

A vertical decorative strip on the left side of the slide, featuring a close-up of autumn leaves in shades of red, orange, and yellow.

Java Programming for android

Dieudonne U



Object–Oriented Programming

- object-oriented concepts
 - classes
 - objects
 - encapsulation
 - inheritance
 - Polymorphism
 - Interface and abstract

Objects and Classes

- *Classes serve as “blueprints” that describe the states and behaviors of objects, which are actual “instances” of classes*
- For example, a Vehicle class describes a motor vehicle’s blueprint:
 - States: “on/off”, driver in seat, fuel in tank, speed, etc.
 - Behaviors: startup, shutdown, drive “forward”, shift transmission, etc.
- There are many possible Vehicles, e.g., Honda Accord, Mack truck, etc. These are *instances of the Vehicle blueprint*
- Many Vehicle states are specific to each Vehicle object, e.g., on/off, driver in seat, fuel remaining.
- Other states are specific to the class of Vehicles, not any particular Vehicle (e.g., keeping track of the “last” Vehicle ID # assigned). These correspond to *instance fields and static fields in a class*.
- Notice: we can operate a vehicle without knowing its implementation “under the hood”.
- Similarly, a class makes public *instance methods by which objects* of this class can be manipulated. Other methods apply to the set of all Vehicles (e.g., set min. fuel economy). These correspond to *static methods in a class*

Objects and Classes

```
1 public class Vehicle
2 {
3     // Instance fields
4     private boolean isOn = false;
5     private boolean isDriverInSeat = false;
6     private double fuelInTank = 10.0;
7     private double speed = 0.0;
8
9     // Static fields
10    private static String lastVin = "4A4AP3AU*DE999998";
11
12    // Instance methods (some omitted for brevity)
13    public Vehicle() { ... }
14
15    // Constructor
16    public void startUp() { ... }
17    public void shutOff() { ... }
18    public void getIsDriverInSeat() { ... }
19
20    //getter, setter methods
21    public void setIsDriverInSeat() {...}
22    private void manageMotor(){...} //More private methods ...
23
24    //Static methods public
25    static void setVin(String newVin){...}
26 }
```

Objects and Classes

Objects




Class

```
Mobile.java
1 package oopsconcept;
2
3 public class Mobile {
4     private String manufacturer;
5     private String operating_system;
6     public String model;
7     private int cost;
8
9     //Constructor to set properties/characteristics of object
10    Mobile(String man, String o,String m, int c){
11        this.manufacturer = man;
12        this.operating_system=o;
13        this.model=m;
14        this.cost=c;
15    }
16    //Method to get access Model property of Object
17    public String getModel(){
18        return this.model;
19    }
20    // We can add other method to get access to other properties
21 }
22
```



Encapsulation

- Encapsulation means putting together all the variables (instance variables) and the methods into a single unit called Class.
- It also means hiding data and methods within an Object.
- Encapsulation provides the security that keeps data and methods safe from inadvertent changes.
- Programmers sometimes refer to encapsulation as using a “black box,” or a device that you can use without regard to the internal mechanisms.
- A programmer can access and use the methods and data contained in the black box but cannot change them.
- Below example shows Mobile class with properties, which can be set once while creating object using constructor arguments.
- Properties can be accessed using getXXX() methods which are having public access modifiers.



```
package oopsconcept;
public class Mobile {
    private String manufacturer;
    private String operating_system;
    public String model;
    private int cost;

    //Constructor to set properties/characteristics of object
    Mobile(String man, String o,String m, int c){
        this.manufacturer = man;
        this.operating_system=o;
        this.model=m;
        this.cost=c;
    }
    //Method to get access Model property of Object
    public String getModel(){
        return this.model;
    }
    // We can add other method to get access to other properties
}
```



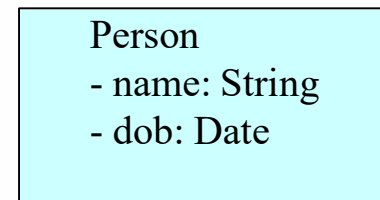
Inheritance terms and Terminology

- **superclass, base class, parent class:** terms to describe the parent in the relationship, which shares its functionality
- **subclass, derived class, child class:** terms to describe the child in the relationship, which accepts functionality from its parent
- **extend, inherit, derive:** become a subclass of another class

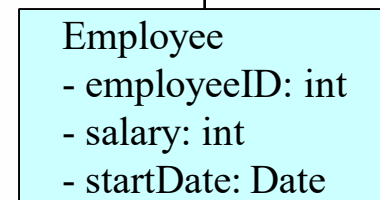
Inheritance terms and Terminology

- Inheritance is a fundamental Object Oriented concept
- A class can be defined as a "subclass" of another class.
 - The subclass inherits all data attributes of its superclass
 - The subclass inherits all methods of its superclass
 - The subclass inherits all associations of its superclass
- The subclass can:
 - Add new functionality
 - Use inherited functionality
 - Override inherited functionality

superclass:



subclass:





Inheritance: Definition

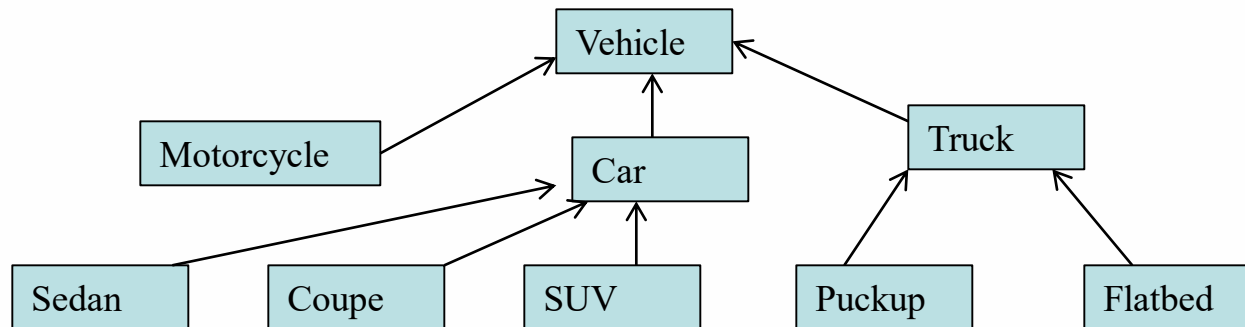
- **inheritance: a parent-child relationship between classes**
- **allows sharing of the behavior of the parent class into its child classes**
 - **one of the major benefits of object-oriented programming (OOP) is this code sharing between classes through inheritance**
- **child class can add new behavior or override existing behavior from parent**

Inheritance in Java

- in Java, you specify another class as your parent by using the keyword `extends`
 - `public class CheckingAccount
 extends BankAccount {`
 - the objects of your class will now receive all of the state (fields) and behavior (methods) of the parent class
 - constructors and static methods/fields are not inherited
 - by default, a class's parent is `Object`
- Java forces a class to have exactly one parent ("single inheritance")
 - other languages (C++) allow multiple inheritance

Inheritance Example

- Types of Vehicles:
 - Motorcycle, Car, Truck, etc.
- Types of Cars:
 - Sedan, Coupe, SUV. Types of Trucks: Pickup, Flatbed.
- Subclasses inherit fields/methods from superclasses.
- Subclasses can add new fields/methods, override those of parent classes
- For example, Motorcycle's `driveForward()` method differs from Truck's `driveForward()` method



Inheritance Other Example

```
class BankAccount {  
    private double myBal;  
    public BankAccount() { myBal = 0; }  
    public double getBalance() { return myBal; }  
}
```

```
class CheckingAccount extends BankAccount {  
    private double myInterest;  
    public CheckingAccount(double interest) { }  
    public double getInterest() { return myInterest;  
    }  
    public void applyInterest() { }  
}
```

- CheckingAccount objects have myBal and myInterest fields, and getBalance(), getInterest(), and applyInterest() methods



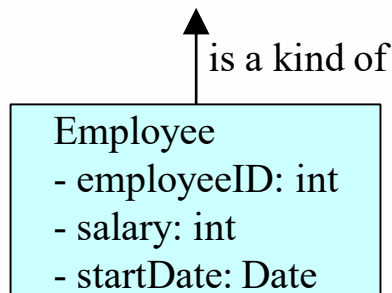
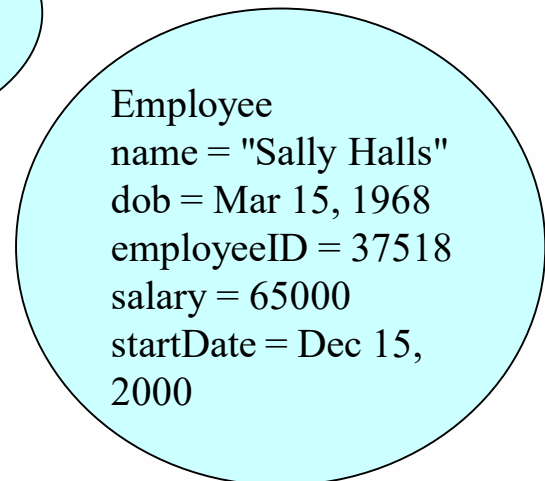
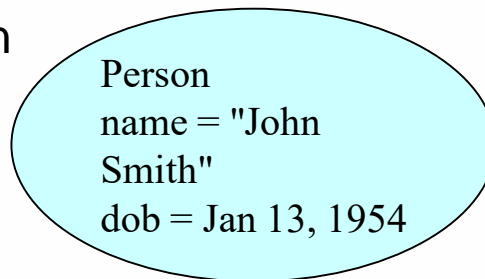
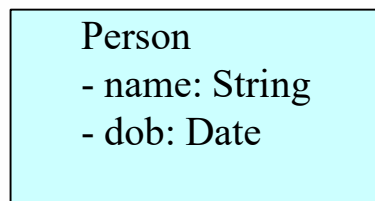
Inheritance Other Example

The objectives are:

- To explore the concept and implications of inheritance
 - Polymorphism
- To define the syntax of inheritance in Java
- To understand the class hierarchy of Java
- To examine the effect of inheritance on constructors

What really happens?

- When an object is created using new, the system must allocate enough memory to hold all its instance variables.
 - This includes any inherited instance variables
- In this example, we can say that an Employee "is a kind of" Person.
 - An Employee object inherits all of the attributes, methods and associations of Person



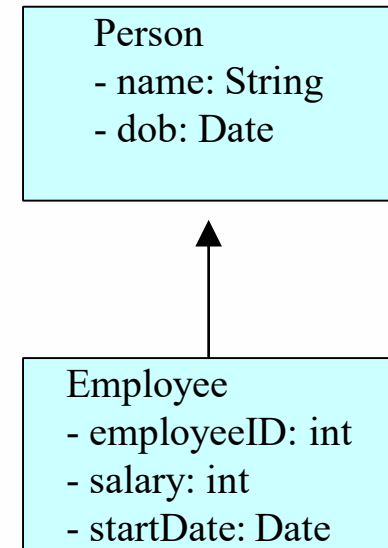
Inheritance in Java

- Inheritance is declared using the "extends" keyword
 - If inheritance is not defined, the class extends a class called Object

```
public class Person
{
    private String name;
    private Date dob;
    [...]
```

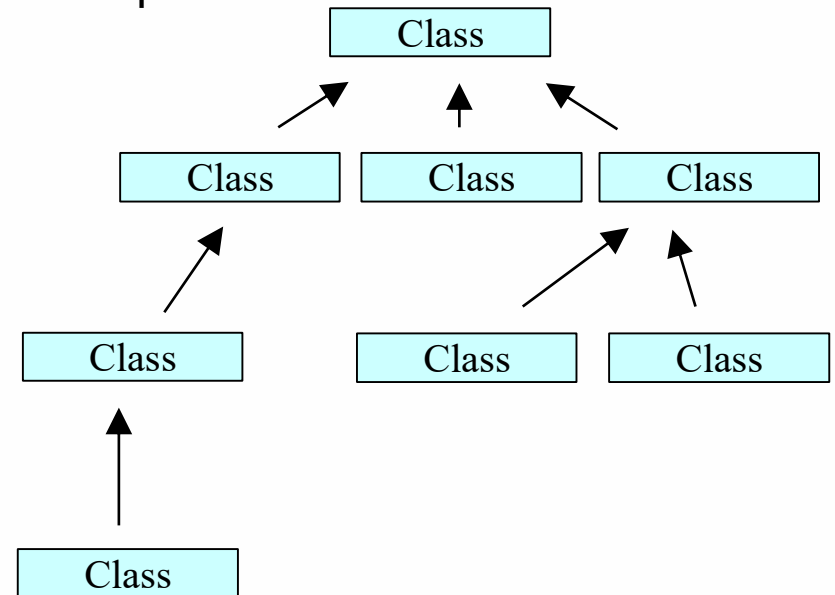
```
public class Employee extends Person
{
    private int employeeID;
    private int salary;
    private Date startDate;
    [...]
```

```
Employee anEmployee = new Employee();
```



Inheritance Hierarchy

- Each Java class has one (and only one) superclass.
 - C++ allows for multiple inheritance
- Inheritance creates a class hierarchy
 - Classes higher in the hierarchy are more general and more abstract
 - Classes lower in the hierarchy are more specific and concrete
- There is no limit to the number of subclasses a class can have
- There is no limit to the depth of the class tree.





Protected Members

- In a superclass
 - public members
 - Accessible anywhere program has a reference to a superclass or subclass type
 - private members
 - Accessible only in methods of the superclass
 - protected members
 - Intermediate protection between private and public
 - Only accessible by methods of superclass, of subclass, or classes in the same package
- Subclass methods
 - Can refer to public or protected members by name
 - Overridden methods accessible with `super.methodName`



Relationship between Superclass Objects and Subclass Objects

- Object of subclass
 - Can be treated as object of superclass
 - Reverse not true
 - Suppose many classes inherit from one superclass
 - Can make an array of superclass references
 - Treat all objects like superclass objects
 - Explicit cast
 - Convert superclass reference to a subclass reference (downcasting)
 - Can only be done when superclass reference actually referring to a subclass object
 - instanceof operator
 - if (p instanceof Circle)
 - Returns true if the object to which p points "is a" Circle

"Has-a" Relationships

- "Has-a" relationship: when one object contains another as a field

```
public class BankAccountManager {  
    private List myAccounts;  
    // ...  
}
```

- a BankAccountManager object "has-a" List inside it, and therefore can use it

"Is-a" relationships

- "Is-a" relationships represent sets of abilities; implemented through interfaces and inheritance

```
public class CheckingAccount  
  extends BankAccount {  
    // ...  
  }
```

- a CheckingAccount object "is-a" BankAccount
 - therefore, it can do anything an BankAccount can do
 - it can be substituted wherever a BankAccount is needed
 - a variable of type BankAccount may refer to a CheckingAccount object

The class called Object

- At the very top of the inheritance tree is a class called Object
- All Java classes inherit from Object.
 - All objects have a common ancestor
 - This is different from C++
- The Object class is defined in the java.lang package
 - Examine it in the Java API Specification

Object

Constructors and Initialization

- Classes use constructors to initialize instance variables
 - When a subclass object is created, its constructor is called.
 - It is the responsibility of the subclass constructor to invoke the appropriate superclass constructors so that the instance variables defined in the superclass are properly initialized
- Superclass constructors can be called using the "super" keyword in a manner similar to "this"
 - It must be the first line of code in the constructor
- If a call to super is not made, the system will automatically attempt to invoke the no-argument constructor of the superclass.

Constructors – Example

```
public class BankAccount
{
    private String ownersName;
    private int accountNumber;
    private float balance;

    public BankAccount(int anAccountNumber, String aName)
    {
        accountNumber = anAccountNumber;
        ownersName = aName;
    }
    [...]
}

public class OverdraftAccount extends BankAccount
{
    private float overdraftLimit;

    public OverdraftAccount(int anAccountNumber, String aName, float
aLimit)
    {
        super(anAccountNumber, aName);
        overdraftLimit = aLimit;
    }
}
```

Method Overriding

- Subclasses inherit all methods from their superclass
 - Sometimes, the implementation of the method in the superclass does not provide the functionality required by the subclass.
 - In these cases, the method must be overridden.
- To override a method, provide an implementation in the subclass.
 - The method in the subclass **MUST** have the exact same signature as the method it is overriding.

Method overriding - Example

```
public class BankAccount
{
    private String ownersName;
    private int accountNumber;
    protected float balance;

    public void deposit(float anAmount)
    {
        if (anAmount>0.0)
            balance = balance + anAmount;
    }

    public void withdraw(float anAmount)
    {
        if ((anAmount>0.0) && (balance>anAmount))
            balance = balance - anAmount;
    }

    public float getBalance()
    {
        return balance;
    }
}
```


Method overriding - Example

```
public class OverdraftAccount extends BankAccount
{
    private float limit;

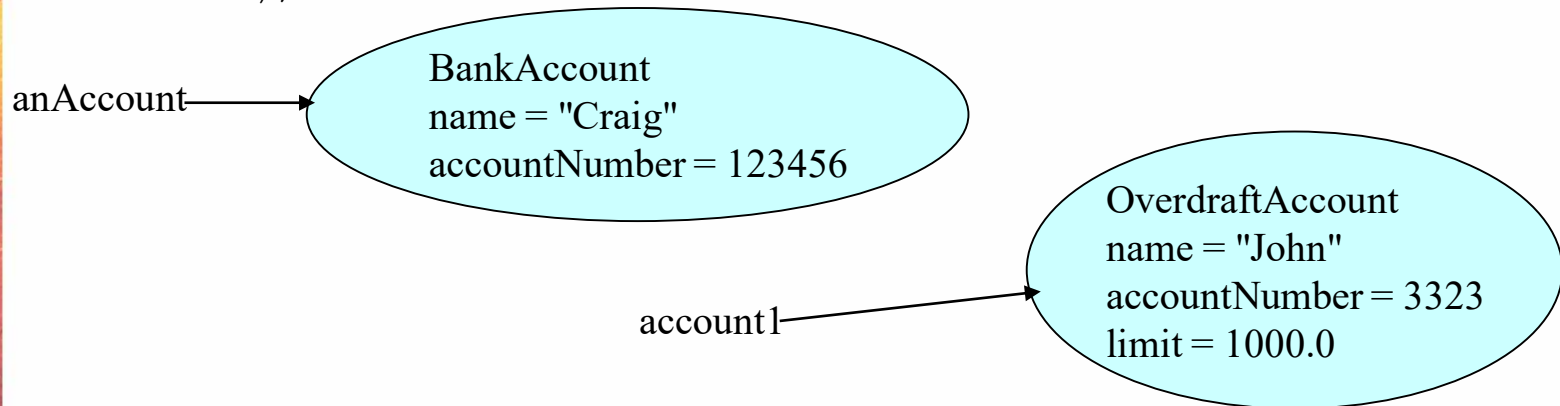
    public void withdraw(float anAmount)
    {
        if ((anAmount>0.0) &&
            (getBalance()+limit>anAmount))
            balance = balance - anAmount;
    }
}
```

Object References and Inheritance

- Inheritance defines "a kind of" relationship.
 - In the previous example, OverdraftAccount "is a kind of" BankAccount
- Because of this relationship, programmers can "substitute" object references.
 - A superclass reference can refer to an instance of the superclass OR an instance of ANY class which inherits from the superclass.

```
BankAccount anAccount = new BankAccount(123456, "Craig");
```

```
BankAccount account1 = new OverdraftAccount(3323, "John",  
1000.0);
```



Polymorphism

- In Core, Java Polymorphism is one of easy concept to understand.
- Polymorphism definition is that Poly means many and morphos means forms.
- It describes the feature of languages that allows the same word or symbol to be interpreted correctly in different situations based on the context.
- There are two types of Polymorphism available in Java.
- For example, in English, the verb “run” means different things if you use it with “a footrace,” a “business,” or “a computer.” You understand the meaning of “run” based on the other words used with it.
- Object-oriented programs are written so that the methods having the same name works differently in different context. Java provides two ways to implement polymorphism



Static Polymorphism (compile time polymorphism/ Method overloading):

The ability to execute different method implementations by altering the argument used with the method name is known as method overloading.

In below program, we have three print methods each with different arguments.

When you properly overload a method, you can call it providing different argument lists, and the appropriate version of the method executes.

Static Polymorphism (compile time polymorphism/ Method overloading):

```
package oopsconcept;
class Overloadsample {
    public void print(String s){
        System.out.println("First Method with only String- "+ s);
    }
    public void print (int i){
        System.out.println("Second Method with only int- "+ i);
    }
    public void print (String s, int i){
        System.out.println("Third Method with both- "+ s + "--" + i);
    }
}
public class PolymDemo {
    public static void main(String[] args) {
        Overloadsample obj = new Overloadsample();
        obj.print(10);
        obj.print("Amit");
        obj.print("Hello", 100);
    }
}
```



Dynamic Polymorphism (run time polymorphism/ Method Overriding)

When you create a subclass by extending an existing class, the new subclass contains data and methods that were defined in the original superclass.

In other words, any child class object has all the attributes of its parent.

Sometimes, however, the superclass data fields and methods are not entirely appropriate for the subclass objects; in these cases, you want to override the parent class members. Let's take the example used in inheritance explanation.

Dynamic Polymorphism (run time polymorphism/ Method Overriding)

```
package oopsconcept;
public class OverridingDemo {
    public static void main(String[] args) {

        //Creating Object of SuperClass and calling getModel Method
        Mobile m = new Mobile("Nokia", "Win8", "Lumia",10000);
        System.out.println(m.getModel());

        //Creating Object of Sublcass and calling getModel Method
        Android a = new Android("Samsung", "Android", "Grand",30000);
        System.out.println(a.getModel());

        //Creating Object of Sublcass and calling getModel Method
        Blackberry b = new Blackberry("BlackB", "RIM", "Curve",20000);
        System.out.println(b.getModel());
    }
}
```

Abstraction

- All programming languages provide abstractions. It can be argued that the complexity of the problems you're able to solve is directly related to the kind and quality of abstraction.
- An essential element of object-oriented programming is an abstraction.
- Humans manage complexity through abstraction.
- When you drive your car you do not have to be concerned with the exact internal working of your car(unless you are a mechanic).
- What you are concerned with is interacting with your car via its interfaces like steering wheel, brake pedal, accelerator pedal etc.
- Various manufacturers of car have different implementation of the car working but its basic interface has not changed (i.e. you still use the steering wheel, brake pedal, accelerator pedal etc to interact with your car). Hence the knowledge you have of your car is abstract.



Abstraction

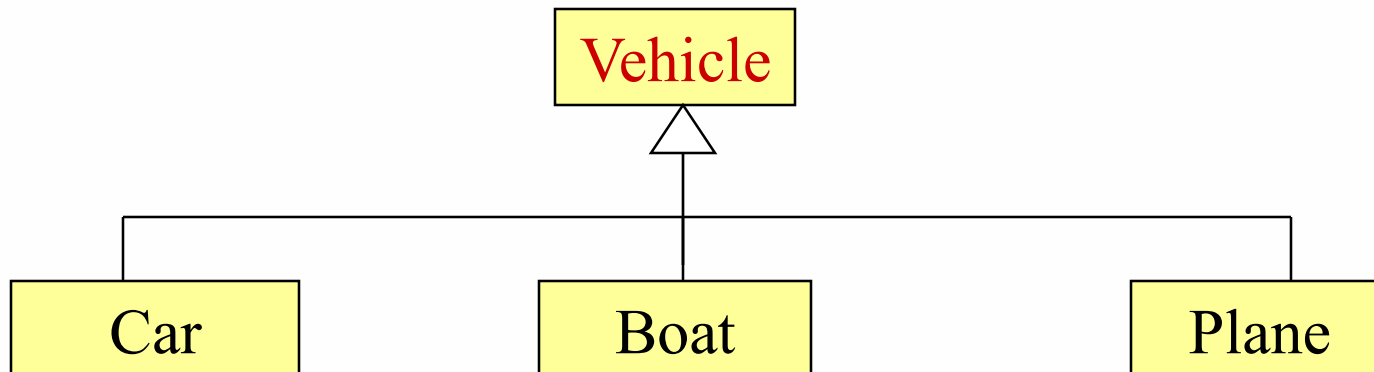
- A powerful way to manage abstraction is through the use of hierarchical classifications.
- This allows you to layer the semantics of complex systems, breaking them into more manageable pieces.
- From the outside, a car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on.
- In turn, each of these subsystems is made up of more specialized units. For instance, the sound system consists of a radio, a CD player, and/or a tape player.
- The point is that you manage the complexity of the car (or any other complex system) through the use of hierarchical abstractions.

Abstraction

- An abstract class is something which is incomplete and you can not create an instance of the abstract class.
- If you want to use it you need to make it complete or concrete by extending it.
- A class is called concrete if it does not contain any abstract method and implements all abstract method inherited from abstract class or interface it has implemented or extended.
- Java has a concept of abstract classes, abstract method but a variable can not be abstract in Java.
- Let's take an example of Java Abstract Class called Vehicle. When I am creating a class called Vehicle, I know there should be methods like start() and Stop() but don't know start and stop mechanism of every vehicle since they could have different start and stop mechanism
 - e.g some can be started by a kick or some can be by pressing buttons.
- The advantage of Abstraction is if there is a new type of vehicle introduced we might just need to add one class which extends Vehicle Abstract class and implement specific methods.

Abstract Classes

- Java allows **abstract** classes
 - use the modifier abstract on a class header to declare an abstract class
abstract class Vehicle
{ ... }
- An abstract class is a placeholder in a class hierarchy that represents a generic concept



Abstract Classes

- An abstract class often contains *abstract methods*, though it doesn't have to
 - Abstract methods consist of only methods *declarations*, without any method body
- The non-abstract child of an abstract class must override the abstract methods of the parent
- An abstract class cannot be instantiated (why?)
- The use of abstract classes is a design decision; it helps us establish common elements in a class that is too general to instantiate

abstract class

```
package oopsconcept;
public abstract class VehicleAbstract {
    public abstract void start();
    public void stop(){
        System.out.println("Stopping Vehicle in abstract class");
    }
}

class TwoWheeler extends VehicleAbstract{
    @Override
    public void start() {
        System.out.println("Starting Two Wheeler");
    }
}

class FourWheeler extends VehicleAbstract{
    @Override
    public void start() {
        System.out.println("Starting Four Wheeler");
    }
}

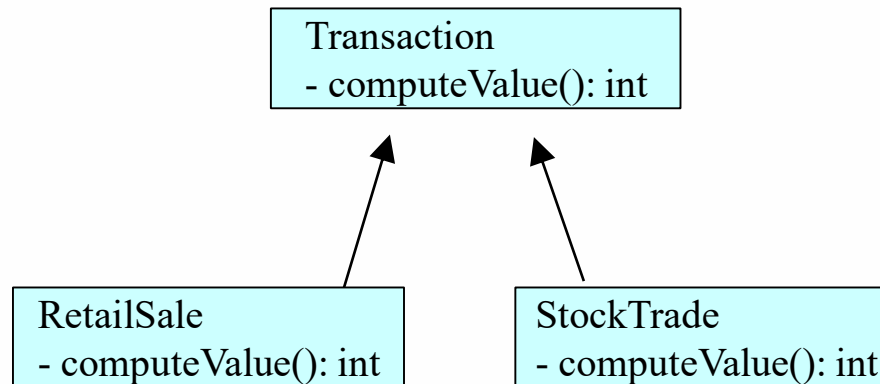
package oopsconcept;
public class VehicleTesting {
    public static void main(String[] args) {
        VehicleAbstract my2Wheeler = new TwoWheeler();
        VehicleAbstract my4Wheeler = new FourWheeler();
        my2Wheeler.start();
        my2Wheeler.stop();
        my4Wheeler.start();
        my4Wheeler.stop();
    }
}
```


Abstract Methods

- Methods can also be abstracted
 - An abstract method is one to which a signature has been provided, but no implementation for that method is given.
 - An Abstract method is a placeholder. It means that we declare that a method must exist, but there is no meaningful implementation for that methods within this class
- Any class which contains an abstract method **MUST** also be abstract
 - Any class which has an incomplete method definition cannot be instantiated (ie. it is abstract)
- Abstract classes can contain both concrete and abstract methods.
 - If a method can be implemented within an abstract class, and implementation should be provided.

Abstract Method Example

- In the following example, a Transaction's value can be computed, but there is no meaningful implementation that can be defined within the Transaction class.
- How a transaction is computed is dependent on the transaction's type
- Note: This is polymorphism.



Defining Abstract Methods

- Inheritance is declared using the "extends" keyword
 - If inheritance is not defined, the class extends a class called Object

```
public abstract class Transaction
{
    public abstract int computeValue();
}
```

Note: no implementation

Transaction
- computeValue(): int

```
public class RetailSale extends Transaction
{
    public int computeValue()
    {
        [...]
    }
}
```

RetailSale
- computeValue(): int

StockTrade
- computeValue(): int

```
public class StockTrade extends Transaction
{
    public int computeValue()
    {
        [...]
    }
}
```

Java Interface

- A Java *interface* is a collection of **constants** and **abstract methods**
 - abstract method: a method header without a method body; we declare an abstract method using the modifier abstract
 - since all methods in an interface are abstract, the abstract modifier is usually left off
- Methods in an interface have public visibility by default

Interface: Syntax

```
public interface Doable
{
    public static final String NAME;

    public void doThis();
    public int doThat();
    public void doThis2 (float value, char ch);
    public boolean doTheOther (int num);
}
```



Implementing an Interface

- A class formally implements an interface by
 - stating so in the class header in the implements clause
 - a class can implement multiple interfaces: the interfaces are listed in the implements clause, separated by commas
- If a class asserts that it implements an interface, it must define all methods in the interface or the compiler will produce errors

Implementing Interfaces

```
public class Something implements Doable
{
    public void doThis ()
    {
        // whatever
    }

    public void doThat ()
    {
        // whatever
    }

    // etc.
}
```


Interfaces: Examples from Java Standard Class Library

- A class that implements an interface can implement other methods as well
- The Java Standard Class library defines many interfaces:
 - the **Iterator** interface contains methods that allow the user to move through a collection of objects easily
 - **hasNext()**, **next()**, **remove()**
 - the **Comparable** interface contains an abstract method called **compareTo**, which is used to compare two objects

```
if (obj1.compareTo(obj2) < 0)
    System.out.println("obj1 is less than obj2");
```

Interfaces

- In Java, only single inheritance is permitted. However, Java provides a construct called an interface which can be implemented by a class.
- Interfaces are similar to abstract classes (we will compare the two soon).
- A class can implement any number of interfaces. In effect using interfaces gives us the benefit of multiple inheritance without many of its problems.
- Interfaces are compiled into bytecode just like classes.
- Interfaces cannot be instantiated.
- Can use interface as a data type for variables.
- Can also use an interface as the result of a cast operation.
- Interfaces can contain only abstract methods and constants.



Interface Hierarchies

- Inheritance can be applied to interfaces as well as classes
- One interface can be used as the parent of another
- The child interface inherits all abstract methods of the parent
- A class implementing the child interface must define all methods from both the parent and child interfaces
- Note that class hierarchies and interface hierarchies are distinct (they do not overlap)

Interfaces (cont)

- An interface can extend other interfaces with the following syntax:

```
Public interface interfaceID extends, interface2, interface3  
{  
    //constants/method signatures  
}
```

- Obviously, any class which implements a “sub-interface” will have to implement each of the methods contained in it’s “super-interfaces”



Interface vs. abstract class

	Interface	Abstract class
Fields	Only constants	Constants and variable data
Methods	No implementation allowed (no abstract modifier necessary)	Abstract or concrete

Interface vs. abstract class (cont)

	Interface	Abstract class
Inheritance	A subclass can implement many interfaces	A subclass can inherit only one class
	Can extend numerous interfaces	Can implement numerous interfaces
	Cannot extend a class	Extends one class

Final Methods and Final Classes

- Methods can be qualified with the final modifier
 - Final methods cannot be overridden.
 - This can be useful for security purposes.

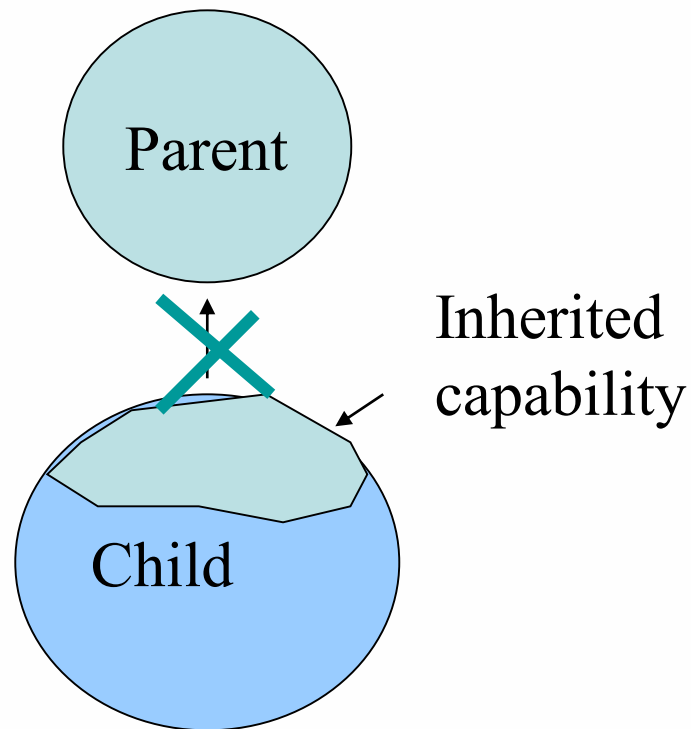
```
public final boolean validatePassword(String username, String  
    Password)  
{  
    [...]
```

- Classes can be qualified with the final modifier
 - The class cannot be extended
 - This can be used to improve performance. Because there'll be no subclasses, there will be no polymorphic overhead at runtime.

```
public final class Color  
{  
    [...]
```


Final and Abstract Classes

Restricting Inheritance



Final Members:

A way for Preventing Overriding of Members in Subclasses

- All methods and variables can be overridden by default in subclasses.
- This can be prevented by declaring them as final using the keyword “final” as a modifier. For example:
 - `final int marks = 100;`
 - `final void display();`
- This ensures that functionality defined in this method cannot be altered any. Similarly, the value of a final variable cannot be altered.

Final Classes:

A way for Preventing Classes being extended

- We can prevent an inheritance of classes by other classes by declaring them as final classes.
- This is achieved in Java by using the keyword final as follows:

```
final class Marks
```

```
{ // members  
}
```

```
final class Student extends Person
```

```
{ // members  
}
```

- Any attempt to inherit these classes will cause an error.

Java - Arrays

- Java provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type.
- An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.
- Instead of declaring individual variables, such as number0, number1, ..., and number99,
- You declare one array variable such as numbers and use numbers[0], **numbers[1]**, **and ...**, **numbers[99]** to represent individual variables.

```
double[] myList = new double[10]; or  
double[] myList = {1.9, 2.9, 3.4, 3.5};
```

```
// Print all the array elements for loop  
for (int i = 0; i < myList.length; i++) {  
    System.out.println(myList[i] + " ");  
}
```

```
// Print all the array elements with foreach  
for (double element: myList) {  
    System.out.println(element);  
}
```



Java - Collections

- The Collection interface is the foundation upon which the collections framework is built.
- It declares the core methods that all collections will have.
- These methods include:
add, addAll, clear, contains, equal, isEmpty, remove, size and more others

List interface

- The List interface extends **Collection** and declares the behavior of a collection that stores a sequence of elements.
- Elements can be inserted or accessed by their position in the list, using a zero-based index.
- A list may contain duplicate elements.
- In addition to the methods defined by **Collection**, List defines some of its own, which are summarized in the following table.

`void add(int index, Object obj)`

`Object get(int index)`

`Object remove(int index)`

`int indexOf(Object obj)`

List interface Implementations

- Since List is an interface you need to instantiate a concrete implementation of the interface in order to use it.
 - You can choose between the following List implementations in the Java Collections API:
 - `java.util.ArrayList`
 - `java.util.LinkedList`
 - `java.util.Vector`
 - `java.util.Stack`
-
- `List listA = new ArrayList();`
 - `List listB = new LinkedList();`
 - `List listC = new Vector();`
 - `List listD = new Stack();`

Adding and Accessing Elements

- To add elements to a **List** you call its **add()** method.
- This method is inherited from the **Collection** interface. Here are a few examples:

```
List listA = new ArrayList();  
listA.add("element 1");  
listA.add("element 2");  
listA.add("element 3");  
listA.add(0, "element 0");
```

- The first three **add()** calls add a **String** instance to the end of the list.
- The last **add()** call adds a **String** at index 0, meaning at the beginning of the list.

Adding and Accessing Elements

- By default you can put any **Object** into a **List**, but from Java 5, Java Generics makes it possible to limit the types of object you can insert into a **List**.

```
List<MyObject> list = new ArrayList<MyObject>();
```

```
List<String> list = new ArrayList<String>();
```

```
//add a fixed-size elements to list
```

```
List<String> messages = Arrays.asList("Hello", "World!", "How", "Are", "You");
```

```
//or simply use array and add it as a list
```

```
String[] myStrings = new String[] {"Elem1","Elem2","Elem3","Elem4","Elem5"};
```

```
List mylist = Arrays.asList(myStrings );
```

```
//accessing
```

```
for(String obj : list) {  
    }
```

Adding and Accessing Elements

- The order in which the elements are added to the **List** is stored, so you can access the elements in the same order.

```
List listA = new ArrayList();  
listA.add("element 0");  
listA.add("element 1");  
listA.add("element 2");
```

```
//access via index  
String element0 = listA.get(0);  
String element1 = listA.get(1);  
String element3 = listA.get(2);
```

- You can do so using either the **get(int index)** method, or via the **Iterator** returned by the **iterator()** method.

//access via new for-loop

```
for(Object object : listA) {  
    String element = (String) object;  
}
```

Clearing a List & List Size

- The Java **List** interface contains a **clear()** method which removes all elements from the list when called. Here is simple example of clearing a **List** with **clear()**:

```
List list = new ArrayList();  
list.add("object 1");  
list.add("object 2");  
//etc.  
list.clear();
```

- You can obtain the number of elements in the **List** by calling the **size()** method. Here is an

```
List list = new ArrayList();  
list.add("object 1");  
list.add("object 2");  
//etc.  
list.Size();
```

```
//using it in FOR loop  
for(int i=0; i < list.size(); i++) {  
    Object obj = list.get(i);  
}
```

ArrayList class

- The ArrayList class extends AbstractList and implements the List interface. ArrayList supports dynamic arrays that can grow as needed.
- Standard Java arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold.
- Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.
 - It implements all optional list operations and it also permits all elements, includes null.
 - It provides methods to manipulate the size of the array that is used internally to store the list.

```
ArrayList<Integer> list = new ArrayList<Integer>();
```

Review

- What is inheritance? What is a superclass? What is a subclass?
- Which class is at the top of the class hierarchy in Java?
- What are the constructor issues surrounding inheritance?
- What is method overriding? What is polymorphism? How are they related?
- What is a final method? What is a final class?