Faculty of Computer Science

Programming Language 2

Object oriented design using JAVA

Dr. Ayman Ezzat

Email: ayman@fcih.net

Web: www.fcih.net/ayman

Introduction

Unit 3: Introduction to classes

Outline

1. Introduction
2. Objects and classes
3. Information hiding
4. Constructors
5. Some examples of Java classes
6. Class Diagrams
7. Inheritance revisited
8. Polymorphism
9. Interfaces

Why classes not struct

- Struct data variables and methods are defined as public by default.
- Class data variables and method are defined as private by default.
- Classes can have relationship between each other.
- Classes can have concepts like overloading, inheritance, and polymorphism.
- Classes have a new access level for its' members called protected.

2. Objects and classes

Objects to model the world

- An object is something that has a **state**: it has associated with it some variables whose values define that state. *For example*:
  - **A bank account** could be modeled as an object; its **state** may be:
    - the name of the account holder,
    - the current balance,
    - the overdraft limit
    - and a list of recent transactions.
  - **An airplane in an air-traffic control system**; its **state** might consist of
    - data that identifies the plane,
    - its position
    - and its eventual destination.
  - **A pull-down menu in the user interface of an applet**; its **state** may be:
    - a list of the commands for the pull-down menu,
    - the type of box that encloses the menu when it is on screen
    - and its colour.
- Another important feature of objects is that we can **invoke methods** on them.
  - **For example, the expression obj.setA(2);**

Unit in strategy game

2. Objects and classes

Objects to model the world (Cont'd)

- **An applet or application** is a collection of communicating objects.
  - The means of this communication is **via methods**, which allow data to be passed in **to objects** (by method arguments) and **out of objects** (as values returned by methods). For example:
    - The **user interface object** invokes a method of the **pull-down menu object**, resulting in the display of the pull-down menu.
    - The **pull-down menu object** returns a value identifying that the user has selected the 'word count' command.
    - The **document object** invokes a method to count the words in the document, with a count being incremented each time a word is found.
    - A method is invoked on the **window object**, asking it to display the word count on the user's computer screen.

2. Objects and classes

Defining classes and constructing objects

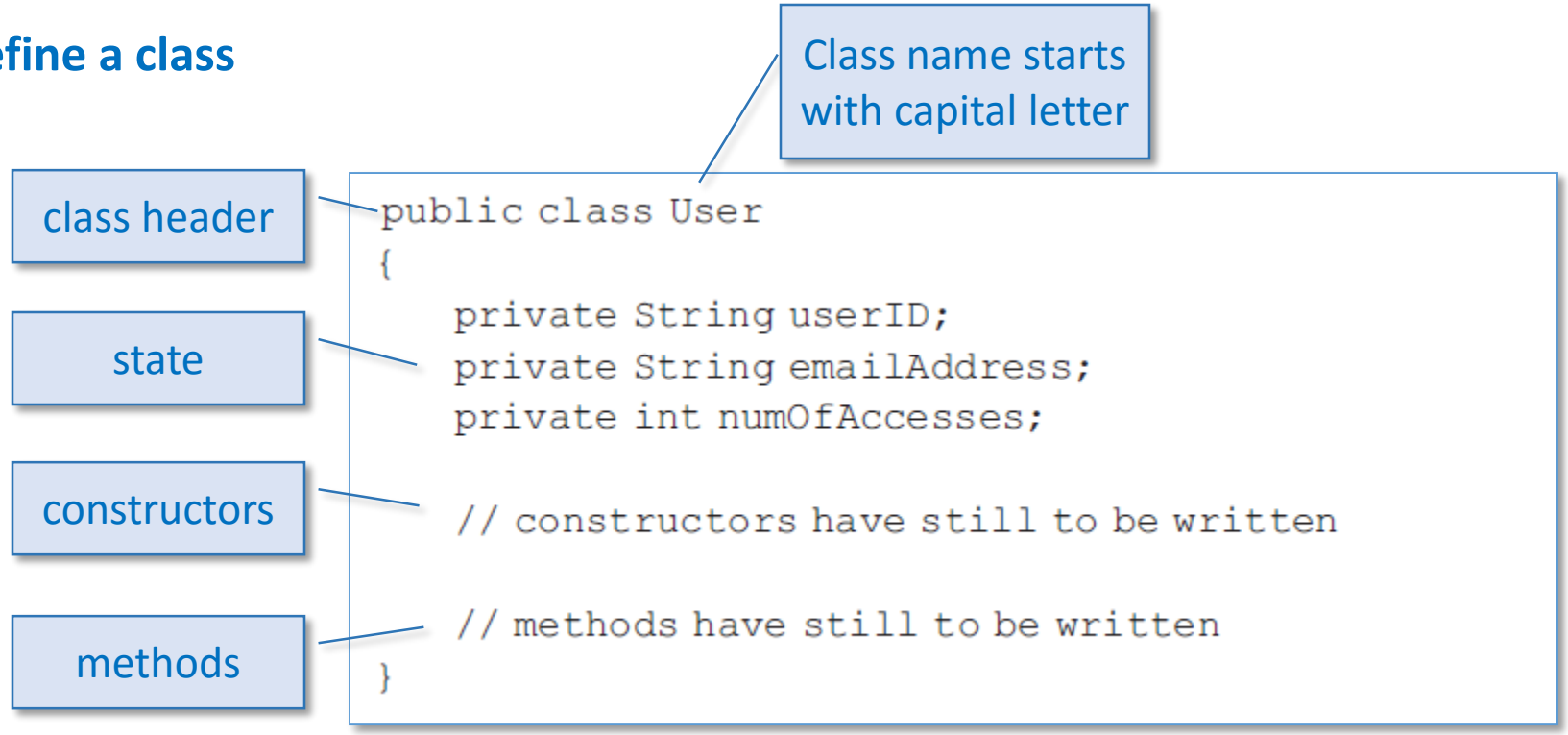
- **A class** is a definition of a category of objects.
  - A class defines:
    - **the items of data** that make up the state of an object (**reference variables**), including their type: for example, whether they are integers, strings, or types defined by other classes;
    - **the 'behavior'** of objects of the class, by which we mean the **methods** that can be invoked on objects of the class.
  - **Class members**: The instance variables and methods of a class or object are known collectively as its members.
  - Classes **are like templates or blueprints** for objects. They do not make objects!

- There are usually three steps to using an object:
  (1) Define a class.
  (2) Construct an object.
  (3) Store an object reference.

2. Objects and classes

Defining classes and constructing objects (Cont'd)

(1) Define a class



2. Objects and classes

```
User john;
john = new User();
```

```
User john = new User();
```

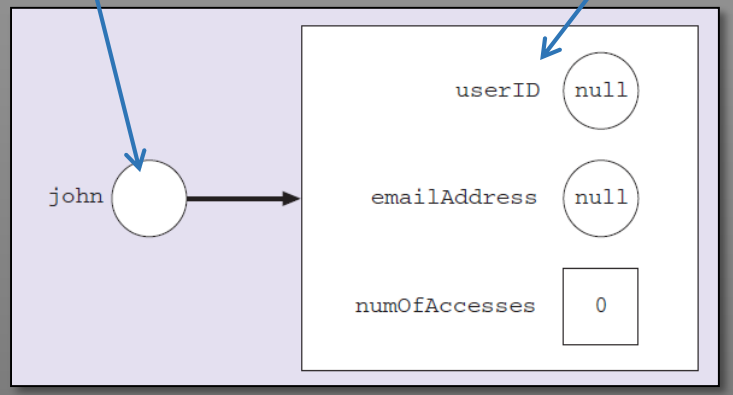
Store an object reference: storing a reference to a User object. The general statement is:

ClassName objectName;

In the above example, the declaration creates a reference variable, "**john**" of the **type** "**User**"

Construct an object. We call this **invoking a constructor**

After execution →



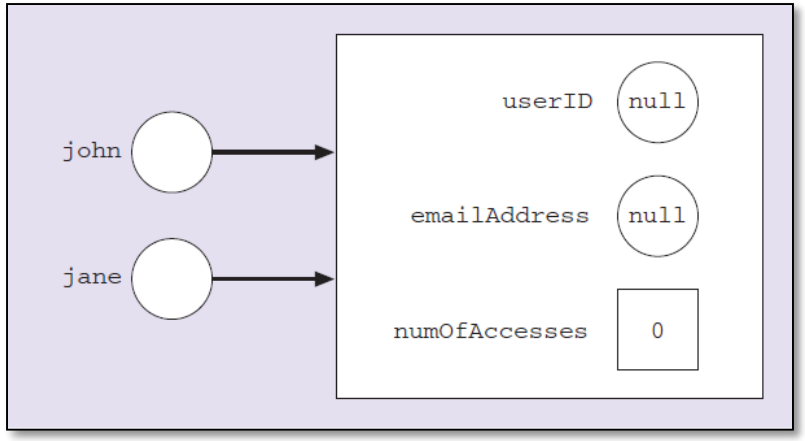
The keyword **null** represents a reference that does not point at any object.

2. Objects and classes

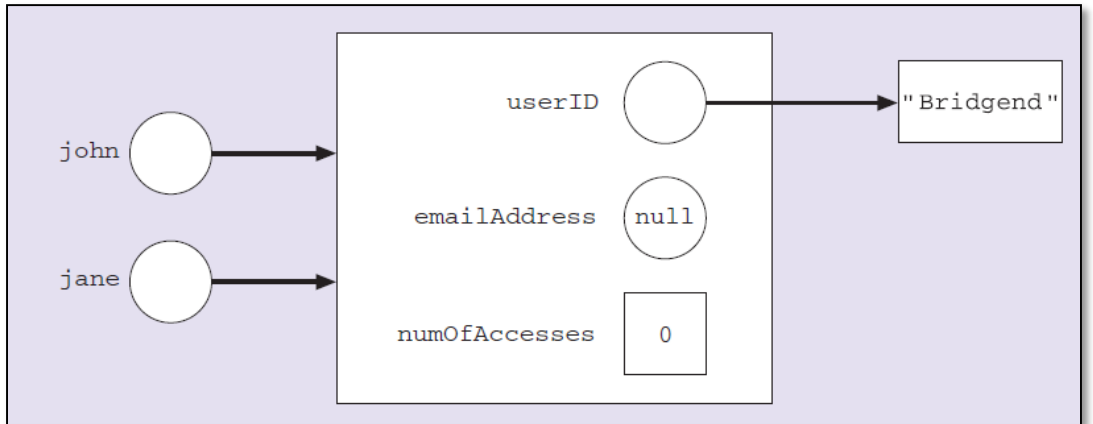
Reference variables and assignment

- If we write:

```
jane = john;
```



- If we make the userID variable in this object reference the string "Bridgend"

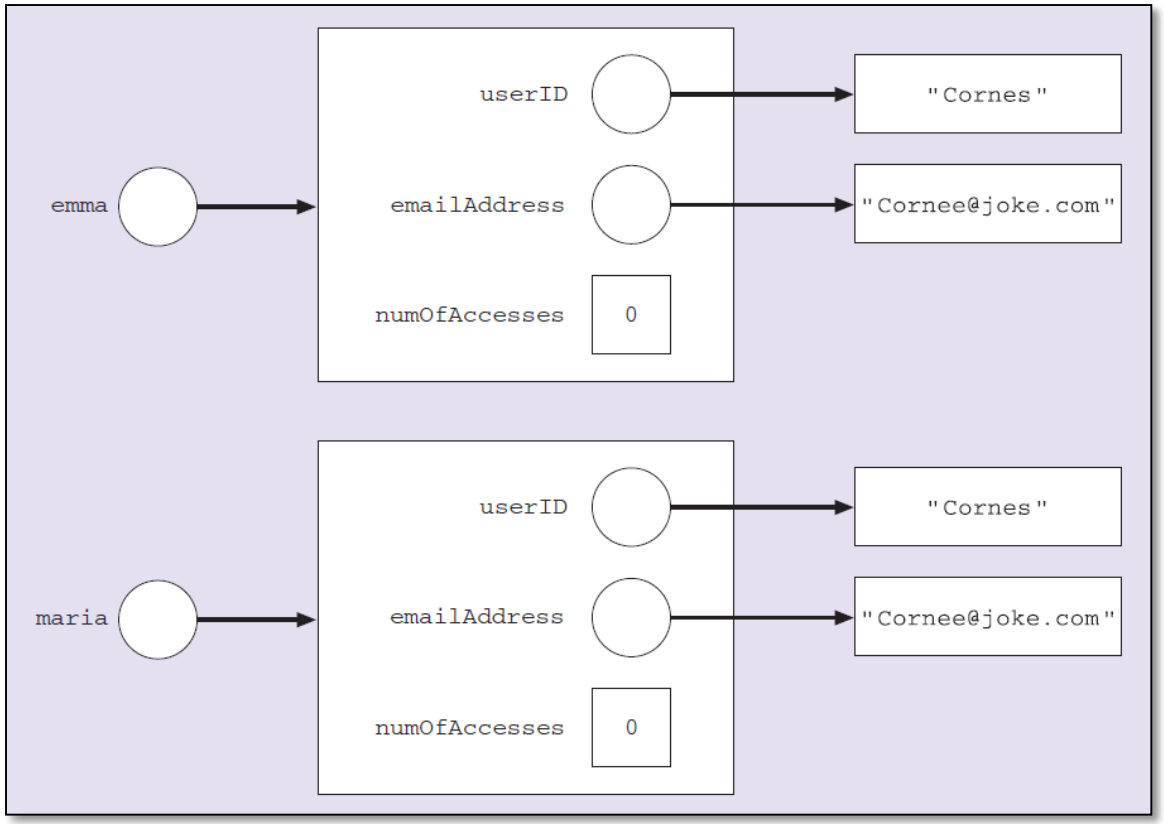


2. Objects and classes

Reference variables and assignment (Cont'd)

- If we write:

```
User emma = new User("Cornes", "cornee@joke.com");
User maria = new User("Cornes", "cornee@joke.com");
```



3. Information hiding

Getter and setter methods

- A **getter method** (also called an **'accessor'** method):
  - It is one that accesses some attribute of an object.
  - A getter method **should not change** the state of the object.

- A **setter method** (also called a **'mutator'** method)
  - It is one that changes the state of an object by setting the value of an instance variable.
  - Setter methods **do not** normally **return** a value.

Usage of getter and setter methods

1. If you like to put conditions on your data input i.e If statement on the data entered by the user.
2. If you like to set a property as read only
3. If you like to set a property as write only
4. If you plan in the future to do something before setting value

Other than that public variables is ok, however not advised at all

3. Information hiding

Getter and setter methods (Cont'd)

- **Example**: Getter and setter methods for the User class

```
public class User
{
    private String userID;
    private String emailAddress;
    private int numOfAccesses;

    // constructors have still to be written

    // setter methods
    public void setUserID (String usIdVal)
    {
        userID = usIdVal;
    }

    public void setEmailAddress (String emailAddressVal)
    {
        emailAddress = emailAddressVal;
    }
```

```
    // getter methods
    public String getUserID ()
    {
        return userID;
    }

    public String getEmailAddress ()
    {
        return emailAddress;
    }

    public int getNumOfAccesses ()
    {
        return numOfAccesses;
    }

    // update number of accesses by this user
    public void updateAccesses ()
    {
        numOfAccesses++;
    }
}
```

3. Information hiding

Member access by dot notation (.)

- The keyword **public** in front of a member in the class User indicates that it may be accessed using dot notation with a User reference variable.
- In contrast, because we have used the keyword **private** in front of the User instance variables, there is no access outside the class itself to these variables.

This is NOT allowed →

```
john.numOfAccesses = -1; // not allowed
```

This is allowed →

```
john.updateAccesses(); // allowed
```

3. Information hiding

Packages

- A package is used to associate a number of classes that are closely related to one another.
  - if classes form a cohesive group, they should be marked as belonging to the same package.
  - For example, Java has a collection of classes named java.math.
  - If we wanted to make use of this package, we would add a statement to advise the Java system of this, by adding an **`import`** statement at the beginning of our program. In this case we would say:

```
import java.math.*;
public class User
{
    // etc.
}
```

  - The .* notation means 'all the classes in the package'.

3. Information hiding

Access modifiers

- There are two kinds of access to class members:
  - Using a **qualified name** → with an object reference and a dot notation,
    - for example **john.getNumOfAccesses();**
  - Using a **simple name** → a name without a reference and dot notation,
    - for example **getNumOfAccesses()**

- When to use them?
  - **Simple name**:
    - Class members could be accessed using the simple name within their class.
    - If we say a member is inherited we mean that the subclass has the same access to it as if it were defined in the subclass.
  - **Qualified name**:
    - Other classes would have to have to use a qualified name.

3. Information hiding

Access modifiers (Cont'd)

- There are three keywords associated with controlling levels of access to class members in Java: **`public, protected & private`**. These are known as **access modifiers**.
  - The following table & example illustrates the next few slides:

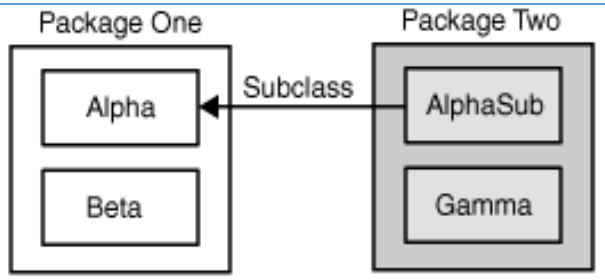
Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

(source: http://java.sun.com/docs/books/tutorial/java/javaOO/accesscontrol.html)

Main program

Example



Visibility for a ref. var. defined in Alpha

Modifier	Alpha	Beta	Alphasub	Gamma
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

3. Information hiding

Access modifiers (Cont'd)

(1) The public access level

- **Accessing public members** (instance variables or methods): any class can access **public members** of a **public class** using a **qualified name**.

```
package AppletUser;
public class UserV2
{
    // public instance variable
    public int numOfAccesses;

    // public setter method
    public void setNumOfAccesses (int n)
    {
        numOfAccesses = n;
    }
}
```

```
import AppletUser.*;
public class TestUserV2
{
    public static void main (String[] args)
    {
        // this method has a UserV2 reference
        UserV2 myUser;

        // once initialized, the reference can be used
        // to access members of a UserV2 object
        myUser = new UserV2();

        // qualified access to instance variable numOfAccesses
        myUser.numOfAccesses = 1;

        // qualified access to setNumOfAccesses method
        myUser.setNumOfAccesses(1);
    }
}
```

3. Information hiding

Access modifiers (Cont'd)

(1) The public access level (Cont'd)

- **Inheritance of public members**: Any public members are inherited by all subclasses, so they can be accessed within those classes without a dot notation.

```
package AppletUser;
public class UserV2
{
    // public instance variable
    public int numOfAccesses;

    // public setter method
    public void setNumOfAccesses (int n)
    {
        numOfAccesses = n;
    }
}
```

```
import AppletUser.*;
public class MyUser extends UserV2
{
    public void doThings ()
    {
        // simple name access to numOfAccesses variable
        numOfAccesses = 1;

        // simple name access to setNumOfAccesses method
        setNumOfAccesses(1); // same effect as line above
    }
}
```

3. Information hiding

Access modifiers (Cont'd)

(3) The private access level

- **Accessing private members**: **private members** can be accessed **only** within its defining class.
- **Inheritance**: private members are never inherited, so can never be accessed by subclasses as if they were part of the subclass.

Which access level should I use?

- A rule of thumb is to have **public methods** and **private data**.
- **Helper methods:** it is normal practice for them to be **private**.
  - Helper methods (or helpers) are those which provide a service needed only inside the class.
- We will not be making extensive use of protected or default access types, but your understanding of Java would be incomplete without us having discussed them.

Data Hiding summary

- **Private**: An element that can be seen inside the class definition only, you can never access from an object or a driven class.
- **Public**: An element can be accessed in every place (inside class, from objects of class, and from driven classes).

4. Constructors

- **A constructor** is a piece of code that contains instructions on how to initialize objects of a class.
- A constructor must have **the same name** as its class.

Example:

The Coordinate class:

```
public class Coordinate{
    // instance variables
    private int xPos, yPos;
    // a public Coordinate constructor
    public Coordinate (int xVal, int yVal){
        xPos = xVal;
        yPos = yVal;
    }
    // methods of the class
}
```

Defining a new object:

```
Coordinate newCoord = new Coordinate(20, -2)
```

- Although constructors look like methods, according to the language specification they are not.
  - e.g. the code for the constructor **does not declare a returned type**.

4. Constructors

Multiple constructors:

- you may provide several constructors, catering for a selection of initialization possibilities. Each constructor must have a unique set of arguments (if any).

Example →

```
public class Coordinate{
    private int xPos, yPos;
    private static final int DEFAULT_X = 0;
    private static final int DEFAULT_Y = 0;
    // two-argument constructor
    public Coordinate (int xVal, int yVal){
        xPos = xVal;
        yPos = yVal;
    }
    // one-argument constructor
    public Coordinate (int xVal){
        xPos = xVal;
        yPos = DEFAULT_Y;
    }
    // zero-argument constructor
    public Coordinate (){
        xPos = DEFAULT_X;
        yPos = DEFAULT_Y;
    }
    // methods of the class
}
```

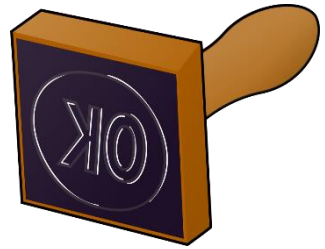
Constructor Overloading

Constructor Overloading

4. Constructors

Constructor access level

- We have created all our constructors as **public** because it is usual that we want to create objects from outside the class in which we define them.
- However, we can make it illegal to create objects of the type outside the defining class by making the constructor **private**.
  - **Example: you cannot create an object of the Math class because its only constructor is private.**
- Constructors can also be given protected or default access level.



Sealed Class

Inheritance of constructors

- Constructors are **not considered class members** and **are not inherited**.

The default constructor

- If you do not write a constructor for your class, Java provides you with a special constructor called the default constructor
- **The default constructor has no arguments.**

4. This Keyword

The keyword `this`

- The word **`this`** can be used to **invoke a constructor**.
- You can think of **`this`** as a reference to the class in which it appears.
- Consider the example on the right:
  - The first three constructors **make use of the three-argument constructor**. This is a way of reusing constructors you have written and can **assist with readability** when you have several constructors.
  - Do not think of this(a, b, 0) as a call to a method. It is only possible to invoke constructors in this way.
  - a constructor cannot invoke itself (whereas a method can).

Reference to the calling object

```
public class ThreeInt{
    private int p, q, r;
    public ThreeInt (){
        this(0, 0, 0);
    }
    public ThreeInt (int a){
        this(a, 0, 0);
    }
    public ThreeInt (int a, int b){
        this(a, b, 0);
    }
    public ThreeInt (int a, int b, int c){
        p = a; q = b; r = c;
    }
    // methods associated with ThreeInt
}
```

4. Super

The keyword `super`

- You can think of **`super`** as a reference to the **superclass** of the class in which it appears. For example:
  - Assume that we want to extend the class **`ThreeInt`** presented earlier.

```
public class FourInt extends ThreeInt{
    private int s;
    // methods
}
```

  - Now we would want to ensure correct initialization of the instance variables p, q and r defined in ThreeInt, but we would have to rely on ThreeInt to do this, as the data is private.
  - Some code for a constructor in FourInt that would do this is shown below:

```
public FourInt (int a, int b, int c, int d){
    super(a, b, c);
    s = d;
}
```

  - The code super(a, b, c) results in the three-argument constructor of ThreeInt being executed, which results in the values of a, b and c being assigned to the instance variables p, q and r associated with this class. After this, the instance variable s associated with FourInt is assigned the value d.

4. Constructors

Default use of superclass constructors

- If you do not call a superclass constructor, **Java will call it for you** by default.
  - The only superclass constructor it call by default is a **zero-argument one**.
- The default constructor for every class ClassName looks like this:

```
public ClassName ()
{
    super();
}
```

- Only the class **`Object`**, which is at the top of the class hierarchy, does not call a superclass constructor, because it has no superclass.

CLASS DIAGRAMS

Class diagram symbols

Classes can have relationships between each other; first type of relations is called

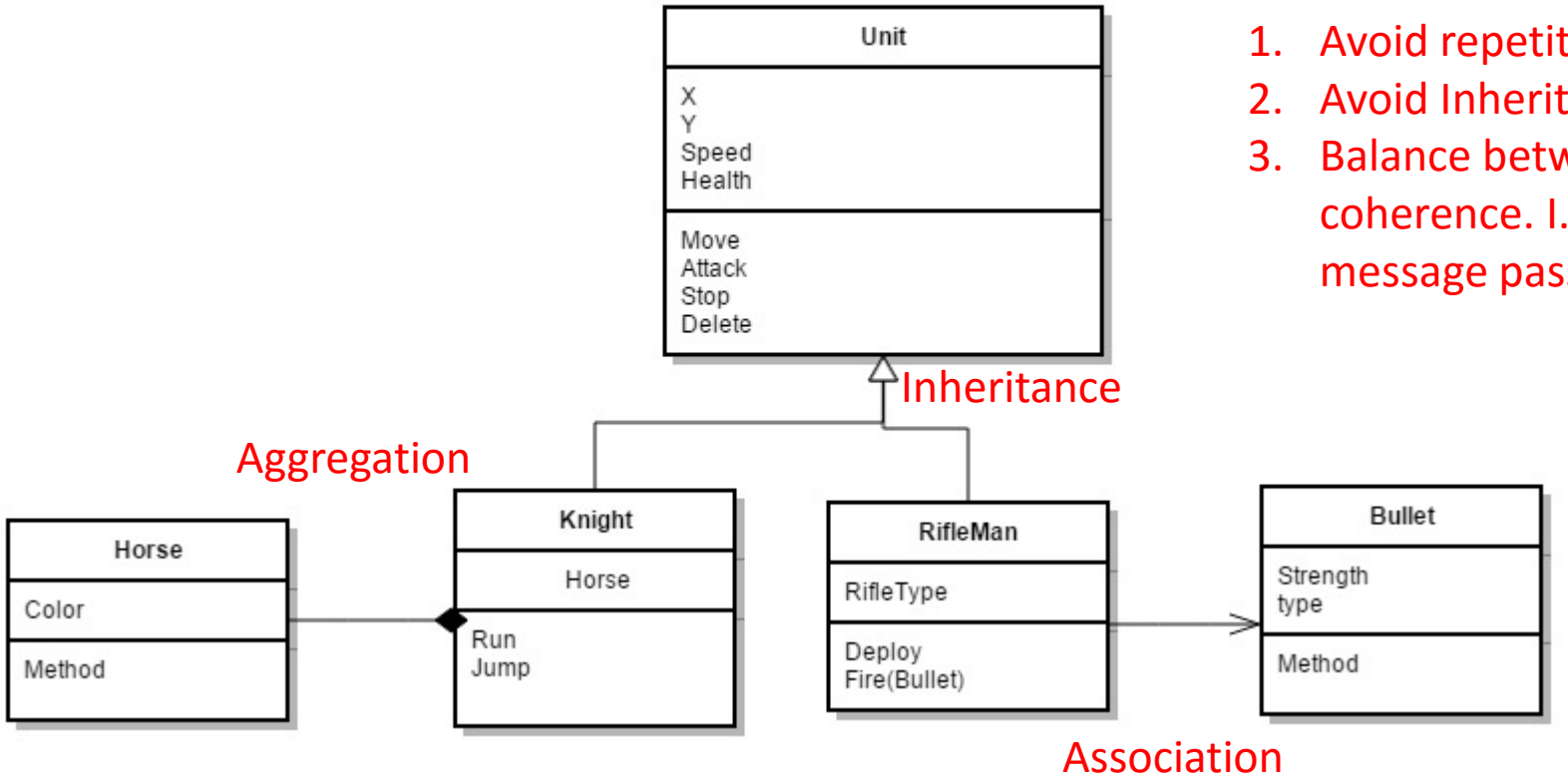
1. Aggregation: A class is composed of another class drawn by 
2. Association: A class is being assisted by another class drawn by →
3. Inheritance: A class is driven from parent class drawn by 

- In order to simplify the communication between developers, UML "unified modeling language" was developed.
- UML has many diagrams that simplify the design of programs; the most important one is the class diagram, as it shows each class details and the relations between other classes.

Steps to draw a class diagram

- Mark all the nouns in your problem (Classes)
- Mark all verbs (Methods)
- Mark all characteristics (Properties)
- Encapsulate and inherits whenever you have repeated properties or methods.
- Do not have inheritance cycle
- Keep balance between coupling (Association) and Cohesion (Aggregation)

Strategy Game example



1. Avoid repetition
2. Avoid Inheritance Cycle
3. Balance between coupling and coherence. I.e. balance between message passing and containing.

Inheritance

- Use whenever you have common properties and methods in your class diagram
- It is direct implementation for concept Never repeat your code.

```
public class Vechile {

    public int id=10;
    public String ModelName;
    public int MaxSpeed;
    public int CurrentSpeed;
    private int Gravity;
    public Engine MyEngine;
```

```
public class RacingCar extends Vechile{
    public int TurboValue;
    public void FireTurbo()
    {

        System.out.println("Racing car has
    }
```

Aggregation/Composition

- A composition relationship always used for having Cohesive classes
- A class is composed from another class.
- Aggregation sometimes might make creating objects memory consuming.
- Composition means we can not create an object from engine class stand alone.
- Aggregation means that engine class must be
- inside Vehicle class

```
public class Vechile {

    public int id=10;
    public String ModelName;
    public int MaxSpeed;
    public int CurrentSpeed;
    private int Gravity;
    public Engine MyEngine;
```

```
public class Engine {
  public String EngineType;
  public int NoValves;
}
```

Association/ Message Passing

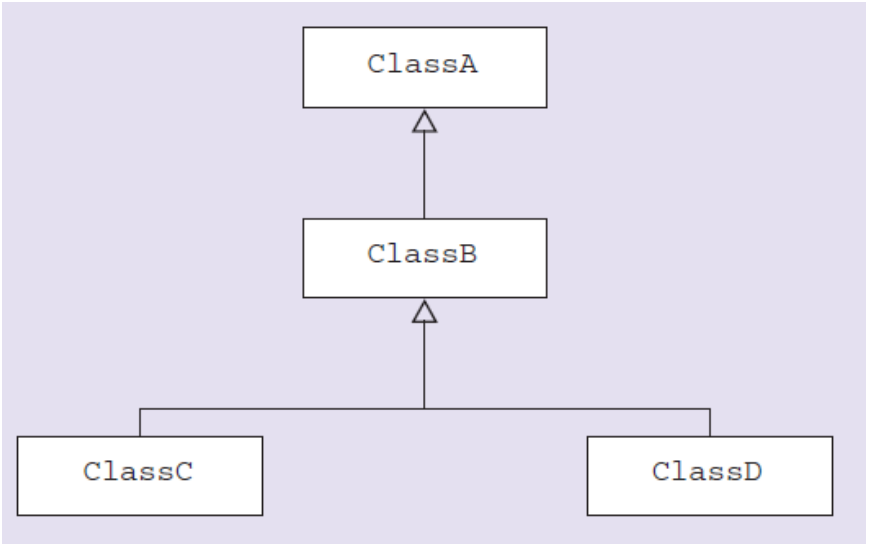

```
public void DriveCar(Vechile x)
{

    System.out.println("Driver " + this.FullName +
}
```

- Most used relationship by classes
- Pass a reference to an object to function header.
- Then you can use this reference to do some operation.
- Many association with single class will lead to coupling problem, hence changing some method name or property name will need many changes and will coast a lot.

The class hierarchy

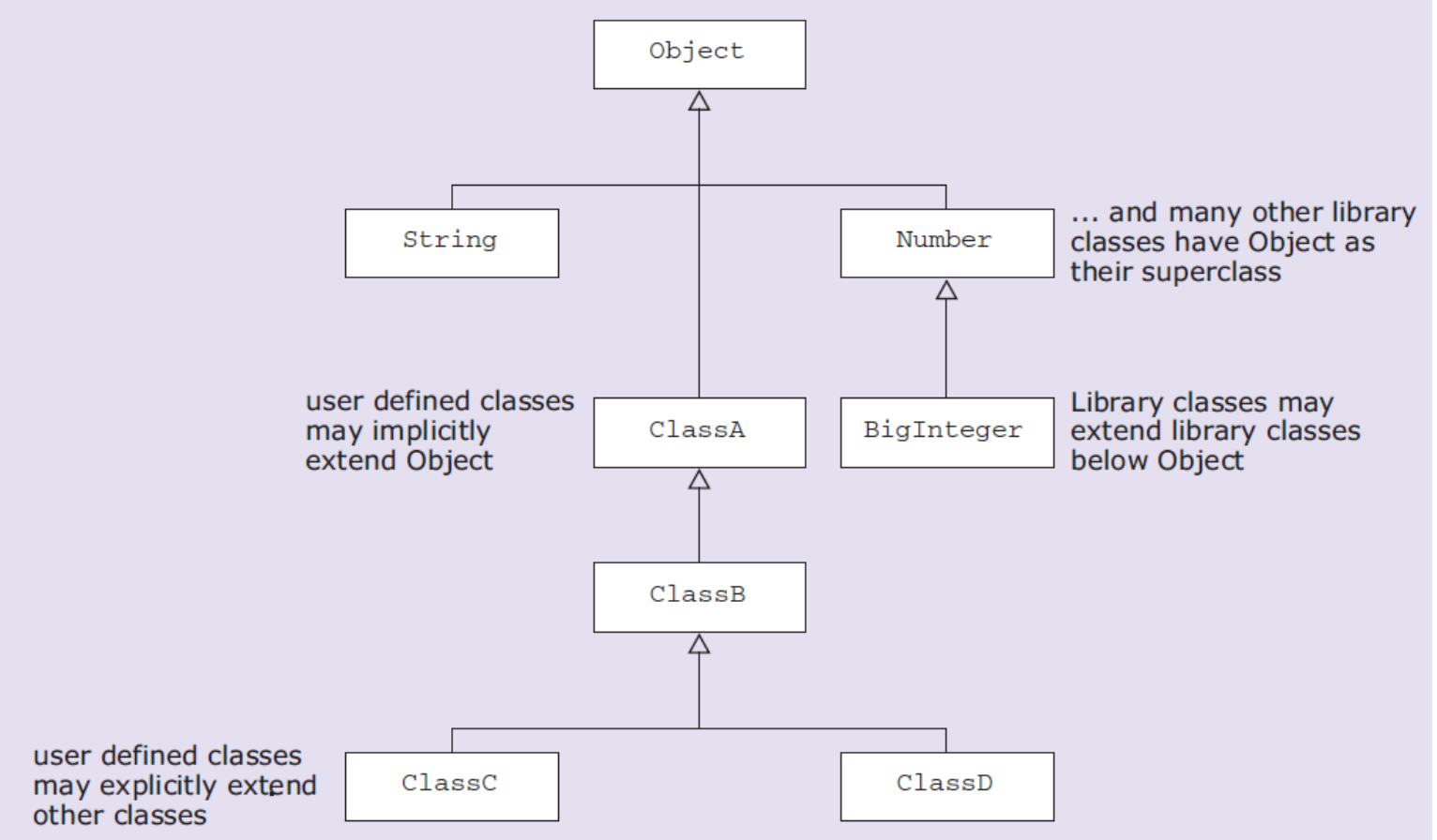
- There are two types of class in a Java hierarchy:
  1. **user-defined**
  2. **Java class library**.

- The figure below shows a user-defined hierarchy
  - classC and classD inheriting from classB which, in turn, inherits from classA.



The class hierarchy

- The figure below shows part of the Java inheritance hierarchy, including the root of the hierarchy, which is the class whose name is Object. The class Object has no superclass.



The class hierarchy

- The classes that programmers develop will:
  - either explicitly inherit from some named class by using the keyword extends;
  - or implicitly inherit from the class whose name is Object.

```
public class ClassName
{
    // instance variables
    // constructors
    // methods
}
```

≡

```
public class ClassName extends Object
{
    // instance variables
    // constructors
    // methods
}
```

The class hierarchy

The `equals` method

- `==` operator is used to compare **the references** of the objects.
  - Comparing two references for equality **does not** compare the contents of the objects referenced.
- `public bollean equals(Object O)` is a method provided by `Object` class.
- The default implementation uses == operator to compare two objects. But it is expected that programmers will **override** this method so that it is used to compare the values of two objects.
- Example of writing an `equals` method for the `User` class →

```
public boolean equals (Object o)
{
    // cast the Object argument to a User type
    User u = (User) o;
    return userID.equals(u.userID) &&
           emailAddress.equals(u.emailAddress) &&
           numOfAccesses == u.numOfAccesses;
}
```

Note1: the argument of the method has to be of an `Object` type, otherwise this method won't override the `equals` method of `Object`.

Note2: we had to use the equals method of the String class to compare the contents of the string instance variables.

Source: http://technologiquepanorama.wordpress.com/ & M257 textbook

The class hierarchy

```
String str1 = new String("MyName");
String str2 = new String("MyName");
if(str1 == str2){
    System.out.println("Objects are equal")
}else{
    System.out.println("Objects are not equal")
}
if(str1.equals(str2)){
    System.out.println("Objects are equal")
}else{
    System.out.println("Objects are not equal")
}
```

-->Output:

Objects are not equal
Objects are equal

```
String str2 = "MyName";
String str3 = str2;
if(str2 == str3){
    System.out.println("Objects are equal")
}else{
    System.out.println("Objects are not equal")
}
if(str3.equals(str2)){
    System.out.println("Objects are equal")
}else{
    System.out.println("Objects are not equal")
}
```

-->Output:

Objects are equal
Objects are equal

Information hiding

Access modifiers (Cont'd)

(2) The protected and default access levels

- **default access** of a member means it has **no access modifier**.
- **Accessing protected/default members**: classes in the same package can access **protected and default members** using a **qualified name**.

```
package AppletUser;
public class UserV3
{
    // protected instance variable
    protected int numOfAccesses;

    // default access level method
    void setNumOfAccesses(int n)
    {
        numOfAccesses = n;
    }
}
```

```
import AppletUser.*; // this class does not compile!
public class TestUserV3
{
    public static void main (String[] args)
    {
        // this method has a UserV3 reference
        UserV3 myUser;

        // initialize the reference
        myUser = new UserV3();

        // qualified access to instance variable numOfAccesses
        // is not allowed!
        myUser.numOfAccesses = 1;

        // qualified access to setNumOfAccesses method
        // is not allowed!
        myUser.setNumOfAccesses(1);
    }
}
```

• The class TestUserV3 does not compile, **because** it is not in the same package as UserV3, and therefore it cannot access either the protected or the default access level members of UserV3.

• Instead of importing the classes in the AppletUser package, the TestUserV3 class would have to be **preceded by the words "package".**

3. Information hiding

Access modifiers (Cont'd)

(2) The protected and default access levels (Cont'd)

- **Inheritance of protected members**:
  - Any subclass inherits any **protected** and **public** members of its superclass.

- **Inheritance of default members**:
  - **Default members** are inherited **only** by subclasses in **the same package** as the superclass.
    - i.e. any subclass in **the same package** inherits the **protected, public**, and **default** access level members of its superclass.
  - **Default access** level is sometimes known as '**package access**' for this reason.

- **Protected**: A level between private and public members, it can be accessed inside the class, from driven classes but never accessed from objects.

Overloading

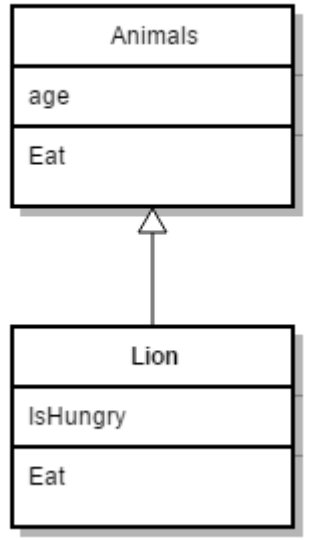
- Having a function with same name but different datatype parameters.
- Or different number of parameters

```
public void Move (int x,int y)
{
    System.out.println("Move int int");
}
public void Move (int x,int y, int z)
{
    System.out.println("Move int int int");
}
public void Move (int x,double y)
{
    System.out.println("Move int double");
}
```

Overriding

- Override the functionality of an existing method.
- The function remains with same parameters ,same name and return type.
- You can not override final and static methods
- Super key word refers always to parent class

```
public class Animal {
    public void eat()
    {
        System.out.println("Animal is easting");
    }
}
```




```
@Override
public  void eat()
{
    super.eat();
    System.out.println("Lion is easting");
}
```

Overriding tostring()

The `toString` method

- The purpose of this method is to produce a readable text representation of an object's contents.
- The method that you inherit from **`Object`** produces:

ClassName@NumericValueComputedUsingTheObject

For example, making use of a two-argument `User` constructor:

```
public User(String anID, String anAddress)
{
    numOfAccesses = 0;
    userID = anID;
    emailAddress = anAddress;
}
```

you might have written:

```
User a = new User("Tembo", "zamby@lusaka.zm");
System.out.println(a.toString());
```

The output would look something like this:

```
User@3fbdb0
```

→**This is not very useful output**

The class hierarchy

Overriding the `toString` method

If we want to have a particular form of output for our `User` objects, we can define a `toString` method in the `User` class. It might look something like this:

```
public String toString ()
{
    return "User " + userID + ", Email " + emailAddress;
}
```

Now:

```
User a = new User("Tembo", "zamby@lusaka.zm");
System.out.println(a);
```

produces the output:

```
User Tembo, Email zamby@lusaka.zm
```

Interfaces

What are interfaces?

- **Interfaces** define a standard behavior required from a class.
  - An interface specifies a list of methods that a group of unrelated classes should implement, so that their instances can interact together by responding to a common subset of messages.
- Interfaces are "like" classes, but they may ONLY have:
  - **Abstract methods, i.e. method header** but NO method code.
    - All methods in an interface are automatically **`public`**.
      No need to writ public in the method header.
  - **Constants**, but NO instance variables.
    - All constants in an interface are automatically **`public static final`**.
  - Note that an interface is **NOT** a class and will have no instances of its own, and so it has no instance variables and no constructor.

To define an interface, we write:

```
public interface InterfaceName {
    //some constants and abstract methods}
```

- Interfaces can extend several other interfaces

```
public interface VenPre extends Predator, Venomous{
    //interface body }
```

```
public interface RGBColours
{
    int RED = 1;
    int GREEN = 2;
    int BLUE = 3;
    void showColour(int colour);
}
```

Example of an interface

Interfaces

How to use interfaces?

- To make use of an interface we need a class to **`implement`** the interface. The class does this by providing the code for any abstract methods in the interface.
- Any class that implements an interface, e.g. the **`RGPColors`** interface from the previous slide, must have a method called **`showColor`**.

```
public class Crayon implements RGPColors{
  public void showColor(){
        //the code for the method goes here!
  }
}
```

Why interfaces?

- To have unrelated classes implement similar methods (behaviors)
- To model multiple inheritance (i.e. to "inherit" from multiple interfaces) - you want to impose multiple sets of behaviors to your class.
- Some Java methods can be applied to objects only if implement certain interfaces.
- **To force having contract between developers**

Interfaces

SAQ:

Say whether each of the following interface definitions is valid or invalid and state your reason.

(a)
```
public interface FriendlyInterface
{
    void displayHello()
    {
        System.out.println("Hello");
    }
}
```

(b)
```
public interface UnfriendlyInterface{}
```

(c)
```
public interface FixedInterface
{
    int MAX_WORD = 200;
}
```

(d)
```
public interface Textable1
{
    final int MAX_WORD = 200;
    String text;
    void displayText();
}
```

(e)
```
public interface Textable2
{
    boolean displayText(int maxWords);
}
```

Answer

(a) Invalid – because it contains a concrete method.

(b) Valid – although perhaps not very useful.

(c) Valid – this defines a constant as the modifiers public static final are implicitly applied.

(d) Invalid – because text is either an attempt at an instance variable or is an uninitialized constant.

(e) Valid – this defines a single method, which is implicitly defined as public abstract.

Interfaces

SAQ: Assume `Employee` and `Parent` are classes; `Comparable` and `Serializable` are standard Java interfaces. Are these declarations valid? If not, why not?

(a)
```
class MonthlyEmployee extends Employee, Parent
implements Comparable
```

(b)
```
class MonthlyEmployee extends Comparable
```

(c)
```
class MonthlyEmployee extends Employee implements Parent
```

(d)
```
class MonthlyEmployee implements Comparable, Serializable
```

(e)
```
public interface Sortable extends Comparable
```

(f)
```
public interface Sortable extends Parent
```

(g)
```
Comparable employee1 = new Comparable();
```

ANSWER.....

(a) Invalid – a class cannot inherit from more than one superclass.

(b) Invalid – a class cannot extend an interface, although it can implement it.

(c) Invalid – a class cannot implement another class: we cannot get multiple inheritance this way!

(d) Valid – a class can implement more than one interface.

(e) Valid – an interface can inherit from another interface.

(f) Invalid – an interface cannot inherit from a class.

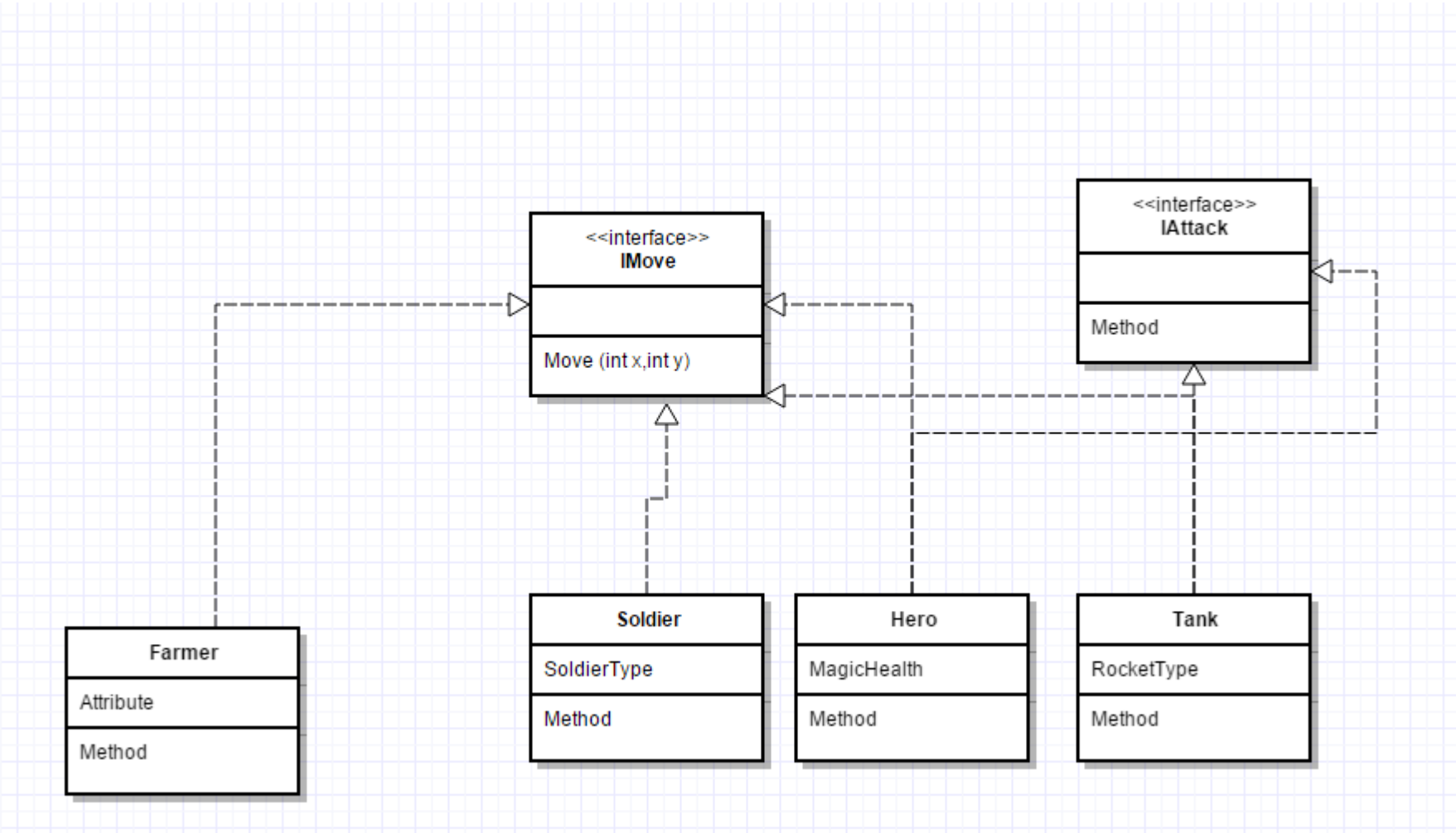
(g) Invalid – you can define a variable (like `employee1`) of an interface type but you cannot create an object of an interface type.

Polymorphism

- Many Faces (Ability to change faces) execute behavior of the pointed environment.
- A way that you point a reference from parent class to child class then fire some function.
- It is well known to select different object with same function name but **different behavior** then polymorphism is used.
- In strategy game and you like to select different objects (soldier, Hero, Farmer) and fire the function move per each object.
- When you select the objects you have created an array of parent class each element pointing to child object.
- We always use interfaces with polymorphism

Implementing Interfaces

- All classes implement 2 interfaces except Farmer class only implement the Imove interface.
- We can have array of pointers of type Imove.



Polymorphism Code Sample

```java
public class Hero implements  IAttack, IMove{
    public void Attack()
    {
        System.out.println("Hero Attacking");
    }
    public String ExtraAttack()
    {
    return "Hero Extra Attack";
    }
    public void Move(int x,int y)
    {
        System.out.println("Hero is moving x y");
    }
}
```

```java
public interface IAttack {
    public void Attack();
    public String ExtraAttack();
}
```

Concrete Objects

```java
Soldier S1=new Soldier();
Hero H1=new Hero();
Tank T1=new Tank();
```

Selecting Objects and filling Array of references

```java
IAttack []AllAttackUnits=new IAttack[5];
AllAttackUnits[0]=S1;
AllAttackUnits[0]=T1;
AllAttackUnits[0]=H1;
```

Doing Polymorphismic call to function attack

```java
for (int i=0;i<3;i++)
{
    AllAttackUnits[i].Attack();
}
```

Information hiding

Static variable (also known as a **class variable)**

- When you declare a static variable in a class, you are specifying that there is **exactly one copy** of the variable **for all objects** defined by that class.
  - Static variables and methods are both introduced by means of the keyword **`static`**, which is another example of a modifier.

```
public class User{
    private int numOfAccesses;
    // static (class) variable declaration
    private static int totalAccesses;

    public void updateAccesses (){
        numOfAccesses++;
    }
    public int getNumOfAccesses (){
        return numOfAccesses;
    }
    public void updateTotalAccesses (){
        totalAccesses++;
    }
    public int getTotalAccesses (){
        return totalAccesses;
    }
    // and other methods
}
```

```
public class TestUser{
    public static void main (String []args){
        // this method has two User references
        User uOne, uTwo;
        // once initialized, the references can be used
        // to access members of the User objects they
        // reference
        uOne = new User();
        uTwo = new User();
        uOne.updateAccesses();
        uOne.updateTotalAccesses();
        uTwo.updateTotalAccesses();
        System.out.println(uOne.getNumOfAccesses());
        System.out.println(uTwo.getNumOfAccesses());
        System.out.println(uOne.getTotalAccesses());
        System.out.println(uTwo.getTotalAccesses());
    }
}
```

The output of the TestUser program would be:

1
0
2
2

Information hiding

Static variables (Cont'd)

When to use static variables?

- They are usually used to define:
  - constants,
  - one value for all objects

Dot notation for static variables

- Suppose that class User has a public static variable day, recording the day of the week as an integer:

```
public class User
{
    public static int day;
    // etc.
}
```

- We can create an object of type User referenced by jane:

```
User jane = new User();
```

- However, to access day **it is preferable** to write `User.day` rather than `jane.day`, as the latter suggests instance data.

Information hiding

Static methods (also known as a **class methods)**

- Static methods carry out general functions **not associated** with objects.
  - For example, the static method **`max`** within the class **`Math`** returns the greater of its two arguments. The method header for one version of max looks like this:

```
public static double max (double a, double b)
```

Thus the code:

```
double val = Math.max(Math.PI, 4.0);
```

- Static methods are allowed only simple name access **to static (not instance) variables**.

- The **`main`** method is a static method.

```
public static void main (String[] args)
{
    // main code
}
```

Information hiding

Constants: the keyword `final`

- When you precede an identifier name with the word final, it means that once its value is specified, nothing in your code can change (or attempt to change) its value.
  - For example:

```
static final int MAX_VALUE = 12;
```

```
public static final double PI = 3.14159265358979323846;
```