University of Nottingham
UK | CHINA | MALAYSIA

COMP2059
Developing Maintainable Software

LECTURE 02 – OBJECT ORIENTATED AND JAVA REFRESHER

Boon Giin Lee (Bryan)

Basic OO Concepts

- Object-oriented programming is founded on these ideas:
  - **Abstraction**: Simple things like objects represent more complex underlying code and data.
    - A class is a blueprint for a category of objects.
    - An object is an entity that combines data with behavior that acts on that data.
  - **Encapsulation** (information hiding): The ability to protect some components of the object from external access.
    - E.g., keeping fields within a class private, then providing access to them via public methods.
  - **Inheritance**: The ability for a class ("subclass") to extend or override functionality of another class ("superclass").

Basic OO Concepts

- Object-oriented programming is founded on these ideas:

  - **Polymorphism**: The provision of a single interface to entities of different types.
    - Compile time (static) polymorphism through …
      - Method overloading: Create multiple methods with same name with different signatures.
    - Run time polymorphism through …
      - Method overriding: Create method in derived class with same name and signature than in base class.
      - Sub classing: reference of base class can reference, instantiate and destroy objects of derived class.

  - **Interface**: A specification of method signatures (without implementations) as a mechanism for enabling polymorphism in a declarative way.

```java
package com.ae2dms.zooproject.animal;

import com.ae2dms.zooproject.animal.Animal;

public class Reptile extends Animal {
    private int numTeeth;

    public Reptile(String name, int numTeeth) {
        super(name);
        this.setNumTeeth(numTeeth);
    }

    public Reptile(Reptile reptile) {
        super(reptile.getName());
    }

    public int getNumTeeth() {
        return numTeeth;
    }

    public void setNumTeeth(int numTeeth) {
        this.numTeeth = numTeeth;
    }

    @Override
    public void eat() {
        System.out.println(this.getClass().getSim
    }
```

```java
package com.ae2dms.zooproject.animal;

import com.ae2dms.zooproject.misc.Maintainable;

public abstract class Animal implements Maintainable {
    private String name;

    public Animal(String name) {
        this.setName(name);
    }

    public abstract void eat();

    public void enjoy() {
        System.out.println(this.getClass().getSimpleName()
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

```java
package com.ae2dms.zooproject.misc;

public interface Maintainable {
    public void maintain();
}
```

Abstraction
Encapsulation
Inheritance
Polymorphism
Interface

```java
package com.ae2dms.zooproject.animal;

import com.ae2dms.zooproject.animal.Animal;

public class Reptile extends Animal {
    private int numTeeth;

    public Reptile(String name, int numTeeth) {
        super(name);
        this.setNumTeeth(numTeeth);
    }

    public Reptile(Reptile reptile) {
        super(reptile.getName());
    }

    public int getNumTeeth() {
        return numTeeth;
    }

    public void setNumTeeth(int numTeeth) {
        this.numTeeth = numTeeth;
    }

    @Override
    public void eat() {
        System.out.println(this.getClass().getSim}
    }
```

Encapsulation

Inheritance

Abstraction

```java
package com.ae2dms.zooproject.animal;

import com.ae2dms.zooproject.misc.Maintainable;

public abstract class Animal implements Maintainable {
    private String name;

    public Animal(String name) {
        this.setName(name);
    }

    public abstract void eat();

    public void enjoy() {
        System.out.println(this.getClass().getSimpleName()
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

Polymorphism (overload)

Polymorphism (override)

Interface

```java
package com.ae2dms.zooproject.misc;

public interface Maintainable {
    public void maintain();
}
```

What are Coming Up …

- Public vs Private.
- Accessors and Modifiers.
- Encapsulation.
- The “this” keyword.
- Constructors.
- Passing by Value or Reference?
- More Hacks.
- Static fields and methods.

Public vs Private; Accessors & Modifiers

- What are the ***general rules*** for constructors, methods, helper methods, fields, and static constants?
  - Constructors and methods: Usually declared public (they constitute the interface of a class).
  - Helper methods that are needed by only inside the class: Usually declared private.
  - Fields: Usually declared private (to support encapsulation).
  - Static constants: Usually declared public.
  - Accessors (also called Getters): Methods that return values of private fields; Name often starts with `get`.
  - Modifiers (also called Mutators or Setters): Methods that set values of private fields; Name often starts with `set`.

Encapsulation

- **Hiding the implementation details of a class** (making all fields and helper methods private) is called encapsulation.
- Encapsulation helps in program maintenance: a change in one class does not affect other classes.
- A client of a class interacts with the class only through well-documented public constructors and methods; this facilitates team development.

The Keyword "this"

- "this" refers to the implicit parameter inside your class.
  - A variable that stores the object on which a method is called.

- Refer to a field.
  - `this.field;`

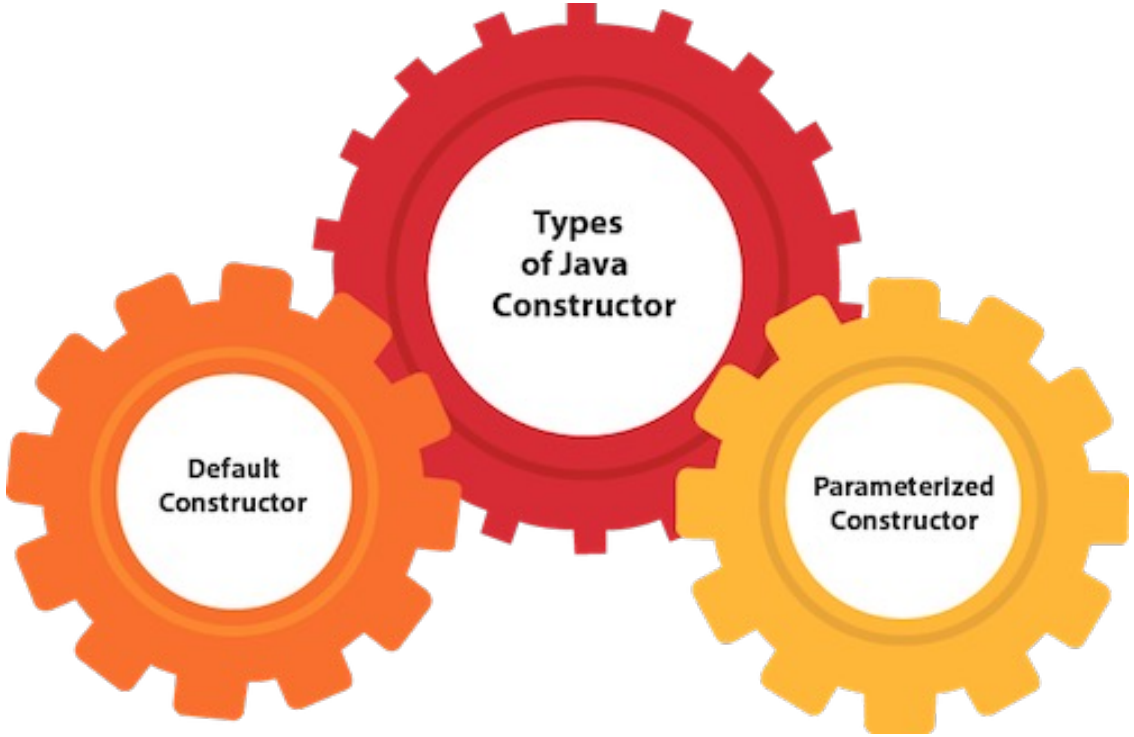
- Call a method.
  - `this.method (parameters);`

- One constructor can call another.
  - `this(parameters);`

```
private int numTeeth;

public Reptile(String name, int numTeeth) {
    super(name);
    this.setNumTeeth(numTeeth);
}

public Reptile(Reptile reptile) {
    this(reptile.getName(), reptile.getNumTeeth());
}

public void setNumTeeth(int numTeeth) {
    this.numTeeth = numTeeth;
}
```

Constructors

- What are constructors used for?



Constructors

- A constructor is a procedure for creating objects of the class.
  - A constructor often initialises an object's fields.
  - Constructors do not have a return type.
  - All constructors in a class have the same name (the name of the class).
  - Constructors may take parameters.
  - If a class has more than one constructor, they must have different numbers and/or types of parameters (constructor overloading).

- Important!
  - Java **provides** a default constructor for a specific class.
  - If you define a constructor for a class, Java **does not provide** the default constructor anymore.

Constructors

```
public Reptile(String name, int numTeeth) {
    super(name);
    this.setNumTeeth(numTeeth);
}

public Reptile(Reptile reptile) {
    this(reptile.getName(), reptile.getNumTeeth());
}
```

```
public Reptile() {

}
```

Default constructor is replaced if new constructor is defined

```
public static void main(String[] args) {
    Reptile r1 = new Reptile( name: "Turtle",  numTeeth: 24);
    Reptile r2 = new Reptile();
}
```

Cannot resolve constructor 'Reptile()'

Create constructor Alt+Shift+Enter More actions... Alt+Enter

Constructors

- Constructors of a class can call each other using the keyword "`this`" (referred to as constructor chaining) – a good way to avoid duplicating code.

- Constructors are invoked using the operator `new`.
  - Declare a reference variable of the required type and then invoke the constructor method after the "`new`" keyword.

- Parameters passed to "`new`" must match the number, types, and order of parameters expected by one of the constructors.

Constructors

- What does the output look like?
  - Refer to class `Zoo`.

```
public class ZooApp {
    public static void main(String[] args) {
        Zoo zoo1;
        zoo1 = new Zoo( location: "Hamburg");
        zoo1 = new Zoo( location: "Munic");
        Zoo zoo2 = zoo1;
        Zoo zoo3 = new Zoo();
        System.out.println("\nZoo1: " + zoo1);
        System.out.println("Zoo2: " + zoo2);
        System.out.println("Zoo3: " + zoo3);

        zoo3.setLocation("Berlin");
        zoo1.setLocation("Berlin");
        System.out.println("\nZoo1: " + zoo1);
        System.out.println("Zoo2: " + zoo2);
        System.out.println("Zoo3: " + zoo3);

        zoo1 = new Zoo( location: "San Diego");
        System.out.println("\nZoo1: " + zoo1);
        System.out.println("Zoo2: " + zoo2);
        System.out.println("Zoo3: " + zoo3);
    }
}
```

```java
public class ZooApp {
    public static void main(String[] args) {
        Zoo zoo1;
        zoo1 = new Zoo( location: "Hamburg");
        zoo1 = new Zoo( location: "Munic");
        Zoo zoo2 = zoo1;
        Zoo zoo3 = new Zoo();
        System.out.println("\nZoo1: " + zoo1);
        System.out.println("Zoo2: " + zoo2);
        System.out.println("Zoo3: " + zoo3);

        zoo3.setLocation("Berlin");
        zoo1.setLocation("Berlin");
        System.out.println("\nZoo1: " + zoo1);
        System.out.println("Zoo2: " + zoo2);
        System.out.println("Zoo3: " + zoo3);

        zoo1 = new Zoo( location: "San Diego");
        System.out.println("\nZoo1: " + zoo1);
        System.out.println("Zoo2: " + zoo2);
        System.out.println("Zoo3: " + zoo3);
    }
}
```

Similar to pointer assignment in C/C++

Output

```
Zoo1: The zoo is in Munic has 1 compounds.
Zoo2: The zoo is in Munic has 1 compounds.
Zoo3: The zoo is in Unknown has 1 compounds.

Zoo1: The zoo is in Berlin has 1 compounds.
Zoo2: The zoo is in Berlin has 1 compounds.
Zoo3: The zoo is in Berlin has 1 compounds.

Zoo1: The zoo is in San Diego has 1 compounds.
Zoo2: The zoo is in Berlin has 1 compounds.
Zoo3: The zoo is in Berlin has 1 compounds.
```

Assign new memory

Static Fields

- A static field (class field or class variable) is shared by all objects of the class.
- A non-static field (instance field or instance variable) belongs to an individual object.
- Static fields are stored with class code, separately from instance variables that describe an individual object.
- Public static fields, usually global constants, are referred to in other classes using dot notation.
  - `ClassName.constName`
- Usually, static fields are **NOT** initialised in constructors; they are initialised either in declarations or in public static methods or just use their default value.
- If a class has only static fields, there is no point in creating objects of that class, as all of them would be identical.
  - `Math` and `System` are examples of the above (they have no public constructors and cannot be instantiated)

Static Methods

- Static methods can access and manipulate class's static fields. They belong to the class – not an instance of it.
- Static methods cannot access instance fields or call instance methods of the class; instance methods can access all fields and call all methods of their class – both static and non-static.
- Static methods will usually take input from the parameters, perform actions on it, then return some result.
- Static methods are called using dot notation.
  - `ClassName.statMethod(…)`

Static Fields and Methods

```
public class ZooApp {
    public static void main(String[] args) {
        int avgVisitors = 100;

        Zoo zoo1;
        zoo1 = new Zoo( location: "Hamburg");
        zoo1 = new Zoo( location: "Munic");
        Zoo zoo2 = zoo1;
        Zoo zoo3 = new Zoo();
        System.out.println("\nZoo1: " + zoo1);
        System.out.println("Zoo2: " + zoo2);
        System.out.println("Zoo3: " + zoo3);

        zoo3.setLocation("Berlin");
        zoo1.setLocation("Berlin");
        System.out.println("\nZoo1: " + zoo1);
        System.out.println("Zoo2: " + zoo2);
        System.out.println("Zoo3: " + zoo3);

        zoo1 = new Zoo( location: "San Diego");
        System.out.println("\nZoo1: " + zoo1);
        System.out.println("Zoo2: " + zoo2);
        System.out.println("Zoo3: " + zoo3);

        zoo1.changeZoo(zoo1, avgVisitors);
        System.out.println("\n" + zoo1 + " (" + avgVisitors + " visitors)");
        System.out.println("\nThe total zoos created: " + Zoo.getNumZoosCreated());

        test();
    }

    public void test() {
        System.out.println("This is a Test!");
    }
}
```

- Does this compile?
  - Refer to class `Zoo`.

Static Fields and Methods

```
    zoo1.changeZoo(zoo1, avgVisitors);
    System.out.println("\n" + zoo1 + " (" + avgVisitors + " visitors)");
    System.out.println("\nThe total zoos created: " + Zoo.getNumZoosCreated());

    test();
}

public static void test() {
    System.out.println("This is a Test!");
}
```

```
private String location;
private ArrayList<Compound> compounds;
private int avgVisitors;
private static int count = 0;

public Zoo(String location, int numCompounds) {
    this.setLocation(location);
    this.compounds = new ArrayList<Compound>();
    this.count++;
    createCompound(numCompounds);
}
```

```
The zoo is in San Diego has 1 compounds. (100 visitors)

The total zoos created: 4
This is a Test!
```

Output

```
public static int getNumZoosCreated() {
    return count;
}
```

```
zoo1 = new Zoo( location: "Hamburg");
zoo1 = new Zoo( location: "Munic");
```

```
Zoo zoo3 = new Zoo();
zoo1 = new Zoo( location: "San Diego");
```

Java Collections Framework

- What do we understand by "Collections" in Java?
  - A collection is an object that represents a group of objects.
  - The collections API is a unified framework for representing and manipulating collections, independent of their implementation.

- What does the abbreviation API stand for?
  - Application Programming Interface.

- What is the difference between a library and an API?
  - An API is an interface or communication protocol between a client (e.g., a program you are developing) and a server (e.g., a library you are using), intended to simplify the building of client-side software.
  - A library contains re-usable chunks of code.

Java Collections Framework

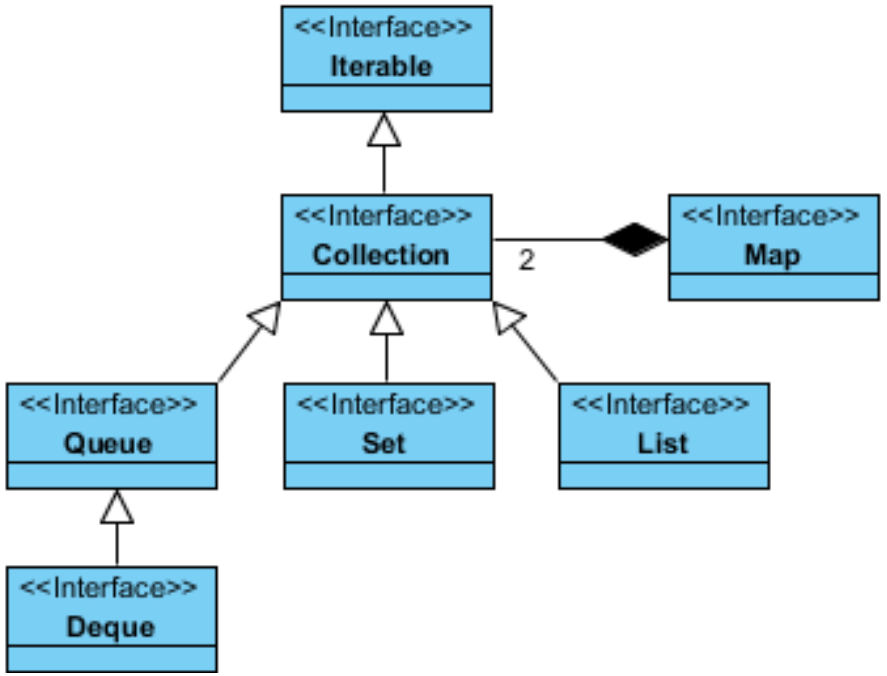
- Java Collections Framework principal ideas:
  - We have container objects that contain objects.
  - All containers are either “collections” or “maps”.
  - All containers provide a common set of method signatures, in addition to their unique set of method signatures.

- The framework contains data structures.
  - E.g., arrays; lists; maps.

- The framework contains algorithmic operations.
  - E.g., searching; sorting.

Java Collections Framework

- Collection
  - Something that holds a dynamic collection of objects.
- Map
  - Defines mapping between keys and objects (two collections).
- Iterable
  - Collections are able to return an iterator object that can scan over the contents of a collection of one object at a time.



Java Collections Framework

- Core collection framework interfaces
  - Iterable – represents an iterator object.
  - Collection – represents a group of objects (elements).
  - Map – maps keys to values; no duplicate keys.
  - Queue – represents FIFO queues or LIFO stacks.
  - Deque – represents a double ended queue.
  - Set – a collection that cannot contain duplicate elements.
  - List – an ordered sequence of elements that allows duplicate elements.

- Interface location
  - Most interfaces can be found in the `java.util.*` package.
  - The "`iterable`" interface resides in the `java.lang.*` package.

Java Collections Framework

- Classes that implement the collection interfaces typically have names in the form of <implementation style><interface>. https://www.infoworld.com/article/2075249/secure-type-safe-collections.html

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

- Legacy classes (**do not use**).
  - `Vector` (now `ArrayList`);
  - `HashTable` (now `HashMap`);
  - `Stack` (now `ArrayDeque`);

Reference 1: https://nikhilmopidevi.github.io/2017/06/19/Overview-of-the-Java-Collections-Framework/
Reference 2: https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html
Reference 3: https://www.studytonight.com/java/legacy-classes-and-interface.php
Reference 4: https://www.techiedelight.com/why-vector-class-java-obsolete/

Module java.base

Package java.util

Class LinkedList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.AbstractSequentialList<E>
        java.util.LinkedList<E>

Type Parameters:

E - the type of elements held in this collection

All Implemented Interfaces:

`Serializable`, `Cloneable`, `Iterable<E>`, `Collection<E>`, `Deque<E>`, `List<E>`, `Queue<E>`

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

Doubly-linked list implementation of the `List` and `Deque` interfaces. Implements all optional list operations, and permits all elements (including `null`).

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

Note that this implementation is not synchronized. If multiple threads access a linked list concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using the `Collections.synchronizedList` method. This is best done at creation time, to prevent accidental unsynchronized access to the list:

```
List list = Collections.synchronizedList(new LinkedList(...));
```

The iterators returned by this class's `iterator` and `listIterator` methods are *fail-fast*: if the list is structurally modified at any time after the iterator is created, in any way except through the Iterator's own `remove` or `add` methods, the iterator will throw a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than

Constructor Summary

Constructors

Constructor	Description
`LinkedList()`	Constructs an empty list.
`LinkedList(Collection<? extends E> c)`	Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

"? Extend E" means some type that either is E or a subtype of E

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
void	`add(int index, E element)`	Inserts the specified element at the specified position in this list.
boolean	`add(E e)`	Appends the specified element to the end of this list.
boolean	`addAll(int index, Collection<? extends E> c)`	Inserts all of the elements in the specified collection into this list, starting at the specified position.
boolean	`addAll(Collection<? extends E> c)`	Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
void	`addFirst(E e)`	Inserts the specified element at the beginning of this list.
void	`addLast(E e)`	Appends the specified element to the end of this list.

Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
void	**add**(int index, **E** element)	Inserts the specified element at the specified position in this list.
boolean	**add**(**E** e)	Appends the specified element to the end of this list.
boolean	**addAll**(int index, **Collection**<? extends **E**> c)	Inserts all of the elements in the specified collection into this list, starting at the specified position.
boolean	**addAll**(**Collection**<? extends **E**> c)	Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
void	**addFirst**(**E** e)	Inserts the specified element at the beginning of this list.
void	**addLast**(**E** e)	Appends the specified element to the end of this list.
void	**clear**()	Removes all of the elements from this list.
Object	**clone**()	Returns a shallow copy of this `LinkedList`.
boolean	**contains**(**Object** o)	Returns `true` if this list contains the specified element.
Iterator<**E**>	**descendingIterator**()	Returns an iterator over the elements in this deque in reverse sequential order.
E	**element**()	Retrieves, but does not remove, the head (first element) of this list.
E	**get**(int index)	Returns the element at the specified position in this list.
E	**getFirst**()	Returns the first element in this list.

Java Collections Framework

- Non typesafe collections (**do not use**).
  - Collection constructors are not able to specify the type of objects the collection is intended to contain.
  - Need to cast objects when using them; a "`ClassCastException`" will be thrown if attempt to cast to the wrong type.

```
public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.add("object 1");
    String s = (String)list.getFirst();
    System.out.println(s);
}
```

Output

```
object 1
```

```
public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.add("object 1");
    int s = (int)list.getFirst();
    System.out.println(s);
}
```

Output

```
Exception in thread "main" java.lang.ClassCastException
```

Java Collections Framework

- Typesafe collections with “Generics”.

```
public static void main(String[] args) {
    LinkedList<Integer> list = new LinkedList();
    list.add(2059);
    int s = (int)list.getFirst();
    System.out.println(s);
}
```

Error

```
public static void main(String[] args) {
    LinkedList<int> list = new LinkedList();
    list.add(2059);
    int s = (int)list.getFirst();
    System.out.println(s);
}
```

- You cannot type a collection using primitive type (e.g., `int`).
  - Values of primitive types need to be put into objects of a suitable wrapper class (e.g., `Integer`) before they can be added to a collection.

ArrayList Class

```java
public static void main(String[] args) {
    ArrayList<String> monthNames = new ArrayList<String>( initialCapacity: 12);
    monthNames.add("Jan");
    monthNames.addAll(Arrays.asList("Feb", "Mar", "Apr"));

    Iterator<String> iter = monthNames.iterator();
    while(iter.hasNext()) {
        String month = iter.next();
        System.out.println(month);
    }

    Collections.shuffle(monthNames);
    System.out.println(monthNames.toString());

    ArrayList<Integer> monthDays = new ArrayList<>( initialCapacity: 12);
    monthDays.add(Integer.valueOf(31));
    monthDays.add(28);

    Object o = monthDays.get(1);
    System.out.println(o instanceof Integer);

    int febNum = monthDays.get(1);
    System.out.println(febNum);
}
```

In Java, iterator is an interface which is used to iterate over a collection of Java object components one by one.

Output

```
Jan
Feb
Mar
Apr
[Jan, Feb, Apr, Mar]
true
28
```

TreeSet Class

- `Treeset` provides an implementation of the `Set` interface that uses a tree for storage.
- Objects are stored in sorted, ascending order.

```
public static void main(String[] args) {
    ArrayList<String> list = new ArrayList<String>();
    list.addAll(Arrays.asList("One", "One", "Two", "Three"));
    System.out.println("List: " + list.toString());

    TreeSet<String> set = new TreeSet<>(list);
    System.out.println("List: " + set.toString());
}
```

Output

```
List: [One, One, Two, Three]
List: [One, Three, Two]
```

HashMap Class

- `HashMap` is a Hash table-based implementation of the `Map` interface.
- This implementation provides all the optional map operations and permits null values and the null key.

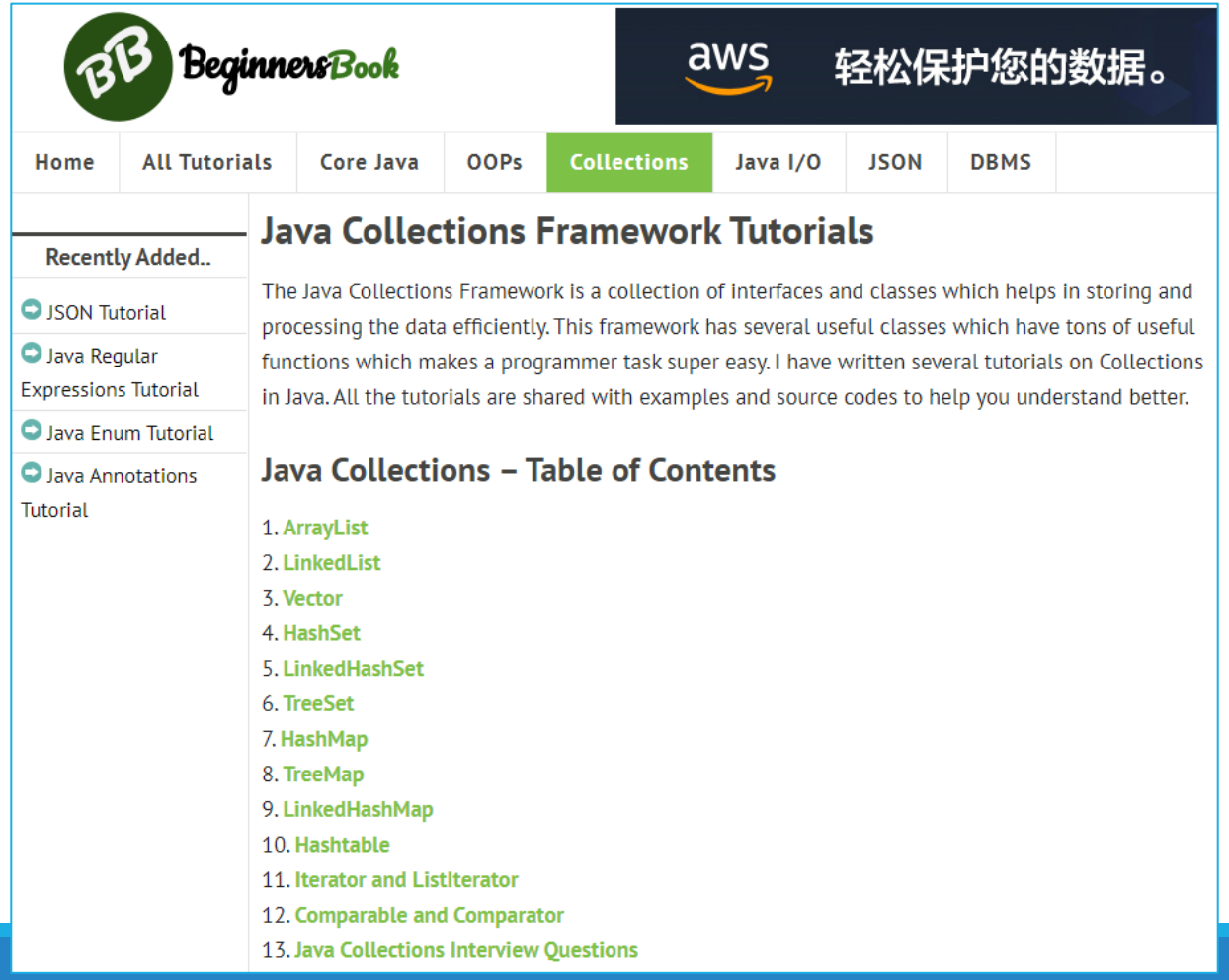
```
public static void main(String[] args) {
    HashMap<String, Integer> userData = new HashMap<>();
    userData.put("Emma", 30);
    userData.put("Paul", 30);
    userData.put("Bernd", 25);
    userData.put("Bernd", 25);
    userData.put("Sophia", null);
    userData.put("Bernd", 28);
    System.out.println(userData);

    Set<String> keys = userData.keySet();
    for(String key:keys) {
        System.out.println(key + " = " + userData.get(key));
    }
}
```

Output

```
{Bernd=28, Emma=30, Paul=30, Sophia=null}
Bernd = 28
Emma = 30
Paul = 30
Sophia = null
```

https://beginnersbook.com/java-collections-tutorials/

Java Collections Examples



University of Nottingham
UK | CHINA | MALAYSIA

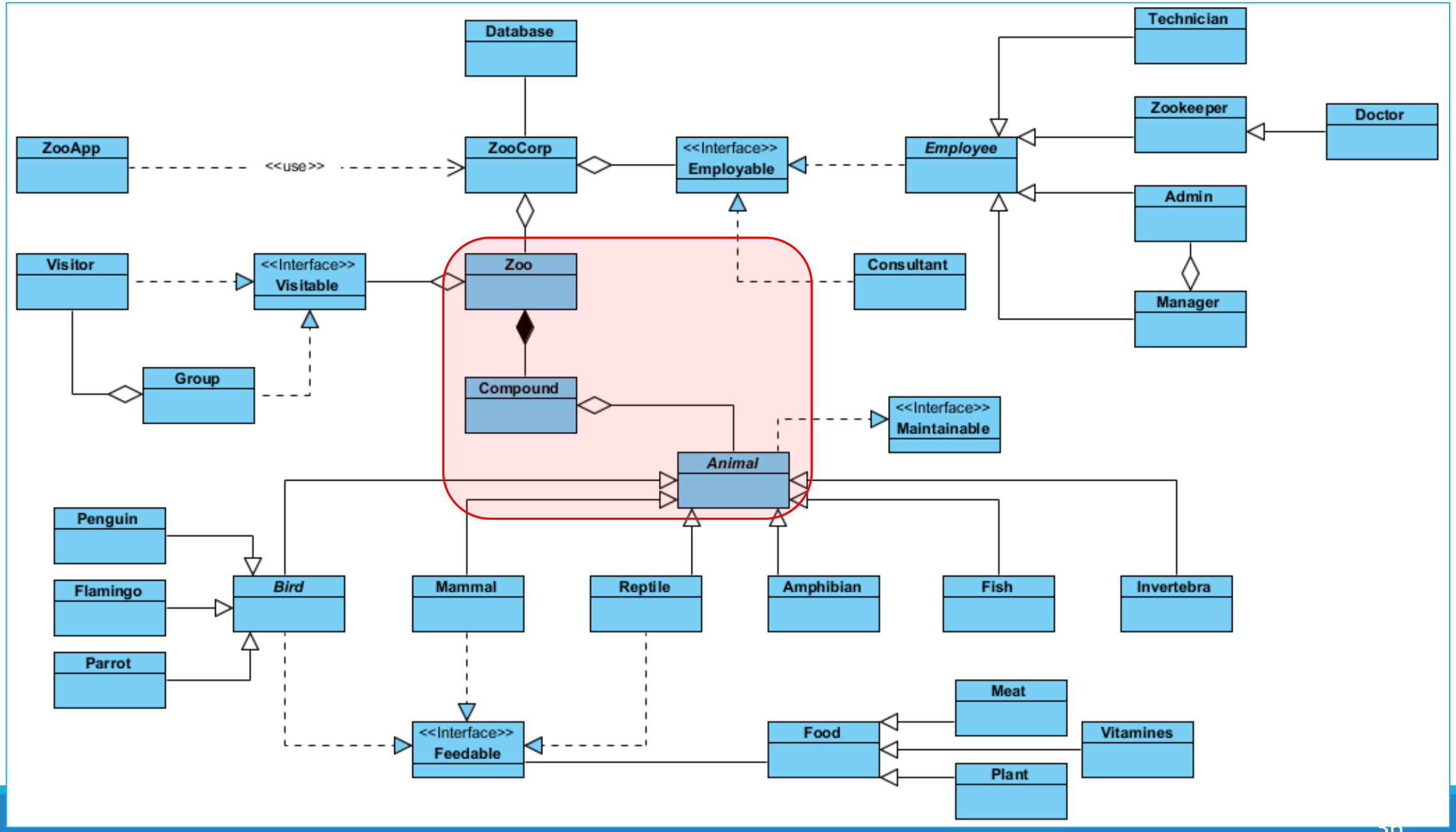
Implementation of Object-Oriented Principles

AGGREGATION AND COMPOSITION; INHERITANCE; POLYMORPHISM; ABSTRACT METHODS AND CLASSES; INTERFACES

Case Study: Zoo Management

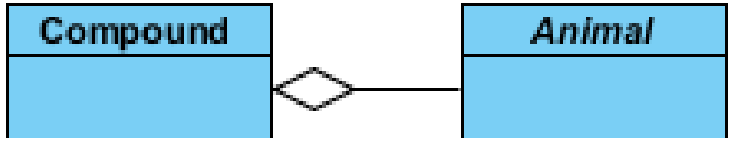
JAVA BASICS

OOP: How to Translate Diagrams into Code That Can be Explained Using Everyday Language!

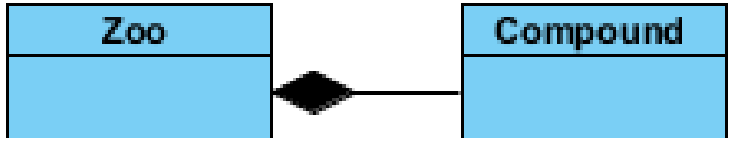


Aggregation and Composition

- What is the difference between the Aggregations and Compositions?



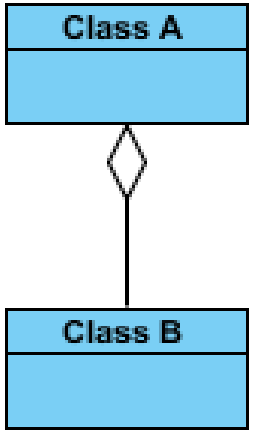
- Aggregation
  - The object exists outside the other, is created outside, so it is passed as an argument (for instance) to the constructor.



- Composition
  - The object only exists, or only makes sense inside the other, as part of the other.

Aggregation in Java

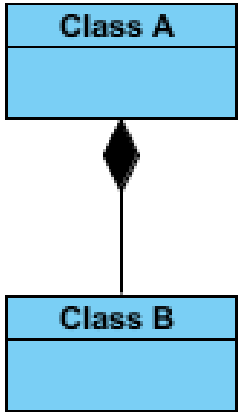
- Class B is part of class A (semantically), but class B can be shared and if class A is deleted, class B is not deleted.
  - Class A stores the reference to class B for later use.
    - Often setter injection is used.




```
public class A {
    private B b;

    public void setB(B b) {
        this.b = b;
    }
}
```

Composition in Java

- Class A owns class B and class B cannot be shared and if class A is deleted, class B is also deleted.
  - The containing object is responsible for the creation and life cycle of the contained object (either directly or indirectly).
    - Composition via member initialisation.
    - Composition via constructor initialisation.

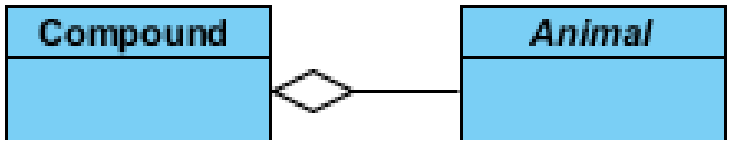



```java
public class A {
    private B b = new B();
}
```

```java
public class A {
    private B b;

    public A() {
        this.b = new B();
    }
}
```

Aggregation

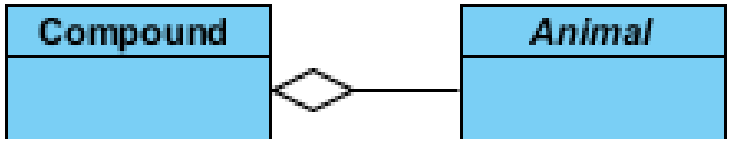
- Provide some implementation code.



Aggregation

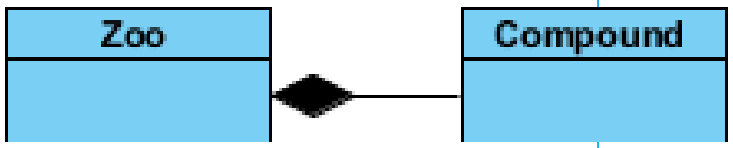
```java
public class Compound {
    private final ArrayList<Animal> animals;

    public Compound() {
        animals = new ArrayList<>();
    }

//    public void addAnimal(Animal animal) {
//        animals.add(new Animal());
//    }

    public void addAnimal(Animal animal) {
        animals.add(animal);
    }

    public void printInfo() {
        System.out.println("The compound has " + animals.size() + " animals.");
    }
}
```

```java
public abstract class Animal {

}
```



Composition

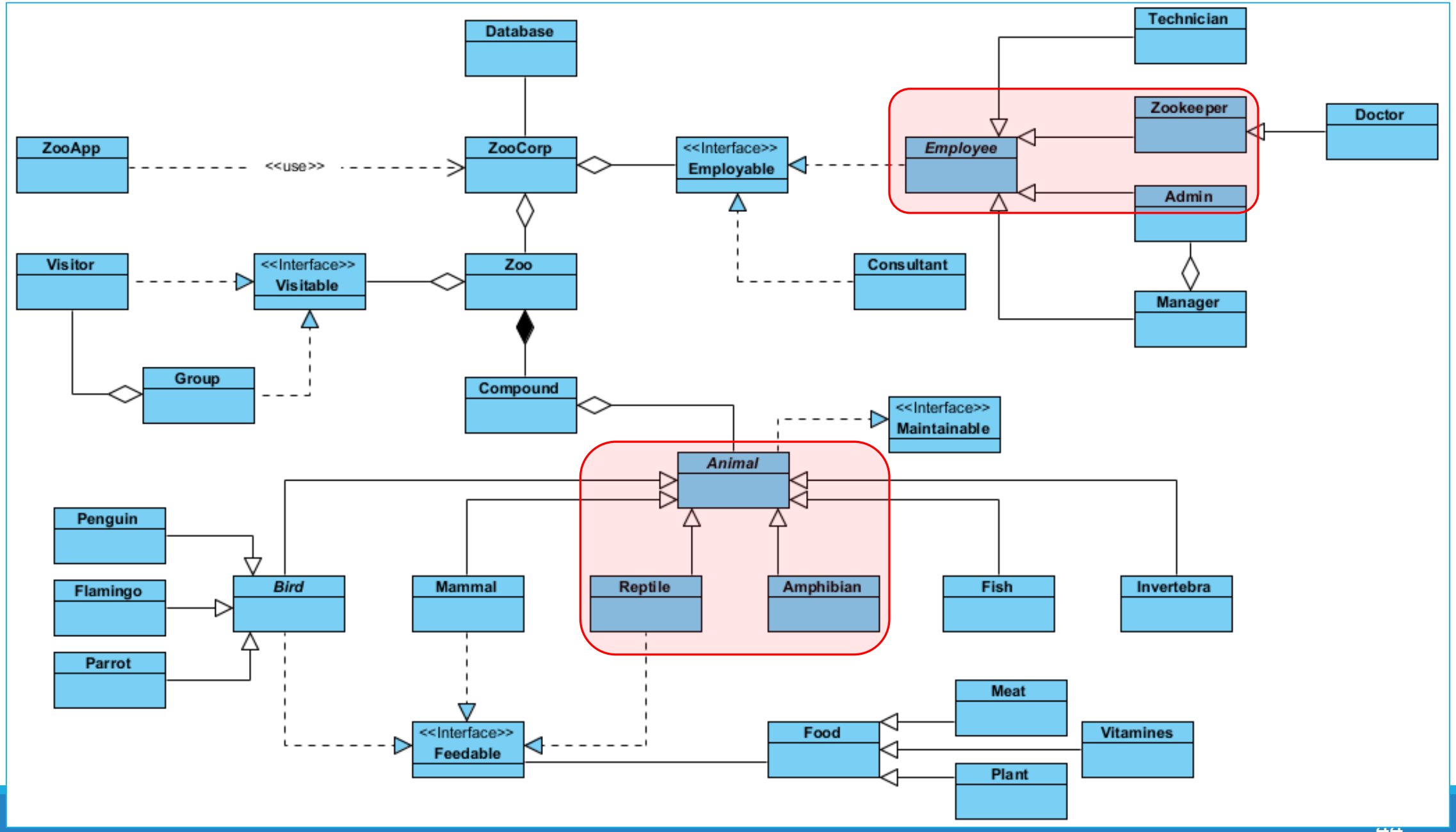
```java
public class Zoo {
    private String location;
    private ArrayList<Compound> compounds;

    public Zoo(String location, int numCompounds) {
        this.setLocation(location);
        this.compounds = new ArrayList<Compound>();
        createCompound(numCompounds);
    }

    public Zoo() {
        this( location: "Unknown",  numCompounds: 1);
    }

    public void createCompound(int numCompounds) {
        if (numCompounds < 1) numCompounds = 1;
        for (int i = 0; i < numCompounds; i++) {
            this.compounds.add(new Compound());
        }
    }

    public String getLocation() {
        return location;
    }

    public void setLocation(String location) {
        this.location = location;
    }

    public void printInfo() {
        System.out.println("The zoo in " + location + " has " + compounds.size() + " compounds.");
    }
}
```



Inheritance

- What is inheritance and why do we use it?

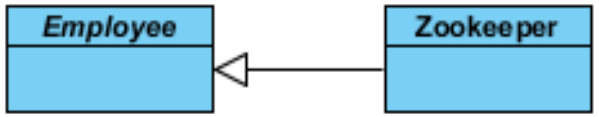
  - Inheritance: Forming new classes based on existing ones.
    - A way to share/reuse code between two or more classes.
  - Superclass: Parent class being extended.
  - Subclass: Child class that inherits behavior from superclass.
    - Gets a copy of every field and method from superclass.
  - "is-a" relationship: Each object of the subclass also "is a(n)" object of the superclass and can be treated as one.


Database
Technician
ZooApp
<<use>>
ZooCorp
<<Interface>>
Employable
Employee
Zookeeper
Doctor
Admin
Visitor
<<Interface>>
Visitable
Zoo
Consultant
Manager
Group
Compound
<<Interface>>
Maintainable
Animal
Penguin
Flamingo
Parrot
Bird
Mammal
Reptile
Amphibian
Fish
Invertebra
<<Interface>>
Feedable
Food
Meat
Vitamines
Plant

Inheritance

- Example:

```
public class Zookeeper extends Employee {
    public Zookeeper(String name) { }

    @Override
    public double getSalary() { }

    @Override
    public void promotion() { }
}
```



- By extending `Employee`, each `Zookeeper` object now:
  - Receives a copy of each method from Employee automatically.
  - Can be treated as an Employee by client code.
  - `Zookeeper` can replace ("override") behavior from Employee.

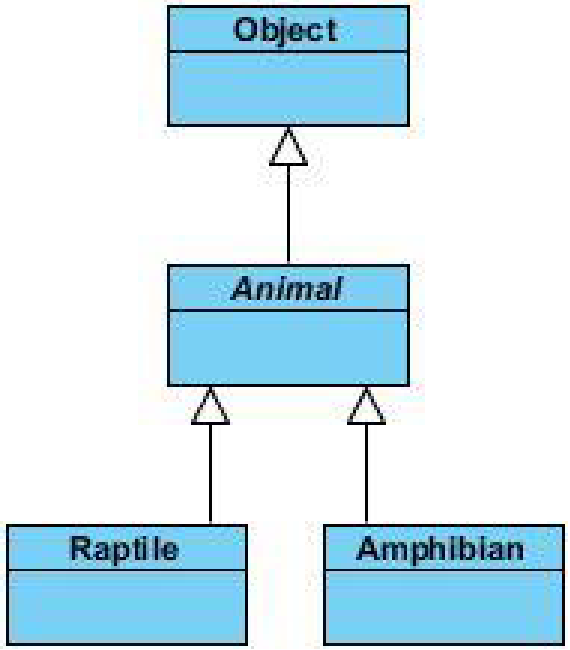
```java
public abstract class Employee {
    private String name;
    private double salary;

    public Employee(String name) {
        setName(name);
        setSalary(2000);
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getSalary() {
        return salary;
    }

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

    public abstract void promotion();
}
```

A subclass can call its parent's method(s)/constructors(s)

```java
public class Zookeeper extends Employee {

    public Zookeeper(String name) {
        super(name);
    }

    @Override
    public double getSalary() {
        double baseSalary = super.getSalary();
        return baseSalary + 1000.00;
    }

    @Override
    public void promotion() {
        super.setSalary(super.getSalary() * 1.1);
    }
}
```

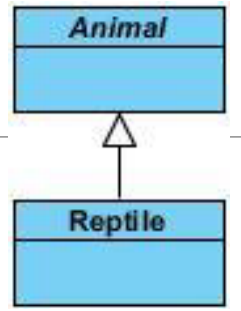
Inheritance

- Every class is either
  - A direct subclass of `Object` (no extends).
  - A subclass of a descendent of `Object` (extends).



- Class `Reptile` extends `Animal`.
- Class `Amphibia` extends `Animal`.
- Class `Animal` extends `Object`.

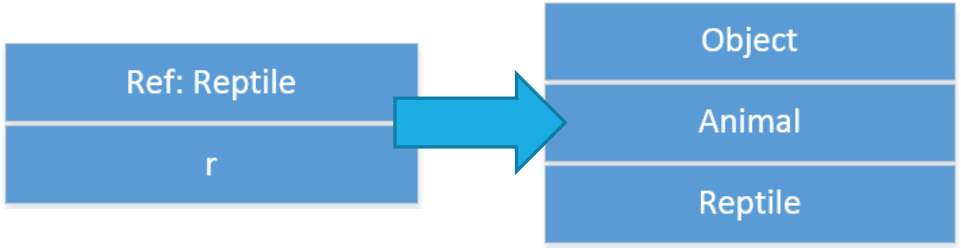
```java
public abstract class Animal {
    private String name;

    public Animal(String name) {
        this.setName(name);
    }

    public abstract void eat();

    public void enjoy() {
        System.out.println(this.getClass().getSimpleName() + " enjoys life as animal.");
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```



```java
public class Reptile extends Animal {
    private int numTeeth;

    public Reptile(String name, int numTeeth) {
        super(name);
        this.setNumTeeth(numTeeth);
    }

    public int getNumTeeth() {
        return numTeeth;
    }

    public void setNumTeeth(int numTeeth) {
        this.numTeeth = numTeeth;
    }

    @Override
    public void eat() {
        System.out.println(this.getClass().getSimpleName() + " eats like a reptile.");
    }
}
```

Inheritance

- Object creation process: `Reptile r = new Reptile();`
  1. Create reference "`r`".
  2. Start creating `Reptile` by entering `Reptile` constructor and making call to parent.
  3. Start creating `Animal` by entering `Animal` constructor and making call to parent.
  4. Create `Object` portion.
  5. Create `Animal` portion.
  6. Create `Reptile` portion.



Inheritance

- Casting object references.

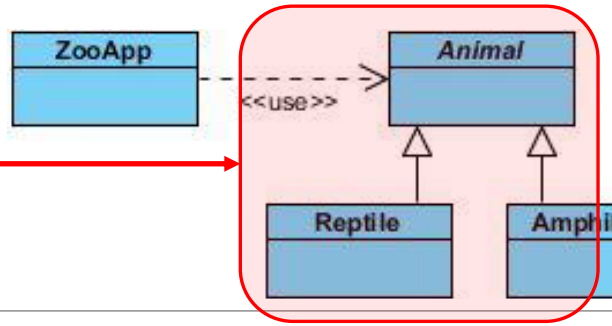
```
public class ZooApp {
    public static void main(String[] args) {
        Object o;
        Reptile r;

        o = r;
        r = o;
    }

    public s
}
```

Required type: Reptile
Provided: Object
Cast to 'com.ae2dms.zooproject.animal.Reptile' Alt+Shift+Enter More actions... Alt+Enter
Object o

Polymorphism

- What is the difference between polymorphism, method overloading, and method overriding?
  - Polymorphism
    - Polymorphism is an object-oriented concept.
    - Method overloading and method overriding are two forms of polymorphism.
  - Method overloading
    - Methods with the same name co-exists in the same class but they must have different method signature.
    - Resolved during compile time (static binding).
  - Method overriding
    - Method with the same name is declared in super and subclass.
    - Resolved during runtime (dynamic binding).

Polymorphism: Dynamic Binding



- At run time (dynamic) when a method is invoked on a reference to the ACTUAL object is examined and the "lowest" or closet version of the method is actually run.

```java
public class Amphibian extends Animal {
    public Amphibian(String name) {
        super(name);
    }

    @Override
    public void eat() {
        System.out.println(this.getClass().getSimpleName() + " eats like an amphibian.");
    }

    @Override
    public void enjoy() {
        System.out.println(this.getClass().getSimpleName() + " enjoys life as amphibian.");
    }
}
```

```java
public static void main(String[] args) {
    Animal a1 = new Amphibian( name: "Frog");
    Animal a2 = new Reptile( name: "Snake",  numTeeth: 4);
    Reptile r1 = new Reptile( name: "Turtle",  numTeeth: 24);

    a1.enjoy();
    a2.enjoy();
    r1.enjoy();
}
```

Output

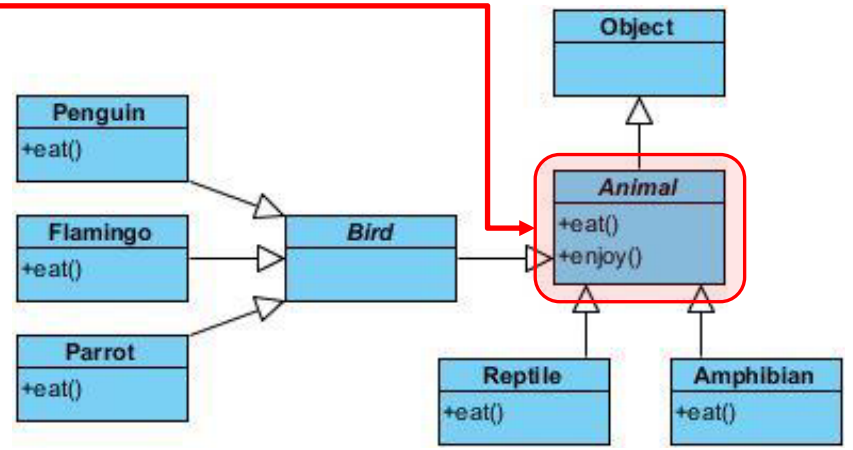
```
Amphibian enjoys life as amphibian.
Reptile enjoys life as animal.
Reptile enjoys life as animal.
```

Abstract Methods and Classes

- Any subclass of class `Animal` has two choices:
  - Define an `eat` method.
  - Be abstract.

- Note:
  - Abstract classes may not be used to instantiate objects.
  - References to abstract classes are legal.

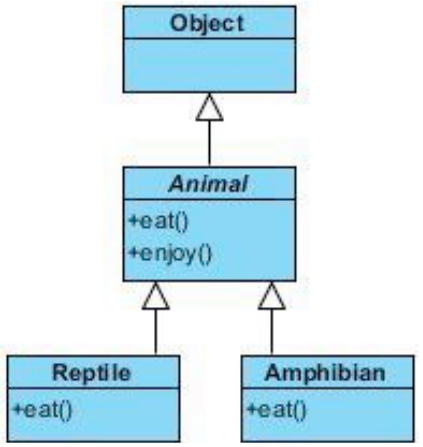
```
public abstract class Animal {
    private String name;

    public Animal(String name) {
        this.setName(name);
    }

    public abstract void eat();

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void enjoy() {
        System.out.println(this.getClass().getSimpleName() + " enjoys life as animal.");
    }
}
```



Abstract Methods and Classes

- Any subclass of class `Animal` has two choices:
  - Define an `eat` method.
  - Be abstract.

- Note:
  - Abstract classes may not be used to instantiate objects.
  - References to abstract classes are legal.



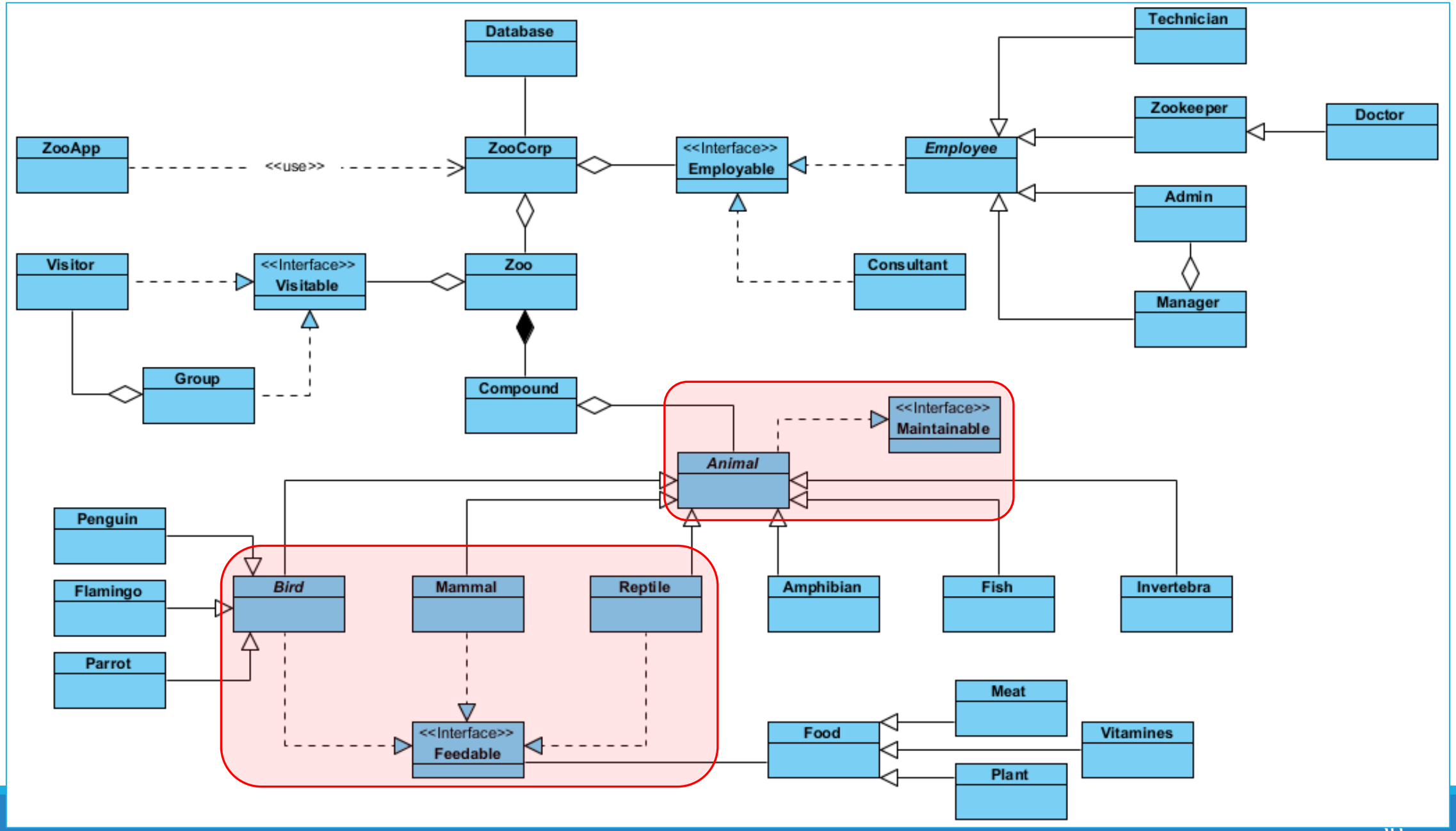

```
public class ZooApp {
    public static void main(String[] args) {
        ArrayList<Object> animals = new ArrayList<>();
        Object o = new Reptile( name: "Snake", numTeeth: 4);
        Reptile r = new Reptile( name: "Turtle", numTeeth: 24);

        animals.add(o);
        animals.add(r);
        animals.add(new Amphibian( name: "Frog"));

        while (animals.size() > 0) {
            o = animals.remove( index: 0);
            System.out.println(o.toString());

            ((Animal) o).eat();
            ((Animal) o).enjoy();
            System.out.println();
        }
    }
}
```

Output

```
com.ae2dms.zooproject.animal.Reptile@378bf509
Reptile eats like a reptile.
Reptile enjoys life as animal.

com.ae2dms.zooproject.animal.Reptile@27d6c5e0
Reptile eats like a reptile.
Reptile enjoys life as animal.

com.ae2dms.zooproject.animal.Amphibian@4f3f5b24
Amphibian eats like an amphibian.
Amphibian enjoys life as amphibian.
```

Interfaces

- What is the difference between an abstract class and an interface?
  - Java abstract class.
    - Can have instance methods that implement a default behavior.
    - May contain non-final variables.
  - Java interfaces.
    - Methods are implicitly abstract and cannot have implementations, HOWEVER, available with "default" for Java SDK 11 and above.
    - Variables declared are by default final.


Database
Technician
Zookeeper
Doctor
ZooApp
<<use>>
ZooCorp
<<Interface>>
Employable
Employee
Admin
Visitor
<<Interface>>
Visitable
Zoo
Consultant
Manager
Group
Compound
<<Interface>>
Maintainable
Animal
Penguin
Flamingo
Parrot
Bird
Mammal
Reptile
Amphibian
Fish
Invertebra
<<Interface>>
Feedable
Food
Meat
Vitamines
Plant

Interfaces

- Some explanations from the Internet.

  - An interface is a **contract**: The guy writing the interface says, “hey, I accept things looking that way”, and the guy using the interface says “OK, the class I write looks that way”.

  - An interface is an **empty shell**, there are only the signatures of the methods, which implies that the methods do not have a body. The interface can’t do anything. It’s just a pattern.

  - Abstract classes look a lot like interfaces, but they have something more: You can define a behavior for them. It’s more about a guy saying, “these classes should look like that, and they have that in common, so fill in the blanks!”.

Reference: https://stackoverflow.com/questions/1913098/what-is-the-difference-between-an-interface-and-abstract-class

Interfaces

- Interfaces are less restrictive when it comes to inheritance.
  - While classes can only ever extend one other class (single inheritance), with interfaces we can choose to **implement as many interfaces** as we like.
  - Implementing an interface means writing implementation code for each of the methods in the interface.
  - Abstract – The parent that offers inheritance in terms of behaviors and patterns.
  - Interface – Ability that can be adopted, e.g., flying capability – Flyable.

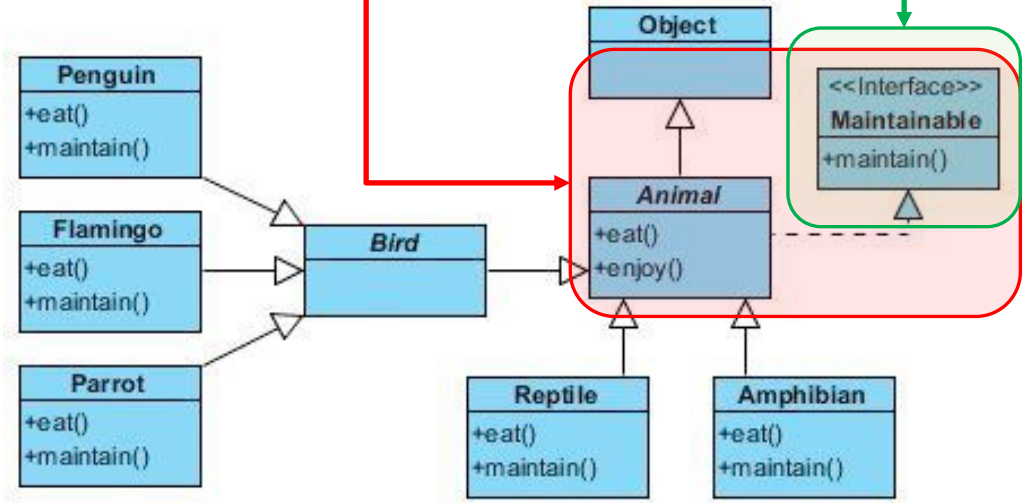
Interfaces

- Some rules:
  - Use the keyword "`interface`" instead of "`class`" to declare an interface.
  - Implement an interface with the "`implement`" keyword. A class that implements an interface must provide implementations for all the methods in the interface.
  - Generally, all interfaces whose name end with "`-able`" mark the interfaces as being able to do something (e.g. `Runnable – run()`, `Executable – execute()`, `Comparable – compareTo()`)
  - Similar to classes, you can build up inheritance hierarchies of interfaces by using the "`extends`" keyword.

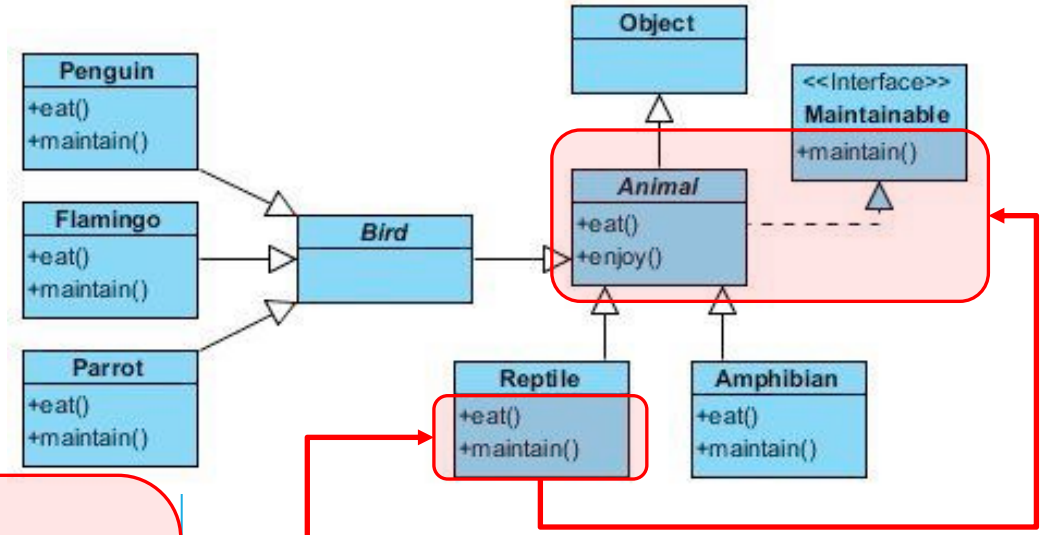
Interfaces

```java
public abstract class Animal implements Maintainable {
    private String name;

    public Animal(String name) {
        this.setName(name);
    }

    public abstract void eat();

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void enjoy() {
        System.out.println(this.getClass().getSimpleName() + " enjoys life as animal.");
    }
}
```

```java
public interface Maintainable {
    public void maintain();
}
```



```java
public class Reptile extends Animal {
    private int numTeeth;

    public Reptile(String name, int numTeeth) {
        super(name);
        this.setNumTeeth(numTeeth);
    }

    public Reptile(Reptile reptile) {
        this(reptile.getName(), reptile.getNumTeeth());
    }

    public int getNumTeeth() {
        return numTeeth;
    }

    public void setNumTeeth(int numTeeth) {
        this.numTeeth = numTeeth;
    }

    @Override
    public void eat() {
        System.out.println(this.getClass().getSimpleName() + " eats like a reptile.");
    }


    @Override
    public void maintain() {
        System.out.println(this.getClass().getSimpleName() + " maintains life as reptile.");
    }
}
```



Reverse Engineering: From Code to Underlying Design

- What can you do?
  - Have a look at the Javadocs (if present).
  - Create UML diagram.
    - Write down all class and interface names.
    - Look for indications in the code that represent relationships.
    - Look for design patterns (if documentation claims they are there).
    - You can use the help of tools (e.g., Visual Paradigm) but don't trust them fully.
      - Be aware that some of these produce "object diagrams" rather than "class diagrams" and some of them mess up aggregations and compositions.
      - Depending on required diagram complexity, you might NOT want to put all fields and methods and dependency relationships into the diagram (if you are asked to produce a high-level diagram).

Employee Management System

A SAMPLE SCENARIO

Scenario

- You are a software engineer working at a mid-sized company that uses an **Employee Management System** to handle payroll for different types of employees. Recently, your manager assigned you to **refactor the existing system** because it has become difficult to maintain and extend. The system currently supports **full-time and part-time employees**, but the codebase is old, lacks proper structure, and is challenging to add new features.
- Your task includes:
  - **Analyzing the current code** and identifying areas that are missing or poorly implemented.
  - **(Basic) Refactoring the code** to apply these OOP principles, making it easier to add more types of employees in the future.
  - **Implementing a new feature**: Support for **interns** as employee type.

Sample Non-Clean Code

```
class Employee {
    public String name;
    public double salary;
    public String employeeType;  // "FullTime", "PartTime", etc.

    public Employee(String name, double salary, String employeeType) {
        this.name = name;
        this.salary = salary;
        this.employeeType = employeeType;
    }

    public void calculatePay() {
        if (employeeType.equals("FullTime")) {
            System.out.println(name + " receives a full-time salary of " + salary);
        } else if (employeeType.equals("PartTime")) {
            System.out.println(name + " receives a part-time salary of " + (salary / 2));
        }
    }

    public void log() {
        System.out.println("Logging employee data for " + name);
    }
}
```

Code smell:

- What can you observed?
- What are the ISSUES?

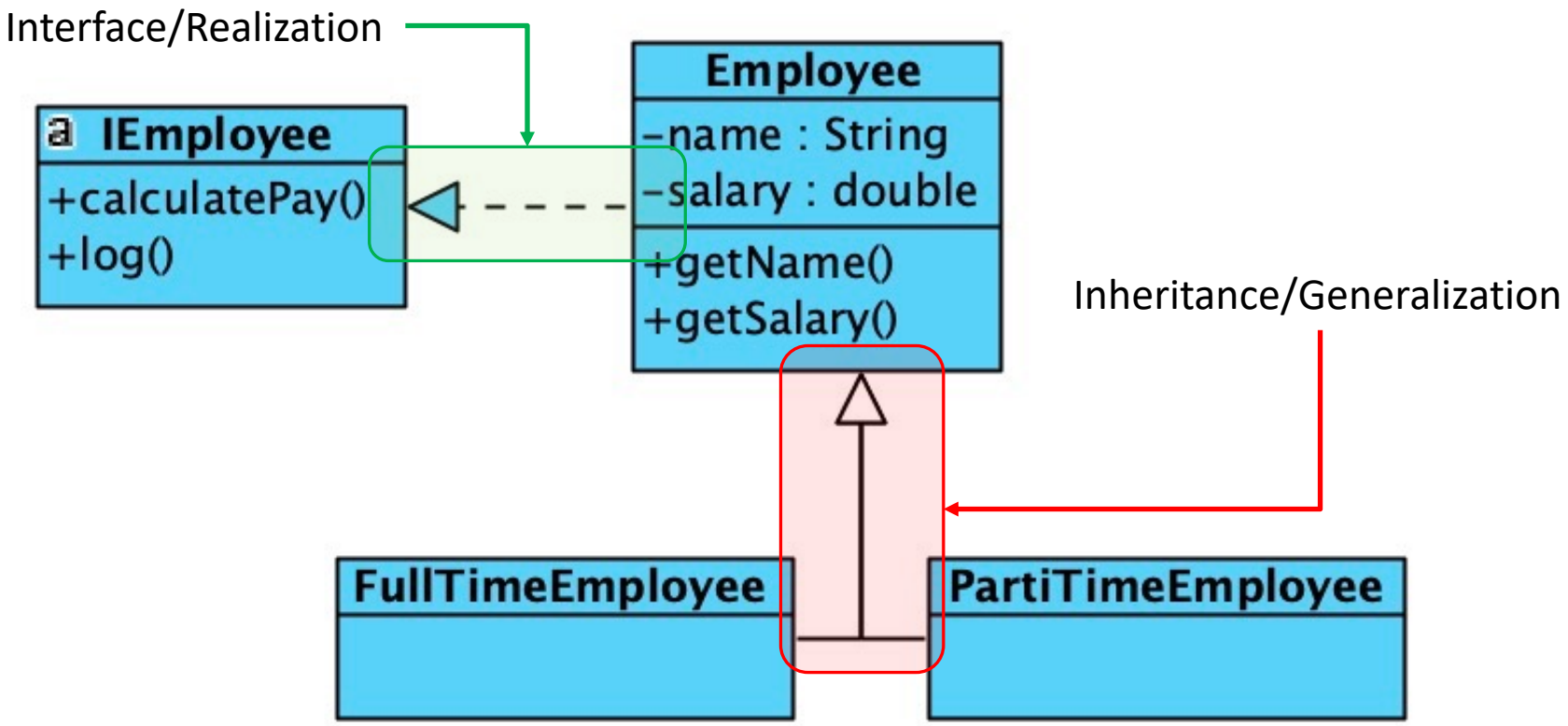
Current Situation

- The current system is built using one large class called `Employee`. This class has multiple responsibilities and does not follow proper OOP principles.
  - It manages both full-time and part-time employees in a single `calculatePay()` method by using **if-else conditions**.
  - The class has public fields for storing employee data such as name, salary, and employee type, violating the principle of **encapsulation**.
  - Adding new employee types (like contractors or interns) requires modifying the existing `Employee` class, which makes the code fragile and prone to errors.

Identifying OOP Issues

- **Abstraction**: The `Employee` class handles different types of employees in a single method, making it hard to extend or maintain.
- **Encapsulation**: Public attributes (name, salary, and employeeType) violate encapsulation, allowing them to be modified directly.
- **Inheritance**: Different employee types (e.g., full-time, part-time) are handled using *conditional statements* rather than subclassing.
- **Polymorphism**: The `calculatePay()` method relies on *if-else blocks* instead of utilizing polymorphism.
- **Interface**: There is no use of interfaces to define *common behavior* for different employee types.

Refactoring with OOP Principles



Fix the Code!

- Refactor the code using _____ by creating an IEmployee interface and applying _____ by making class fields private:

```
interface IEmployee {
    void calculatePay();
    void log();
}
```

Which OOP?

```
abstract class Employee implements IEmployee {
    private String name;
    private double salary;

    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }
}
```

```
public String getName() {
        return name;
    }

    public double getSalary() {
        return salary;
    }

    public abstract void calculatePay();

    public void log() {
        System.out.println("Logging employee data for " + name);
    }
```

Which OOP? **WHY**?

Which OOP?

Fix the Code!

```
class FullTimeEmployee extends Employee {
    public FullTimeEmployee(String name, double salary) {
        super(name, salary);
    }

    @Override
    public void calculatePay() {
        System.out.println(getName() + " receives a full-time salary of " + getSalary());
    }
}


class PartTimeEmployee extends Employee {
    public PartTimeEmployee(String name, double salary) {
        super(name, salary);
    }

    @Override
    public void calculatePay() {
        System.out.println(getName() + " receives a part-time salary of " + (getSalary() / 2));
    }
}
```

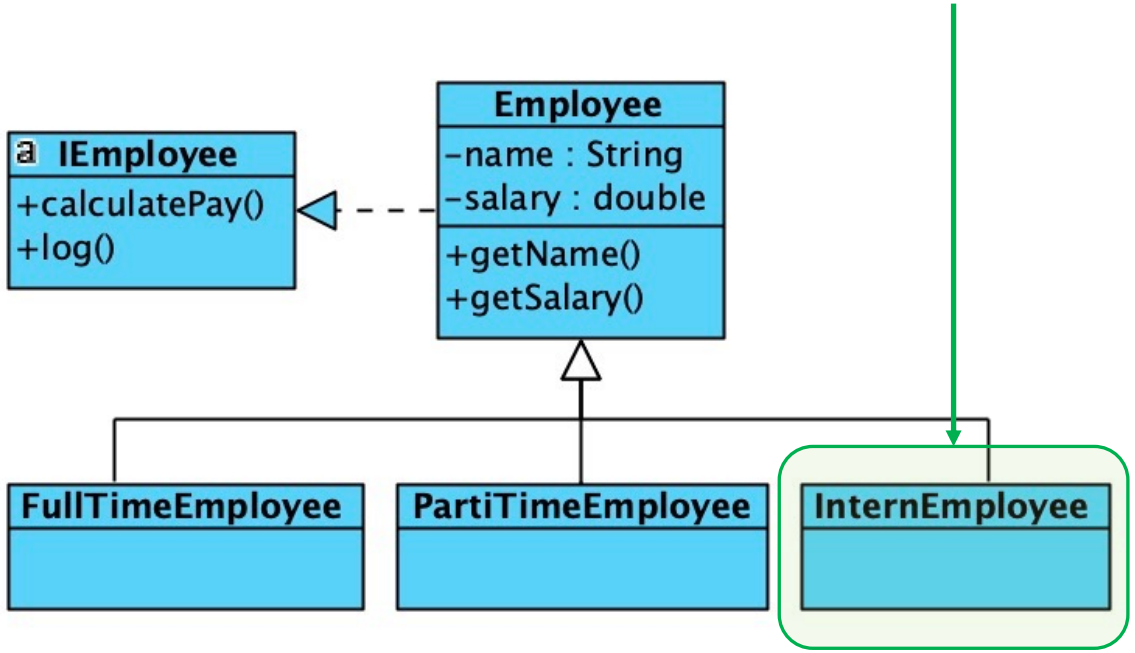
Fix the Code!

- Use _____ to create specific classes for different employee types, each implementing its own `calculatePay()` method.

- Use _____ allows us to call `calculatePay()` without worrying about the specific employee type.

Adds a New Requirement

- Now that the code is refactored with clean OOP principles, it's easier to implement **new requirements**.
- Need to add a new employee type, **`InternEmployee`**, with a fixed stipend.



Adds a New Requirement

```
class InternEmployee extends Employee {
    public InternEmployee(String name, double stipend) {
        super(name, stipend);
    }

    @Override
    public void calculatePay() {
        System.out.println(getName() + " receives a fixed stipend of " + getSalary());
    }
}
```

Refactored Code Explanations

- **Abstraction**: The `Employee` class provides a generalized structure for different types of employees.
- **Encapsulation**: Employee details (name, salary) are encapsulated within the class and only accessible through getter methods.
- **Inheritance**: Specific employee types (FullTime, PartTime, etc.) inherit common behavior from the `Employee` class.
- **Polymorphism**: The `calculatePay()` method is overridden in each subclass, and polymorphism ensures that the correct method is called at runtime. Removing the need for if-else conditions.
- **Interface**: The `IEmployee` interface ensures a standard set of behaviors for all employee types.

Key Points

- Ensure that the system is **easily extendable**, allowing new employee types to be added without modifying existing code.

- This scenario demonstrates how OOP principles such as **abstraction, encapsulation, inheritance, polymorphism,** and **interfaces** are important in designing a flexible and maintainable system.

- By refactoring the non-clean code into a clean, modular design, the system becomes easier to extend and manage, **fulfilling new requirements without breaking existing functionality**.

Useful Resources

FOR STUDYING JAVA IN MORE DEPTH

References

- Sommerville (1992) – "Software Engineering" 4e, Pearson.
- https://www.tutorialspoint.com/java/

Java Tutorial

- Java - Home
- Java - Overview
- Java - Environment Setup
- Java - Basic Syntax
- Java - Object & Classes
- Java - Basic Datatypes
- Java - Variable Types
- Java - Modifier Types
- Java - Basic Operators
- Java - Loop Control
- Java - Decision Making
- Java - Numbers
- Java - Characters
- Java - Strings
- Java - Arrays
- Java - Date & Time
- Java - Regular Expressions
- Java - Methods
- Java - Files and I/O
- Java - Exceptions
- Java - Inner classes

Java Advanced

- Java - Data Structures
- Java - Collections
- Java - Generics
- Java - Serialization
- Java - Networking
- Java - Sending Email
- Java - Multithreading
- Java - Applet Basics
- Java - Documentation

Java Object Oriented

- Java - Inheritance
- Java - Overriding
- Java - Polymorphism
- Java - Abstraction
- Java - Encapsulation
- Java - Interfaces
- Java - Packages

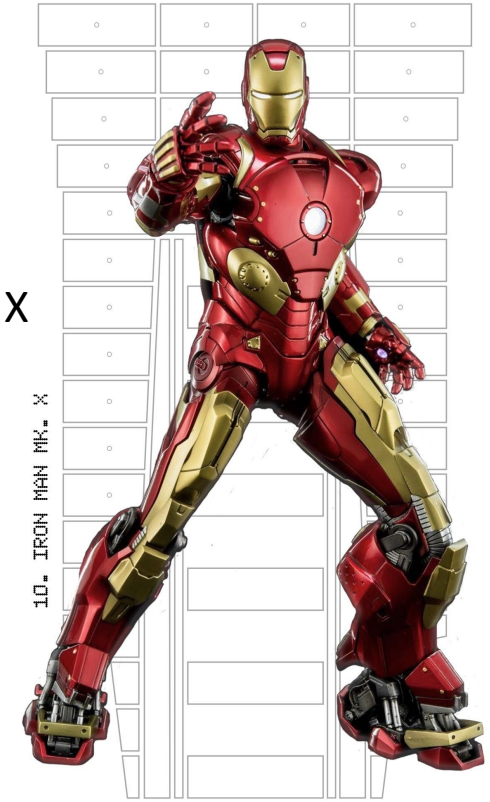
Iron Man

AN EXAMPLE

Iron Man Suit

Mark I

Mark X



Body Plate

Blueprint of BodyPlate

Properties/Item/Equipment/Functionality

Abstraction

Interface

```
public abstract class BodyPlate {
    2 implementations
    public abstract void setFireResistance();
```

```
public interface Flyable {
    1 implementation
    void getSpeed();
}
```

Inheritance

```
public class SilverSoul extends BodyPlate {
    @Override
    public void setFireResistance() {
        double fireResistance = 1.0;
    }
}
```

SilverSoul is a BodyPlate

```
public class Mithral extends BodyPlate implements Flyable {
    @Override
    public void getSpeed() {
        int speed = 40;
    }

    @Override
    public void setFireResistance() {
        double fireResistance = 0.5;
    }
}
```

Mithral is a BodyPlate

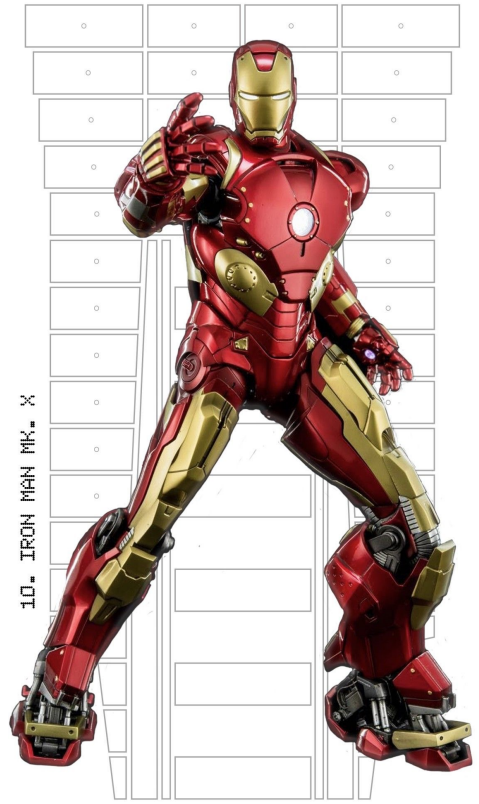
Iron Man Suit

```
public class Mark_I extends Suit {
    1 usage
    public Mark_I() {
        setBodyPlate(new SilverSoul());
        setGauntlet(new MetalGauntlet());
        setWeapon(new FireBeam());
    }
}
```

Mark I is a Suit

Mark X is a Suit

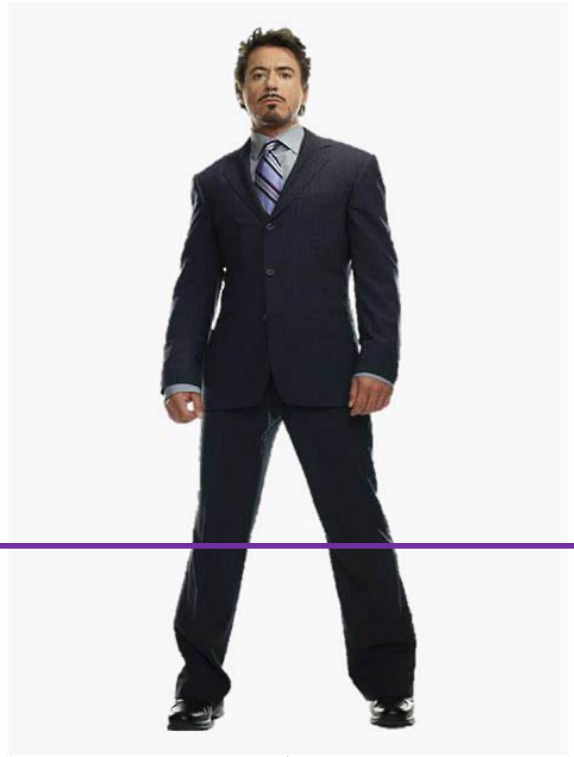
```
public class Mark_X extends Suit {
    1 usage
    public Mark_X() {
        setBodyPlate(new Mithral());
        setGauntlet(new TitaniumGauntlet());
    }
}
```

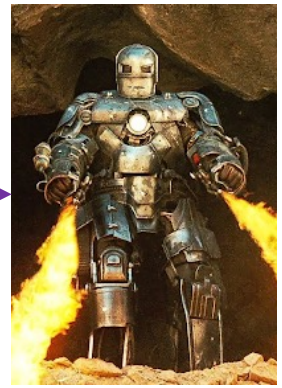


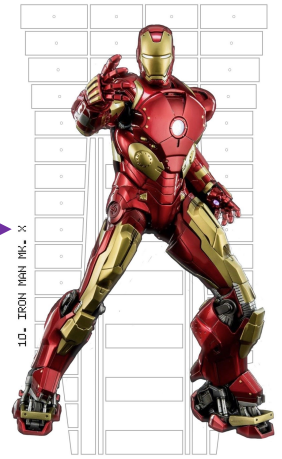
Iron Man!

```java
public class Person {
    1 usage
    private String name;
    1 usage
    private Suit suit;

    1 usage
    public Person(String name) { this.name = name; }

    1 usage
    public void SuitUp(Suit suit) { this.suit = suit; }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Mark_I mark_I = new Mark_I();
        Mark_X mark_X = new Mark_X();

        Person person = new Person( name: "Tony Stark");

        person.SuitUp(mark_I);
    }
}
```



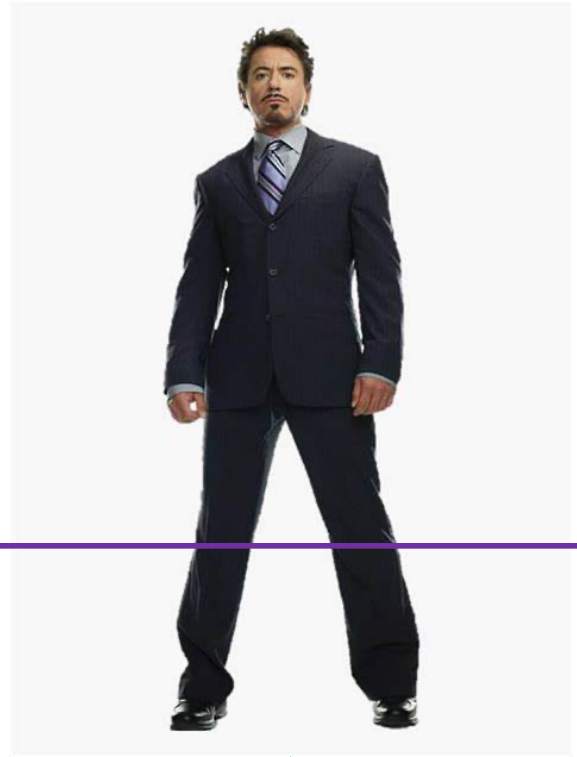


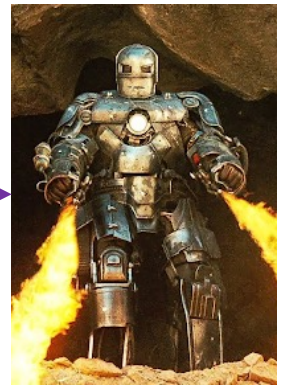


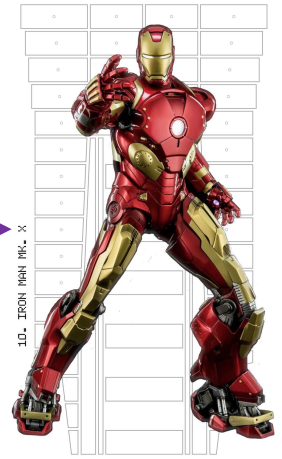
```java
public class Person {
    2 usages
    private String name;
    2 usages
    private Suit suit;

    1 usage
    public Person(String name) { this.name = name; }

    public Person(String name, Suit suit) {
        this.name = name;
        this.suit = suit;
    }

    1 usage
    public void SuitUp(Suit suit) { this.suit = suit; }
}
```

Polymorphism

```java
public class Main {
    public static void main(String[] args) {
        Mark_I mark_I = new Mark_I();
        Mark_X mark_X = new Mark_X();

        Person person = new Person( name: "Tony Stark");

        person.SuitUp(mark_I);
    }
}
```







Abstraction
Encapsulation
Inheritance
Polymorphism
Interface

Things to Consider

- Is it easy to add/remove a body plate?
- Is it easy to introduce a new armor (e.g., greave) or remove an existing armor?
- Is it easy to add/remove propert(ies) in a suit?
- Is it easy to add/remove a suit?
- Is it easy to add/remove weapon(s) from a suit?
- Is it easy to reuse the same suit for different person?
- Is it easy to …

OOP in Maintainable Software

- Modularity – can we divide the code in modular form?
- Reusability – can we reuse the same module/code?
- Analyzability – is it easy to analyze the code?
- Changeability – is it easy to change the code?
- Modification Stability – will it affect other code?
- Testability – how can we test the code?
- Compliance – does the code works in different OS or version?

Put Your Mind in Maintenance Mode