Java Object-Oriented Programming

by BPI

What We’ll Cover Today

1. Four Pillars of OOP
2. Abstract Classes & Interfaces
3. Benefits of OOP

Object-Oriented Programming

- “A method of programming based on a hierarchy of classes, and well-defined and cooperating classes”- Oracle Website
- A programming paradigm focused on concepts of objects for effective modeling of systems
- Java is inherently object-oriented
- Provide brief example by mentioning a program that takes an object and processes the object

Four Pillars of OOP

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction

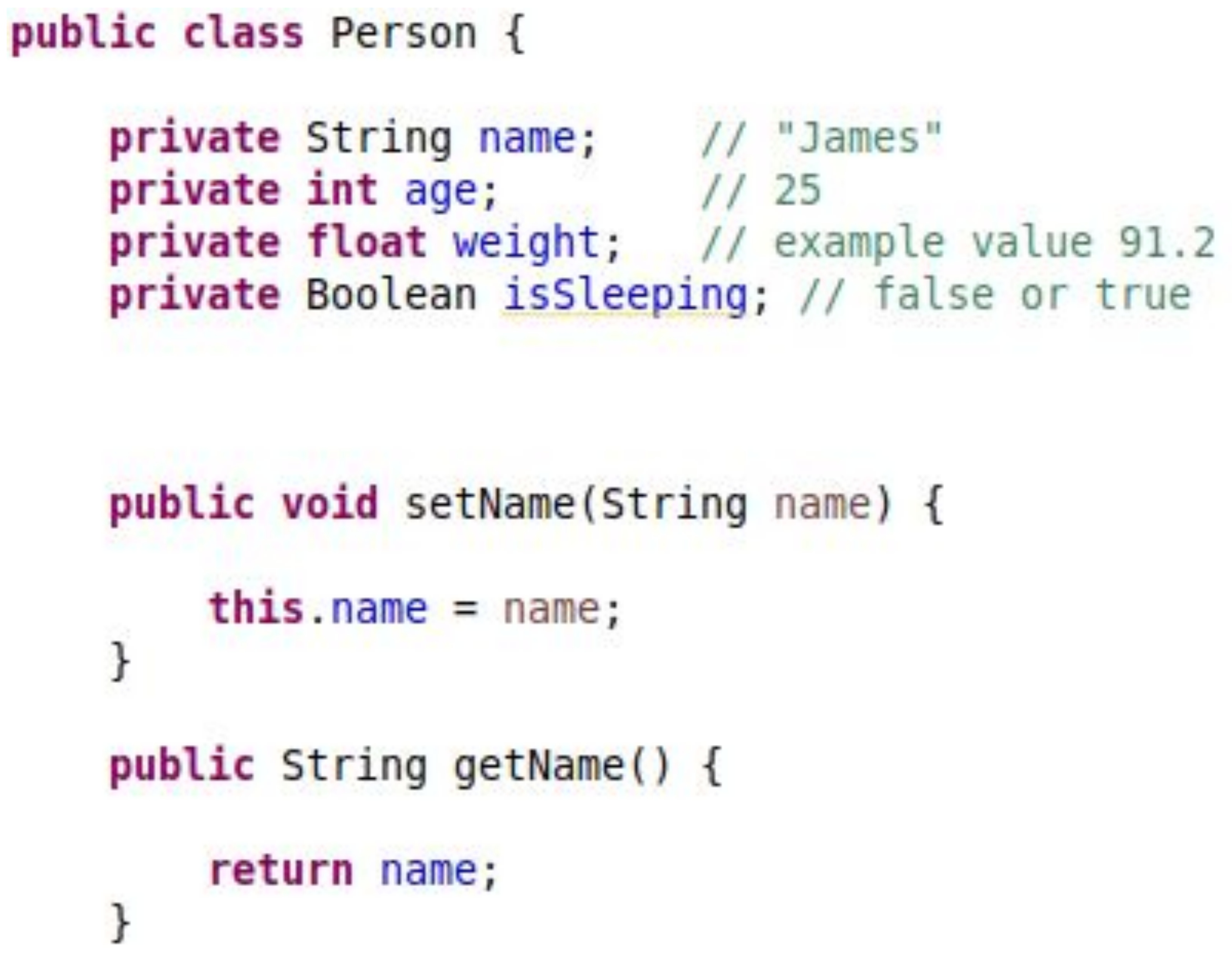
Encapsulation

- Bundling of data and methods into a single unit called a class.
- Encapsulate the internal data of an object and hiding it from outside access

Encapsulation - Example

- A Person has a name, age, height, weight (all marked as private) and can move() and speak()

Encapsulation - Example



```java
public class Person {

    private String name;     // "James"
    private int age;         // 25
    private float weight;    // example value 91.2
    private Boolean isSleeping; // false or true



    public void setName(String name) {

        this.name = name;
    }

    public String getName() {

        return name;
    }
```

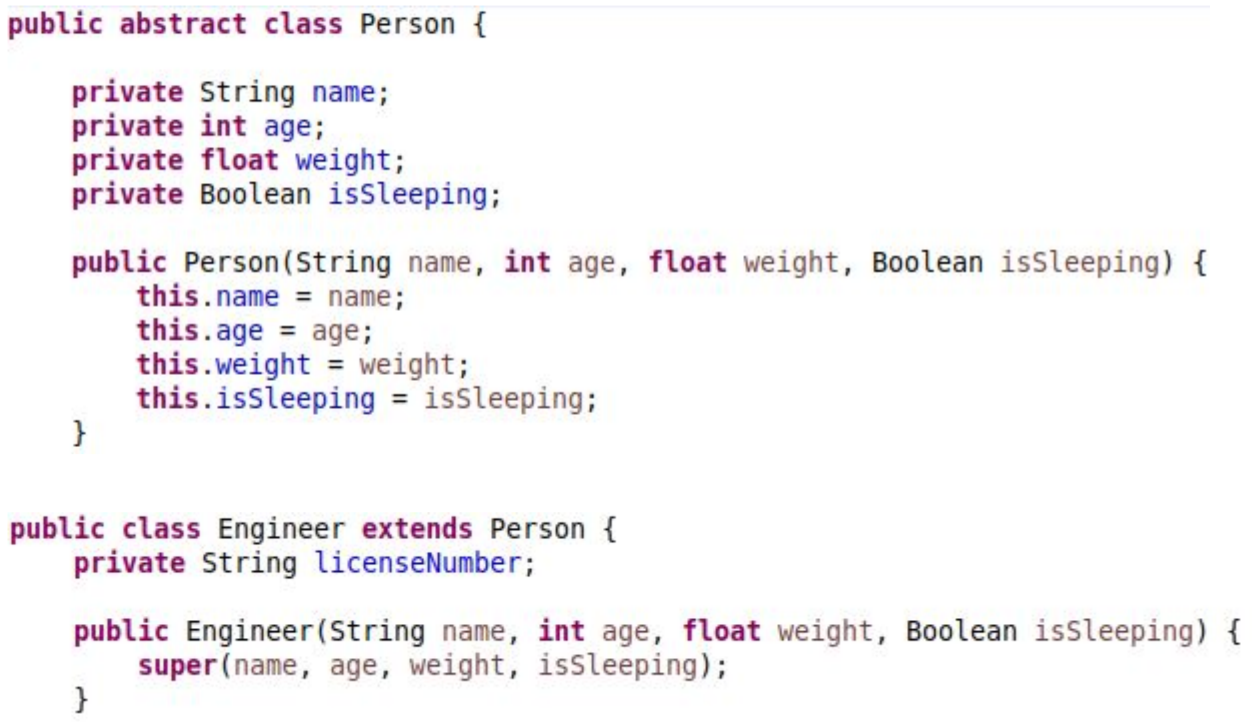
Inheritance

- allows a class to inherit the attributes and methods of a parent class
- Promotes code reusability
- Related concepts are access modifiers, “super” keyword

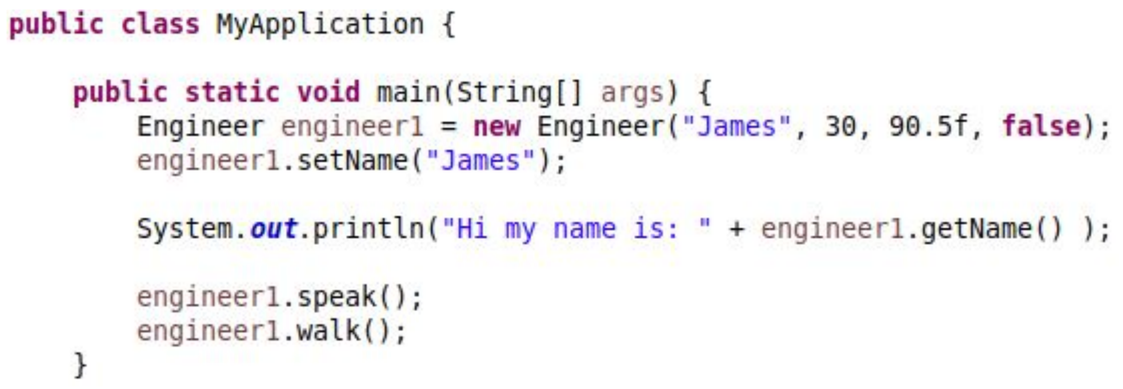
Inheritance - Example

- A subclass Engineer can extend the abstract class "Person". An engineer inherits the traits of a Person. They can also move(), speak() and they also have a name, age, etc.

Inheritance - Example



```
public abstract class Person {

    private String name;
    private int age;
    private float weight;
    private Boolean isSleeping;

    public Person(String name, int age, float weight, Boolean isSleeping) {
        this.name = name;
        this.age = age;
        this.weight = weight;
        this.isSleeping = isSleeping;
    }


public class Engineer extends Person {
    private String licenseNumber;

    public Engineer(String name, int age, float weight, Boolean isSleeping) {
        super(name, age, weight, isSleeping);
    }
```



```
public class MyApplication {

    public static void main(String[] args) {
        Engineer engineer1 = new Engineer("James", 30, 90.5f, false);
        engineer1.setName("James");

        System.out.println("Hi my name is: " + engineer1.getName() );

        engineer1.speak();
        engineer1.walk();
    }
```

Polymorphism

- Allows different objects of different classes to be treated as one common super class
- Promotes code flexibility
- Can be implemented using “implements”, using overriding and overloading
- Focuses on how an object behaves differently
- Programming to an Interface

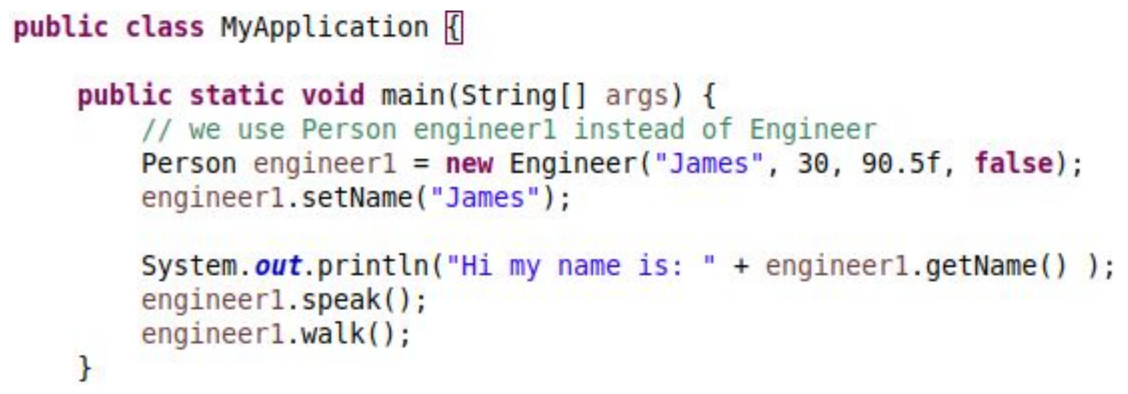
Polymorphism

- “Programming to an interface” means writing code that depends on an Interface rather than its subclasses
- Programming with objects based on what they can do and not what they are

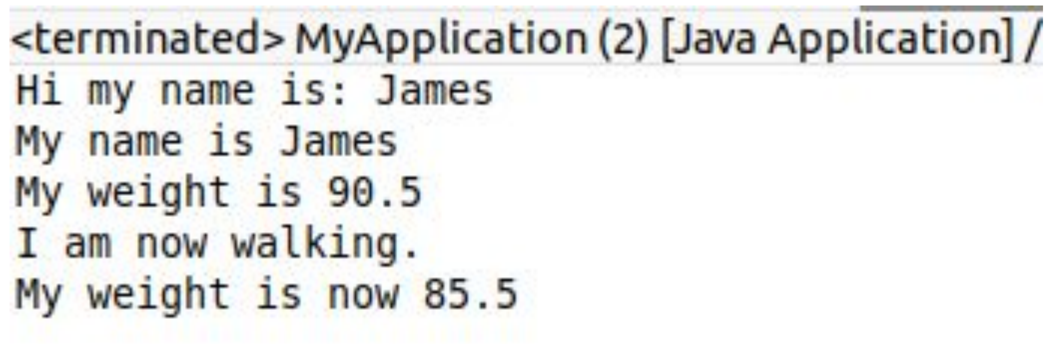
Polymorphism - Example

- An Engineer is a Person. A Programmer is also a person. They can share behavior and they can also have different behavior but they are all treated as a “Person”

Polymorphism - Example



```
public class MyApplication {

    public static void main(String[] args) {
        // we use Person engineer1 instead of Engineer
        Person engineer1 = new Engineer("James", 30, 90.5f, false);
        engineer1.setName("James");

        System.out.println("Hi my name is: " + engineer1.getName() );
        engineer1.speak();
        engineer1.walk();
    }
```



```
<terminated> MyApplication (2) [Java Application] /
Hi my name is: James
My name is James
My weight is 90.5
I am now walking.
My weight is now 85.5
```

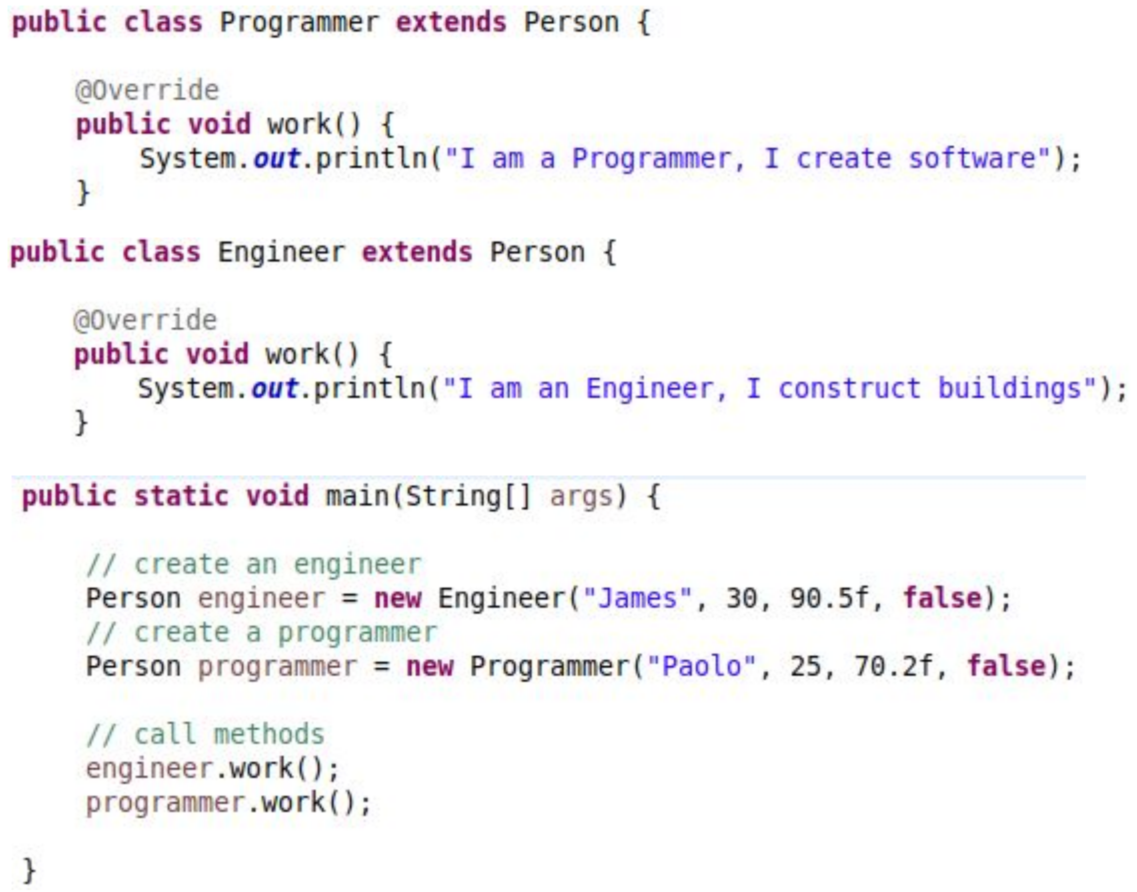
Abstraction

- Simplifies classes by hiding the complexities or processes not relevant for outside objects and showing only the essential information
- Promotes intuitive and efficient programming as we are only concerned with what an object has and what an object does
- Focuses on what an object does
- Can be implemented using abstract classes or interfaces

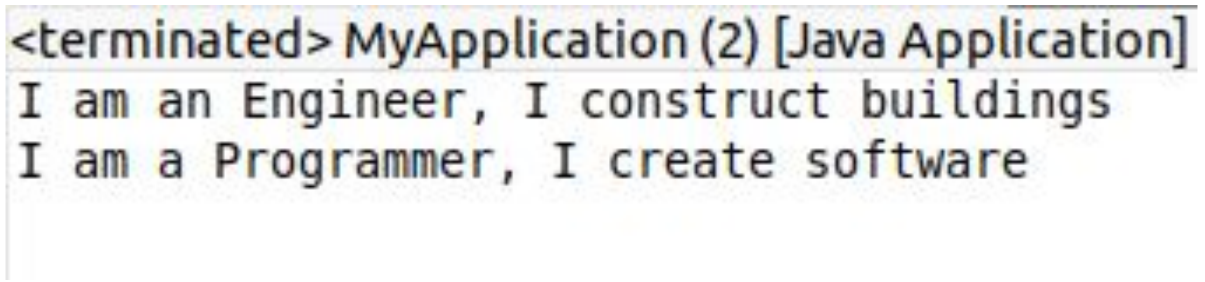
Abstraction - Example

- A Person can have the method work(), its subclasses can have their own implementation of this method. work() for the Engineer can be “I’m an Engineer, I construct buildings” and work() for the Programmer can be “I’m a Programmer, I create software!”.

Abstraction - Example



```
public class Programmer extends Person {

    @Override
    public void work() {
        System.out.println("I am a Programmer, I create software");
    }

public class Engineer extends Person {

    @Override
    public void work() {
        System.out.println("I am an Engineer, I construct buildings");
    }

public static void main(String[] args) {

    // create an engineer
    Person engineer = new Engineer("James", 30, 90.5f, false);
    // create a programmer
    Person programmer = new Programmer("Paolo", 25, 70.2f, false);

    // call methods
    engineer.work();
    programmer.work();

}
```



```
<terminated> MyApplication (2) [Java Application]
I am an Engineer, I construct buildings
I am a Programmer, I create software
```

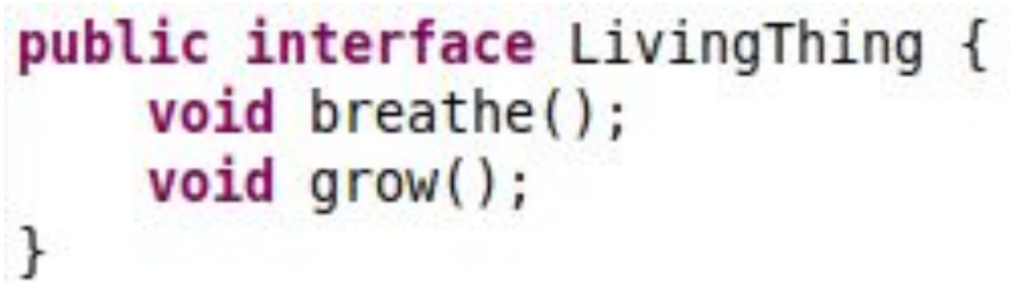
Interfaces

- A reference type similar to a class; uses keyword interface
- Treated as a kind of contract or blueprint that implementing classes must adhere to
- Can contain only fields that are constants, abstract methods, default methods, static methods
- All declared fields are automatically public, static, and final. All declared methods are public
- Cannot be instantiated; only implemented by classes or extended by other interfaces

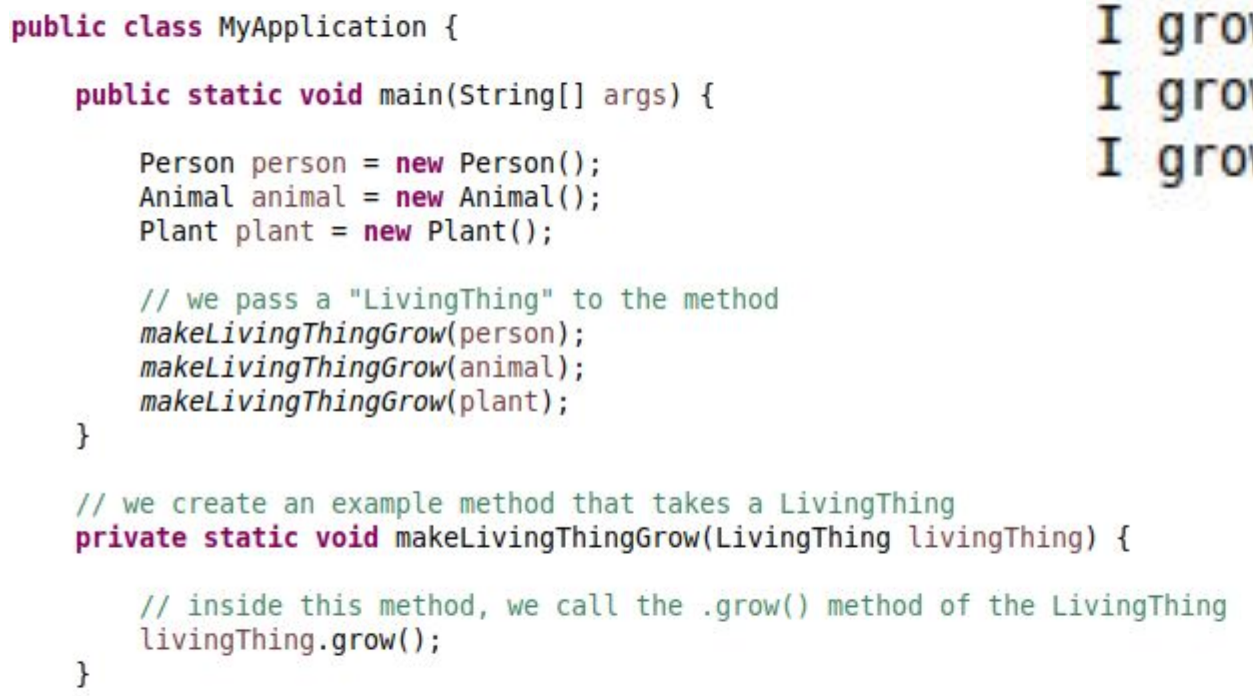
Interfaces - Example

- *LivingThings* interface and Person, Animal, Plant classes

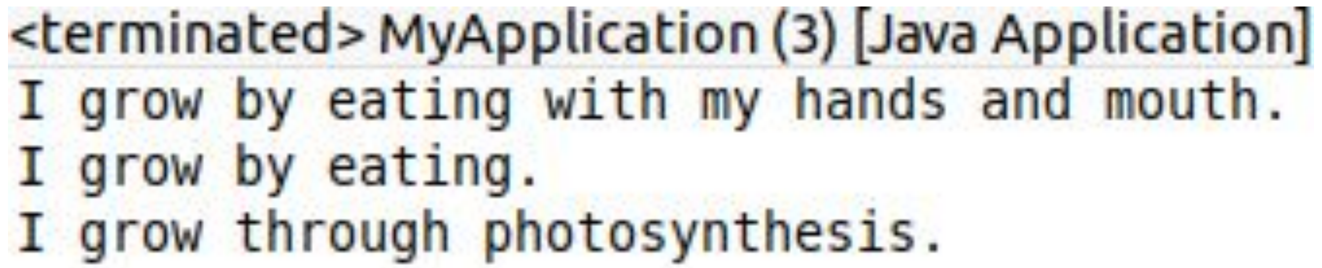
Interface - Example



```
public interface LivingThing {
    void breathe();
    void grow();
}
```



```
public class MyApplication {

    public static void main(String[] args) {

        Person person = new Person();
        Animal animal = new Animal();
        Plant plant = new Plant();

        // we pass a "LivingThing" to the method
        makeLivingThingGrow(person);
        makeLivingThingGrow(animal);
        makeLivingThingGrow(plant);
    }

    // we create an example method that takes a LivingThing
    private static void makeLivingThingGrow(LivingThing livingThing) {

        // inside this method, we call the .grow() method of the LivingThing
        livingThing.grow();
    }
```



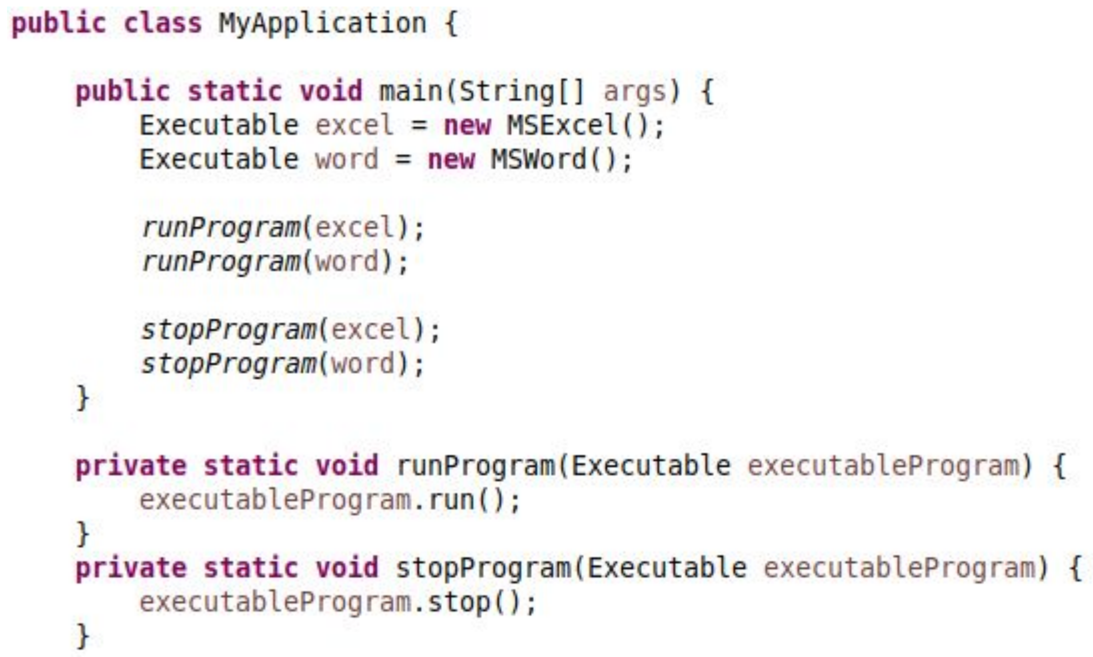
```
<terminated> MyApplication (3) [Java Application]
I grow by eating with my hands and mouth.
I grow by eating.
I grow through photosynthesis.
```

Interface - Example (2)

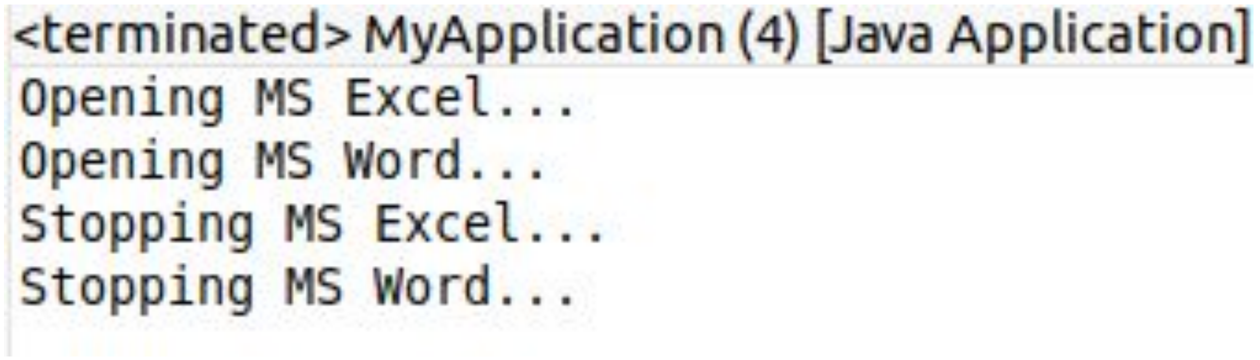
- Provide another example that uses services

Seatwork - Individual (M2_ACTIVITY4)

- Modify the classes: Executable.java, MSWord.java, MSExcel.java to make the program runnable.



```
public class MyApplication {

    public static void main(String[] args) {
        Executable excel = new MSExcel();
        Executable word = new MSWord();

        runProgram(excel);
        runProgram(word);

        stopProgram(excel);
        stopProgram(word);
    }

    private static void runProgram(Executable executableProgram) {
        executableProgram.run();
    }
    private static void stopProgram(Executable executableProgram) {
        executableProgram.stop();
    }
```



```
<terminated> MyApplication (4) [Java Application]
Opening MS Excel...
Opening MS Word...
Stopping MS Excel...
Stopping MS Word...
```

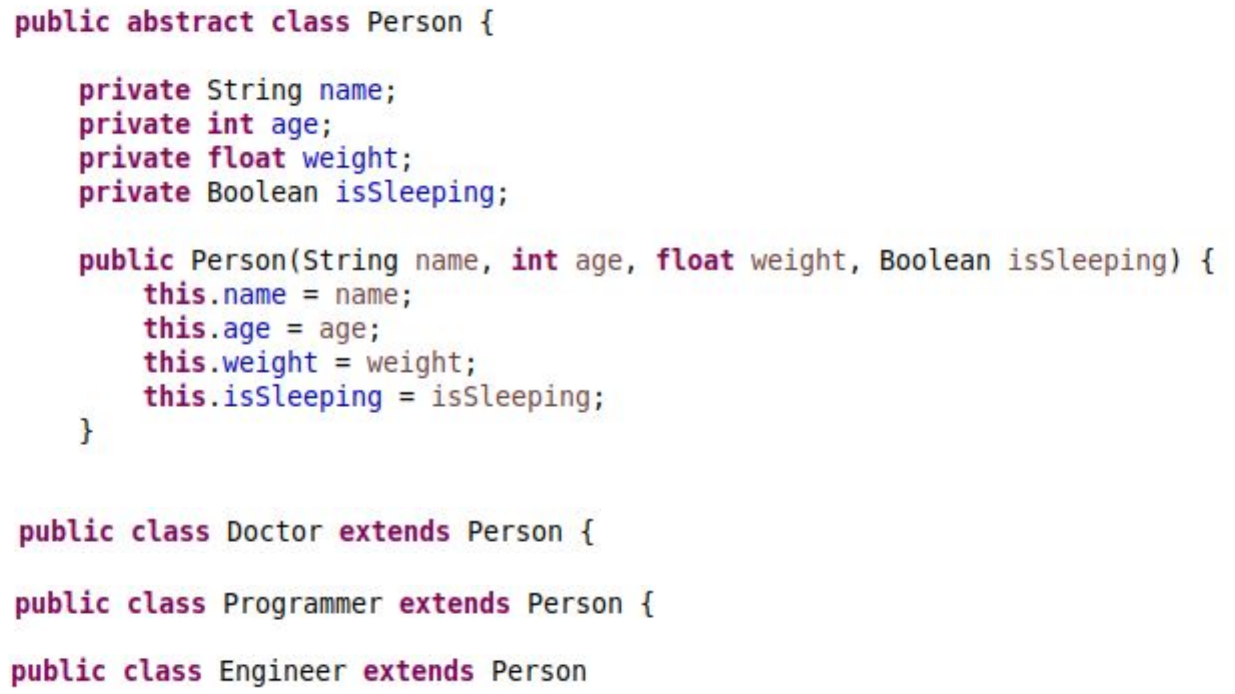
Abstract Classes

- Allows for partial implementation or shared logic
- Can contain fields that are not static and final, abstract & concrete methods
- Commonly used as a “base class”
- Cannot be instantiated; only extended

Abstract Classes - Example

- Person abstract class, Doctor, Engineer, and Programmer classes

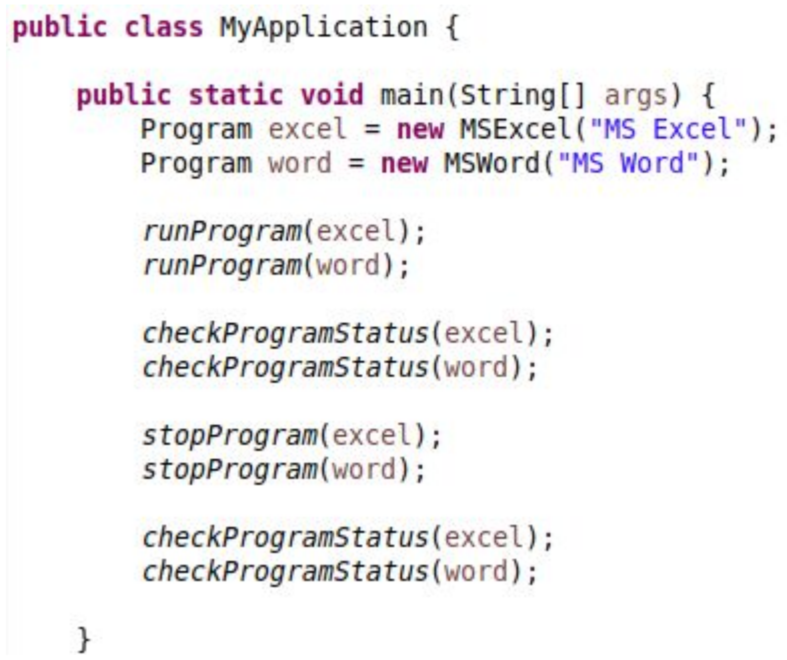
Abstract Classes - Example



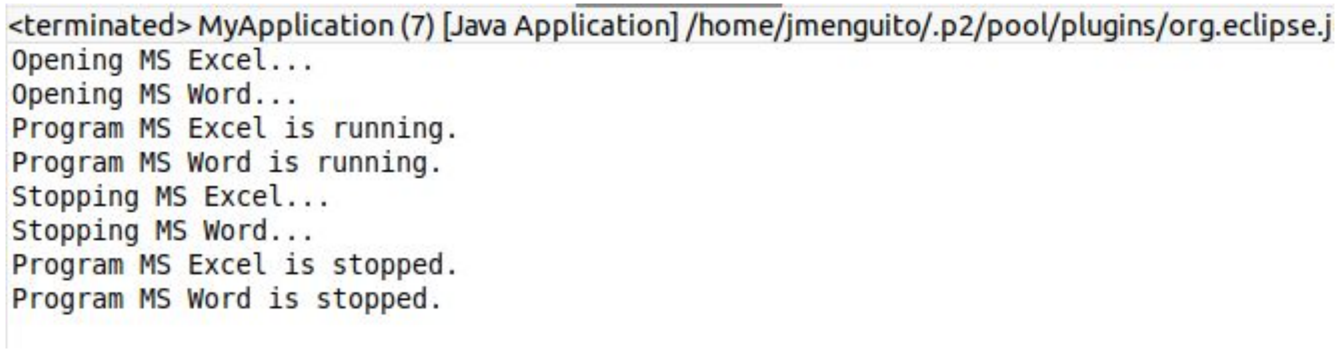
```
public abstract class Person {

    private String name;
    private int age;
    private float weight;
    private Boolean isSleeping;

    public Person(String name, int age, float weight, Boolean isSleeping) {
        this.name = name;
        this.age = age;
        this.weight = weight;
        this.isSleeping = isSleeping;
    }


public class Doctor extends Person {

public class Programmer extends Person {

public class Engineer extends Person
```

Seatwork - Individual (M2_ACTIVITY5)

- Modify the classes: Program.java, MSWord.java, MSExcel.java to make the program runnable.



```java
public class MyApplication {

    public static void main(String[] args) {
        Program excel = new MSExcel("MS Excel");
        Program word = new MSWord("MS Word");

        runProgram(excel);
        runProgram(word);

        checkProgramStatus(excel);
        checkProgramStatus(word);

        stopProgram(excel);
        stopProgram(word);

        checkProgramStatus(excel);
        checkProgramStatus(word);

    }
```



```
<terminated> MyApplication (7) [Java Application] /home/jmenguito/.p2/pool/plugins/org.eclipse.j
Opening MS Excel...
Opening MS Word...
Program MS Excel is running.
Program MS Word is running.
Stopping MS Excel...
Stopping MS Word...
Program MS Excel is stopped.
Program MS Word is stopped.
```

When to use Abstract Classes

- You have similar classes with shared logic
- Your subclasses will have many common fields or methods and will take advantage of access modifiers

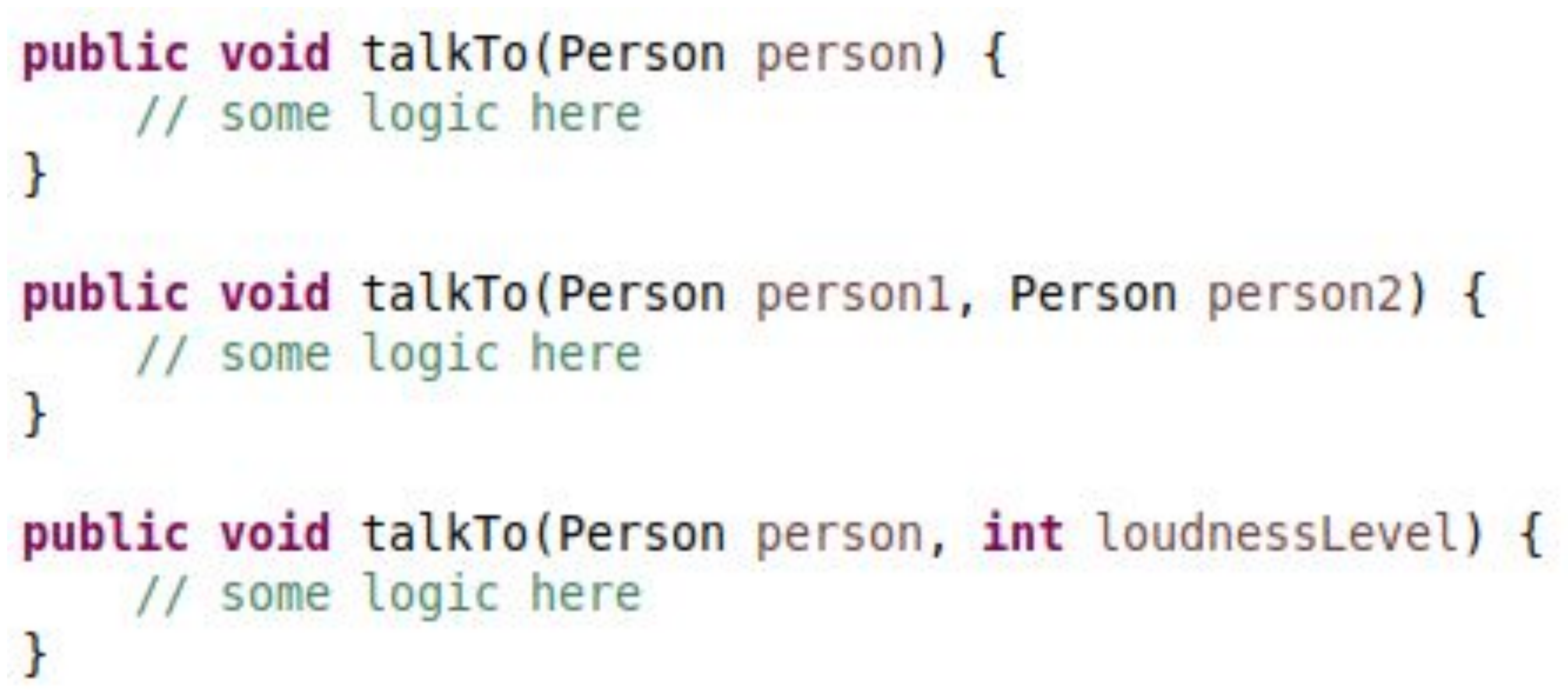
When to use Interfaces

- You have seemingly unrelated classes that share a characteristic or trait
- You want to specify behavior but you are not concerned who implements the behavior
- Multiple inheritance

Method Overloading

- Same method name but different parameters (data type or number of accepted parameters)
- Example: void talkTo(Person person) and void talkTo(Animal animal) and void talkTo(Person person1, Person person2)

Method Overloading - Example

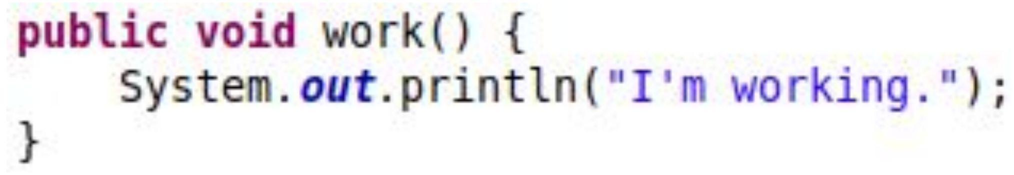


```java
public void talkTo(Person person) {
    // some logic here
}

public void talkTo(Person person1, Person person2) {
    // some logic here
}

public void talkTo(Person person, int loudnessLevel) {
    // some logic here
}
```

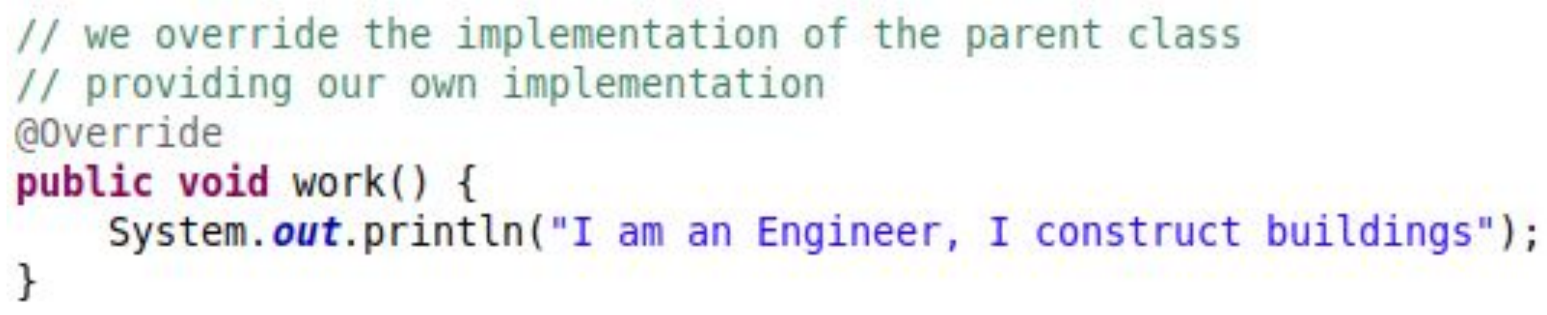
Method Overriding

- Same method signature, but a different implementation from the superclass
- “super” keyword allows a subclass to invoke the superclass’s method.
- Example: Engineer work() can be different from Programmer work();

Method Overriding - Example



```java
public void work() {
    System.out.println("I'm working.");
}
```



```java
// we override the implementation of the parent class
// providing our own implementation
@Override
public void work() {
    System.out.println("I am an Engineer, I construct buildings");
}
```

The Importance of OOP and Related Concepts

- Code Flexibility
- Code Reusability
- Code Maintainability
- Code Readability

Code Flexibility

- An abstract method’s implementation can be changed without affecting the external invocation of the method in a public interface
- Example 1: refer to previous example given Engineer and Programmer work();
- Example 2: demonstrate “programming to an Interface” (swap implementations in Main)

Code Reusability

- Inheritance allows code reuse and shared logic
- Example 1: extending subclasses to use a method that was only written once

Code Maintainability

- Respecting the concept of Encapsulation and Abstraction enforces code maintainability and reduces coupling.
- Coupling is the degree of interdependence between two pieces of code.
- Using public instead of private for attributes increases coupling as one change could potentially require changing other parts of the code as opposed to the use of private attributes and public methods

Code Maintainability - Example

- show an example of a public attribute vs a private attribute. Showcase how changing the data type will add more work

Code Readability

- Effective use of the OOP concepts make code more readable and easier to use with other pieces of code
- Developers reading the code just see what the object does rather than how it does it
- A method (and effective naming convention) communicates intent directly.

Exercise 2 - Individual (M2_ACTIVITY6)

1. Create Interface “Refuelable”. Implementing classes should be able to implement method “refuel()”
2. Create Abstract Class “Vehicle”. Extending classes should have attributes “numberOfWheels” and “brand” and should have an abstract method “startEngine()”, it should have a concrete method “void destroy()”.
3. Create two concrete classes “Car” and “Truck”. It must implement “Refuelable” and extend “Vehicle”.
4. In your Main Application, create one Car and one Truck and call the methods they inherited / implemented.
5. In your Main Application, create a method called “destroyVehicle” that takes ONE parameter (either a Car or Truck) and call their “destroy()” method.