CMSC 132: Object-Oriented Programming II

Inheritance

Mustang vs Model T

Ford Mustang

Ford Model T

Interior: Mustang vs Model T

Frame: Mustang vs Model T

Mustang

Model T

Compaq: old and new

Price: US$3590
Weight: 28 pounds
CPU: Intel 8088, 4.77MHz
RAM: 128K, 640K max

Object Oriented Programming

- An Object-Oriented Language supports the following fundamental concepts:
  - Polymorphism
  - Inheritance
  - Encapsulation
  - Abstraction
  - Classes
  - Objects
  - Instance
  - Method

Object

- Objects have states and behaviors.
- Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating.
- An object is an instance of a class.
  - If we consider the real-world, we can find many objects around us, cars, dogs, humans, etc. All these objects have a state and a behavior.

Class

- A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.

```
public class Bicycle{
  public int gear;
  public int speed;
  public Bicycle(int startSpeed, int startGear) {
      gear = startGear;
      speed = startSpeed;
  }
  public void setGear(int v){gear = v;}
  public void applyBrake(int dec){speed -= dec;}
  public void speedUp(int inc) { speed += inc; }
}
```

Java Class Example

- Fraction Class
  - Numerator
  - Denominator
  - Reduce a Fraction to Lowest Terms
  - Addition, Multiplication
  - …

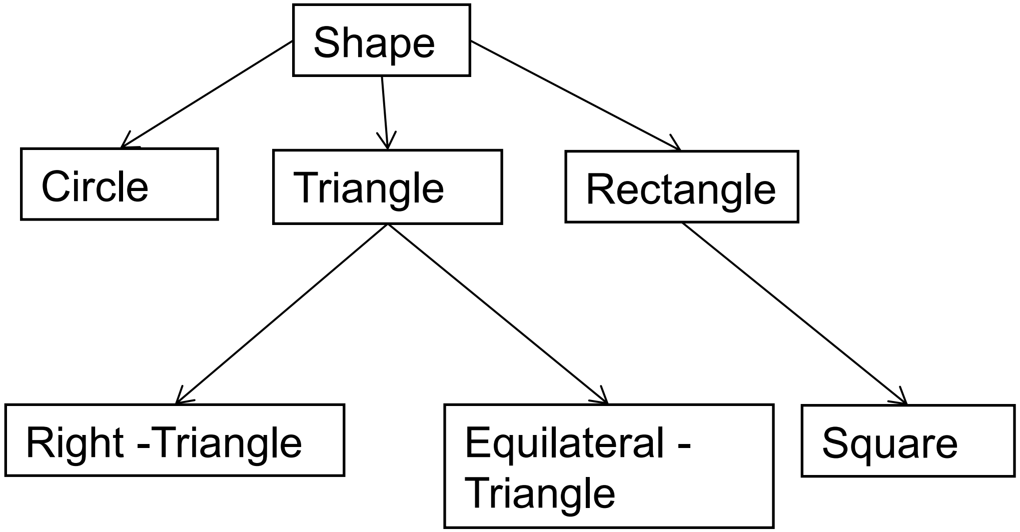
  - Now, let us implement the Fraction class.
  - Code will be posted on course site.

Inheritance

- Classes can be *derived* from other classes, thereby *inheriting* fields and methods from those classes.
- A class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*).
- The class from which the subclass is derived is called a *superclass* (also a *base class* or a *parent class*).
- Derived (Child) class can be base (parent) class

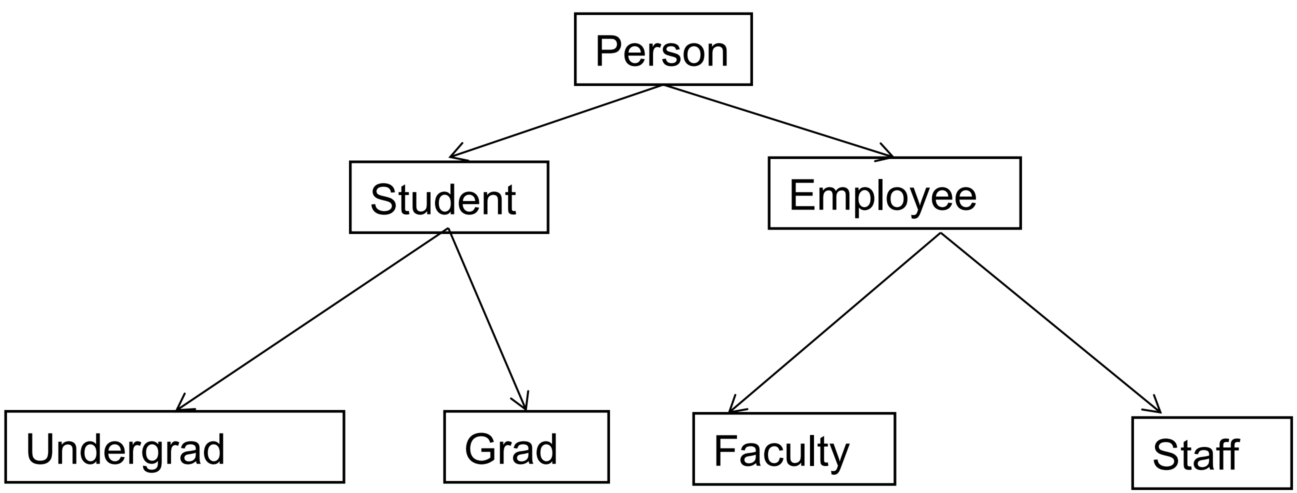
Inheritance

Motivation: In real life objects have a hierarchical structure:

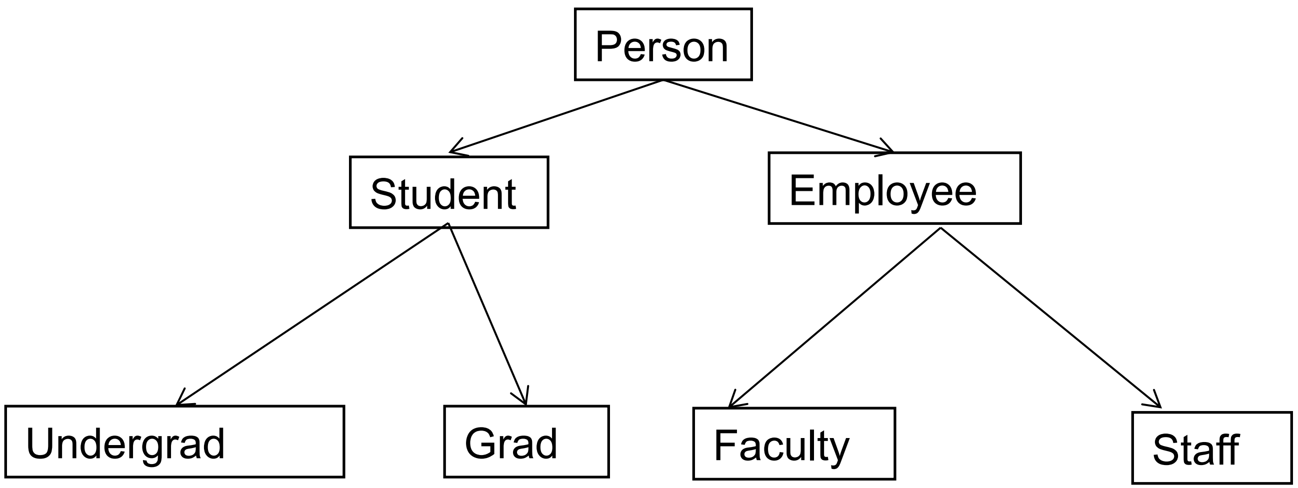


Inheritance

- Define a general class
- Later, define specialized classes based on the general class
- These specialized classes inherit properties from the general class



Inheritance



Person: name, address, phone, email
Student: college, major, gpa
Employee: Salary, dateHired, office
Faculty: rank, officeHours
Staff: title
Undergrad: freshman,sophomore, junior, or senior)
Grad: advisor, level (ms or phd)

Inheritance cont.

- What are some properties of a Person?
  - name, height, weight, age
- How about a Student?
  - ID, major, gpa
- Does a Student have a name, height, weight, and age?
  - Student inherits these properties from Person

is-a relationship

- This inheritance relationship is known as an is-a relationship

- A Grad student is a Student
- A Student is a Person.

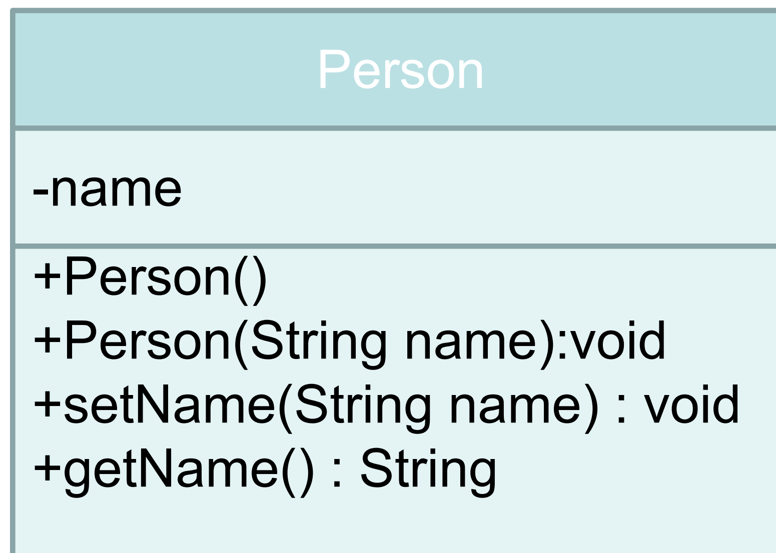
- Is a Person a Student? – Not necessarily!

Why inheritance is useful

- Enables you to define shared properties and actions once
- Derived classes can perform the same actions as base classes without having to redefine the actions
- If desired, the actions can be redefined – method overriding

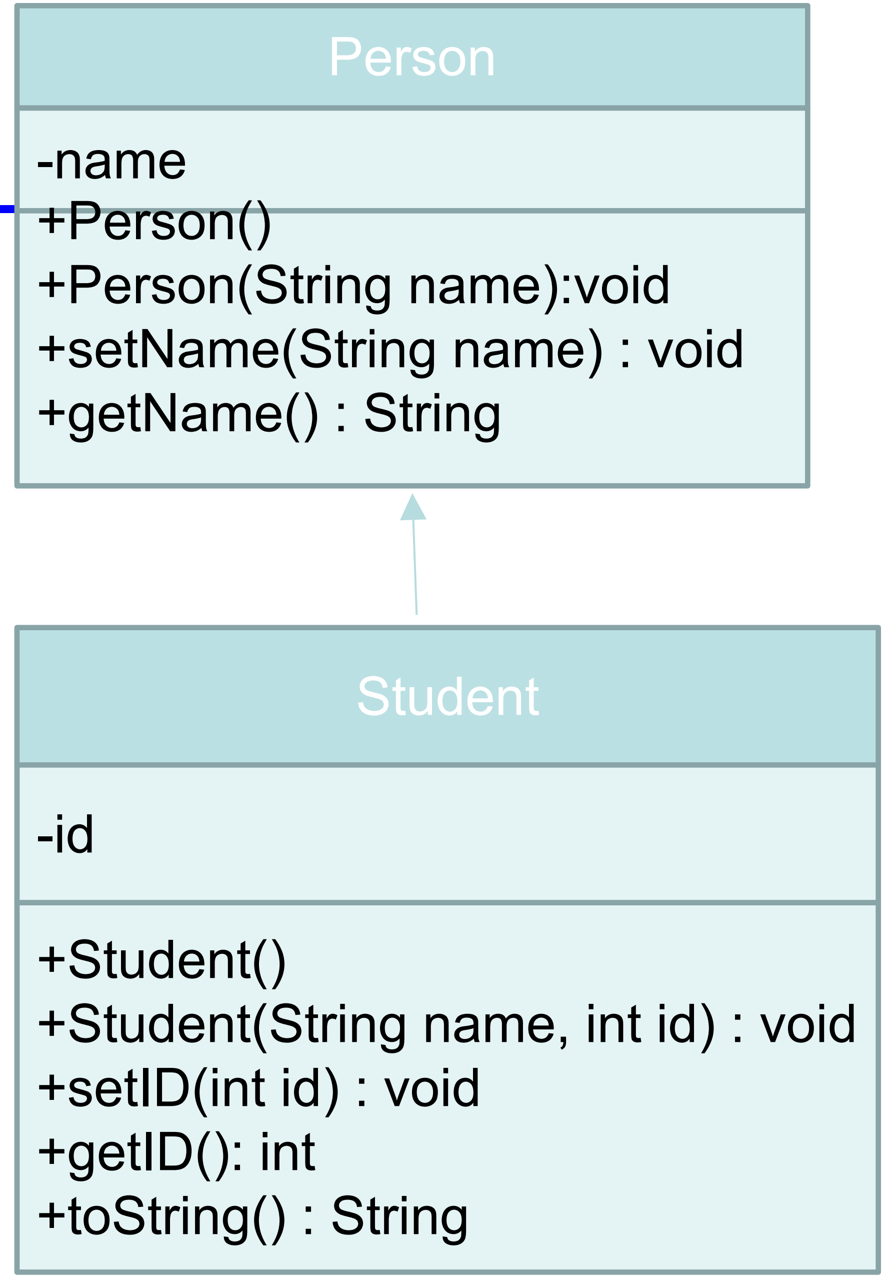
Person Class

```
public class Person {
    private String name;
    public Person(){
      name = "";
    }
    public Person(String name){
      this.name = name;
    }
    public void setName(String newName){
        name = newName;
    }
    public String getName(){
        return name;
    }
    @Override
    public String toString(){
        return "Name:"+name;
    }
}
```



Student Class

```
public class Student extends Person{
  private int id;
  public Student() {
    id = 0;
  }
  public Student(String name, int id) {
     super(name);
     this.id = id;
  }
  public void setID(int idNumber) {
     id = idNumber;
  }
  public int getID(){
      return id;
  }
  @Override
  public String toString(){
     return "Id:"+ id +"\tName:" +
                                getName();
  }
}
```



Dissecting the Student Class

- **Extends**: To specify that Student is a **derived class** (subclass) of Person we add the descriptor "extends" to the class definition:

```
public class Student extends Person {
  …
}
```

- Notice that a Student class
  - Inherits everything from the Person class
  - A Student IS-A Person (wherever a Person is needed, we can use a Student).

Super()

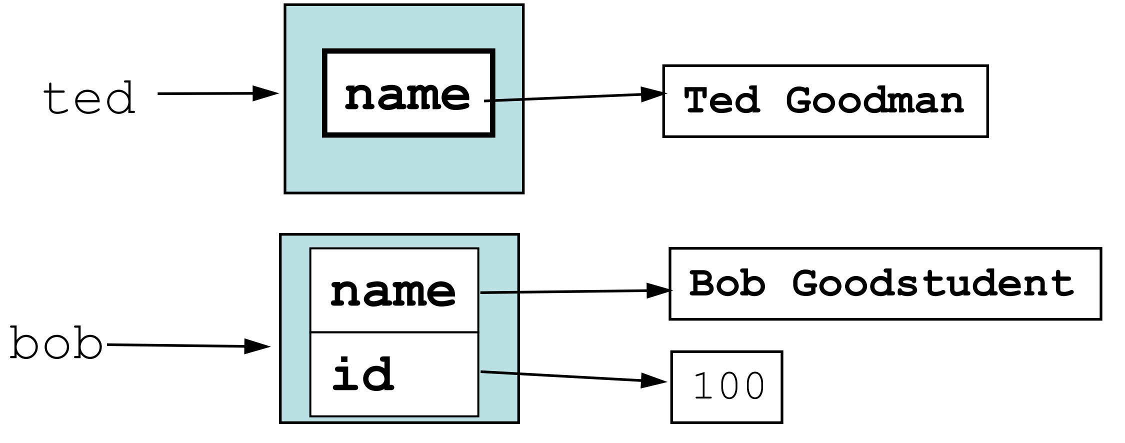
- **super( )**: When initializing a new Student object, we need to initialize its **base class** (or **superclass**). This is done by calling **super(…)**. For example, **super(name)** invokes the constructor **Person(name)**
  - super(…) must be the **first statement** of your constructor
  - If you **do not** call super( ), Java will automatically invoke the base class's **default constructor**
  - What if the base class's default constructor is **undefined**? **Error**
  - You must use "**super(…)**", not "**Person(…)**".

Memory Layout and Initialization Order

- When you create a new derived class object:
  - Java allocates space for **both** the **base class** instance variables and the **derived class** variables
  - Java initializes the **base class variables first**, and then initializes the derived class variables
- **Example**:

  Person ted = new Person("Ted Goodman");

  Student bob = new Student("Bob Goodstudent", 100);



Inheritance

- **Inheritance:** Since Student is derived from Person, a Student object can invoke any of the Person methods, it **inherits** them

```
Student bob = new Student("Bob Goodstudent", 100);

String bobsName = bob.getName( ) );

bob.setName( "Robert Goodstudent" );

System.out.println( "Bob's new info: " +
                                bob.toString( ) );
```

Inheritance

- **A Student "is a" Person**:
  - By inheritance a Student object is also a Person object. We can use a Student reference anywhere that a Person reference is needed

```
Person robert = bob;              // Okay: A Student is a Person
```

  - We cannot reverse this. (A Person need not be a Student.)

```
Student bob2 = robert; // Error! Cannot convert Person to Student
```

Overriding Methods

- **New Methods**: A derived class can define **entirely new** instance variables and new methods
- **Overriding**: A derived class can also **redefine existing** methods

```
public class Person {
  …
  public String toString() { … }
}
public class Student extends Person {
    …
    public String toString() { … }
}
Student bob = new Student( "Bob Goodstudent", 100);
System.out.println("Bob's info: " + bob);
```

> The derived class can redefine this method.

> Since bob is of type Student, this invokes the Student toString( )

Overriding and Overloading

- Don't confuse method **overriding** with method **overloading**.

  Overriding: occurs when a derived class defines a method with the **same name** and **parameters** as the base class.

  Overloading: occurs when two or more methods have the **same name**, but have **different parameters** (different signature).

  Example:

```
public class Person {
    public void setName(String n) { name = n; }
    …
}
public class Faculty extends Person {
    public void setName(String n) {
        super.setName("The Evil Professor " + n);
    }
    public void setName(String first, String last) {
        super.setName(first + " " + last);
    }
}
```

The base class defines a method setName( )

Overriding: Same name and parameters; different definition.

Overloading: Same name, but different parameters.

Quiz 1: Output of following program

```
class Test {
  int i;
}
class Main {
  public static void main(String args[]){
      Test t;
      System.out.println(t.i);
 }
}
```

A. 0
B. garbage value
C. compiler error
D. runtime error

Quiz 1: Output of following program

```
class Test {
  int i;
}
class Main {
  public static void main(String args[]){
      Test t;
      System.out.println(t.i);
 }
}
```

A. 0
B. garbage value
C. compiler error: variable not initialized.
D. runtime error

Quiz 2: Output of following program

```
class Test {
  int i;
}
class Main {
  public static void main(String args[]){
      Test t = null;
      System.out.println(t.i);
  }
}
```

A. 0
B. garbage value
C. compiler error
D. runtime error

Quiz 2: Output of following program

```
class Test {
  int i;
}
class Main {
  public static void main(String args[]){
      Test t = null;
      System.out.println(t.i);
 }
}
```

A. 0
B. garbage value
C. compiler error
D. runtime error: Null pointer exception

Quiz 3: Output of following program

```
class Base{
    void display() {System.out.print("Base ");}
}
class Child extends Base{
   void display(){System.out.print("Child ");}
}
Base b= new Base();
Child c =  new Child ();
Base ref = b;
ref.display();
ref = c;
ref.display();
```

A. Compilation error
B. Base Child
C. Child Base
D. Runtime error

Quiz 3: Output of following program

```
class Base{
    void display() {System.out.print("Base ");}
}
class Child extends Base{
   void display(){System.out.print("Child ");}
}
Base b= new Base();
Child c =  new Child ();
Base ref = b;
ref.display();
ref = c;
ref.display();
```

A. Compilation error
B. Base Child
C. Child Base
D. Runtime error

Overriding Variables: Shadowing

- **We can override methods, can we override instance variables too?**
- **Answer**: Yes, it is possible, but **not recommended**
  - Overriding an instance variable is called **shadowing**, because it makes the base instance variables of the base class inaccessible. (We can still access it explicitly using **super.varName**).

```
public class Person {
    String name;
    // …
}
```

```
public class Staff
  extends Person {
    String name;
    // …  name refers to
  Staff's name
}
```

  - This can be **confusing** to readers, since they may not have noticed that you redefined name. Better to just pick a new variable name

Shadowing example

```
class Base {
 public int x;
 public Base(){x = 10;}
 public String foo(){return x+"";}
}
```

```
class Derived extends Base {
 public int x;
 public Derived(){ x = 20;}
 public String foo(){return (x + "\t" + super.x);}
}
```

```
Derived d = new Derived();
d.foo();
```

Shadowing example

```
class Base {
 public int x;
 public Base(){x = 10;}
 public String foo(){return x+"";}
}
```

```
class Derived extends Base {
 public int x;
 public Derived(){ x = 20;}
 public String foo(){return (x + "\t" + super.x);}
}
```

```
Derived d = new Derived();
d.foo();
```

20 10

Shadowing example

```
class Base {
 public int x;
 public Base(){x = 10;}
 public void foo(){return x);}
}
```

```
class Derived extends Base {
 public int x;
 public Derived(){ x = 20;}
 public void foo(){return (x + "\t" + super.x);}
}
```

```
Derived d = new Derived();
Base b = d;
d.x;
b.x;
```

Shadowing example

```
class Base {
 public int x;
 public Base(){x = 10;}
 public String foo(){return x);}
}
```

```
class Derived extends Base {
 public int x;
 public Derived(){ x = 20;}
 public String foo(){return (x + "\t" + super.x);}
}
```

```
Derived d = new Derived();
Base b = d;
d.x;  20
b.x;  10
```

super and this

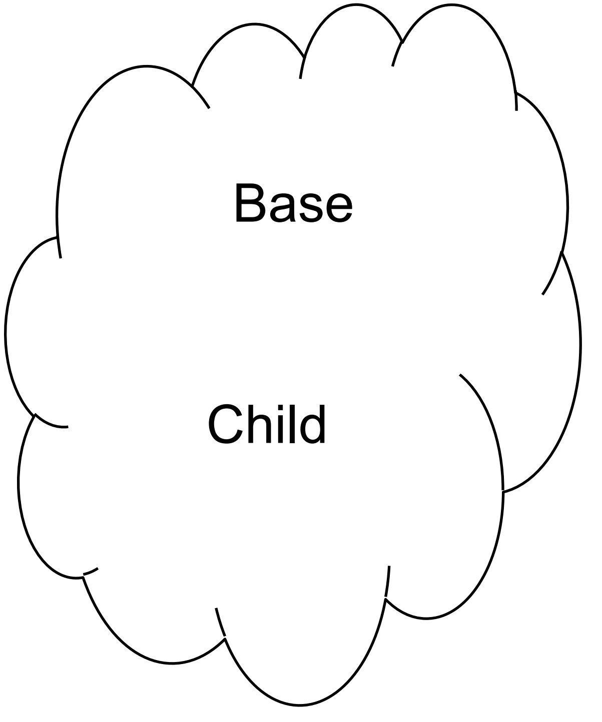
- **super**: refers to the base class object
  - We can invoke any base class constructor using **super(…)**.
  - We can access data and methods in the base class (Person) through **super**. E.g., toString( ) and equals( ) invoke the corresponding methods from the Person base class, using **super.toString( )** and **super.equals( )**.

- **this**: refers to the current object
  - We can refer to our own data and methods using "**this.**" but this usually is not needed
  - We can invoke any of our own constructors using **this(…)**. As with the super constructor, this can only be done **within a constructor**, and must be the **first statement** of the constructor. Example:

```
public Fraction(int n) {
    this(n,1);
}
```

Memory Layout

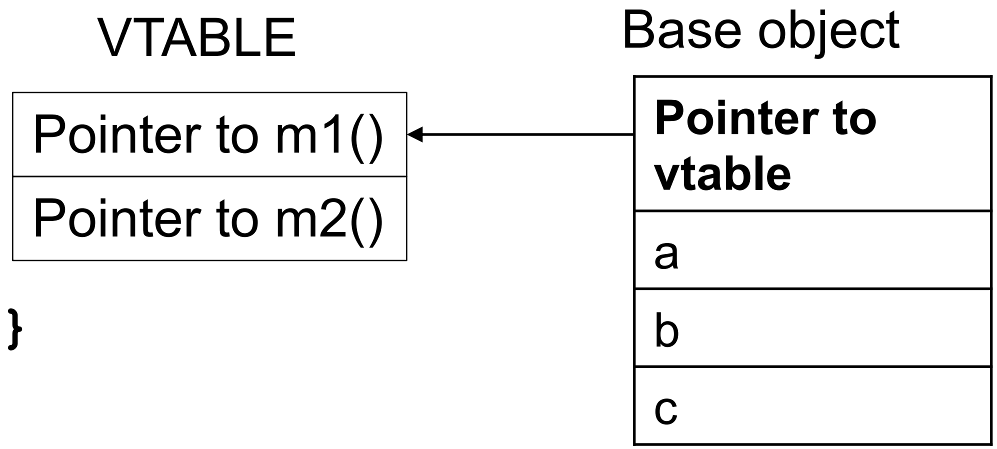
```
class Base{
  private int a;
  protected int b;
  protected int c;
  protected void m1(){}
  public void m2(){}
}

class Child extends Base{
  private int d;
  public void m1(){}
  public void m3(){}
}
```



The Java Virtual Machine does not mandate any particular internal structure for objects.

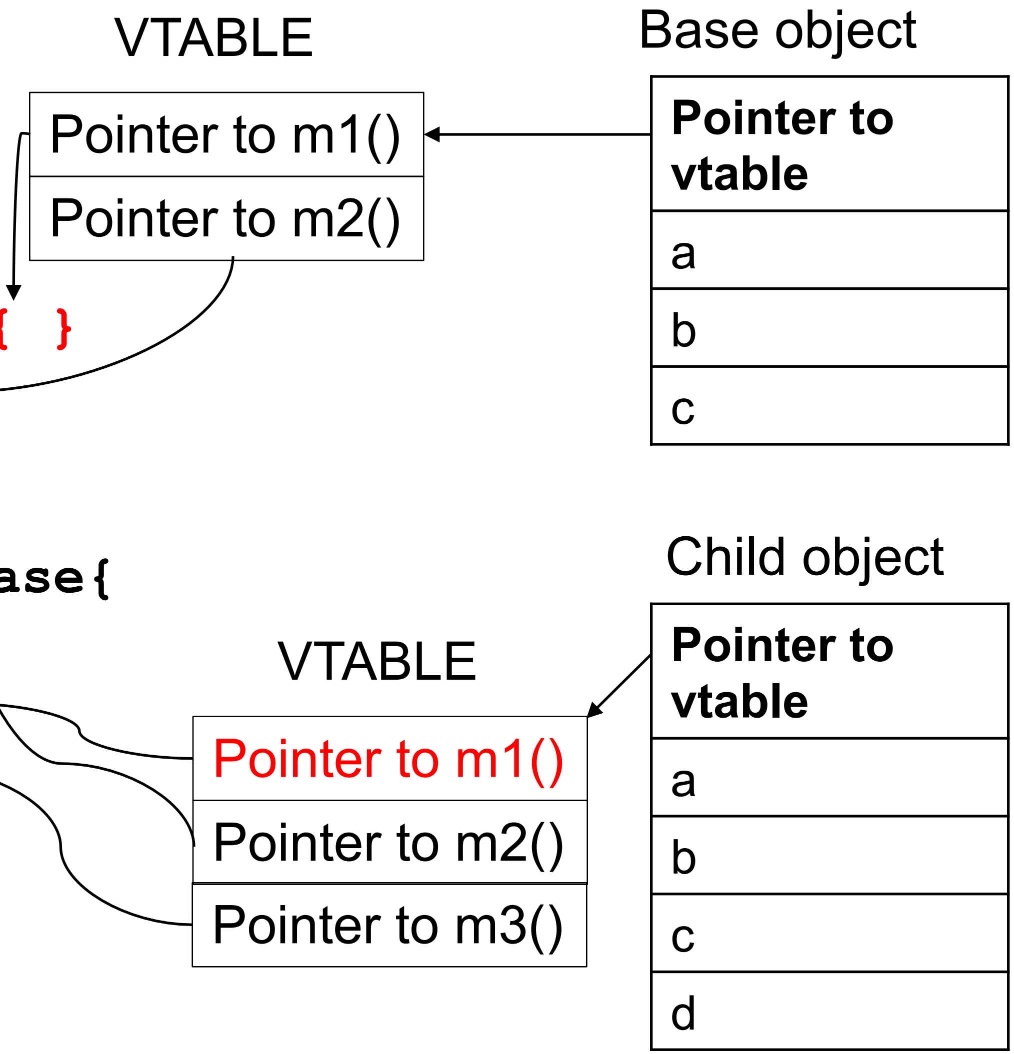
Memory Layout

```
class Base{
  private int a;
  protected int b;
  protected int c;
  protected void m1(){}
  public void m2(){}
}

class Child extends Base{
  private int d;
  public void m1(){}
  public void m3(){}
}
```



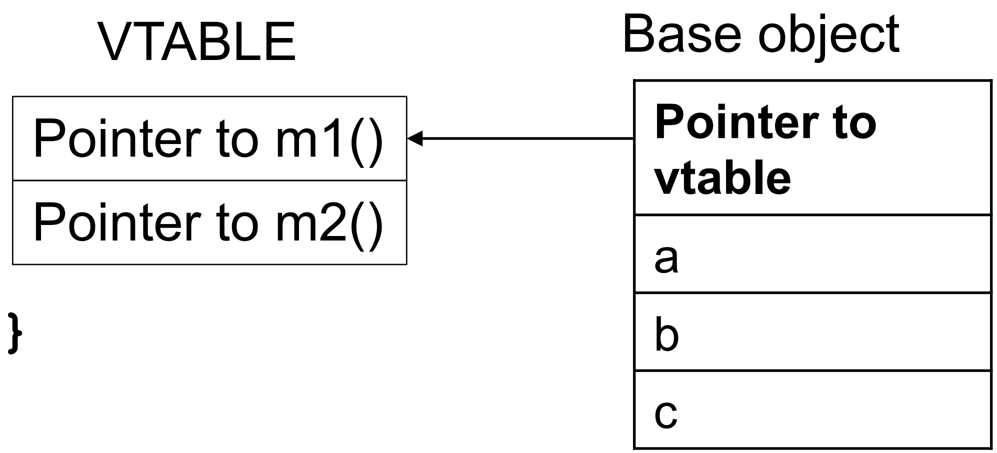
Memory Layout

```
class Base{
  private int a;
  protected int b;
  protected int c;
  protected void m1(){  }
  public void m2(){  }
}

class Child extends Base{
  private int d;
  public void m1(){  }
  public void m3(){  }
}
```

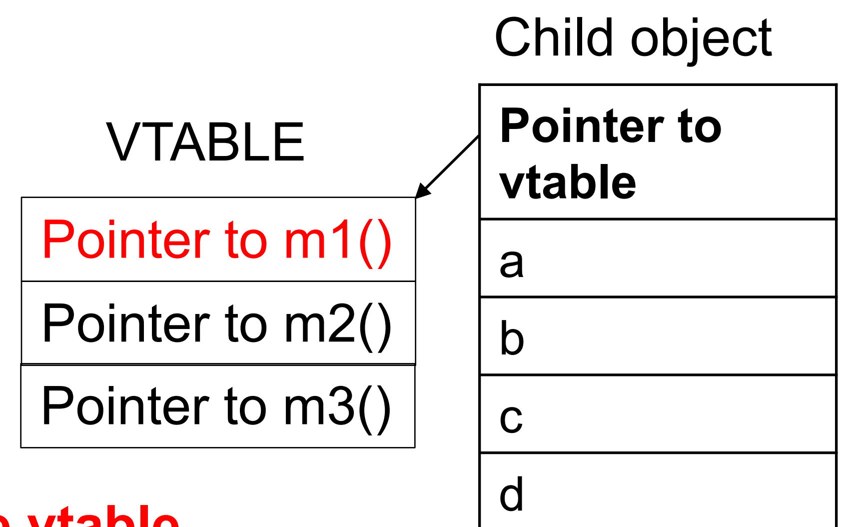


Memory Layout

```
class Base{
   private int a;
   protected int b;
   protected int c;
   protected void m1(){}
   public void m2(){}
}
```




```
class Child extends Base{
   private int d;
   public void m1(){}
   public void m3(){}
}
```



Each class has one vtable.

All objects of the this class shares the vtable.

Inheritance and Private

- **Private members**:
  - Child class **inherits all the private data** of Base class
  - However, **private members** of the base class **cannot** be accessed directly

- **Why is this?** After you have gone to all the work of setting up privacy, it wouldn't be fair to allow someone to simply **extend** your class and now have access to all the **private** information

Quiz 5: True/False

Except Object, which has no superclass, every class has one and only one direct superclass.

A. True
B. False

Quiz5: True/False

Except Object, which has no superclass, every class has one and only one direct superclass.

A. True
B. False

Quiz 6:

```
class Base {
  public void foo(){
    println("Base");
  }
}
class Derived extends Base {
  private void foo(){
    println("Derived");
  }
}
…
  Base b = new Derived();
  b.foo();
…
```

A. Base
B. Derived
C. Compiler Error
D. Runtime Error

Quiz 6:

```
class Base {
  public void foo(){
    println("Base");
  }
}
class Derived extends Base {
  private void foo(){
    println("Derived");
  }
}
…
  Base b = new Derived();
  b.foo();
```

A. Base
B. Derived
C. Compiler Error
D. Runtime Error

It is compiler error to give more restrictive access to a derived class function which overrides a base class function.

Quiz 7:

class Animal has a subclass Mammal. Which of the following is true:

A. Because of single inheritance, Mammal can have no subclasses.
B. Because of single inheritance, Mammal can have no other parent than Animal.
C. Because of single inheritance, Animal can have only one subclass.
D. Because of single inheritance, Mammal can have no siblings.

Quiz 7:

class Animal has a subclass Mammal. Which of the following is true:

A. Because of single inheritance, Mammal can have no subclasses.
B. Because of single inheritance, Mammal can have no other parent than Animal.
C. Because of single inheritance, Animal can have only one subclass.
D. Because of single inheritance, Mammal can have no siblings.

Access level

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

Object

- Object is the superclass of all java classes

- The class **Object** has no instance variables, but defines a number of methods. These include:

  ***toString( )**: returns a String representation of this object*

  equals(Object o): test for equality with another object o

- Every class you define should, overrides these two methods with something that makes sense for your class (hashCode method is also included in the group)

Early and Late Binding

- **Motivation**: Consider the following example:

```
Base b = new Child();
b.toString();
```

- **Q**: Should this call **Base's** toString or **Child's** toString?
- **A**: There are good arguments for either choice:
  - **Early (static) binding**: The variable b is **declared** to be of type **Base**. Therefore, we should call the Base's toString
  - **Late (dynamic) binding**: The object to which b refers was **created** as a "new **Child**". Therefore, we should call the Child's toString
  - **Pros and cons**: Early binding is more efficient, since the decision can be made at compile time. Late binding provides more flexibility
- **Java uses late binding** (by default): so Faculty toString is called (**Note**: C++ uses early binding by default.)

Polymorphism

- Java's **late binding** makes it possible for a single reference variable to refer to objects of many different types. Such a variable is said to be **polymorphic** (meaning having many forms)
- **Example**: Create an array of various university people and print

```
Shape[ ] list = new Shape[3];
list[0] = new Rect(10,20);
list[1] = new Circle (10);
list[2] = new Triangle(3,4,5)
for (int i = 0; i < list.length; i++ )
     System.out.println( list[i].getArea( ) );
```

Output:

- **What type is list[i]?** It can be a reference to any object that is derived from **Shape**. The appropriate **getArea** will be called

getClass and instanceof

- Objects in Java can access their type information **dynamically**
- **getClass( )**: Returns a representation of the class of any object

```
Person bob = new Person( … );
Person ted = new Student( … );

if ( bob.getClass( ) == ted.getClass( ) ) // false (ted
  is really a Student)
```

- **instanceof**: You can determine whether one object is an instance of (e.g., derived from) some class using **instanceof**. Note that it is an **operator** (!) in Java, not a method call

Up-casting and Down-casting

- We have already seen that we can assign a derived class reference anywhere that a base class is expected
  - **Upcasting**: Casting a reference **to a base class** (casting up the inheritance tree). This is done **automatically** and is **always safe**
  - **Downcasting**: Casting a reference **to a derived class**. This may **not be legal** (depending on the actual object type). You can **force** it by performing an explicit cast

- Illegal downcasting results in a **ClassCastException** run-time error

Safe Downcasting

- Can we check for the **legality** of a cast before trying it?
- **A**: Yes, using **instanceof**.

```
For(s:Shape){
        if(s instanceof Circle){
                Circle c = (Circle)s;
                int r = c.getRadius();
        }
}
```

Only Circle has getRadius method

Disabling Overriding with "final"

- Sometimes you do not want to allow method overriding
  - **Correctness**: Your method only makes sense when applied to the base class. Redefining it for a derived class might break things
  - **Efficiency**: Late binding is less efficient than early binding. You know that no subclass will redefine your method. You can force early binding by disabling overriding
- We can disable overriding by declaring a method to be "**final**"

Disabling Overriding with "final"

- **final**: Has two meanings, depending on context:
  - Define **symbolic constants**:

    public static final int MAX_BUFFER_SIZE = 1000;

  - Indicate that a method **cannot be overridden by derived classes**

```
public class Parent {
   …
   public final void someMethod( ) { … }
}
```

```
Subclasses cannot
override this method
```

```
public class Child extends Parent {
   …
   public void someMethod( ) { … }
}
```

```
Illegal! someMethod is
final in base class.
```

Quiz 8

```
class Base {
  final public void show() {
    println("Base");
  }
}
class Derived extends Base {
  public void show() {
    println("Derived");
  }
}
class Main {
  public static void(String[] args){
    Base b = new Derived();
    b.show();
   }
}
```

A. Base
B. Derived
C. Compiler Error
D. Runtime Error

Quiz 8

```
class Base {
  final public void show() {
    println("Base");
  }
}
class Derived extends Base {
  public void show() {
    println("Derived");
  }
}

…
    Base b = new Derived();
    b.show();
…
```

A. Base
B. Derived
C. Compiler Error
D. Runtime Error

Final methods cannot be overridden. Compiler Error: overridden method is final

Quiz 9

```
class Base {
  public static void show() {
    println("Base”);
  }
}
class Derived extends Base {
  public static void show() {
    println("Derived");
  }
}
…
  Base b = new Derived();;
  b.show();
…
```

A. Base
B. Derived
C. Compiler Error

Quiz 9

```
class Base {
  public static void show() {
    println("Base”);
  }
}
class Derived extends Base {
  public static void show() {
    println("Derived");
  }
}
…
  Base b = new Derived();;
  b.show();
…
```

A. Base
B. Derived
C. Compiler Error

when a function is static, runtime polymorphism doesn't happen.

Abstract Class

- Abstract classes cannot be instantiated, but they can be subclassed.
- It may or may not include abstract methods.

```
public abstract class Shape {
  private String id;
  public Shape (String id) {this.id = id};
  public abstract double getArea();
  public String getId() {return id;}
}
```

This abstract method must be defined in a concrete subclass.

Abstract Class

```
public abstract class Shape {
  private String id;
  public Shape (String id) {this.id = id};
  public abstract double getArea();
  public String getId() {return id;}
}

public class Circle extends Shape {
  private double radius;
  public Circle (double r) {
    super("Circle"); radius = r;
  }
  double getArea(){return Math.PI * radius * radius;}
  public double getRadius() {return radius;}
  public void setRadius(double r) {radius = r}
}
```

Must implement

Inheritance versus Composition

- **Inheritance** is but one way to create a complex class from another. The other way is to explicitly have an instance variable of the given object type. This is called **composition**

```
Derive a new
class from
ObjA.
```

Common Object:

```
public class ObjA {
    public methodA( ) { … }
}
```

```
Add ObjA as an
instance variable.
```

Inheritance:

```
public class ObjB extends ObjA {
    …
    // call methodA( );
}
```

Composition:

```
public class ObjB {
    ObjA a;
    // call a.methodA( )
}
```

- **When should I use inheritance vs. Composition?**
  - **ObjB "is a" ObjA**: in this case use **inheritance**
  - **ObjB "has a" ObjA**: in this case use **composition**

Inheritance versus Composition

- **University parking lot permits**: A parking permit object involves a university Person and a lot name ("4", "11", "XX", "Home Depot")

Inheritance:

```
public class Permit extends Person {
    String lotName;

  // …
}
```

Composition:

```
public class Permit {
    Person p;
    String lotName;
    // …
}
```

- **Which to use?**
  - **A parking permit "is a" person?** Clearly no
  - **A parking permit "has a" person?** Yes, because a Person is one of the two entities in a a permit object
  - So **composition** is the better design choice here
- **Prefer Composition over inheritance**
  - **When in doubt or when multiple choices available, prefer composition over Inheritance**