Please note that the slides published AFTER the lectures and workshops are the official slides and are the ones that should be used for revision.

COMP2013 Software Maintenance

Lecture 02

OO and Java Refresher (2/2)

Peer-Olaf Siebers

University of Nottingham
UK | CHINA | MALAYSIA

Week 2 Organisation

- Lecture 2:
  - Going through more advanced Java topics
  - Java Collections framework
  - Implementation of object oriented principles
- Lab 2:
  - Working further on the ZooApp example
  - Looking at packages
- Workshop 2:
  - CW1 Release
  - IDEs + Java 9/10/11 additions
  - Maintaining the ZooApp (basic maintenance)

java collections framework

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?
  - A library contains re-usable chunks of code. These re-usable chunks of code are linked to your program through APIs.

Java Collections Framework

- Java Collections Framework principle 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 signatures

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

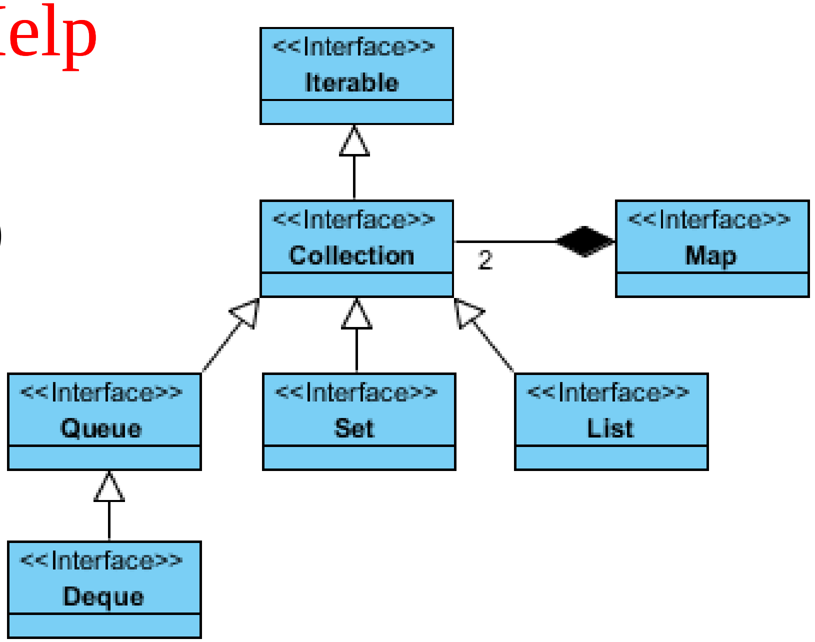
- The framework contains algorithmic operations
  - e.g. searching; sorting

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 one object at a time

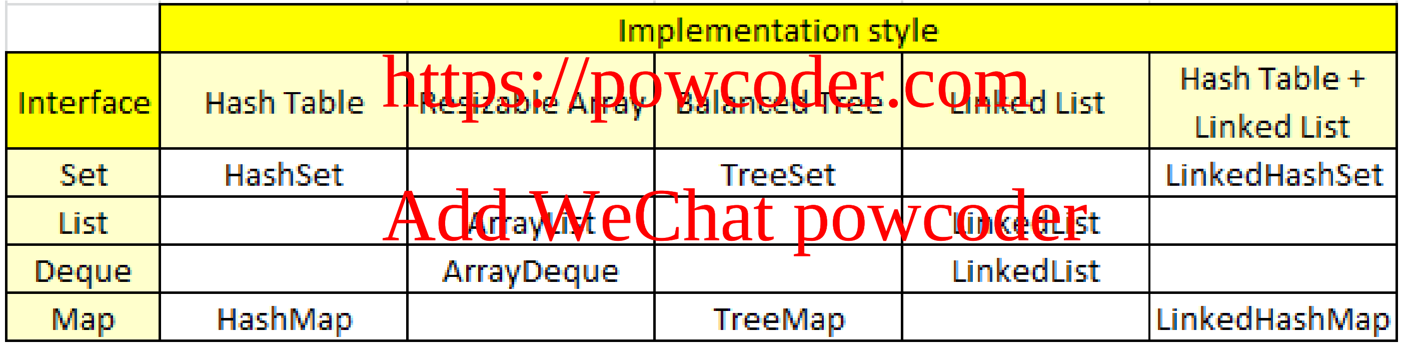


Java Collections Framework

- 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 can be found in the java.lang.* package

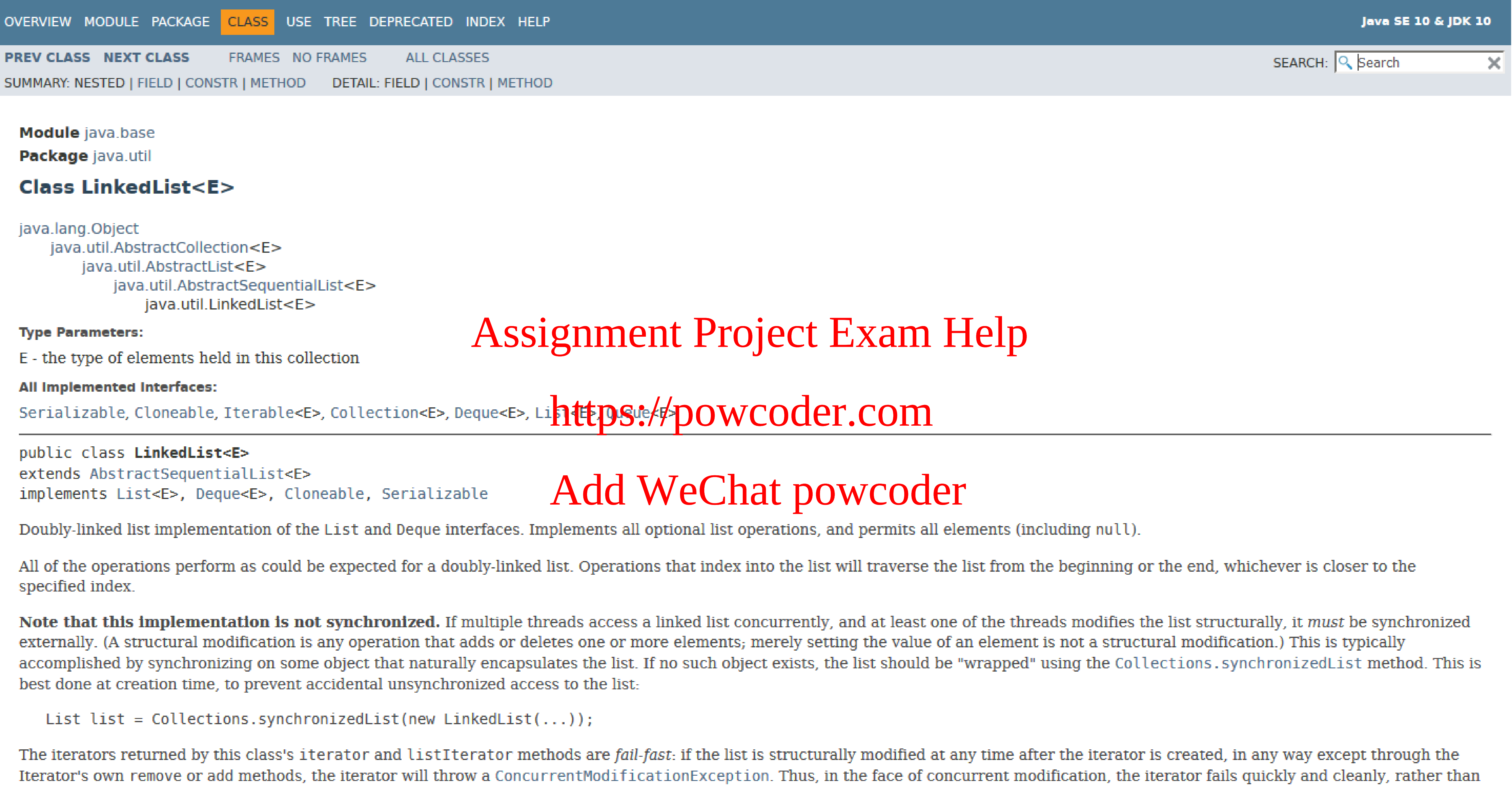
Java Collections Framework

- Classes that implement the collection interfaces typically have names in the form of <Implementation style><Interface>



	Implementation style				
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)


OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
Java SE 10 & JDK 10
PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES
SEARCH: Search
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

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
```

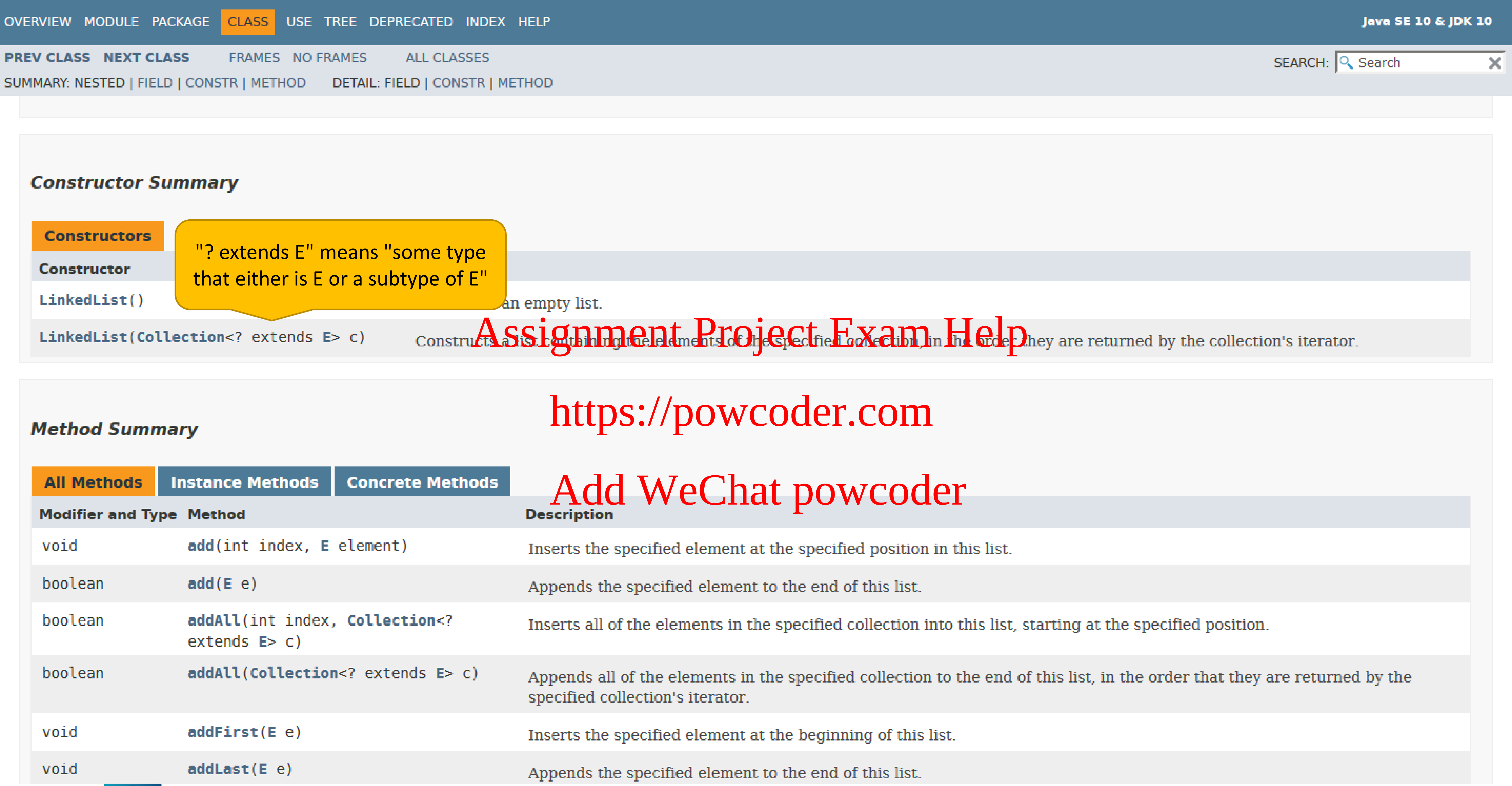
Doubly-linked list implementation of the List and Deque interfaces. Implements all optional list operations, and permits all elements (including null).

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


OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
Java SE 10 & JDK 10
PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES
SEARCH: Search
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Constructor Summary

Constructors

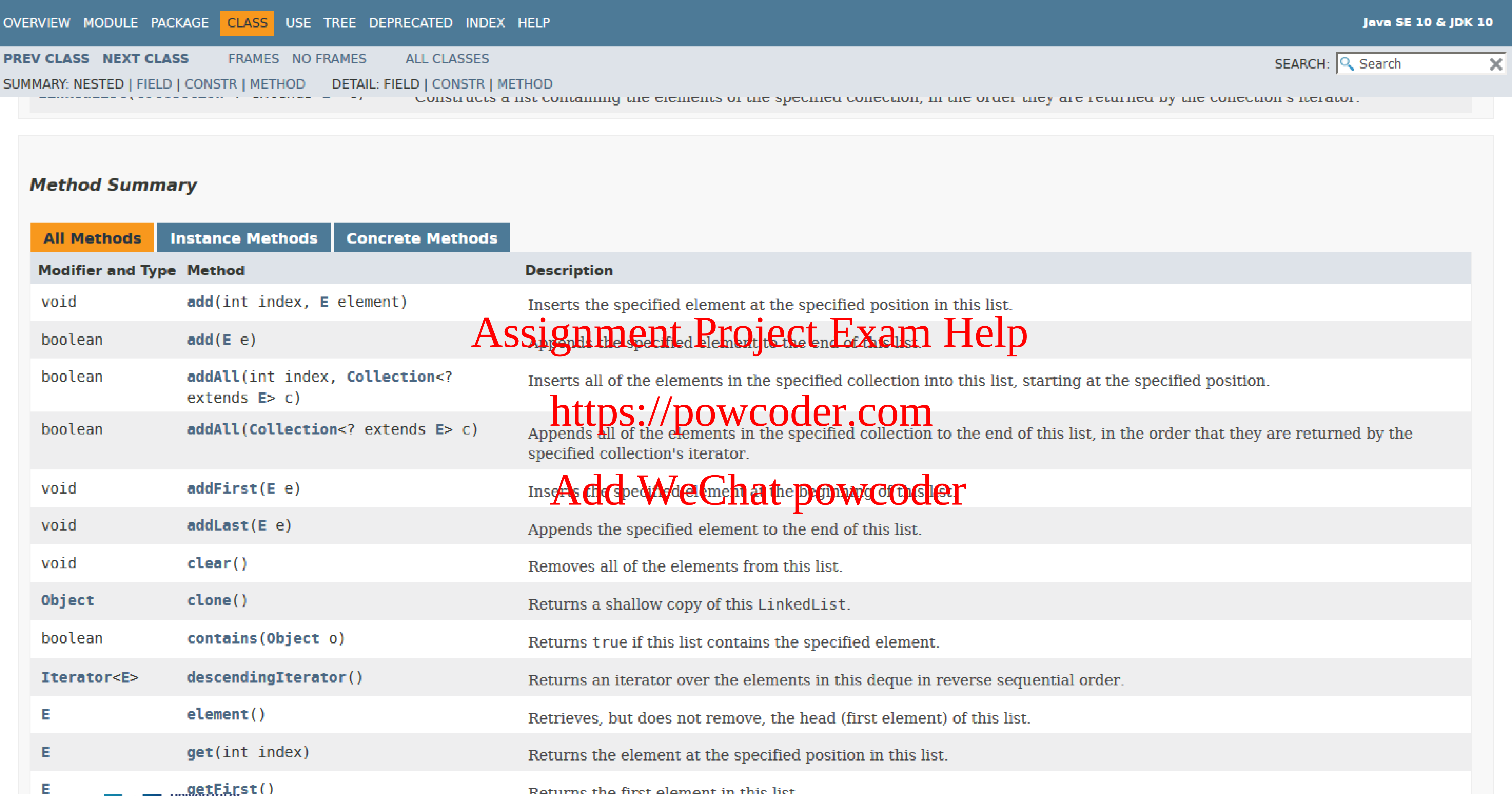
Constructor	Description
`LinkedList()`	an empty list.
`LinkedList(Collection<? extends E> c)`	Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

"? extends 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.



OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
Java SE 10 & JDK 10
PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES
SEARCH: Search
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

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.

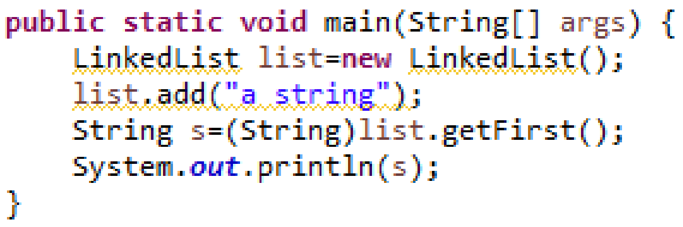
Method Summary

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

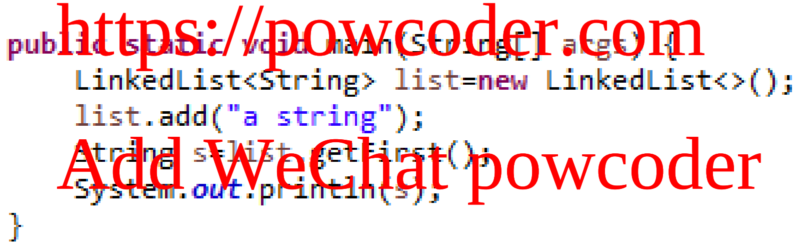
- Non typesafe collections (do not use)
  - Collection constructors are not able to specify the type of objects the collection is intended to contain
  - Need to cast objects when using them; a "ClassCastExeption" will be thrown if we attempt to cast to the wrong type



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

Java Collections Framework

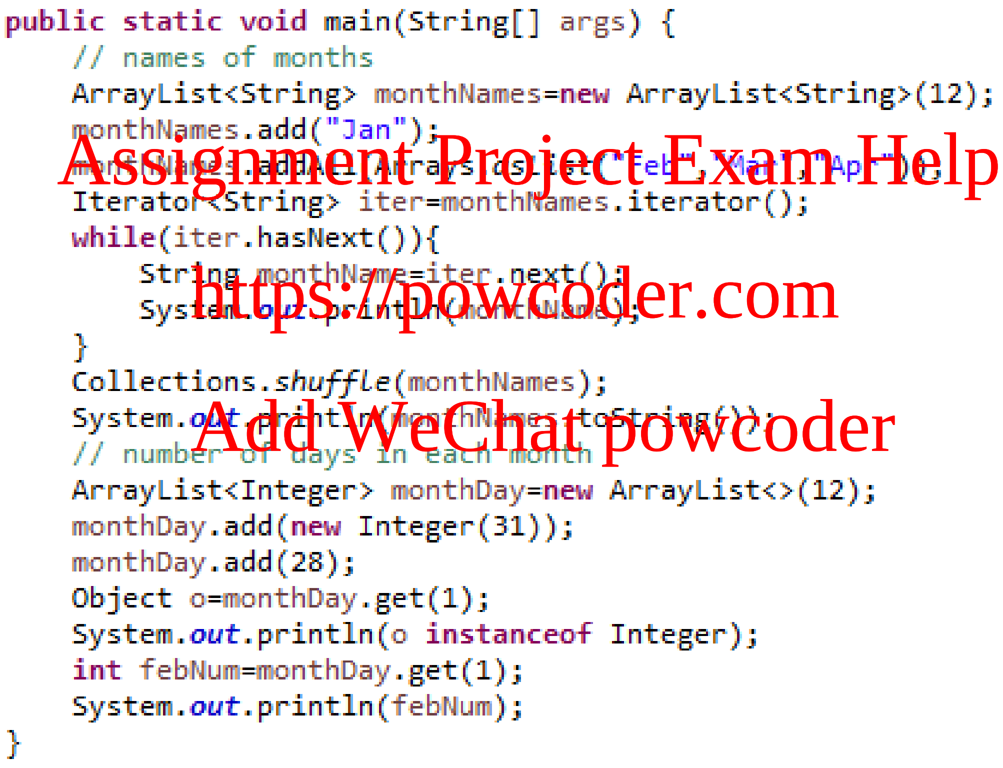
- Typesafe collections with "Generics"
  - Classes support generics by allowing a type variable to be included in their declaration; type are declared for the reference and constructor



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

- You cannot type a collection using a primitive type
  - Values of primitive types need to be put into objects of a suitable wrapper class before they can be added to a collection

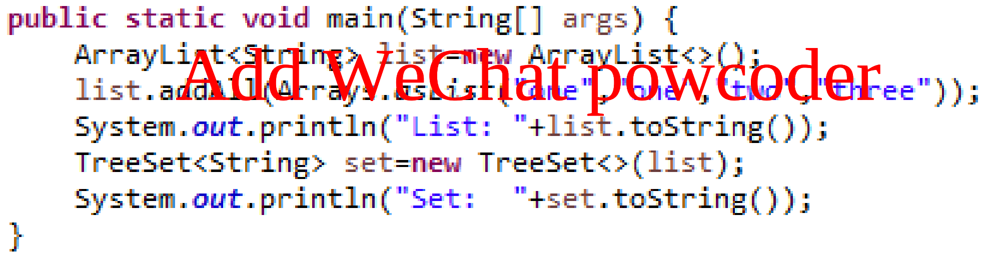
ArrayList Class



```
public static void main(String[] args) {
    // names of months
    ArrayList<String> monthNames=new ArrayList<String>(12);
    monthNames.add("Jan");
    monthNames.addAll(Arrays.asList("Feb","Mar","Apr"));
    Iterator<String> iter=monthNames.iterator();
    while(iter.hasNext()){
        String monthName=iter.next();
        System.out.println(monthName);
    }
    Collections.shuffle(monthNames);
    System.out.println(monthNames.toString());
    // number of days in each month
    ArrayList<Integer> monthDay=new ArrayList<>(12);
    monthDay.add(new Integer(31));
    monthDay.add(28);
    Object o=monthDay.get(1);
    System.out.println(o instanceof Integer);
    int febNum=monthDay.get(1);
    System.out.println(febNum);
}
```

TreeSet Class

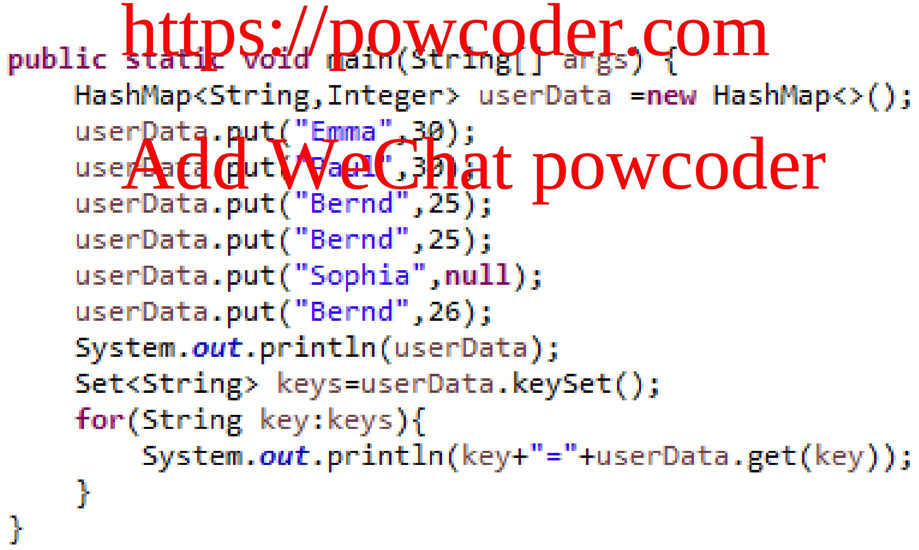
- TreeSet provides an implementation of the Set interface that uses a tree for storage. Objects are stored in sorted, ascending order.



```
public static void main(String[] args) {
    ArrayList<String> list=new ArrayList<>();
    list.addAll(Arrays.asList("one","one","two","three"));
    System.out.println("List: "+list.toString());
    TreeSet<String> set=new TreeSet<>(list);
    System.out.println("Set:  "+set.toString());
}
```

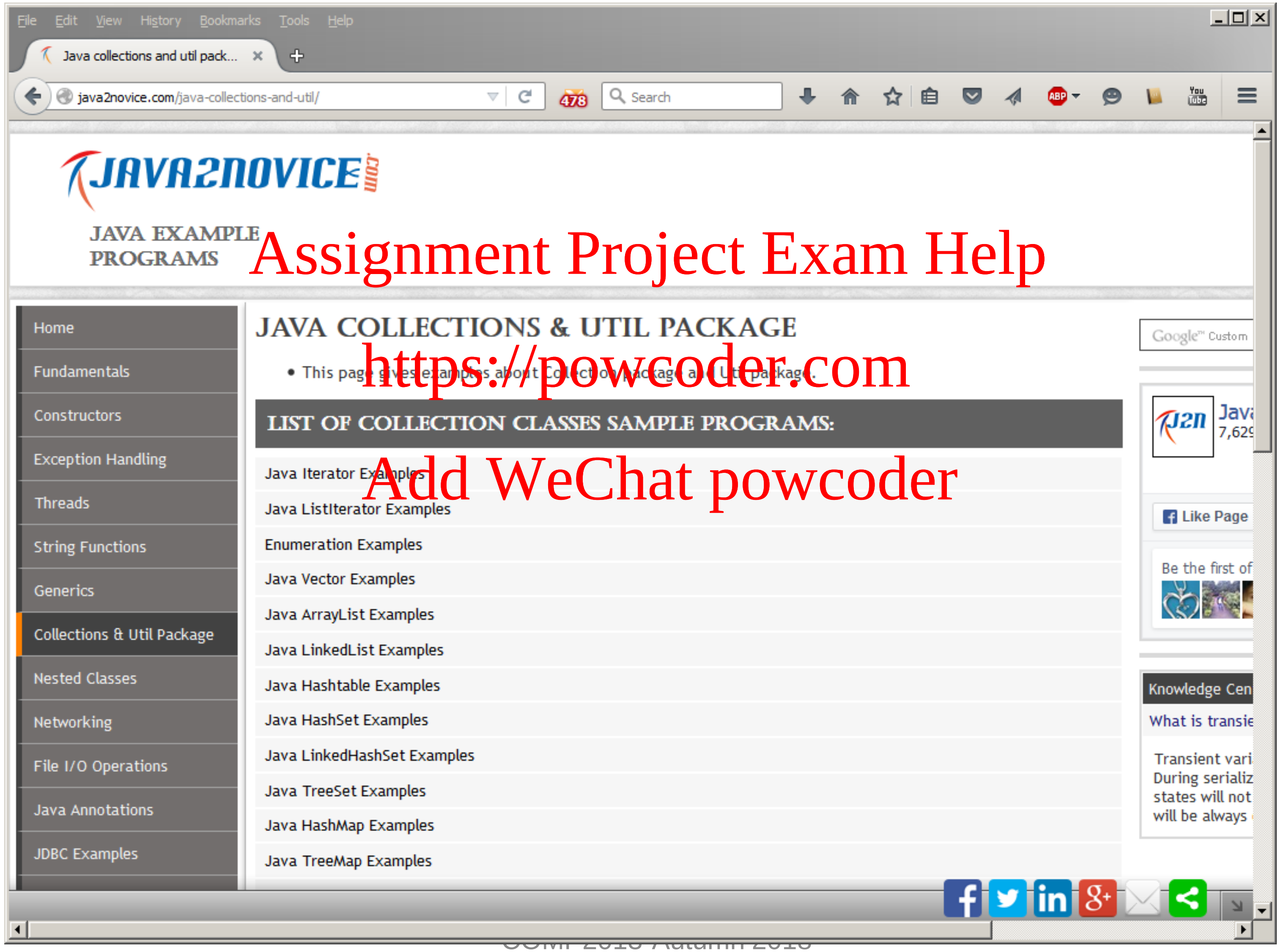
HashMap Class

- HashMap is a Hash table based implementation of the Map interface. This implementation provides all of 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",26);
    System.out.println(userData);
    Set<String> keys=userData.keySet();
    for(String key:keys){
        System.out.println(key+"="+userData.get(key));
    }
}
```

Java Collections Examples



JAVA COLLECTIONS & UTIL PACKAGE

- This page gives examples about Collection package and Util package.

LIST OF COLLECTION CLASSES SAMPLE PROGRAMS:

Java Iterator Examples

Java ListIterator Examples

Enumeration Examples

Java Vector Examples

Java ArrayList Examples

Java LinkedList Examples

Java Hashtable Examples

Java HashSet Examples

Java LinkedHashSet Examples

Java TreeSet Examples

Java HashMap Examples

Java TreeMap Examples

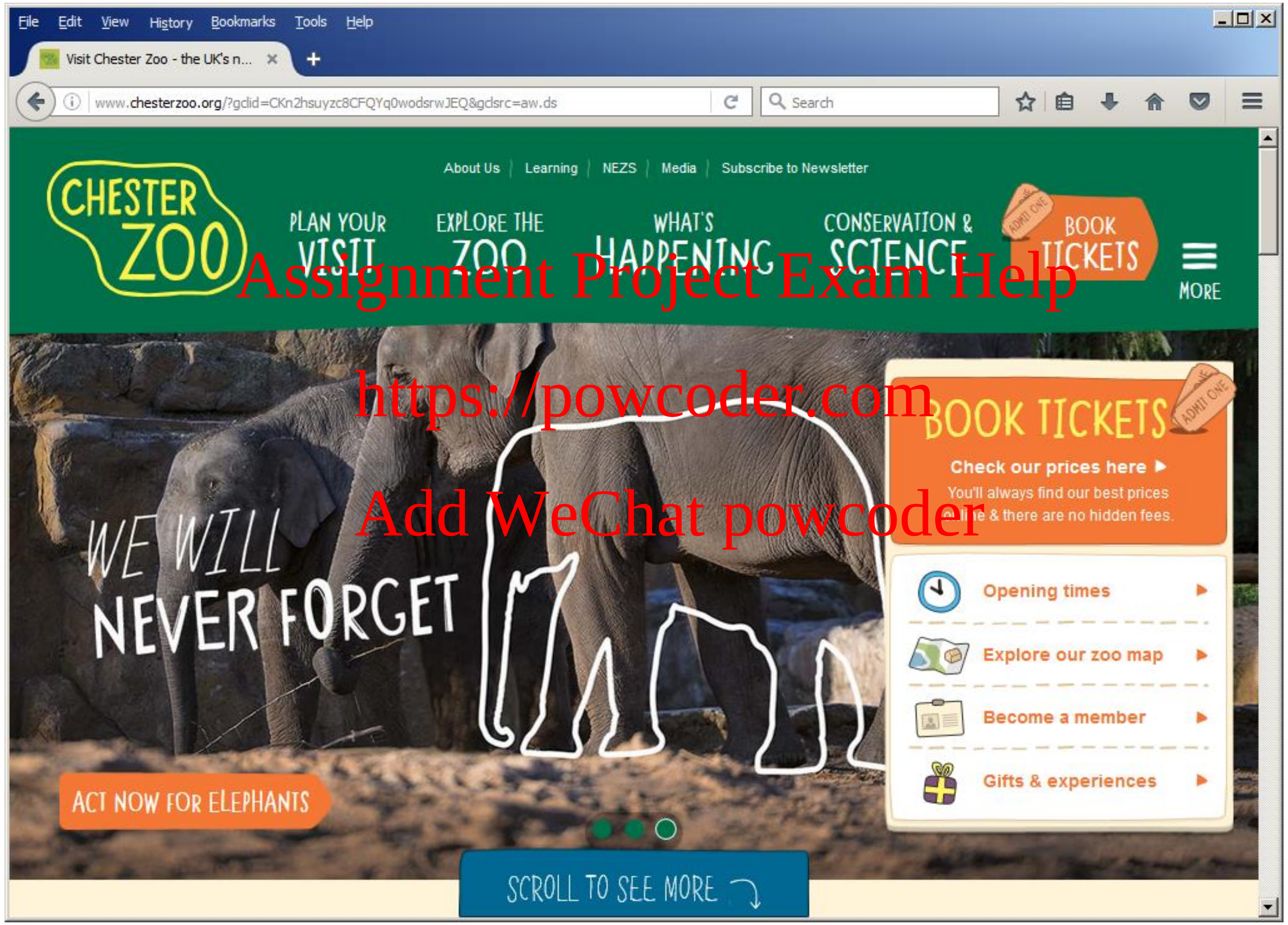
Transient vari
During serializ
states will not
will be always

</>

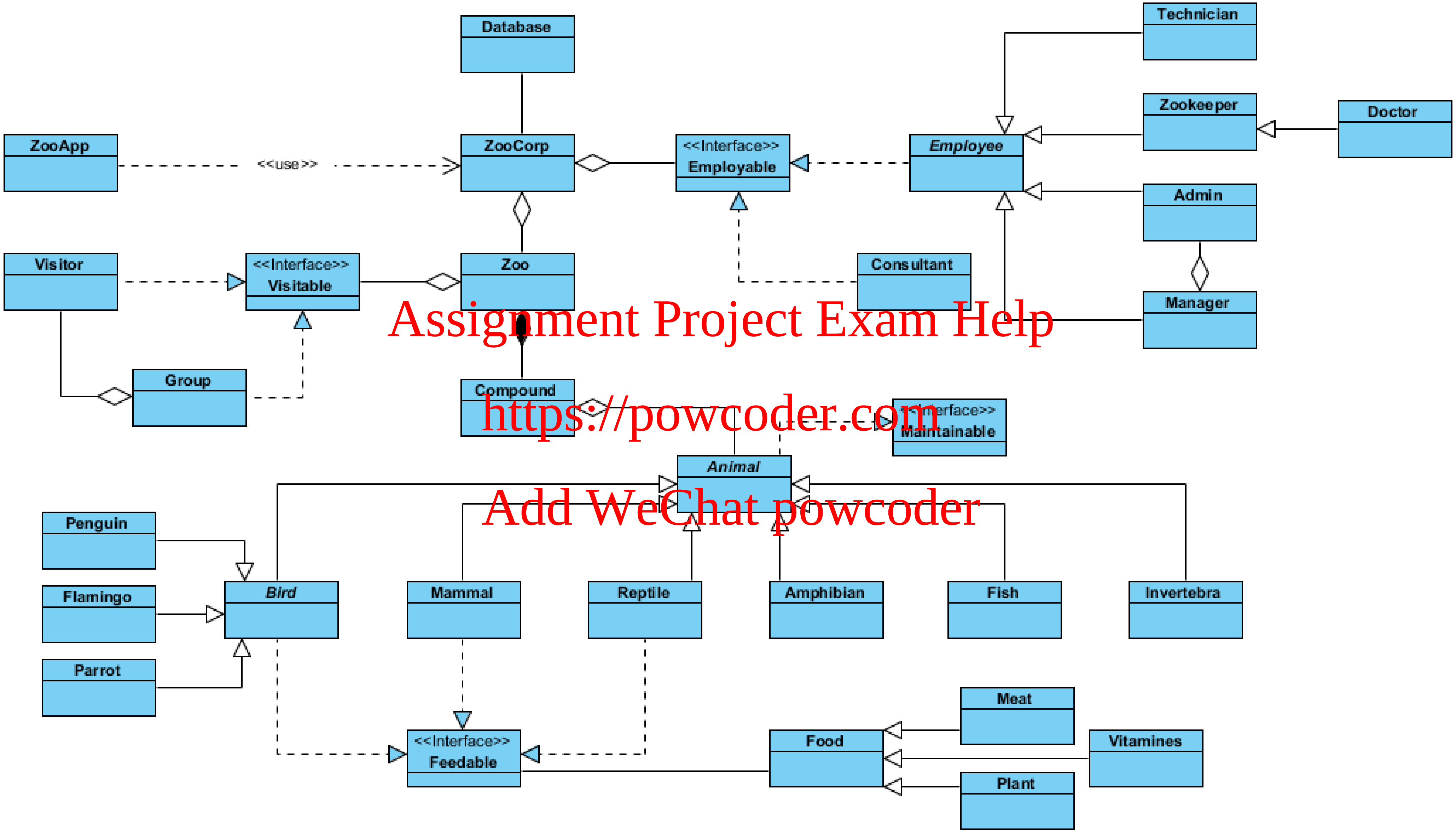
implementation of object oriented principles

Aggregation and Composition; Inheritance; Polymorphism; Abstract Methods and Classes; Interfaces

Case Study: Zoo Management



You'll always find our best prices online & there are no hidden fees.

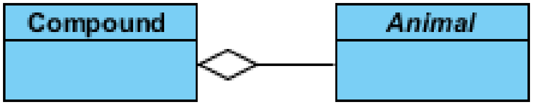

Database
ZooCorp
ZooApp
<<use>>
<<Interface>>
Employable
Employee
Technician
Zookeeper
Doctor
Admin
Manager
Consultant
Visitor
<<Interface>>
Visitable
Zoo
Group
Compound
<<Interface>>
Maintainable
Animal
Penguin
Flamingo
Parrot
Bird
Mammal
Reptile
Amphibian
Fish
Invertebra
<<Interface>>
Feedable
Food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

Aggregation and Composition

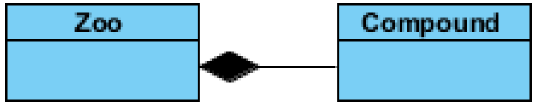
- What is the difference between the Aggregations and Compositions?

  - Aggregation

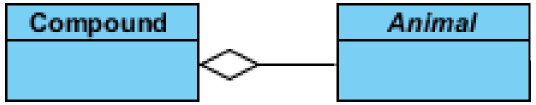
    - The object exists outside the other, is created outside, so it is passed as an argument (for example) to the constructor

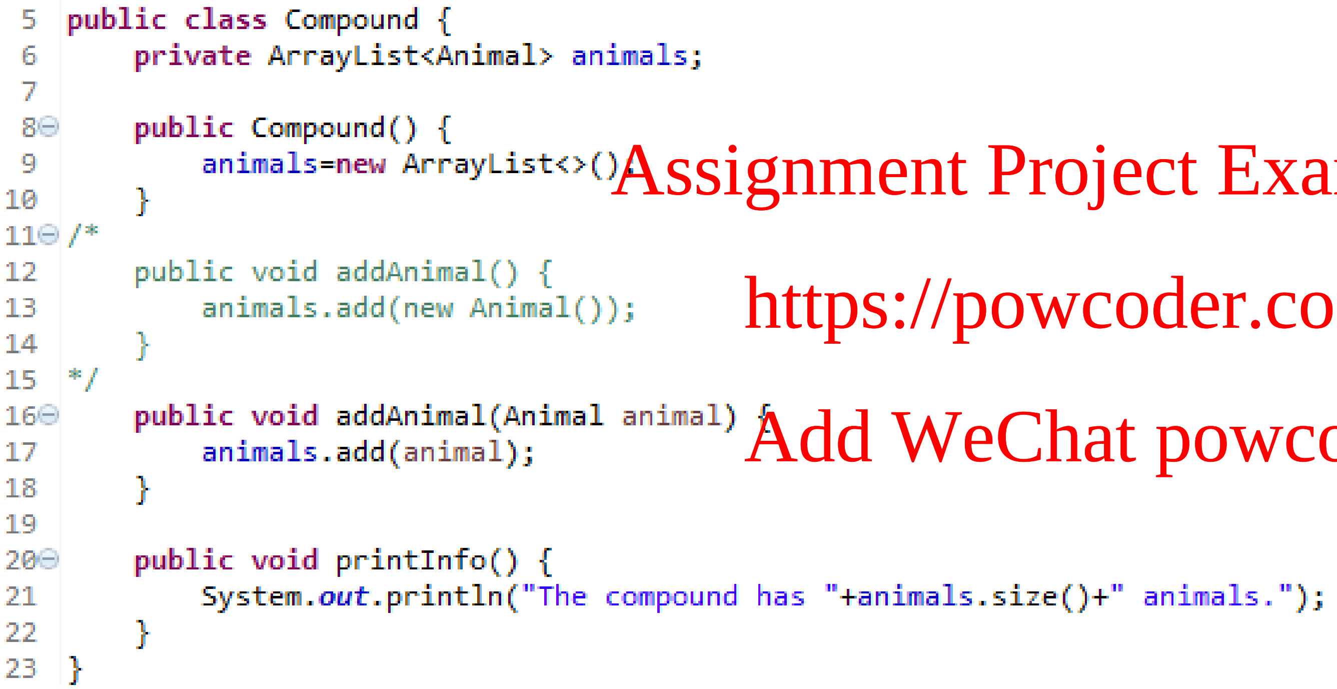


  - Composition
    - The object only exists, or only makes sense inside the other, as a part of the other

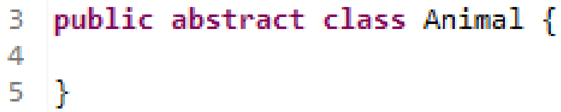


Aggregation



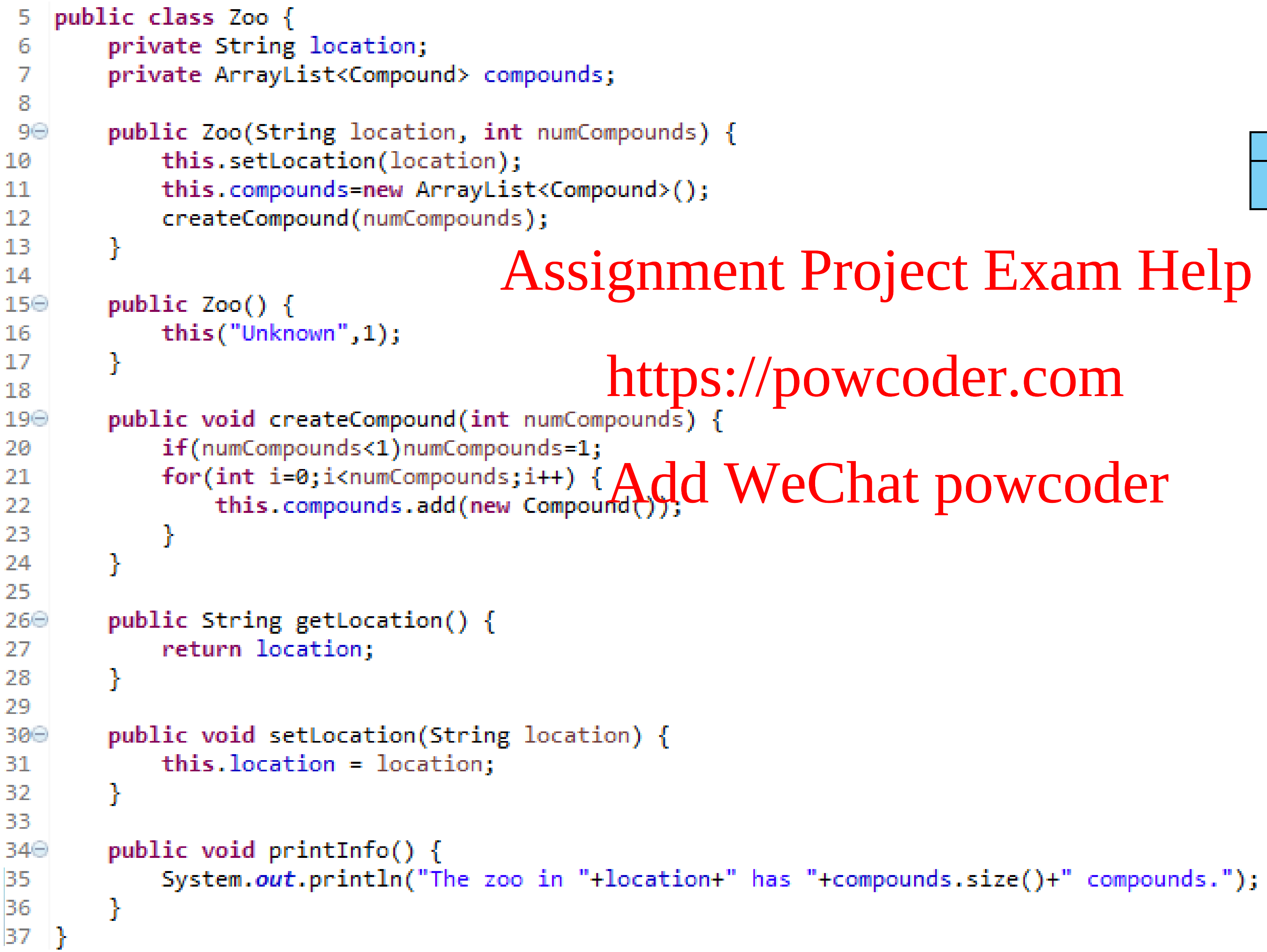


```java
public class Compound {
    private ArrayList<Animal> animals;

    public Compound() {
        animals=new ArrayList<>();
    }
/*
    public void addAnimal() {
        animals.add(new Animal());
    }
*/
    public void addAnimal(Animal animal) {
        animals.add(animal);
    }

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

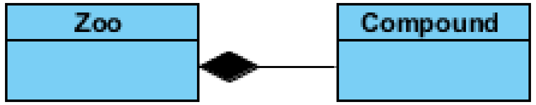


```java
public abstract class Animal {

}
```

Composition



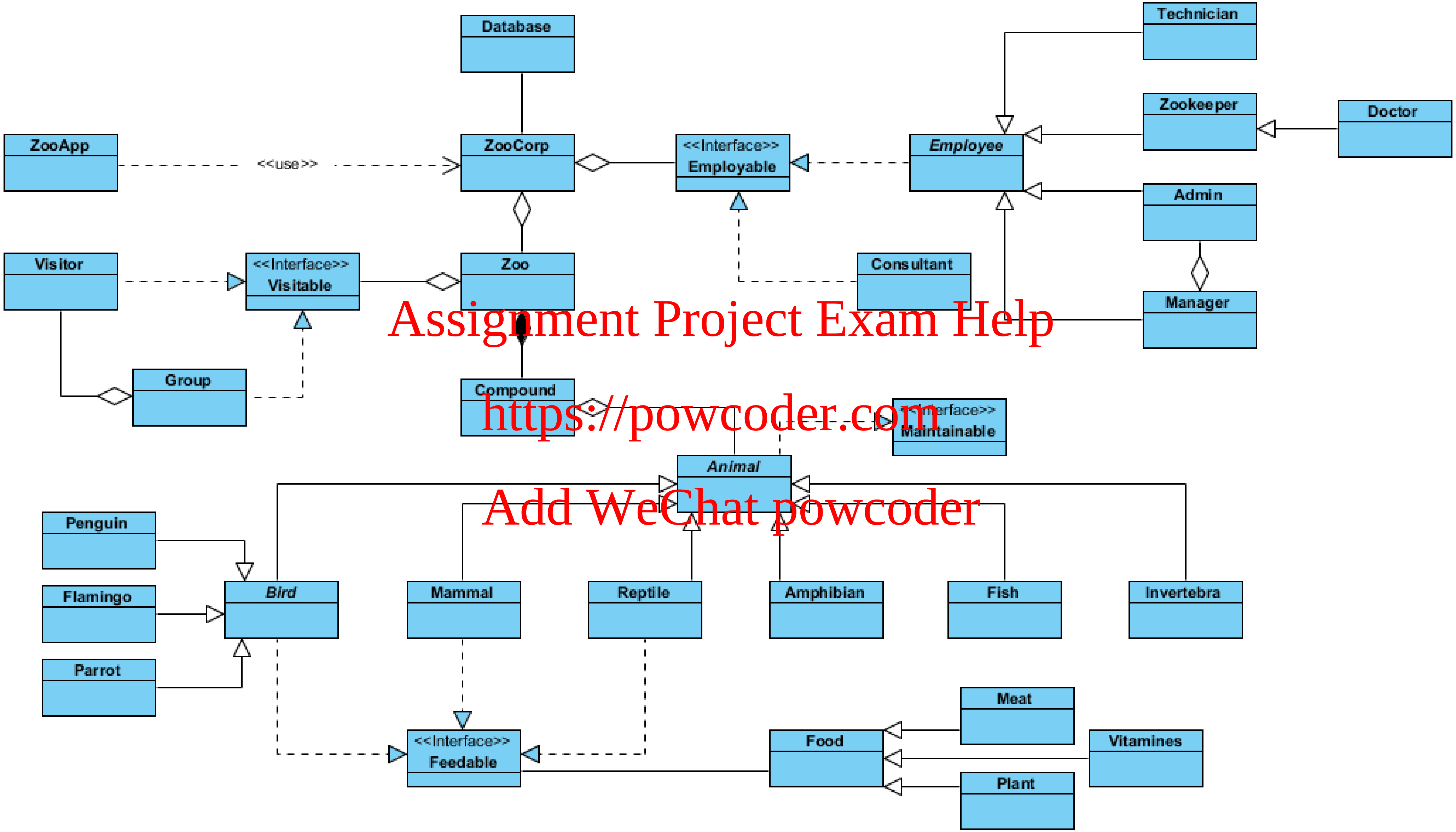
```java
public class Zoo {
    private String location;
    private ArrayList<Compound> compounds;

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

    public Zoo() {
        this("Unknown",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?

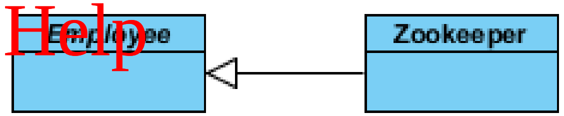
  - Inheritance: Forming new classes based on existing ones
    - A way to share/reuse code between two or more classes
  - Superclass: Parent class being extended
  - Subclass: Child class that inherits behavior from superclass
    - Gets a copy of every field and method from superclass
  - "is-a" relationship: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


Database
Technician
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
Meat
Vitamines
Plant
Food
<<Interface>>
Feedable

Inheritance

- Example:

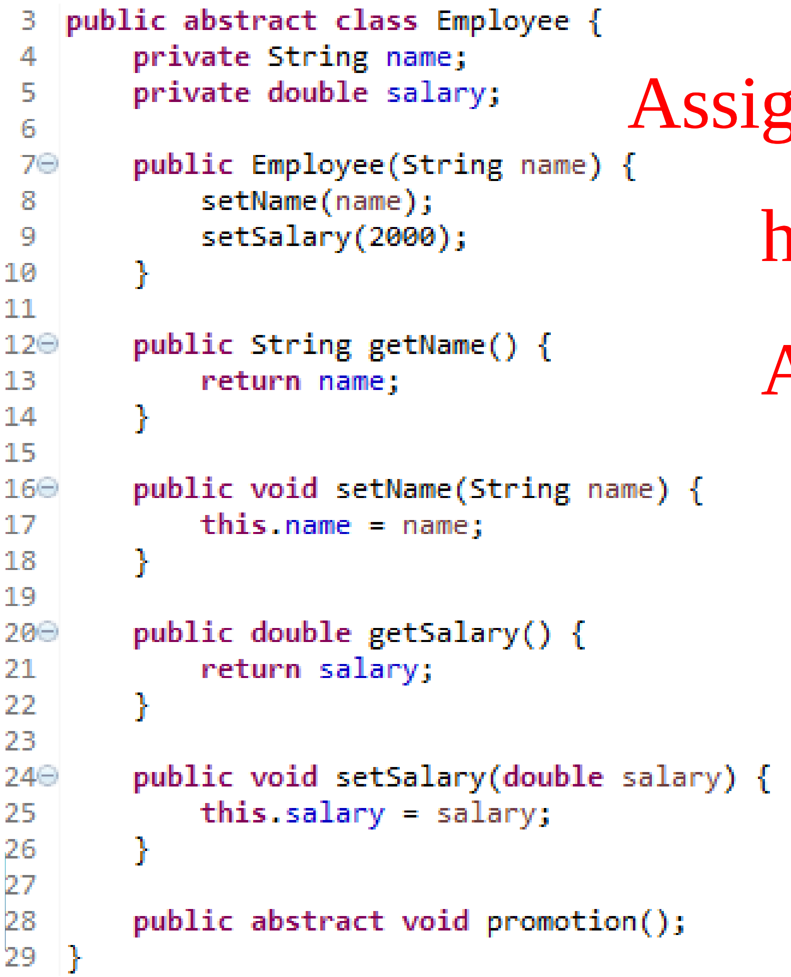
```
public class Zookeeper extends Employee {
...
}
```



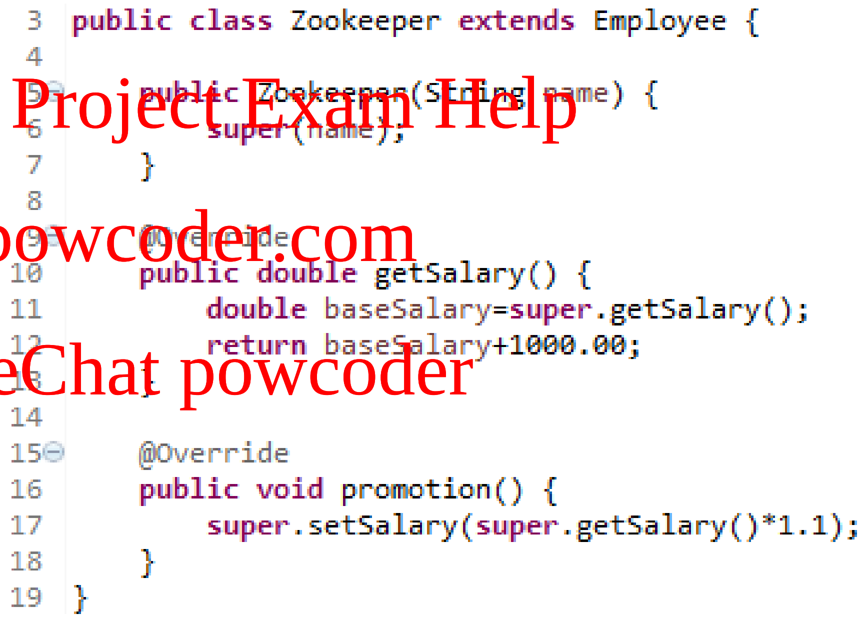
- By extending Employee, each Zookeeper object now:
  - Receives a copy of each method from Employee automatically
  - Can be treated as an Employee by client code
  - Zookeeper can replace ("override") behavior from Employee

Inheritance

- A subclass can call its parent's method/constructor:



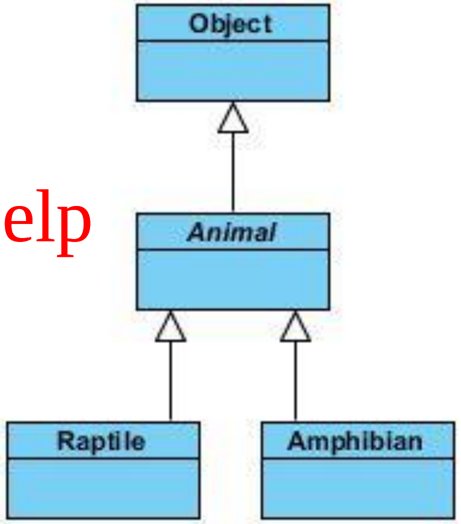
```java
public abstract class Employee {
    private String name;
    private double salary;

    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();
}
```



```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);
    }
}
```

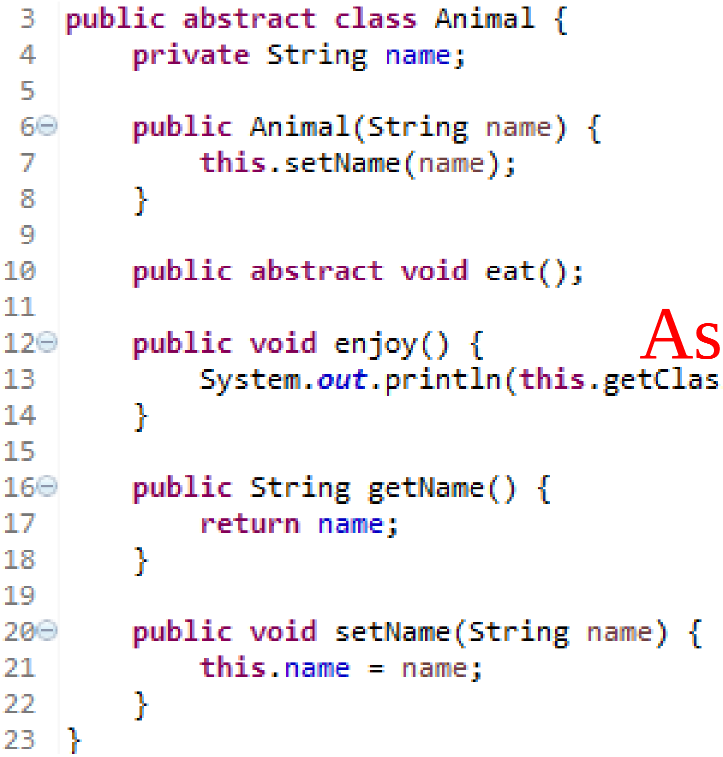
Inheritance

- Every class is either
  - a direct subclass of Object (no extends)
  - a subclass of a descendant of Object (extends)

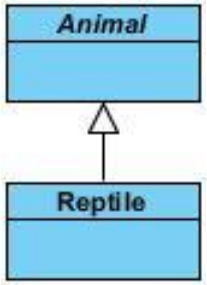


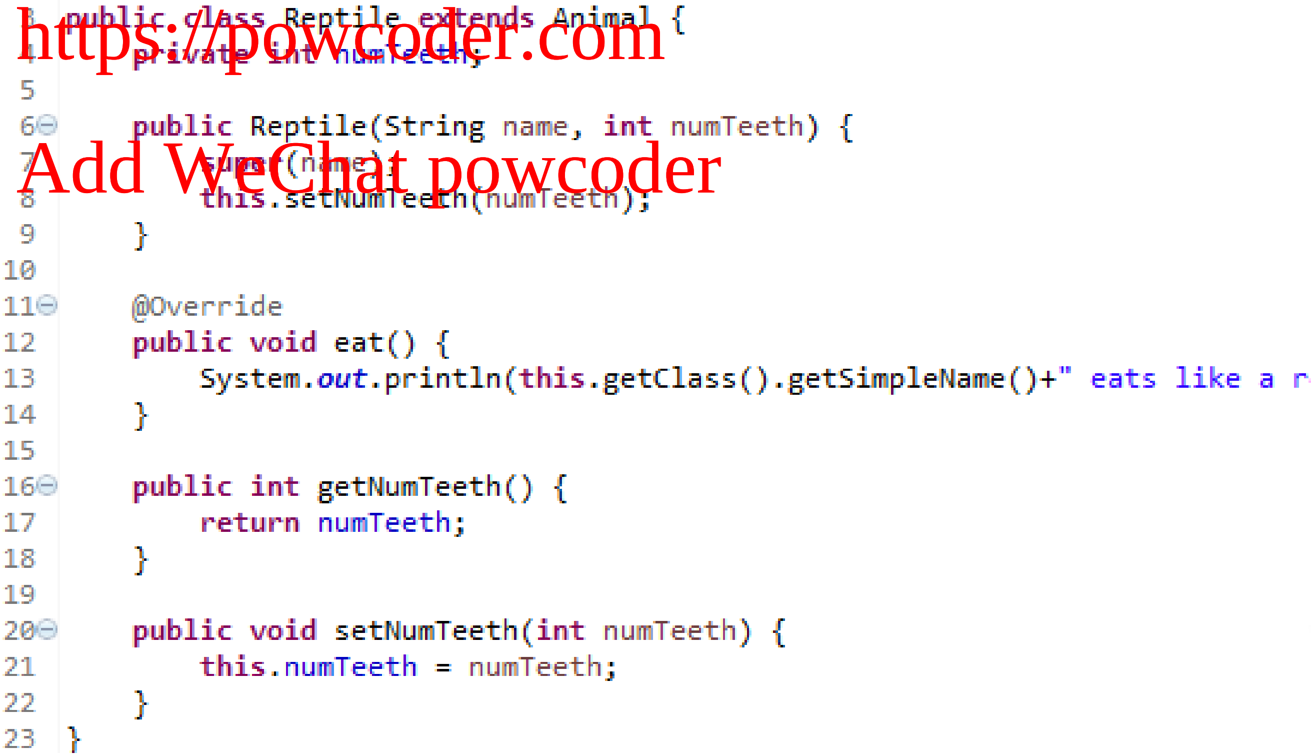
- Class Reptile extends Animal
- Class Amphibia extends Animal
- Class Animal extends Object

Inheritance



```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 an animal.");
    }

    public String getName() {
        return name;
    }

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

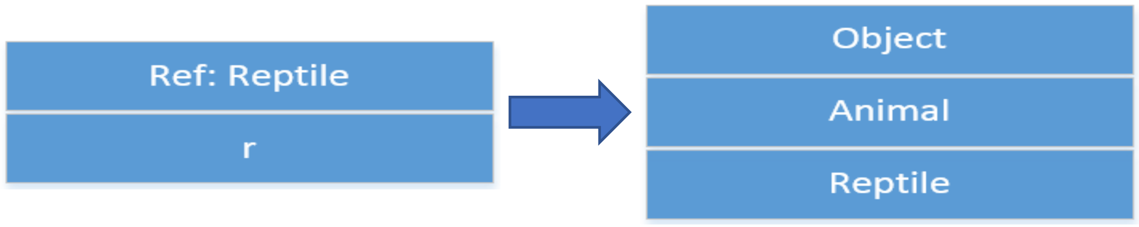




```java
public class Reptile extends Animal {
    private int numTeeth;

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

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

    public int getNumTeeth() {
        return numTeeth;
    }

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

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



Ref: Reptile

r

Object

Animal

Reptile

Inheritance

- Which of these works?
  - Reptile r = new Reptile();
  - Animal a = new Reptile();
  - Object o = new Reptile();
  - Reptile r = new Animal();
  - Animal a = new Object()

Inheritance

- Casting primitives

```
double d;
float f;
d = f;      // legal...no loss of information
f = d;      // illegal...potential loss of information
```

- Casting references

```
Object o;
Reptile r;
o = r;      // legal...a reptile is an object
r = o;      // illegal...not all objects are reptiles
```

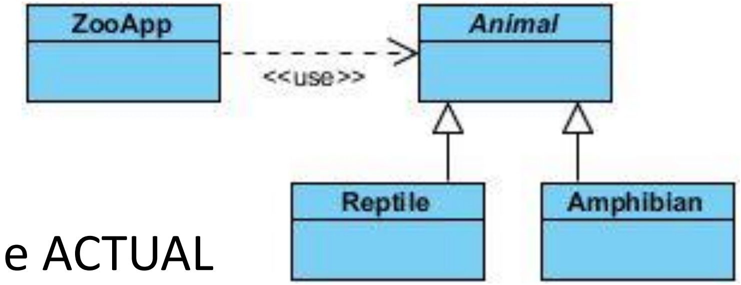
Polymorphism

- What is the difference between polymorphism, method overloading, and method overriding?

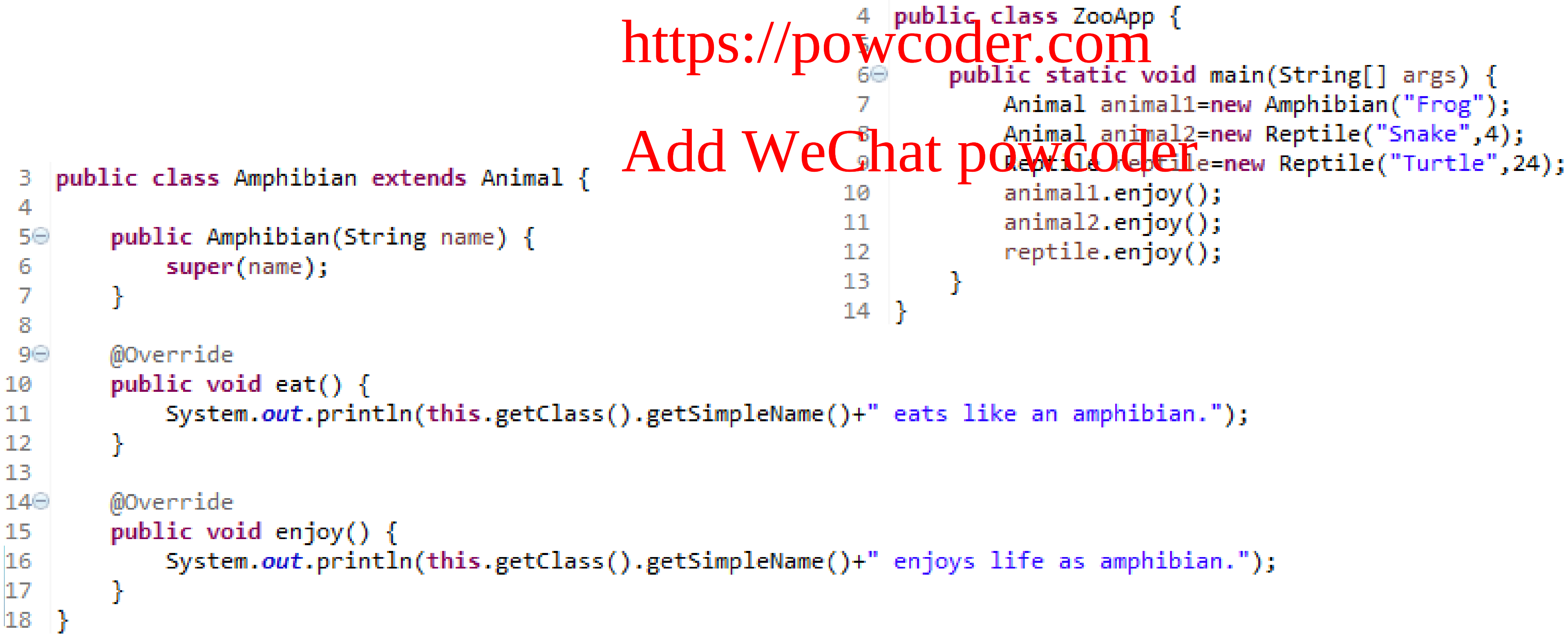
  - Polymorphism
    - Polymorphism is an object oriented concept
    - Method overloading and method overriding are two forms of polymorphism

  - Method overloading
    - Methods with same the 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 sub class
    - Resolved during runtime (dynamic binding)

Polymorphism



- Dynamic Binding
  - At run time (dynamic) when a method is invoked on a reference the ACTUAL OBJECT is examined and the "lowest" or closest version of the method is actually run.

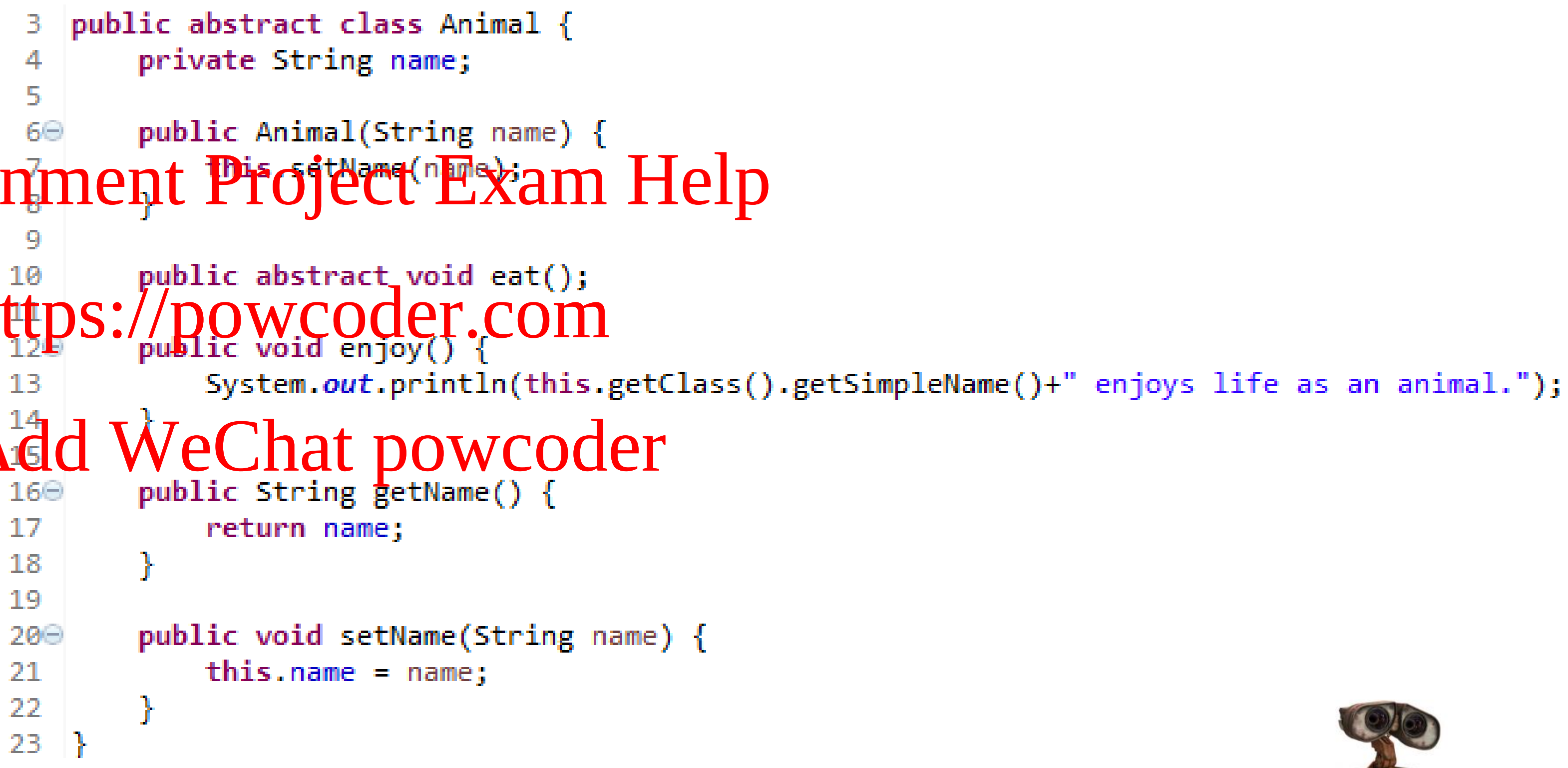


```java
public class ZooApp {

    public static void main(String[] args) {
        Animal animal1=new Amphibian("Frog");
        Animal animal2=new Reptile("Snake",4);
        Reptile reptile=new Reptile("Turtle",24);
        animal1.enjoy();
        animal2.enjoy();
        reptile.enjoy();
    }
}
```

```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.");
    }
}
```

Abstract Methods and Classes

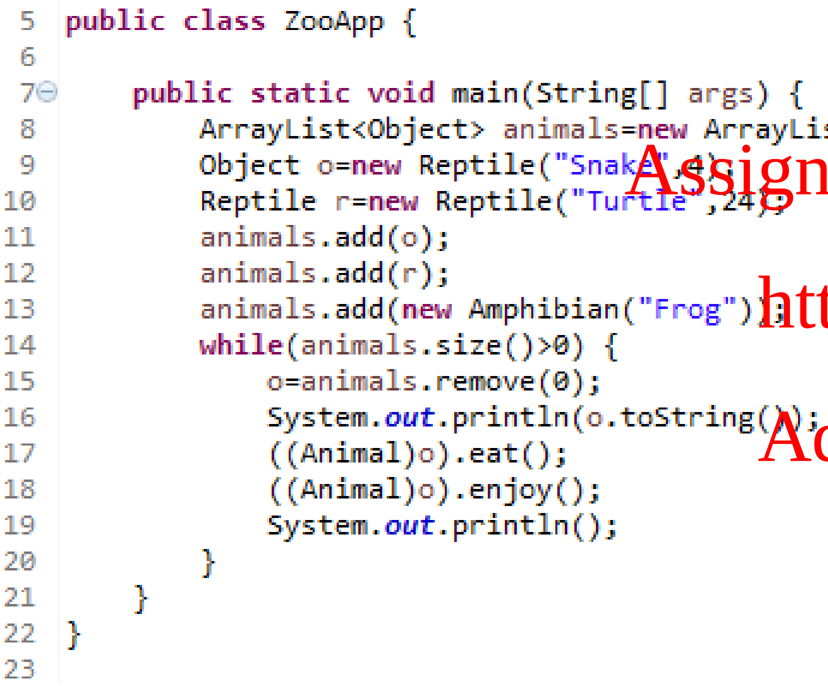
- Any subclass of class Animal has two choices:
  - Define a eat method (i.e. { })
  - Be abstract



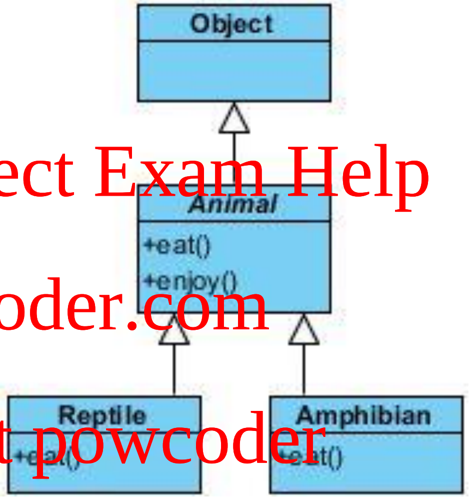
```
public abstract class Animal {
    private String name;

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

    public abstract void eat();

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

    public String getName() {
        return name;
    }

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

- Note:
  - Abstract classes may not be used to instantiate or make objects (new)
  - References to abstract classes are legal

Abstract Methods and Classes

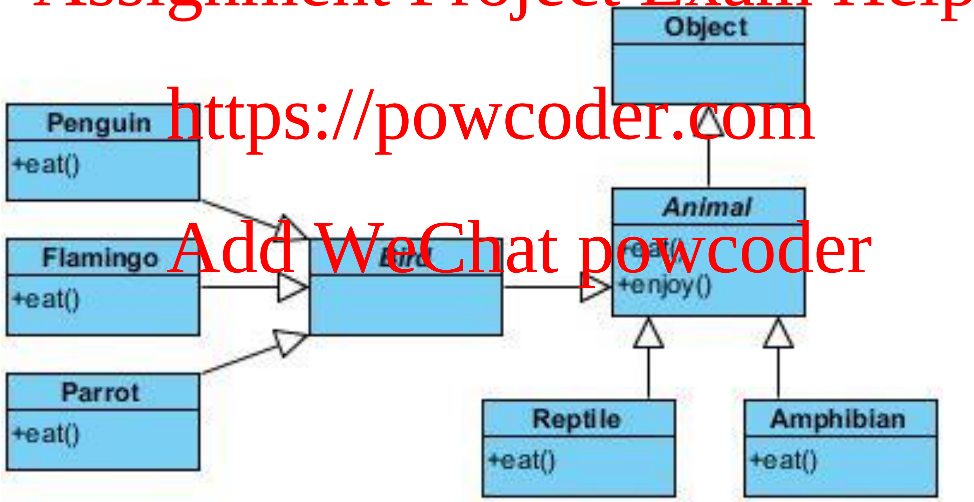


```
public class ZooApp {

    public static void main(String[] args) {
        ArrayList<Object> animals=new ArrayList<>();
        Object o=new Reptile("Snake",4);
        Reptile r=new Reptile("Turtle",24);
        animals.add(o);
        animals.add(r);
        animals.add(new Amphibian("Frog"));
        while(animals.size()>0) {
            o=animals.remove(0);
            System.out.println(o.toString());
            ((Animal)o).eat();
            ((Animal)o).enjoy();
            System.out.println();
        }
    }
}
```



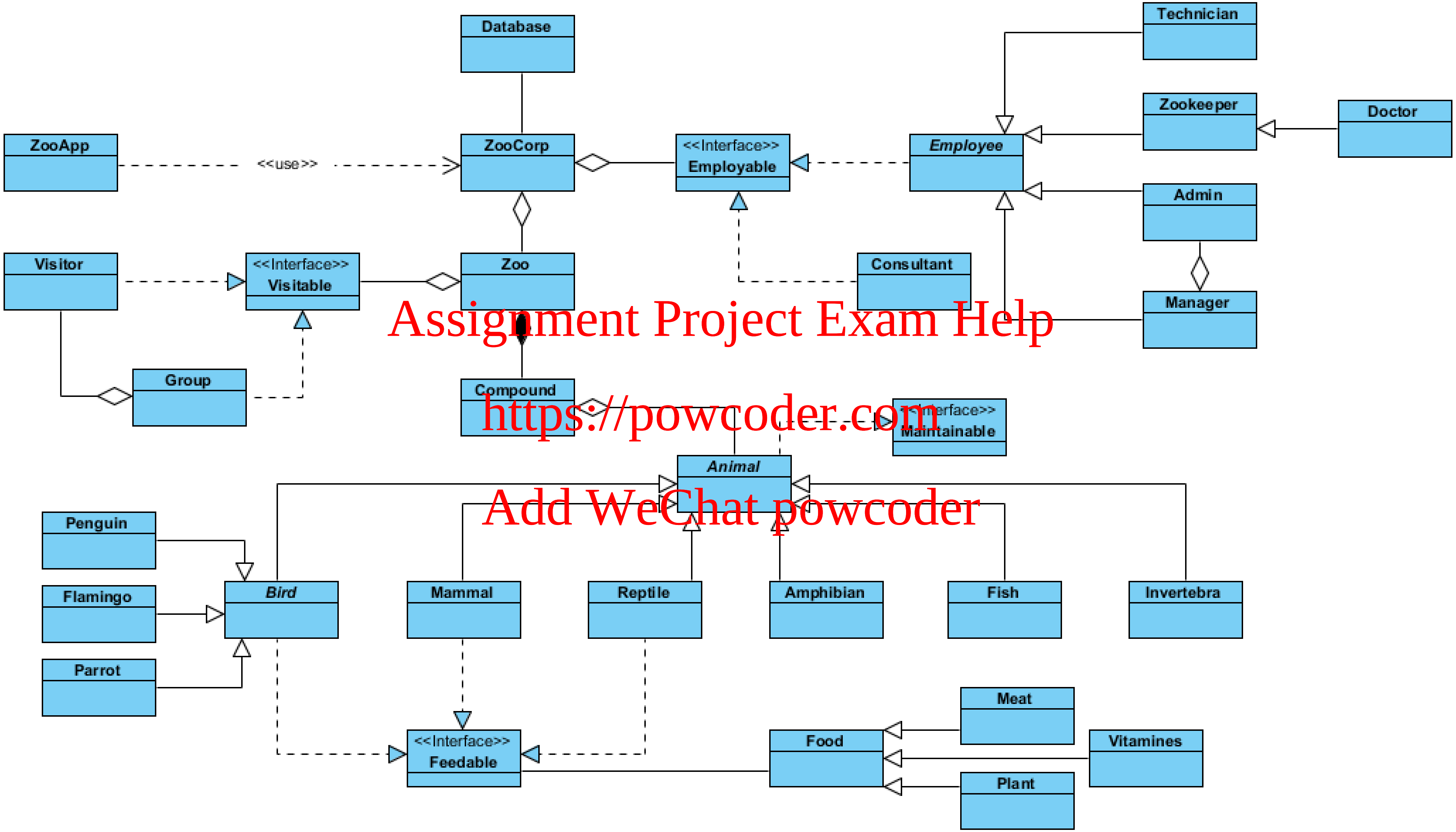
Abstract Methods and Classes

- Abstract subclass



Interfaces

- What is the difference between an abstract class and an interface?
  - Java abstract class
    - Can have instance methods that implement a default behaviour
    - May contain non-final variables
  - Java interfaces
    - Methods are implicitly abstract and cannot have implementations
    - Variables declared are by default final


Database
ZooApp
<<use>>
ZooCorp
<<Interface>>
Employable
Employee
Technician
Zookeeper
Doctor
Admin
Manager
Consultant
Visitor
<<Interface>>
Visitable
Zoo
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: http://stackoverflow.com/questions/1913098/what-is-the-difference-between-an-interface-and-abstract-class

Interfaces

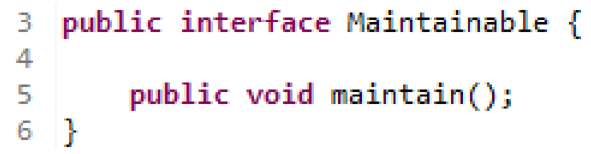
</>

- 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

Interfaces

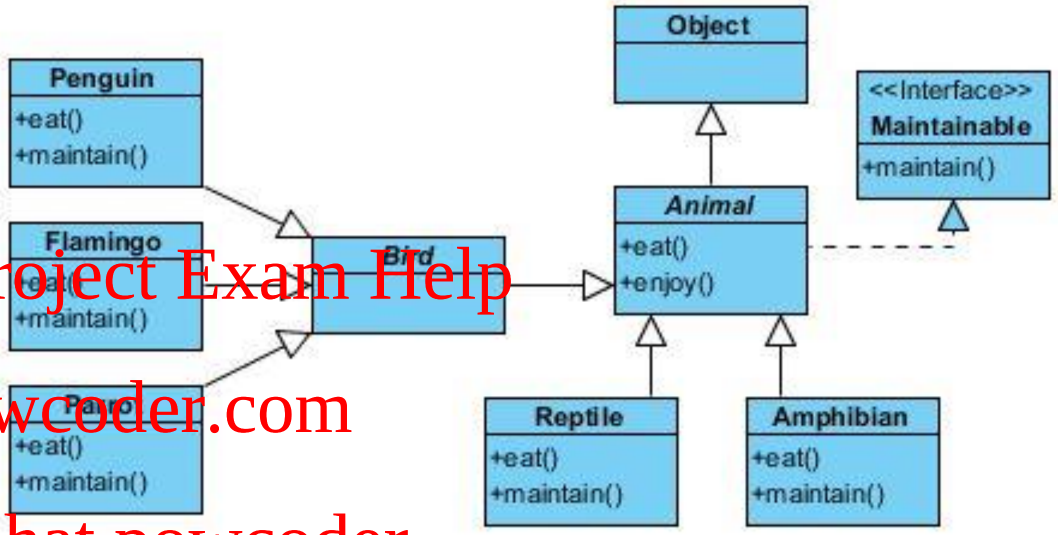
- Some rules:
  - Use the keyword "interface" instead of "class" to declare an interface
  - Implement an interface with the "implements" keyword
  - Because interfaces have no state and are only about method actions, using an action name (ending in "able") is often appropriate
  - A class that implements an interface must provide implementations for all the methods in the interface
  - Similar to classes, you can build up inheritance hierarchies of interfaces by using the "extends" keyword

Interfaces

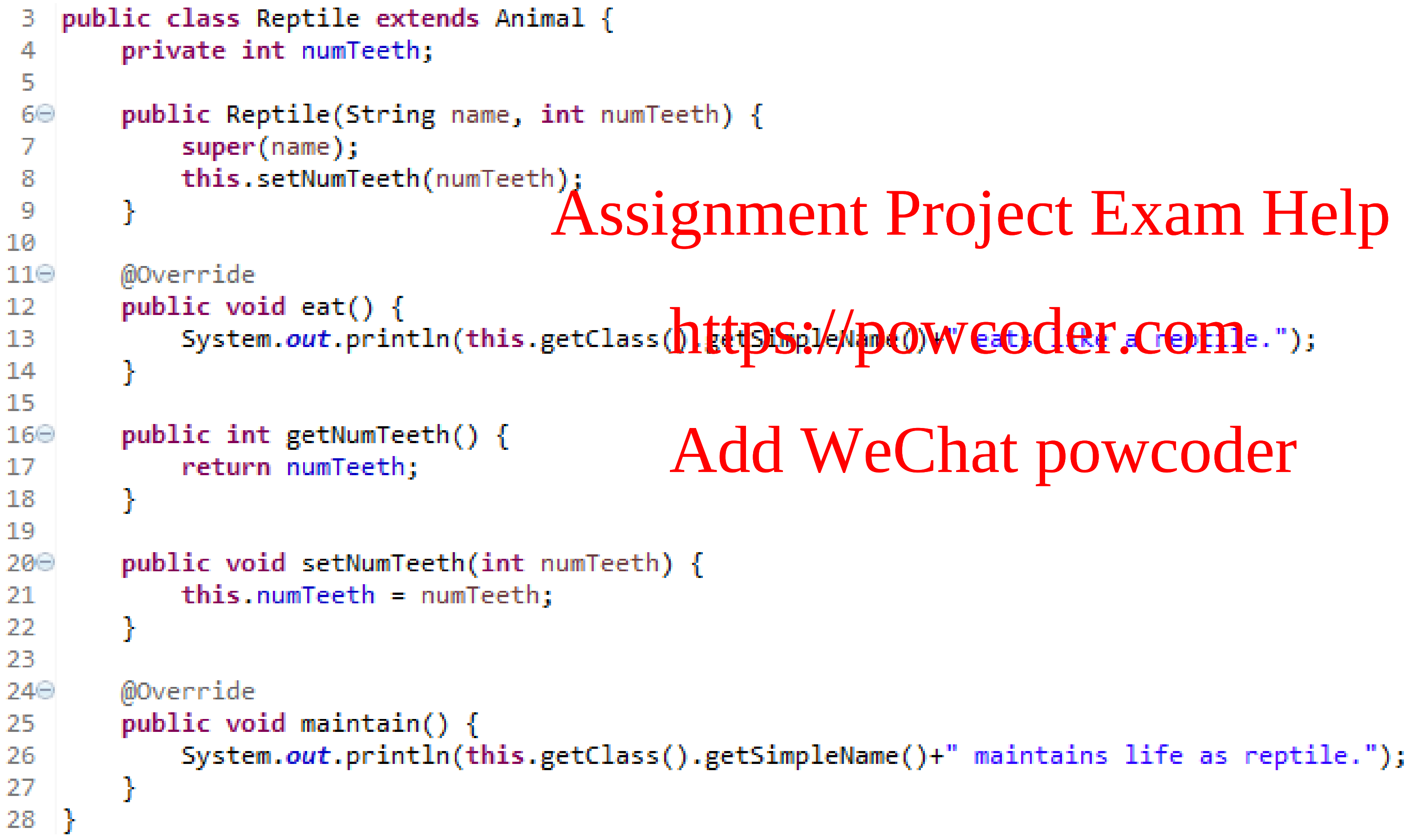


```java
public interface Maintainable {

    public void maintain();
}
```

```java
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()+" enjoys life as animal.");
    }

    public String getName() {
        return name;
    }

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

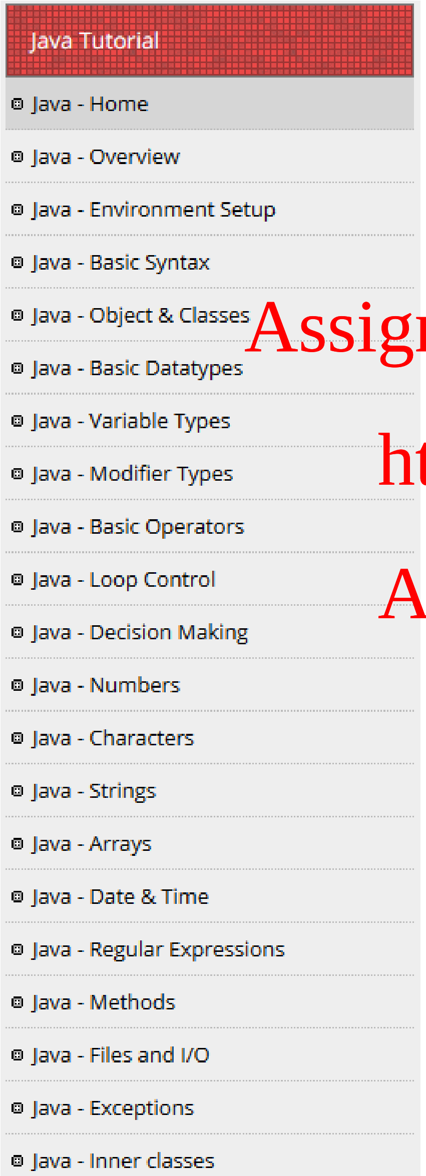


Interfaces



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

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

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

    public int getNumTeeth() {
        return numTeeth;
    }

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

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

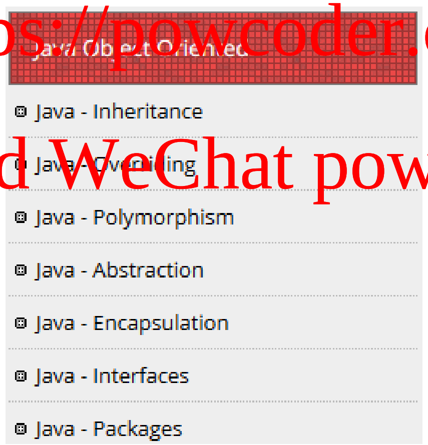
Useful Website



Java Tutorial

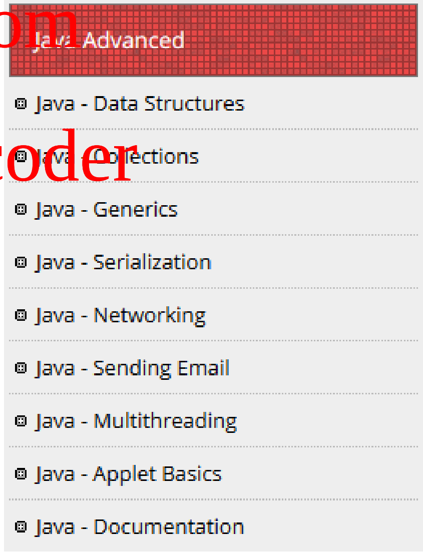
- Java - Home
- Java - Overview
- Java - Environment Setup
- Java - Basic Syntax
- Java - Object & Classes
- Java - Basic Datatypes
- Java - Variable Types
- Java - Modifier Types
- Java - Basic Operators
- Java - Loop Control
- Java - Decision Making
- Java - Numbers
- Java - Characters
- Java - Strings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

https://www.tutorialspoint.com/java/



Java Object Oriented

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



Java Advanced

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

And finally ...

</>

Acknowledgement

- Slides based on material from
  - Bill Leahy's lecture slides
    - http://www.cc.gatech.edu/~bleahy/xjava/cs1311xjava05_poly.ppt
  - Maria Litvin's & Gary Litvin's book slides
    - http://skylit.com/javamethods/ppt/Ch10.ppt
  - Marty Stepp's lecture slides
    - http://www.cs.washington.edu/331/
- and others ...