Object Oriented Programming with Java

Banu Prakash C

Objectives

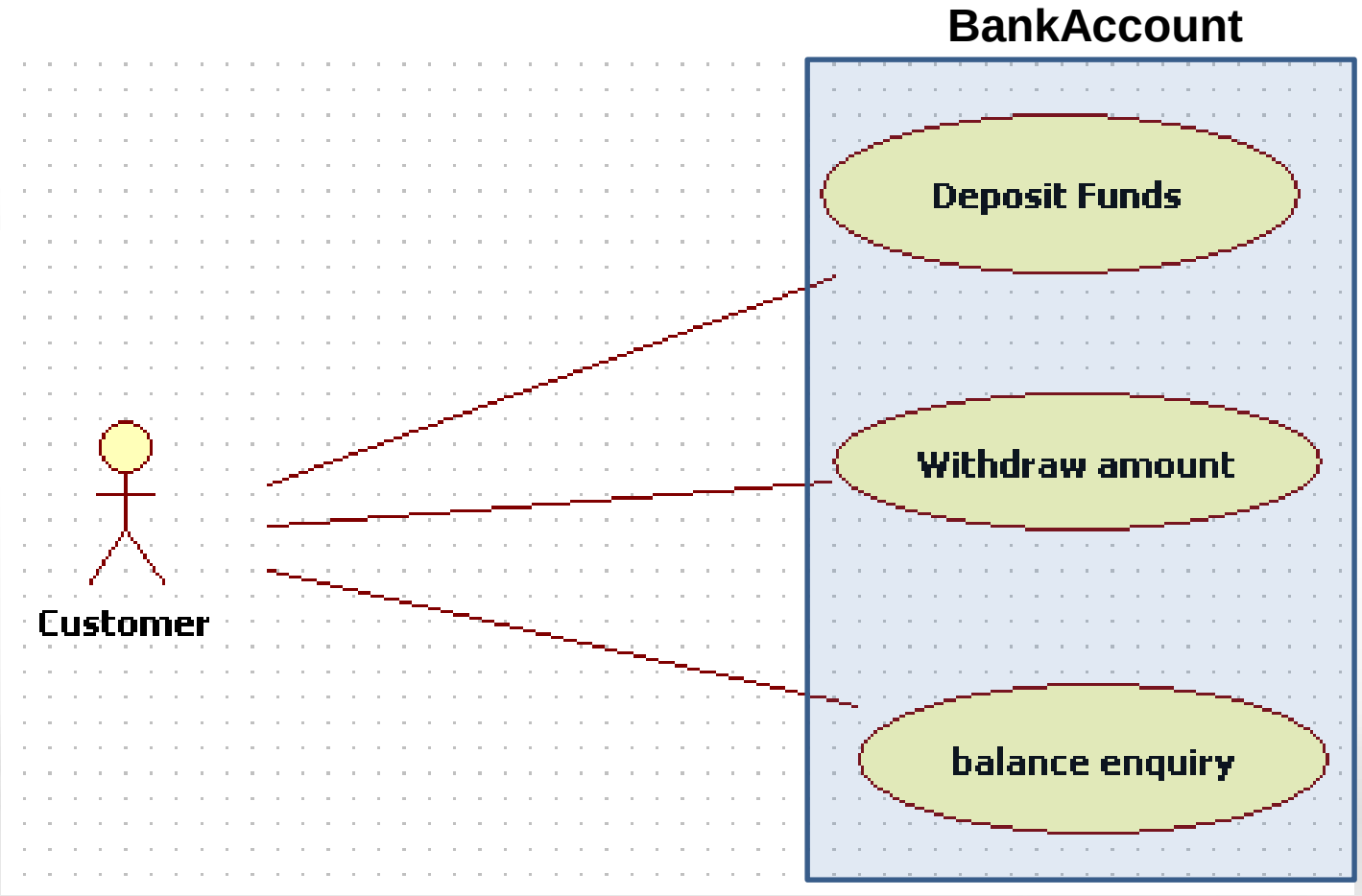
- Understand how to write OOP using Java.
- Understand the difference between instance variables/methods and class(static) variables/methods.
- Understand is-a and has-a relationship
- Understand realization using interface
- Loose coupling and high cohesion.

- Problem statement

Let us build a simple application to store account information using object oriented approach of java language.

All customers can deposit (i.e., credit) money into their BankAccount and withdraw (i.e., debit) money from their accounts. Also customers should be able to check balance status from his accounts.

Bank Account use cases



Designing the public interface of a Class

Method Definition

- access specifier (such as public)
- return type (such as String or void)
- method name (such as deposit)
- list of parameters (double amount for deposit)
- method body in { }

How should the methods look:

- public void deposit(double amount) { . . . }
- public void withdraw(double amount) { . . . }
- public double getBalance() { . . . }

Designing the Public Interface of a Class

Package declaration

Class declaration

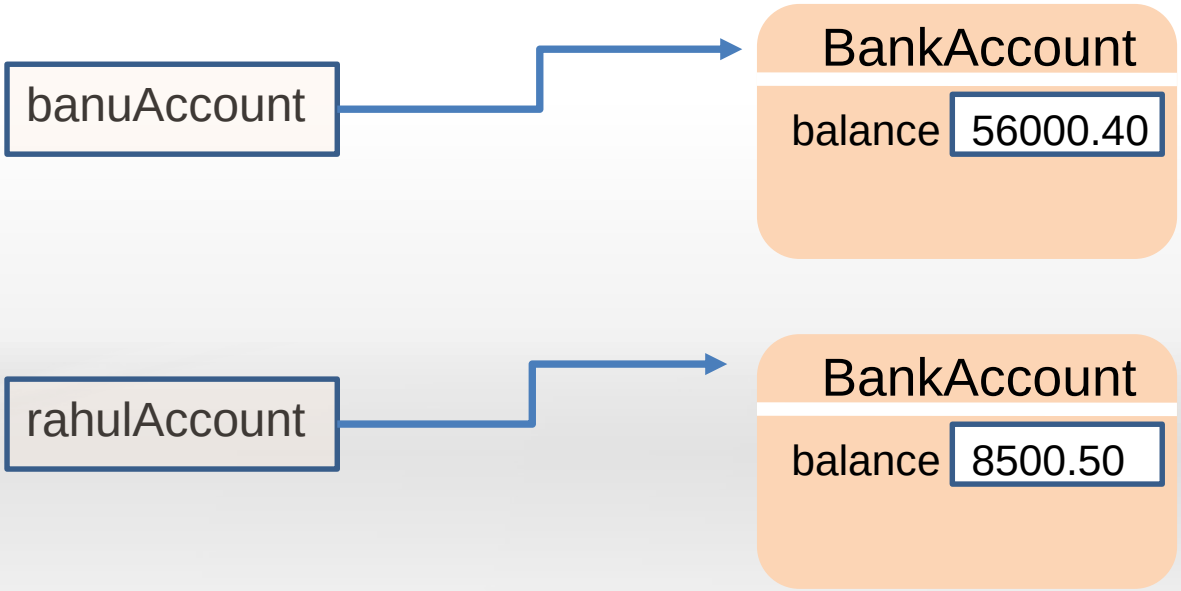
Methods:

Public interface of BankAccount

```
package com. banu    .entity;

/**
 * @author Banu Prakash
 *
 */
public class BankAccount {

    /**
     * @param amount
     */
    public void deposit(double amount) {

    }
    /**
     * @param amount
     */
    public void withdraw(double amount) {

    }
    /**
     * @return
     */
    public double getBalance() {
        return 0.0;
    }
}
```

State of Bank Account

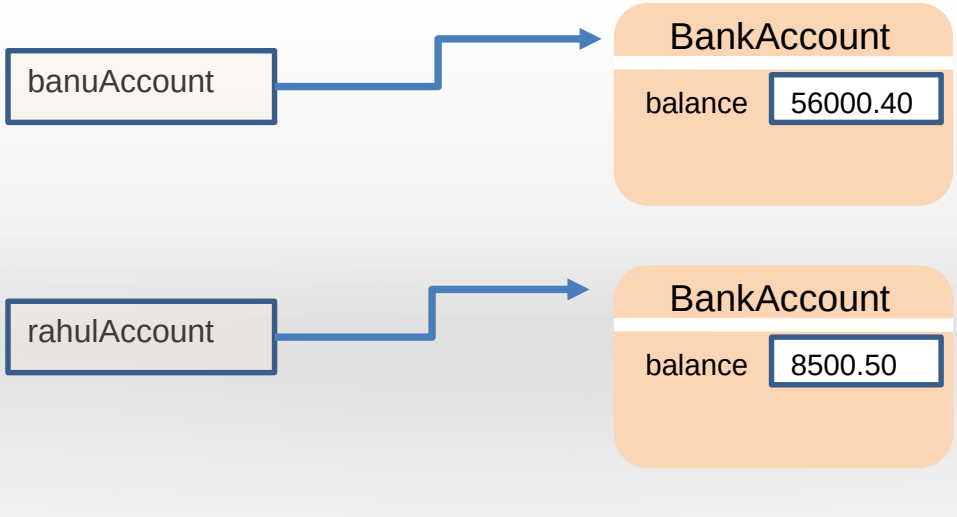
- Now how can we have the balance information of every BankAccount.
- BankAccount should contain a balance instance field to represent the balance of different accounts.



State of BankAccount

- An instance field declaration consists of the following parts:
  - access specifier (such as private)
  - type of variable (such as double)
  - name of variable (such as balance)
- Each object of a class has its own set of instance fields

```
public class BankAccount {
    private double balance;
    // Remaining code
}
```



Modifying/Accessing state of BankAccount

The deposit method of the BankAccount class can access the private instance field:

```
public class BankAccount {
    private double balance;
    public void deposit(double amount) {
      balance = balance + amount;
    }
}
```

```
public class Bank  {
    public static void main(String[] args)   {
      BankAccount rahulAcc = new BankAccount();
      rahulAcc.balance = 1000; // ERROR
      rahulAcc.deposit(1000); // VALID
    }
}
```

Modifying/Accessing state of BankAccount

Implicit and Explicit Method Parameters

- The implicit parameter of a method is the object on which the method is invoked.
- The this reference denotes the implicit parameter

```
public void deposit(double amount) {
        balance = balance + amount;
}
```

Stack

index		parameter
0	reference	this
1	double	500

The above method is converted to

```
public void deposit(double amount) {
        this.balance = this.balance + amount;
}
```

balance is the balance of the object to the left of the dot:
rahulAcc.deposit (500);

Example: Object creation and calling methods

- Video: java_fundamentals_1.swf
  - Illustrates how to create Java project in eclipse.
  - Also illustrates how to create objects and modify / access the state of an object.
- Code Sample: Java_fundamentals_1.zip
  - Contains BankAccount.java and BankAccountExample.java
  - Illustrates how to create objects and modify / access the state of an object.

Constructors

- Constructors are used to initialize the instance variables (fields) of an object.
- When you create a new instance (a new object) of a class using the new keyword, a *constructor* for that class is called.
- **Constructor name is class name**. A Constructor name should be same as that of class.

Creating instances of BankAccount

- BankAccount rahulAcc = new BankAccount();
  - It looks like we are calling a method BankAccount() because of parentheses
  - No, we are calling the BankAccount constructor
- A constructor looks a lot like a method, but it's not a method.
  - It's got the code that runs when you use "new" [when you instantiate an object].

Creating instances of BankAccount

- But where is the Constructor ?. If we didn't write it, who did?
  - The Compiler writes it for you!

The Default Constructor looks like:

```
public BankAccount () {
    Initialization code can be placed here
}
```

Note :

- Same name as that of class.
- No explicit return type.

Constructors

- Imagine if you want need a computer table.
  - Option 1:
    - Ask carpenter to create a Computer table for you

```
public class ComputerTable {
        public ComputerTable( ) {
            Default size for table is considered.
        }
}
```

  - Option 2:
    - Specify that you need an 3 feet width, 4 feet breadth and 2.5 feet length table

```
public class ComputerTable {
    public ComputerTable(double width, double breadth, double length) {
            Explicitly you have specified the
            dimension.
    }
}
```

Creating instances of BankAccount with initial balance

- Generally when an account is created it should have some initial balance.

- Again the best place to put initialization code is in the constructor.

```
public BankAccount (double initialAmount ) {
        this.balance = initialAmount;
}
```

And while creating an object call:

```
BankAccount  rahulAcc = new BankAccount(5000);
```

Answer this?

- What is the output of the following program?

```
class Circle {
    double radius;
    Circle(double radius) {
      this.radius = radius;
    }
    public double getRadius() {
      return radius;
    }
}
public class Tester {
    public static void main(String[] args) {
                Circle circle = new Circle();
                System.out.println(circle.getRadius());
    }
}
```

Answer this

- Imagine you also need to store the owner of the bank account and also for every bank account created we need to assign an account number?
  - What are the extra fields required?
  - Which part of the BankAccount class should have the above initialization code?

Answer this

- Can you design a class for the given specification?
  - We need to create instances of Rectangle.
  - Every rectangle has a width and breadth.
  - Every rectangle object created should be initialized to width=1 and breadth =1 if dimension is not specified.
  - Also provision should be there to explicitly mention the initial dimension of rectangle (example: 4x5 rectangle)
  - Every rectangle's public interface should allow as to change its width and breadth.
  - Also we should be able to know the width, breadth and area of the rectangle. Requirement is we may need only one at a given point of time.

Object references

- The new operation instantiates an object of a particular class, and returns a reference to it. This reference is a handle to the location where the object resides in memory.

```
SomeObject aRef = new SomeObject();
```

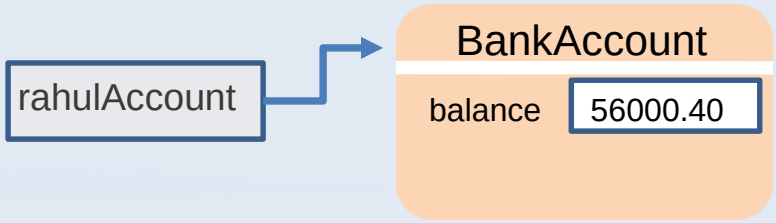
The reference is stored in the variable "aRef."

- Second, you can obtain a reference by assignment. The reference to an existing reference obtained by assignment is just a duplicate reference to the object (and not to a copy of the object).

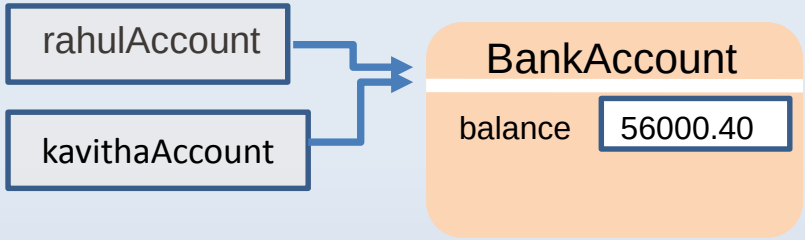
```
SomeObject anotherRef = aRef;
```

Referencing BankAccounts

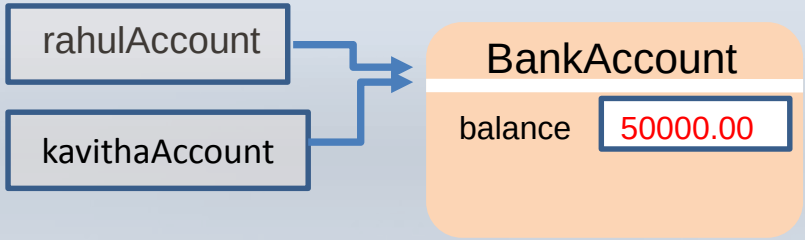
```
BankAccount rahulAccount =
        new BankAccount(56000.40);
```




```
BankAccount  kavithaAccount =
                rahulAccount;
```



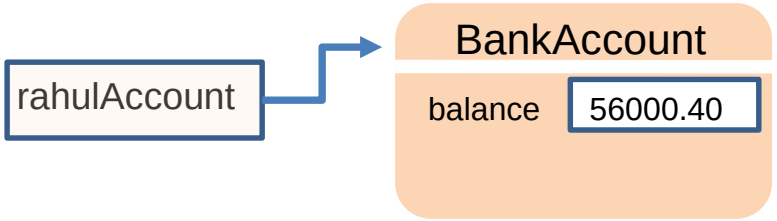

```
kavithaAccount.withdraw(6000.40);
```



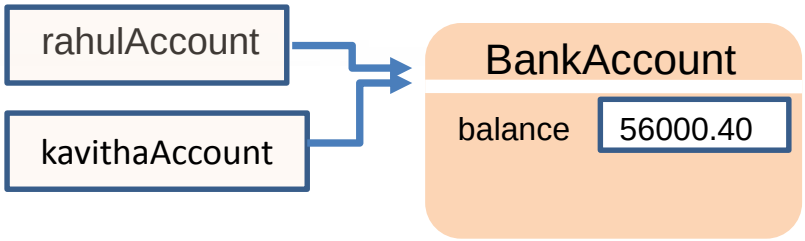
rahulAccount and kavithaAccount balance will be 50000.00

Comparing BankAccounts

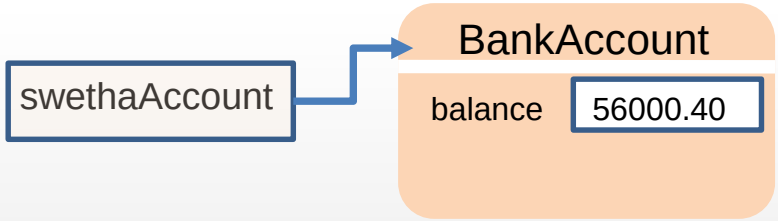
```
BankAccount rahulAccount =
        new BankAccount(56000.40);
```




```
BankAccount  kavithaAccount =
                rahulAccount;

    rahulAccount == kavithaAccount
```




```
BankAccount swethaAccount =
        new BankAccount(56000.40);
    rahulAccount != swethaAccount
    rahulAccount.equals(swethaAccount)
```



Note: rahulAccount and kavithaAccount are one and the same.
swethaAccount and rahulAccount contain same balance, but they are two different accounts

Comparing BankAccounts

- Override equals method in BankAccount.

```
/* (non-Javadoc)
 * @see java.lang.Object#equals(java.lang.Object)
 */
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    BankAccount other = (BankAccount) obj;
    if (accountNumber == null) {
        if (other.accountNumber != null)
            return false;
    } else if (!accountNumber.equals(other.accountNumber))
        return false;
    return true;
}
```

If both the references are same, naturally the contents are same

Objects being compared should belong to same type.

Example

- Video: java_fundamentals_2.swf
  - Example illustrates difference how to use IDE to generate getters/ setters, constructors and also illustrates between equals and == operator.
- Code Sample: java_fundamentals_2.zip
  - BankAccount.java and BankAccountExample.java
  - Illustrates difference between equals and == .

Static members

- Static variables
  - Variables that are common to all objects of class.
- Static methods
  - Methods that can be invoked without an instance of a class.

Static variables

- Fields that have the static modifier in their declaration are called *static fields* or *class variables*.
- They are associated with the class, rather than with any object.
- Every instance of the class shares a class variable, which is in one fixed location in memory.
- Any object can change the value of a class variable
- Class variables can also be manipulated without creating an instance of the class.

Static methods

- Static methods, which have the static modifier in their declarations
- Static methods can be invoked without creating an instance of the class.
- Static methods should be invoked with the class name.

  ```
  ClassName.methodName(arguments);
  ```

- A common use for static methods is to access static fields.

BankAccount Static Members

- Imagine you wanted to count how many BankAccount instances are being created while your program is running.

```
public class BankAccount {
    private int count = 0;
    public BankAccount (){
        count ++;
    }
}
```

This would always set count to 1 each time a BankAccount is made

```
public class BankAccount {
    private static int count = 0;
    public BankAccount (){
        count ++;
    }
}
```

Static variable is initialized ONLY when the class is loaded, NOT each time a new instance is made.

- Static Variables: Value is same for ALL instances of the class.

BankAccount static members.

- To know how many accounts are created, we do not need to have any particular account reference.

- The “behavior not dependent on any particular instance”.

int totalAccounts =
BankAccount.getCount();

```
public class BankAccount {
    private static int count = 0;
    public BankAccount (){
          count ++;
    }
    public static int getCount( ) {
          return count;
    }
}
```

Array of BankAccount

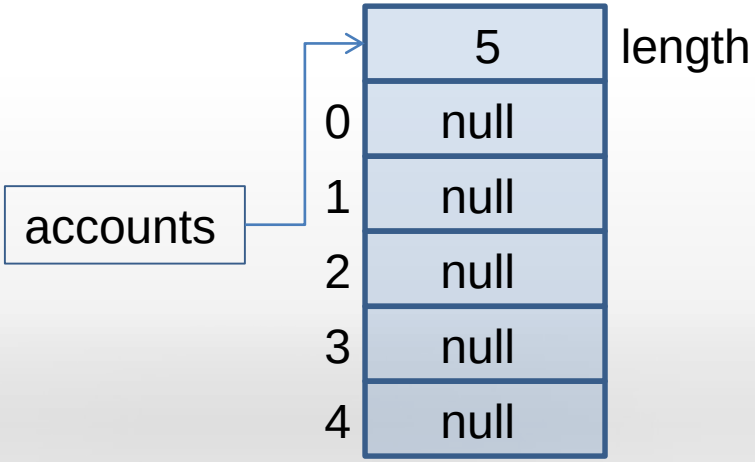
- Bank has to hold reference to many BankAccounts.
- Instead of having references named rahulAccount, swethaAccount, kavithaAccount, and so on.
- Declare a single reference variable to hold all bank accounts and each account identified by their index position.

Declare a BankAccount array variable

```
BankAccount[ ] accounts;
```

Specify that we need to store 5 accounts.

```
accounts = new BankAccount[5];
```

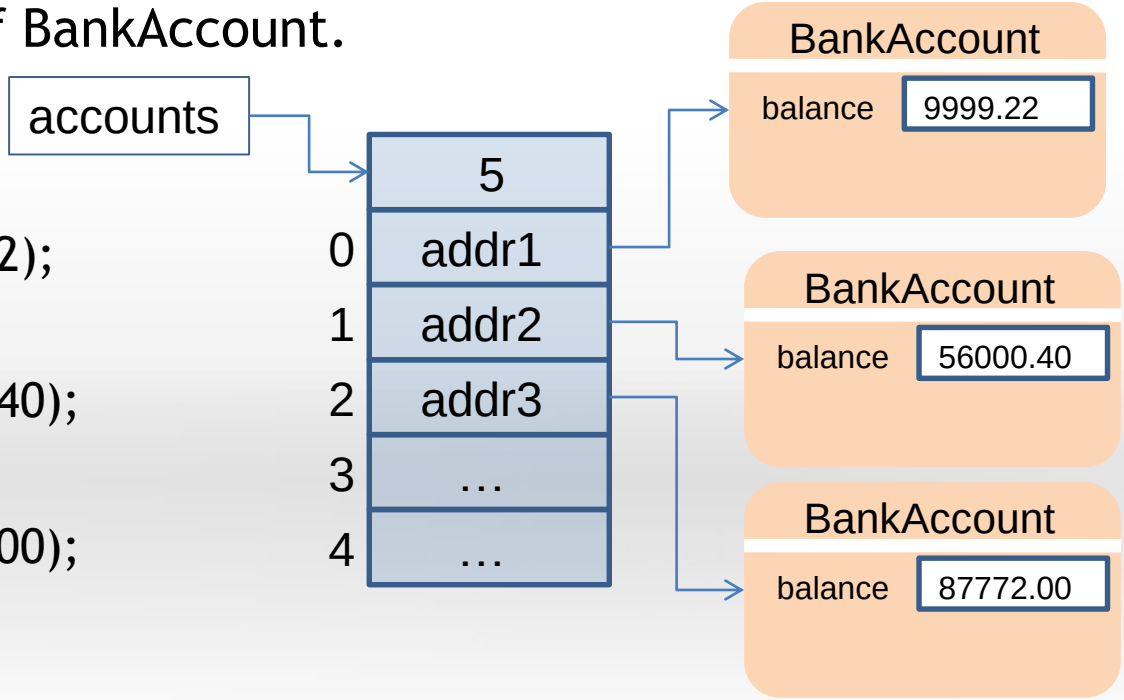


Array of BankAccount

- What's missing?
- We have array of BankAccounts, but no actual account instances.

- Create new instances of BankAccount.

```
accounts[0] =
    new BankAccount(9999.22);
accounts[1] =
    new BankAccount(56000.40);
accounts[2] =
    new BankAccount(87772.00);
```

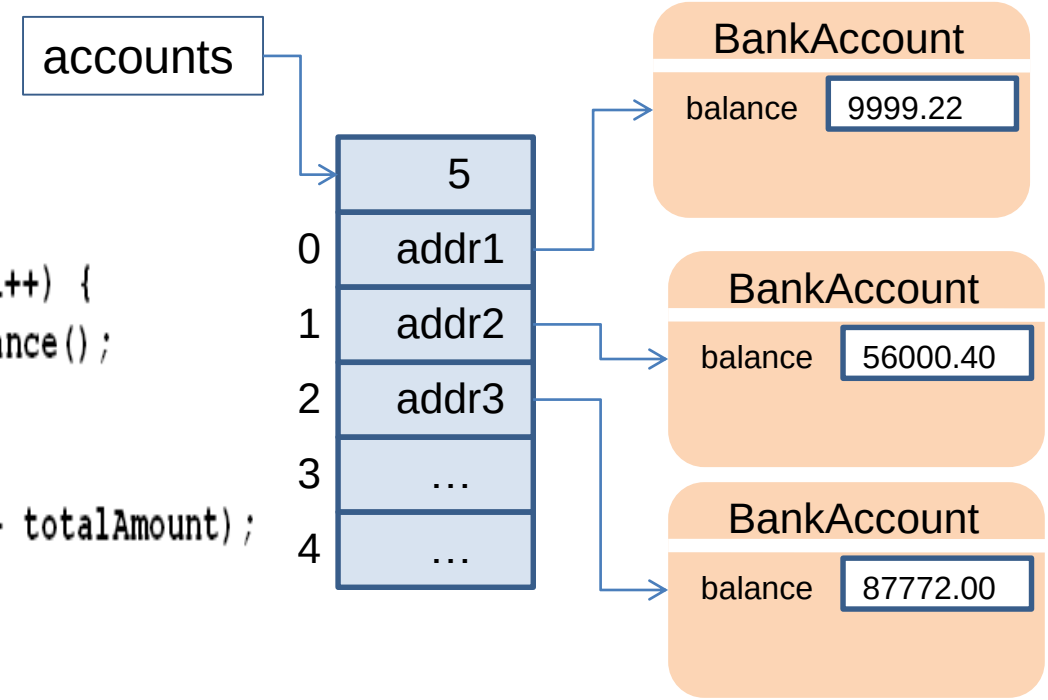


Array of BankAccount

- We need to get the total amount available in the Bank. i.e. the sum of balance of all the accounts

- Traverse through all the accounts and get its current balance.

```
double totalAmount  = 0.0;

for (int i = 0; i < accounts.length; i++) {
    totalAmount += accounts[i].getBalance();
}

System.out.println("Total Amount : " + totalAmount);
```



Example

- Code Sample: Java_fundamentals_3.zip
  - Rectangle.java and RectangleArrayExample.java
  - Illustrates creating array of objects

Relationship between objects

- Association
  - Association represents the relationship shared among the objects of two classes.
  - **Aggregation**
    - Aggregation indicates a relationship between a whole and its parts.
    - *Aggregation* can occur when a class is a collection or container of other objects, but where the contained classes do not have a strong *life cycle dependency* on the container—essentially,
    - if the container is destroyed, its contents are not destroyed.
  - **Composition**
    - Composition is a strong form of aggregation.
    - In this kind of relationship, each part may belong to only one whole.
    - The part is not shared with any other whole.
    - In a composition relationship, when the whole is destroyed, its parts are destroyed as well.

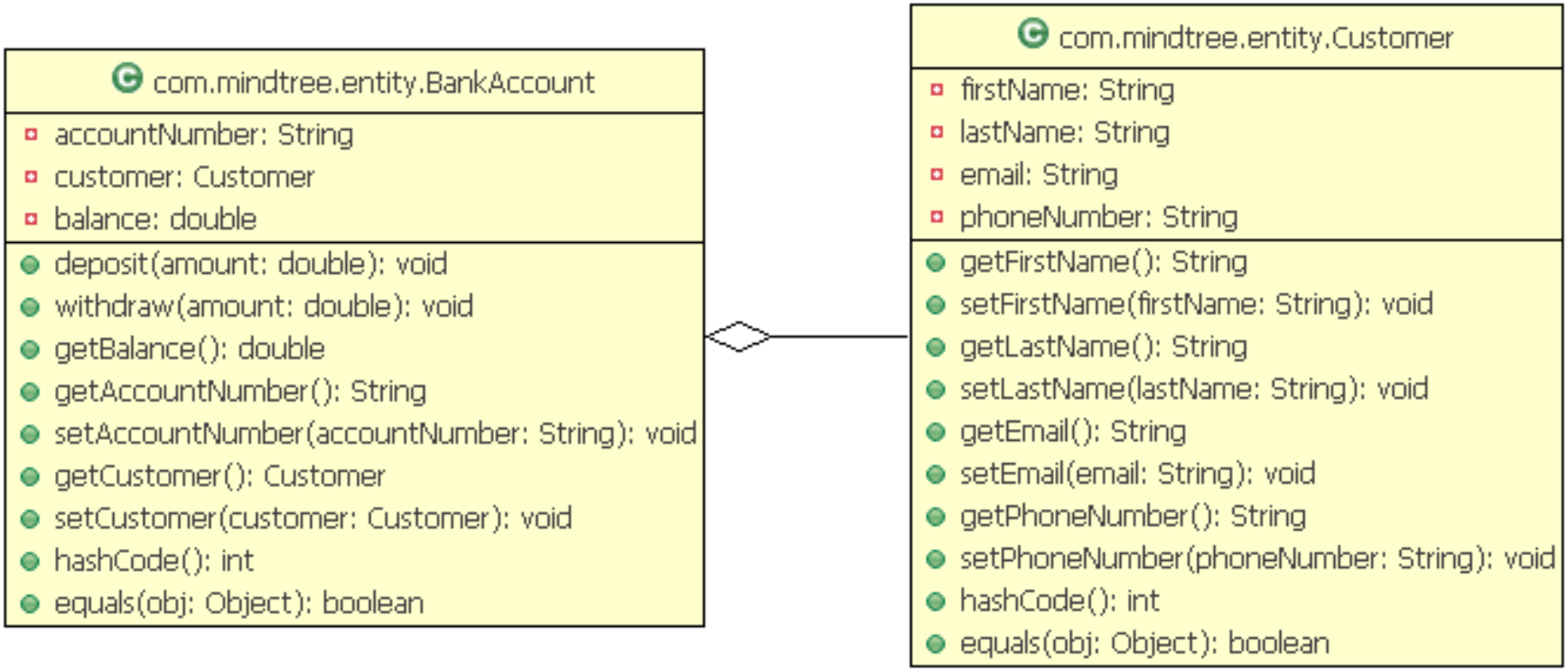
Relationship between objects

- Inheritance (is-A Relationship)
  - The Generalization relationship indicates that one of the two related classes (the *subclass*) is considered to be a specialized form of the other (the *super type*).
  - superclass is considered as '***Generalization***' of subclass
  - Generalization-Specialization relationship
    - A is a type of B
    - Example.
      - "an oak is a type of tree",
      - "an automobile is a type of vehicle“
      - “a human is a mammal”
      - “a mammal is an animal”.

BankAccount belongs to a Customer

- Every BankAccount has a owner, who happens to be a customer of bank.
  - Assumption made is BankAccount does not allow joint account owners.
- Each customer has attributes like firstName, lastName, email and phone numbers.

Aggregation



UML representing an aggregation relationship between BankAccount and Customer

BankAccount and Customer entities

```java
/**
 * @author Banu Prakash
 * © 2011 MindTree Limited.
 */
public class BankAccount {

    private String accountNumber;
    private Customer owner;
    private double balance;

    // remaining code

    /**
     * @return the owner
     */
    public Customer getOwner() {
        return owner;
    }
    /**
     * @param owner the owner to set
     */
    public void setOwner(Customer owner) {
        this.owner = owner;
    }
```

```java
/**
 * @author Banu Prakash
 * © 2011 MindTree Limited.
 */
public class Customer {
    private String firstName;
    private String lastName;
    private String email;
    private String phoneNumber;


    /**
     * @return the firstName
     */
    public String getFirstName() {
        return firstName;
    }
    /**
     * @param firstName the firstName to set
     */
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    // remaining code
```

Code Snippet to assign an Customer to BankAccount

```
/*
 * Instantiate Customer
 * arguments are firstName, lastName, email and phone number
 */
Customer owner = new Customer("Raj", "Kumar", "raj_kumar@mindtree.com", "9833909876");

/*
 * Instantiate BankAccount
 * arguments are account number, owner and intial amount
 */
BankAccount account = new BankAccount("SB122",owner,5000.50);
// Owner of account

Customer accountOwner = account.getOwner();


System.out.println("Owner of Account :" + accountOwner.getFirstName());
```

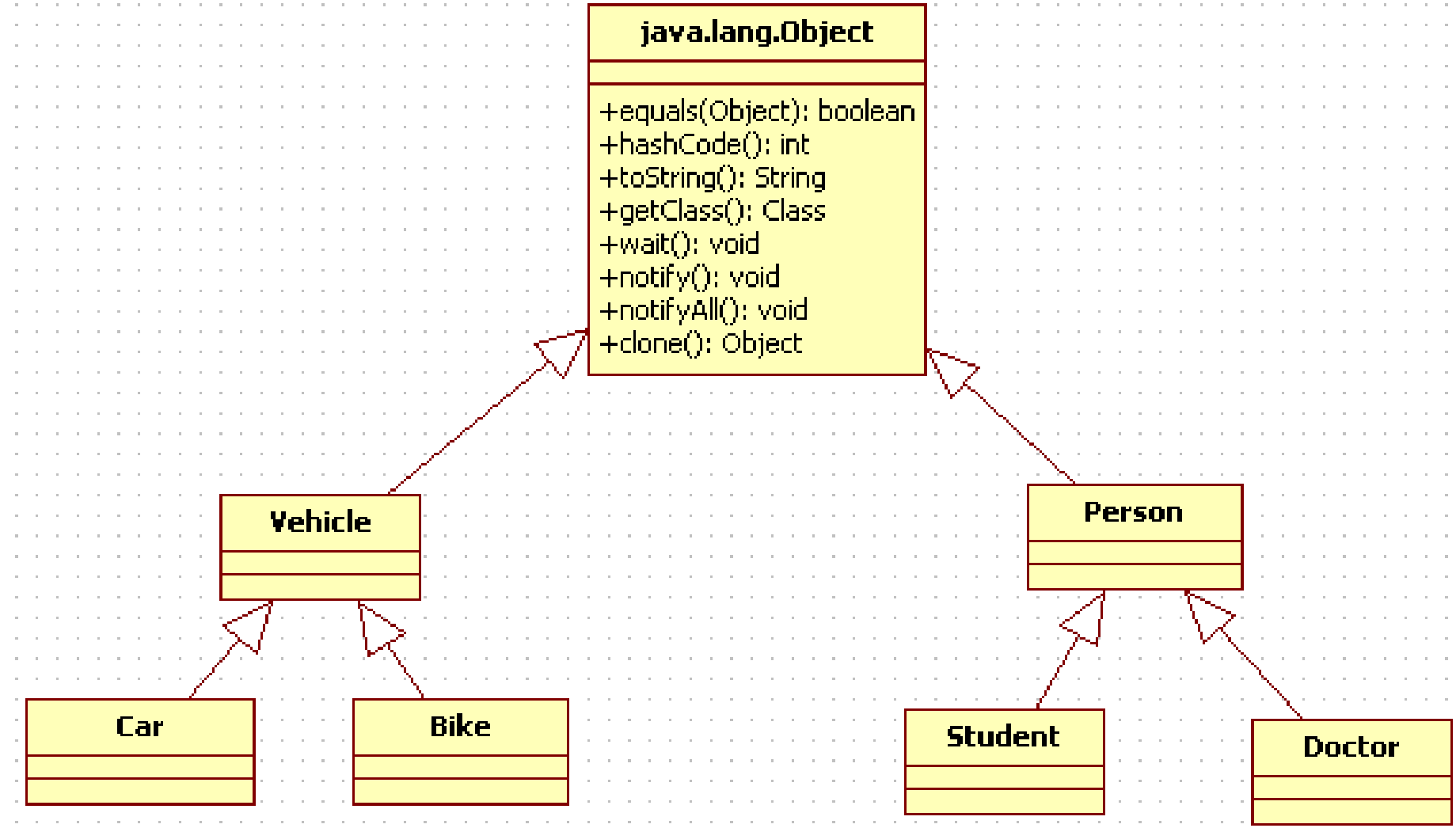
Generalization

- Inheritance is implemented in Java by the *extends* keyword during class declaration

Example:

```
public class Student extends Person {
        // Define additional attributes that make a Person into a
        // Student
}
```

Class Object is the root of the every class hierarchy.



Answer this

```
public class Customer {
        String name;
        public Customer(String name) {
            this.name = name;
        }
        public String getName() {
                return name;
        }
}
public class Test {
     public static void main(String[ ] args) {
                Customer c1 = new Customer("A");
                Customer c2 = new Customer("B");
                boolean eq = c1.equals(c2);          // is this a valid statement,
     }                                               // if yes what is the  the value of
                                                     //"eq"
}
```

Generalization of BankAccount

- More specific types of accounts also exist.
- Savings accounts, for instance earn interest on the money they hold.
  - SavingsAccount should provide a public member function calculateInterest that returns a double indicating the amount of interest earned by an account
- Checking accounts, on the other hand, charge a fee per transaction (i.e., credit or debit).
  - CheckingAccount should redefine member functions credit and debit so that they subtract the fee from the account balance whenever either transaction is performed successfully.
  - CheckingAccount's versions of these functions should invoke the base-class Account version to perform the updates to an account balance

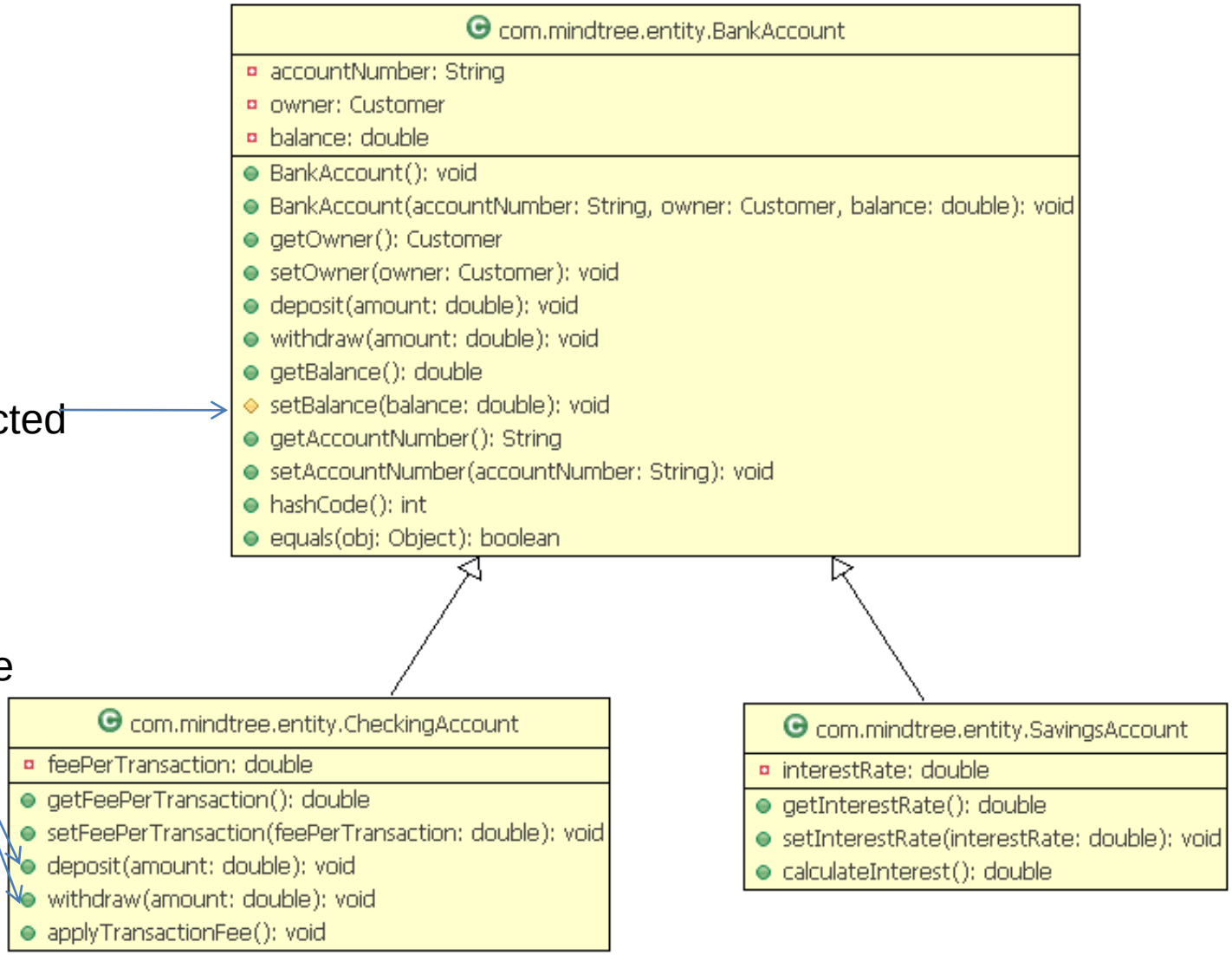
BankAccount hierarchy

Generalization

Specialization

protected

Override



Method overriding

- ***Method Overriding*** allows a subclass to redefine methods of the same signature from the superclass.
- The key benefit of overriding is the ability to define/defer behavior specific to subclasses.
- An overridden method must have:
  - The same name
  - The same number of parameters and types
  - The same return type

Code Snippet: Override and overload

```java
class Person {
    private String name;
    private int age;
    public void set(String name) {
        this.name = name;
    }
    public void set(int age) {
        this.age = age;
    }
    public String getName() {
         return name;
    }
    public int getAge() {
        return age;
    }
}
```

```java
class Doctor extends Person {
    private String specialization;
    public void set(String name, int age,
                    String specialization) {
        set(name);
        set(age);
        this.specialization = specialization;
    }
    public String getName() {
        return "Dr." + super.getName();
    }

    public String getSpecialization() {
        return specialization;
    }
}
```

```java
Doctor doctor = new Doctor();
doctor.set("Dang", 32, "Pediatrician");
System.out.println(doctor.getName());
```

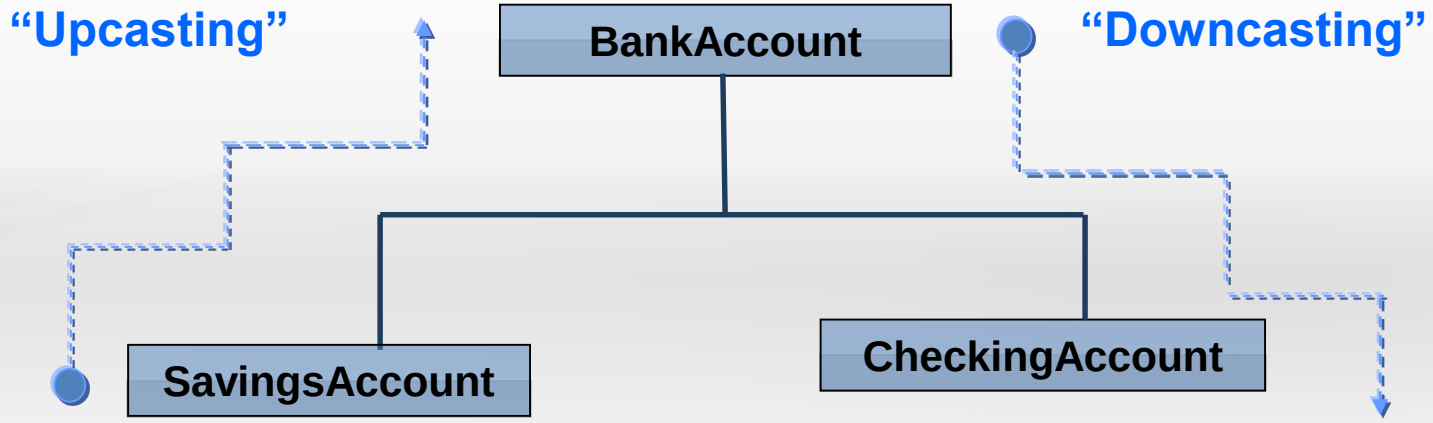
Reference Casting

- To upcast a CheckingAccount object, all you need to do is assign the object to a reference variable of type BankAccount.

  BankAccount account = new CheckingAccount(); //upcasting

- *CheckingAccount checkingAcc =*

  (CheckingAccount)account; // downcasting



What messages BankAccount can send?

[redacted].BankAccount

- getOwner(): Customer
- setOwner(owner: Customer): void
- deposit(amount: double): void
- withdraw(amount: double): void
- getBalance(): double
- setBalance(balance: double): void
- getAccountNumber(): String
- setAccountNumber(accountNumber: String): void
- hashCode(): int
- equals(obj: Object): boolean

What messages BankAccount can send?

```
BankAccount account  = new SavingsAccount();

account.deposit(5000.55); // valid

account.setInterestRate(10.35); // not valid
```

- SavingsAccount contains the method
  `public void setInterestRate(double interestRate);`
- But the reference used is of type BankAccount.
  - BankAccount class can send messages to methods declared only in BankAccount.
  - It has no clue about what the extra functionalities added by the specialized class.

What messages SavingsAccount can send?

[redacted].BankAccount

- getOwner(): Customer
- setOwner(owner: Customer): void
- deposit(amount: double): void
- withdraw(amount: double): void
- getBalance(): double
- setBalance(balance: double): void
- getAccountNumber(): String
- setAccountNumber(accountNumber: String): void
- hashCode(): int
- equals(obj: Object): boolean

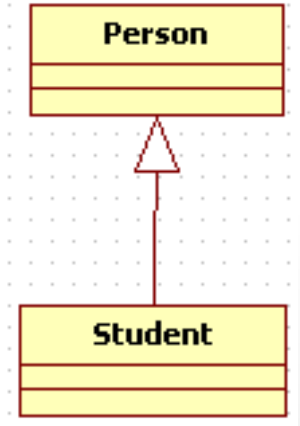
+

[redacted]SavingsAccount

- getInterestRate(): double
- setInterestRate(interestRate: double): void
- calculateInterest(): double

Answer this

- Given the relationship between entities as illustrated. Which of the following statements does not compile?



1. Student student = new Student ();
   Person person = student;
2. Student student = new Person();
3. Student person = (Student) new Person();
4. Person person = new Student( );

Answer this

What is the output of running the following code?

```
public class A {
        public A( ) {
                System.out.println("A Default Constructor");
        }
        public A(double d) {
                System.out.println("A param Constructor");
        }
}
public class  B extends A {
        public B() {
                System.out.println("B Default Constructor");
        }
        public B(int id, double d) {
                System.out.println("B Default Constructor");
        }
        public static void main(String[] args) {
                A obj = new B(5,222.22);
        }
}
```

Answer this

```
public class A {
        public void first(){
                System.out.println("First Method");
        }
        public  void second( ) {
                System.out.println("Second method");
        }
}
    public class  B extends A {
        public  void second(int data ) {  // overloading
                System.out.println("Second method with data");
        }
}
```

What is the output of running the following statements?

1)
```
A obj  = new B();
obj.second();
obj.second( 22 );
```

2)
```
B nobj = new B();
nobj.second();
nobj.second( 22 );
```

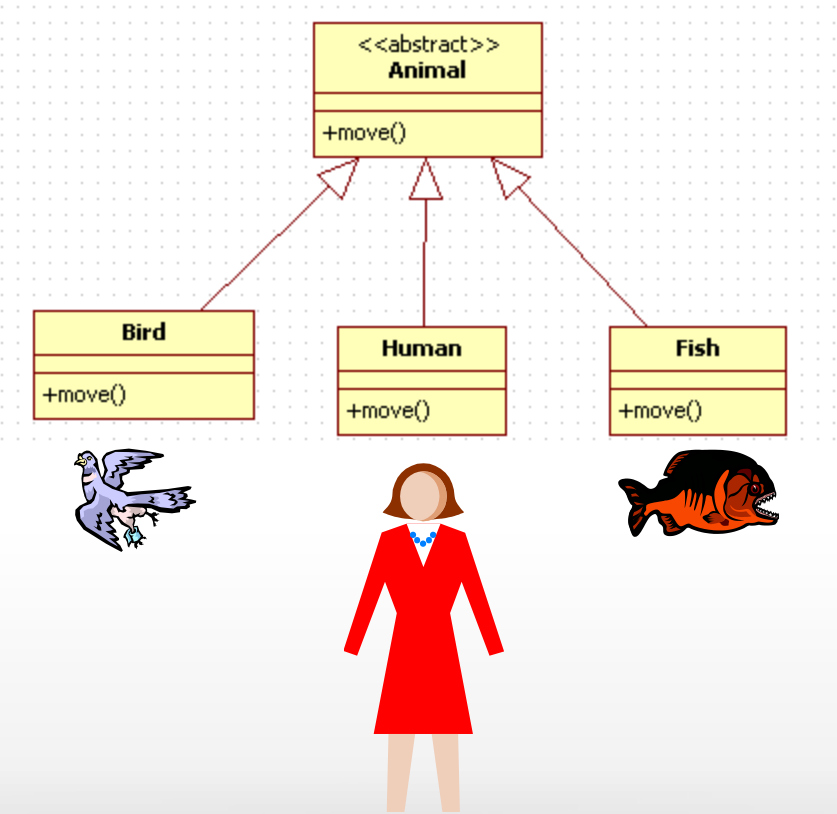
Polymorphism

- All java instance methods are ‘virtual’ and can be overridden by methods that belong to the sub class
- A virtual method is a method whose actual implementation is dynamically determined during runtime

Polymorphism

- Each subclass overrides the move()

```
Animal []  animals = new Animal[3];
animals[0] = new Bird();
animals[1] = new Person();
animals[2] = new Fish();
for (int i = 0; i < animals .length; i++)
            animals [i].move();
```



- References are all the same, but objects are not
- Method invoked is that associated with the Object,

  Not the one present in reference class.

Example

- Code Sample: Java_fundamentals_4.zip
  - Product.java, Tv.java, Mobile.java and ProductExample.java
  - Illustrates upcasting, downcasting and polymorphism.

Answer this

- Given :

```
class A {

}

class B extends A {

}
```

Question 1:

Is the below statement valid:

```
B obj = new A();
```

Answer this

- Given :

```
class A {
        public void test() {
                System.out.println("test method of A");
        }
}

class B extends A {
            public void test() {
                System.out.println("test method of B");
        }
           public void  best( ) {
                    System.out.println("best method of B");
           }
}
```

Question 2 : what is the output?

```
A obj = new B();        // 1
obj.test() ;            //2
obj.best() ;            //3
```

Answer this

Given :

```
class A {
    public static void test() {
        System.out.println("test method of A");
    }
}

class B extends A {
    public static void test() {
        System.out.println("test method of B");
    }

}
```

Question 2 : what is the output?

```
A obj = new B();        // 1
obj.test() ;            //2
```

Interface

- Java interfaces are for realization relationship.
- A Realization is a relationship between two elements, in which one element (the client) realizes the behavior that the other element (the supplier) specifies.

Interface

Signature of interface:

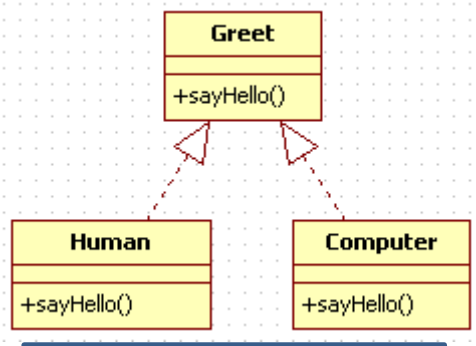
```
<modifier> interface interfaceName  {
        Can contain abstract
        methods and constants only
}
```

How class realizes an interface.

```
class className implements interfaceName {
        Should define methods
        declared in interface
}
```

Interface example

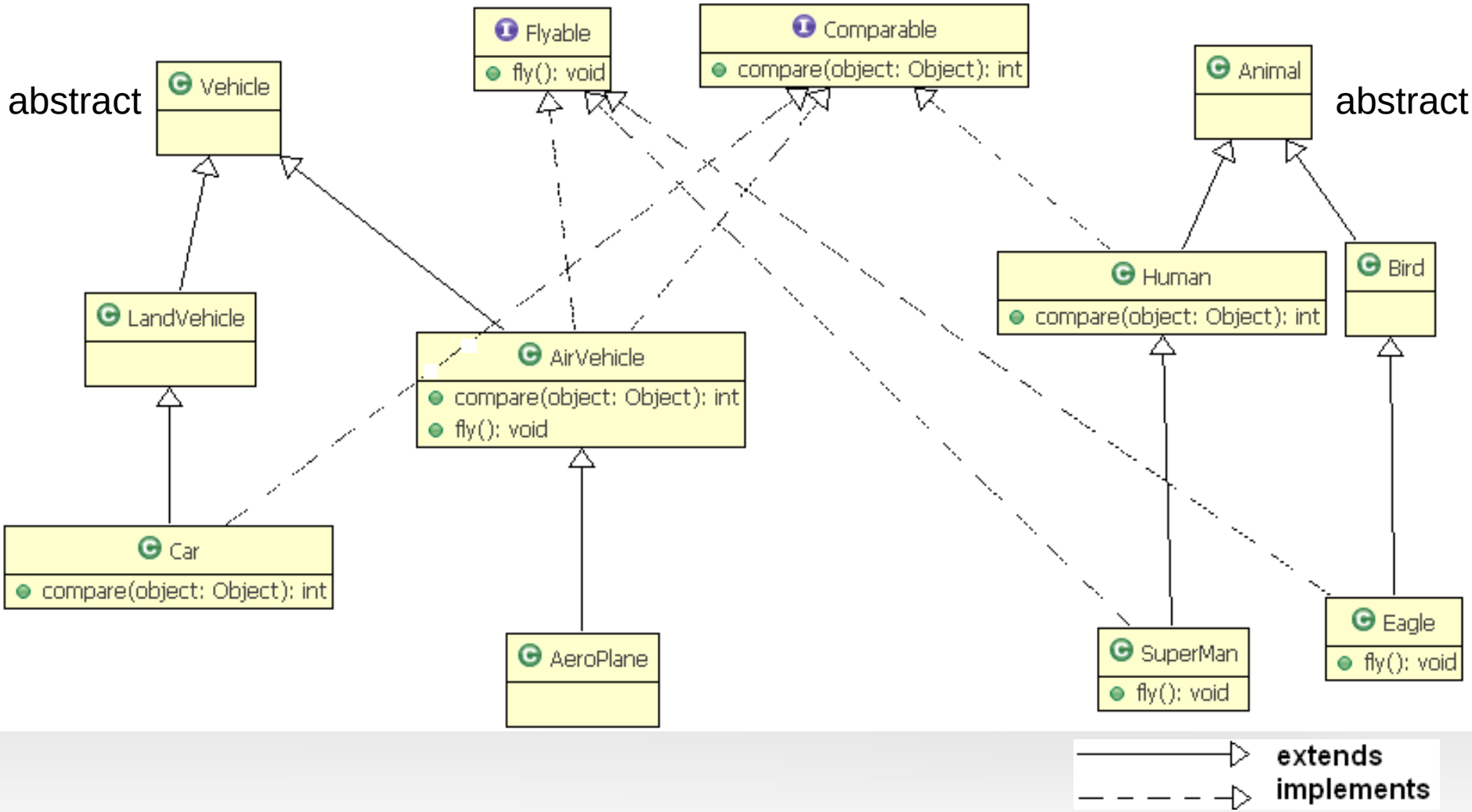
```
public interface Greet {
        public void sayHello();
}
public class Human implements Greet {
        public void sayHello() {
            Bow and say Hello Mr…
        }
}
public class Computer implements Greet {
        public void sayHello() {
            Display a PPT showing greeting
            screen.
        }
}
Greet greet = new Human();
greet.sayHello();     // Bow and say Hello

greet  = new Computer();
greet.sayHello();     // Display  a PPT showing greeting screen.
```



Both the classes Human and Computer realize Greet.

But the way they implement my differ.

Interface


abstract
Vehicle
Flyable
fly(): void
Comparable
compare(object: Object): int
Animal
abstract
LandVehicle
AirVehicle
compare(object: Object): int
fly(): void
Human
compare(object: Object): int
Bird
Car
compare(object: Object): int
AeroPlane
SuperMan
fly(): void
Eagle
fly(): void
extends
implements

Quiz

With reference to previous UML diagram, which of the following are valid statements?

1) Flyable f = new Flyable();

2) Comparable c = new AeroPlane();

3) Flyable f = new AirVehicle();
f.compare(anotherObject);

Answer this

- Given a method signature as

```
public void doTask(Comparable first, Comparable second) {
    // code here
}
```

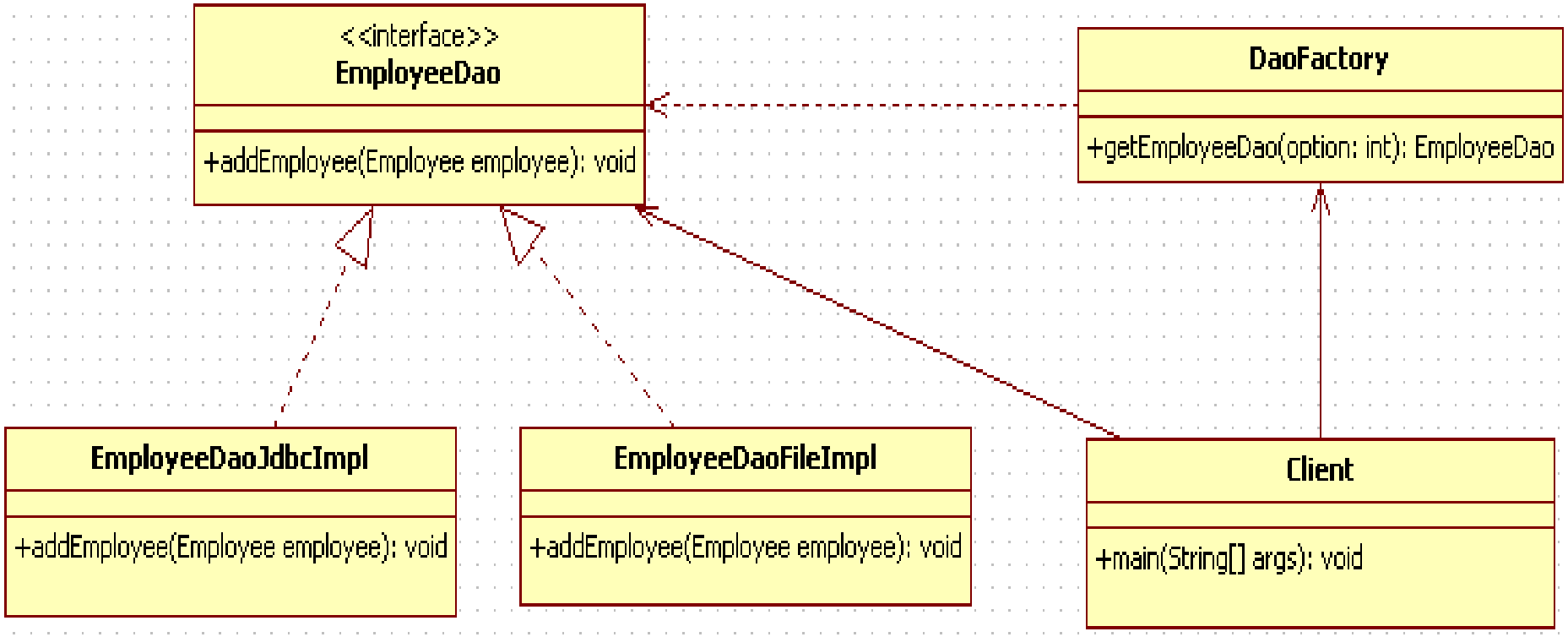
- If we create objects as shown below:
  - Human h = new Human();
  - Bird b= new Bird();
  - Car c = new Car();
  - Comparable sm = new SuperMan();

- Which of the following options are valid?
  1. doTask(h, sm);
  2. doTask(b,c);
  3. doTask(h,c);

Example

- Code Sample: Java_fundamentals_5.zip.
  - Interface : IComparable.java
  - Classes :Book.java, Employee.java, Utility.java, InterfaceExample.java
    - Illustrates using interfaces to implement Open-Close Principle
    - The sort() method is closed for a change, no need to change the algorithm for different entities
    - The sort( ) method is open for extension. It is able to sort Book, Employee entities. But if you create new entities in future, still it will be able to sort them provided the object of that class realize Comparable interface.

Interface and factory



Client access different EmployeeDao implementations using DaoFactory class.

Refer: Java_fundamentals_6.zip

Why code to interfaces?

- **Design**: the methods of an object can be quickly specified and published to all affected developers.
- **Development**: the Java compiler guarantees that all methods of the interface are implemented with the correct signature and that all changes to the interface are immediately visible to other developers
- **Integration**: there is the ability to quickly connect classes or subsystems together, due to their well-established interfaces
- **Testing**: interfaces help in loose coupling between objects and hence help to isolate bugs.

Abstract class versus Interface

Abstract class	Interface
Can have Data fields	Can only have constants
Methods may have implementation	All methods are abstract
Classes and abstract classes extend abstract class	Classes and abstract classes implement interfaces
Class cannot extend multiple abstract classes	Interfaces can extend multiple interfaces
	A class can implement multiple interfaces

Coupling

- **Coupling:**
  - Coupling is the degree to which one class/method knows about another class/method.
  - Coupling Criteria
    - Size : Small is beautiful.
      - Number of connections between modules
      - A method that takes one parameter is more loosely coupled to modules than a method that takes six parameters
      - A class with four well-defined public methods is more loosely coupled to modules that use it than a class that exposes 37 public methods.
    - **Visibility:**
      - **Programming is not like a CID; you don't get credit for being sneaky.**
    - **Flexibility**
      - Flexibility refers to how easily you can change the connections between modules.
      - Ideally, you want something more like the USB connector on your computer than like bare wire and a soldering gun.

Cohesion

- **Cohesion:**
  - Cohesion refers to how closely all the methods in a class or all the code in a method support a central purpose.
  - Example:
    - A function like *Cosine() is perfectly cohesive* because the whole routine is dedicated to performing one function.
    - A function like *CosineAndTan() has lower cohesion because it tries to do more than one* thing.
  - A class should not have a mixture of unrelated responsibilities, that class should be broken up into multiple classes, each of which has responsibility for a cohesive set of responsibilities.

- Good OO design calls for loose coupling and high cohesion

Explore More!!

Never let your curiosity die!

Explore more

- **Reference Books:**
  - **Effective Java (2nd Edition)**
  - **Thinking in Java**
- Software patterns
  - http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns.html
    - Singleton ,Factory and Strategy
- Eclipse Plugin FindBug.
  - https://konnect.mindtree.com/documentrepository/documentDetail.aspx?docID=561
- Java Security
  - http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html

References

Contains the reference that will supplement the self learning and will be needed for completing the assignments & practice questions

References

- Reference books:
  - Head First Java
  - Java Complete reference
  - Thinking in Java
- **Object-Oriented Programming Concepts**
  - http://download.oracle.com/javase/tutorial/java/concepts/
    - Objects, class , inheritance, interface and packages
- Java tutorial for beginners
  - http://download.oracle.com/javase/tutorial/
  - http://www.freejavaguide.com/corejava.htm
  - http://eclipsetutorial.sourceforge.net/totalbeginner.html