()-1);
    }
}
```
Superclasses & Subclasses

Summary

- ______ is defined as a is-a relationship
- Inheritance is achieved using the ______ keyword
- A _______ inherits the features of the super-class
- A _____________ is the general or base-class
- A sub-class has _____ to all non-private properties and behaviors of the super-class
- The ______ keyword refers to the super-class
Superclasses & Subclasses

Summary

- **Inheritance** is defined as a is-a relationship
- Inheritance is achieved using the `extends` keyword
- A **subclass** inherits the features of the **super-class**
- A **super-class** is the general or base-class
- A sub-class has **access** to all non-private properties and behaviors of the super-class
- The **super** keyword refers to the super-class
Superclasses & Subclasses

Summary

- REMINDERS
  - DO NOT EXTEND UNLESS NECESSARY

- JAVA ONLY SUPPORTS SINGLE INHERITANCE
  - Interfaces (introduced later) is a mechanism to do multiple inheritances
An overriding method in the sub-class has the same signature and the same return type as in its super-class.

Typically done for the toString method.

```java
public class Vehicle {
    ...
    public void setSpeed(double speed) {
        _speed = speed;
    }
    ...
}

public class Car extends Vehicle {
    ...
    public void setSpeed(double speed) {
        // scale by 10
        setSpeed(speed * 10);
    }
}
```
overriding vs overloading

**overloading**
- Same .java file
- Multiple methods with same name, but different signatures

**overriding**
- Inheritance
- Same name & same signature
- Different implementation in sub-class
toString method

- Every class in Java is derived from the Object class
  - Implicit inheritance (extends) of the Object class
- Object class
  - toString():String
- Override to print a custom message for our class
public class MainCar {
    public static void main(String[] args) {
        Vehicle v = new Vehicle();
        System.out.println(v.toString());
        Car c = new Car();
        System.out.println(c.toString());
    }
}

public class Car extends Vehicle {
    Car() {
        super();
    }
    Car(int initialSpeed) {
        super(initialSpeed);
    }
    public void accelerate() {
        super.setSpeed(getSpeed() + 1);
    }
    public void decelerate() {
        super.setSpeed(getSpeed() - 1);
    }
    public String toString() {
        System.out.println("Speed is " + getSpeed());
    }
}
Polymorphism

- A variable of a supertype can refer to a subtype object
Polymorphism

- A variable of a supertype can refer to a subtype object

- Advantage
  - Makes Object-Oriented Programming easier to manage
  - Class creator does not need to know how class users plan to use their code
Polymorphism

- A variable of a supertype can refer to a subtype object

**Advantage**

- Makes Object-Oriented Programming easier to manage
  - Class creator does not need to know how class users plan to use their code

**Code Example**
The equals() method is defined in the Object class
- All objects have access to it (similar to the toString() method)

Typically, this will be overridden

Equals vs ==
- The equals method checks if the contents of two objects are the same
- (for objects) == checks if the two objects refer to the same memory location
Equals vs. ==

```
Hello
```

```
Hello
```

```
s1
```

```
s2
```

```
s3
```
String s1 = new String("Hello");
String s2 = s1;
String s3 = new String("Hello");

System.out.println(s1.equals(s2)); // returns true
System.out.println(s1.equals(s3)); // returns true
System.out.println(s1 == s2); // returns true
System.out.println(s1 == s3); // returns false
Code Example
Arrays require to know the size before using.

Arrays have a bounded size.

Very limited in real-world applications.

Limited functionality to manage items.
Arrays

- Arrays require to know the size before using
- Arrays have a bounded size
- Very limited in real-world applications
- Limited functionality to manage items

ArrayLists

- ArrayLists do not require to know the size EVER
- ArrayLists do not have a bounded size (an infinite number of items can be added)
- Very useful for real-world applications
- Extended functionality to manage items efficiently
<table>
<thead>
<tr>
<th>Modifiers</th>
<th>Accessed from the same class</th>
<th>Accessed from the same package</th>
<th>Accessed from a subclass (extends) in a different package</th>
<th>Accessed from a different package</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**protected**
<table>
<thead>
<tr>
<th>Modifiers</th>
<th>Accessed from the same class</th>
<th>Accessed from the same package</th>
<th>Accessed from a subclass (extends) in a different package</th>
<th>Accessed from a different package</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>protected</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Modifiers</td>
<td>Accessed from the same class</td>
<td>Accessed from the same package</td>
<td>Accessed from a subclass (extends) in a different package</td>
<td>Accessed from a different package</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------------------</td>
<td>--------------------------------</td>
<td>----------------------------------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>public</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>protected</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Default (no modifier)</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modifiers</td>
<td>Accessed from the same class</td>
<td>Accessed from the same package</td>
<td>Accessed from a subclass (extends) in a different package</td>
<td>Accessed from a different package</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------------------</td>
<td>-------------------------------</td>
<td>----------------------------------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>public</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>protected</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Default (no modifier)</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>private</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
a final data field (class member) is a constant
- final static int myInt = 5;

a final local variable (declared inside a method) is a constant
- final int localInt = 5;

a final class cannot be extended
- public final class MyClass

a final method cannot be overridden by its sub-classes
- public final void runMyMethod()