

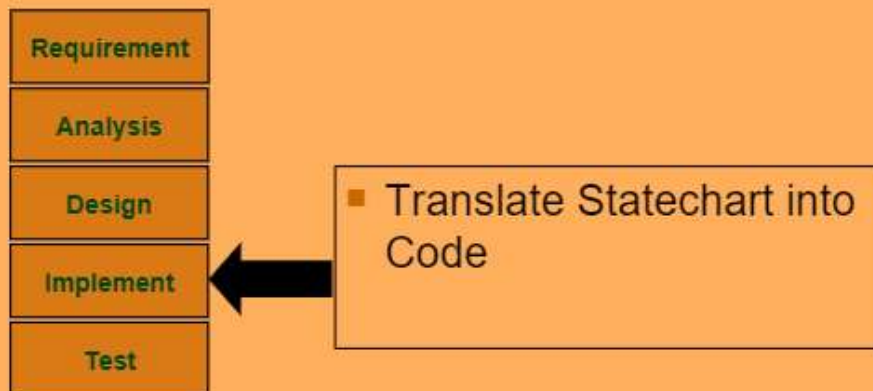
# Lecture 11

Thursday, March 30, 2023 1:23 PM

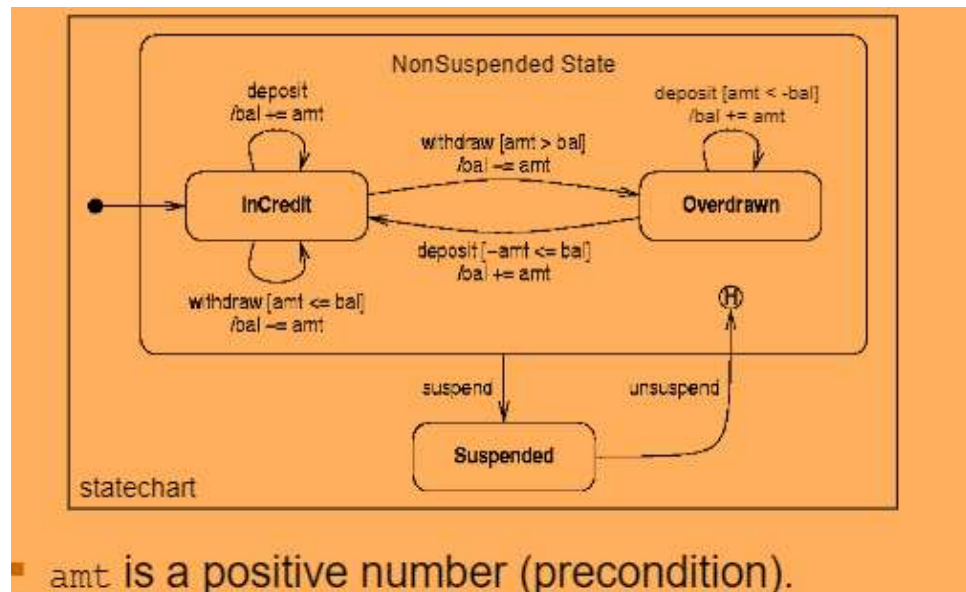
## Overview of This Lecture

- Implementing a Statechart
- Review of Java programming concepts:
  - Inheritance
  - Polymorphism
  - Type Casting
  - Interface

## Where are we now?



- Implementing Statecharts
  - If statechart for a class exists:
    - It gives an overall specification of the class behavior
    - Can be used to structure the implementation of the operations in a systematic way
    - Ex: Accounts class
      - States: InCredit, OverDrawn, Suspended
      - Operations: deposit, withdraw, suspend, unsuspend
      -



- Implementation Steps
  1. Define states
    - Enumerate diff states as constants
  2. Record an object's current state
    - Initial state specifies what state the object is in when made
    - Implement this in constructor
  3. For each op, implement state-dependent behavior using a switch statement:
    - One case for each state
    - Code state-specific actions

## Implementation: Step 1 and 2

```

public class Account {
    private final int InCredit = 0;
    private final int OverDrawn = 1;
    private final int Suspended = 2;

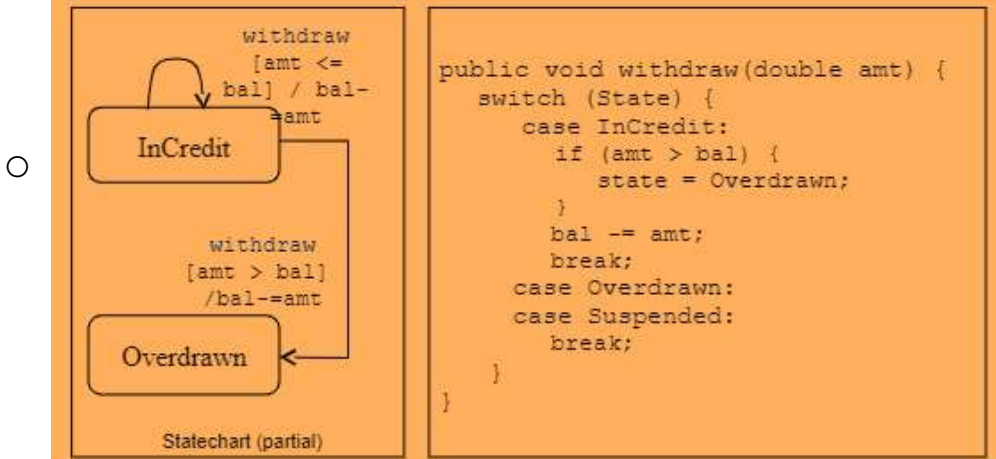
    private int historyState;
    private int state;
    ... ..
    public Account() {
        state = InCredit;
    }
}
  
```

Step 1:  
• Define States.

Step 2:  
• Implements *Initial State* in constructor.

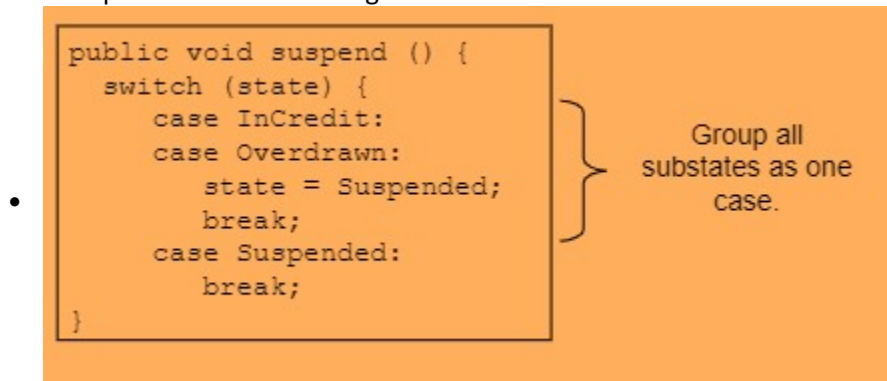
• The recommendation is to use the *State* pattern, but for simplicity we use instead integer constants to identify the states.

## Implementation: Step 3

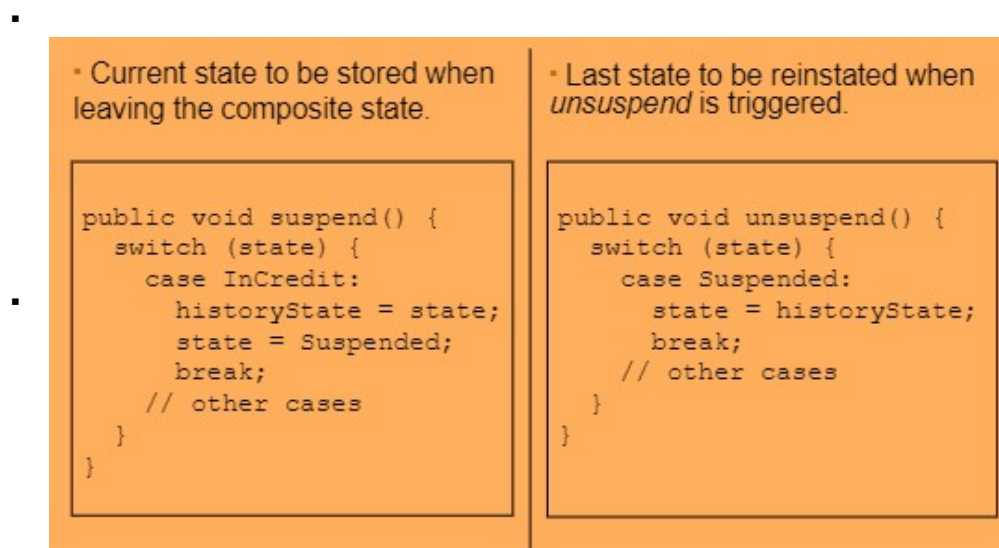


○ Composite States

- Suspend transition leading from the composite state is shared by all substates
  - Group all substates as a single case



○ History State



○ Entry and Exit Actions

- Entry:
  - Code for an entry action should be executed when object enters the state containing the action
- Exit

- Code for an exit action should be executed when the state is left
- Ex:
  - State S has:
  - Entry and exit actions defined for it, and
  - Events eventF and eventG cause transitions to and from S, respectively

## Entry/Exit Action: Implementation



```

public void eventF() {
    switch (state) {
        case S:
            break ;
        case T:
            doEntry() ;
            state = S ;
            break ;
    }
}
  
```

```

public void eventG() {
    switch (state) {
        case S:
            doExit() ;
            state = U ;
            break ;
        case U:
            break ;
    }
}
  
```

▪ For eventF(), the state T is the state that enters S; and for eventG(), the state S goes to state U.

- Review of Java Concepts

○

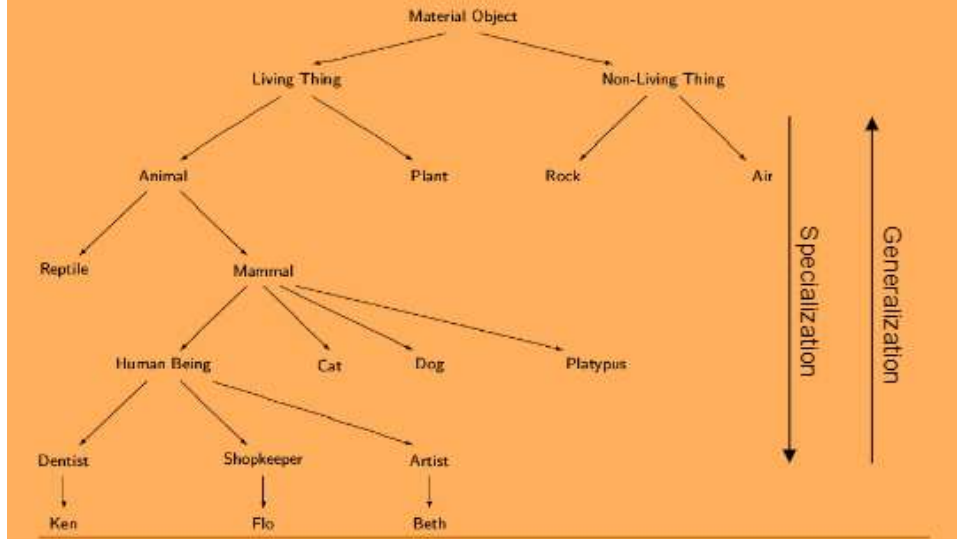
## Overview of Topics

- Intuitive and practical meaning of inheritance.
- Syntax for inheritance.
- Method overloading and overriding.
- ■ Polymorphic assignments and substitution.
- Interfaces.



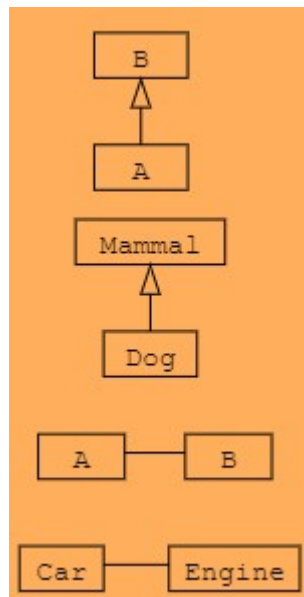
○

# Abstract Idea of Inheritance



## ○ "Is-A" vs "Has-A"

- The "Is-A" rule of thumb
  - If "An A is-a B" sounds right, then A is likely to be a subclass of B
- The "Has-A" rule of thumb:
  - If "An A has-a B" seems right, then A and B should be separate classes, but related by an association
  -



## ○ Motivations for Inheritance

- Reuse of code:
  - Methods defined in the parent class can be made available to the child class w/out rewriting, makes easy to make new abstractions
- Reuse of Concept:
  - Methods described in the parent can be redefined and overridden in the child, concept embodied in the def is shared

## ○ Class Inheritance in Java

- When class A inherits from (extends) class B, class A is "subclass" or an extended class of B, B is "superclass" of A
- All public and protected members of the superclass are accessible in the extended



class

- The subclass inherits features from its superclass, and may add more features
- A subclass extends the capability of its superclass

○ Extending Classes in Java

- A subclass is a specialization of its superclass
- Every instance of a subclass is an instance of a superclass, not vice versa
  - Substitution: an instance of a subtype can be substituted for an instance of its supertype
  - Side issue: subtype vs subclass

○ Type Signature for a Method

- A sequence that has types of method's parameters
- The returned type and parameter names are not part of the type signature
- Parameter order is significant

Method	Type Signature
int Add(int X, int Y)	(int, int)
void Add(int A, int B)	(int, int)
void m(int X, double Y)	(int, double)
void m(double X, int Y)	(double, int)

○ Java Terminology

- - `interface I extends J { }`
    - ▣ I and J contain signatures only, no implementations
    - ▣ I is a subtype of J
  - `class A implements I { }`
    - ▣ objects of class A satisfy interface I
    - ▣ A is an implementation of I
  - - `class A extends B { }`
      - ▣ objects of class A reuse the implementation of B
      - ▣ A is a subclass of B
      - ▣ type of A is also a subtype of type of B
    - `abstract class C { }`
      - ▣ C has unimplemented signatures

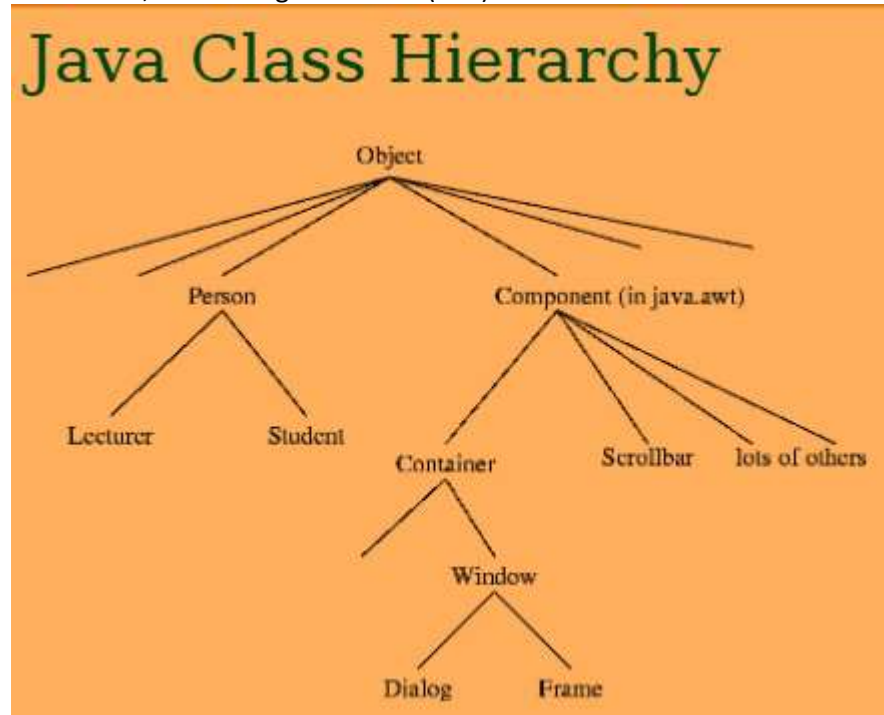
○ Subclassing vs Subtyping

- A is a subclass of B: every A object has a B object inside of it
- A is a Java subtype of B: a var of type B may refer to a var of type A
- Subclassing implies subtyping: if an object A has an object B inside, then a var of type B may refer to a var of type A
- Subtyping does not necessarily imply subclassing: can have a var of interface B that refers to a var of interface A (assuming that interface A extends interface B)

○ Class Hierarchies

- 2 common views:
- All classes are part of a single large class hierarchy, so one class that is the original ancestor of all other classes (Java)

- Classes are only placed in hierarchies if they have a relationship-results in many small hierarchies, but no single ancestor (C++)



## ○ Method Overloading

- 2 methods in a class w/ the same name but diff signatures are known as method overloading
- Overloading is resolved at compile time:

```

public class Account {

    Account () {
        //Signature: ( )
        ...
    }

    Account (String name,
            String number,
            double balance) {
        //Signature:
        //(String,String,double)
    }
    ...
}
  
```

## ○ Method Overriding

- Refers to the presence of an instance method in a subclass that has the same name, type signature, and returned type of a method in the superclass
- The implementation of the method in the subclass replaces the implementation of the method in the superclass

```

public class A {

    M(int X, int Y) {
        //Signature:
        //(int, int)
        ...
    }
}

public class B extends A {
    M(int X, int Y) {
        //Signature:
        //(int, int)
        ...
    }
}

```

## Code Example

```

public class Employee {
    private String name;
    private double salary;
    public void raiseSalary(double byPercent) {
        salary = salary + (salary * byPercent / 100);
    }
    // other methods
}

```

```

public class Manager extends Employee {
    public void raiseSalary(double byPercent) {
        double bonus = 200;
        super.raiseSalary(byPercent + bonus);
    }
    // other methods
}

```



## Code Example (cont)

```
public class Test {
    static public void main(String args[]) {
        Employee e = new Employee ("Mary", 1000);
        e.raiseSalary(10); // Mary's Salary = ?

        Manager m = new Manager ("John", 1000);
        m.raiseSalary(10); // John's Salary = ?
    }
}
```

### Questions:

- What happens in the absence of method `raiseSalary()` in `Manager`?
- Is `Employee m = new Manager("Smith", 1000);` a valid assignment?
- What if `raiseSalary()` is a static method in `Employee` class?
- Does `Manager` redefine name and salary?

### ○ Polymorphic Assignments

- The type of the expression at the RHS of an assignment must be same/subtype of the type of the var at the LHS of the assignment

```
Employee e = new Employee();
Employee m = new Manager();
```

- If class A extends class B, any instance of A can act as an instance of B:

```
B bObject = new A(); // polymorphic assignment
```

**Question:** If A extends B, and B extends C; does A extends C?

```
C cObject = new A(); // valid ?
```

## Code Example

```
class Student {
    protected String name;
    public Student (String name) {this.name = name;}
    public String toString() {
        return "Student: " + name;}
}
```

```
class Undergraduate extends Student {
    public Undergraduate (String name) {super(name);}
    public String toString() {
        return "Undergraduate student: " + name;}
}
```

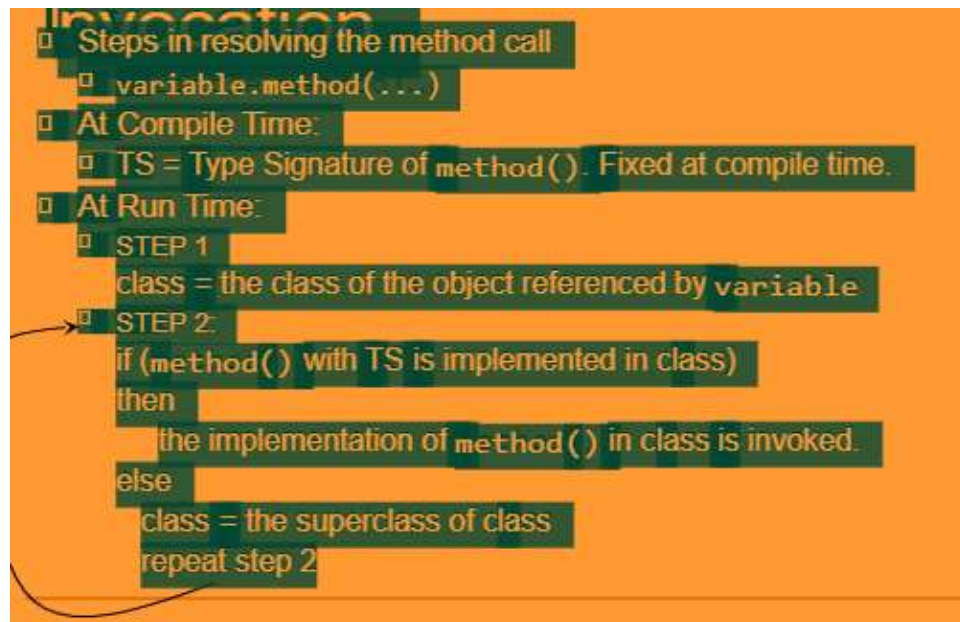
```
class Graduate extends Student {
    public Graduate(String name) {super(name);}
    public String toString() {
        return "Graduate student: " + name;}
}
```

### ○ Polymorphic Method Invocation

- Instance method toString() is overridden in both subclasses:

```
Student s = new Undergraduate();  
s.toString();    // output = ?
```

- The implementation to be invoked depends on the actual class of the object referenced by the var at run-time, not declared type of the var
- Aka polymorphic method invocation aka dynamic binding



### ○ Validity of a Method Invocation

- Assume that Graduate class defines a method `getResearchTopic()` that not defined in the Student class:

```
Student s = new Graduate();  
s.getResearchTopic();    // anything wrong?
```

- Declared type of `s` is Student, not Graduate, even tho `s` holds an instance of Graduate
- Validity of method invocation is checked statically @ compile time and is based on the declared types of vars, not actual classes of objects

### ○ Type Casting

```
Student s1, s2;  
  
s1 = new Undergraduate();    // polymorphic assignment  
s2 = new Graduate();         // polymorphic assignment  
  
Graduate gs;  
gs = s2;                     // anything wrong?
```

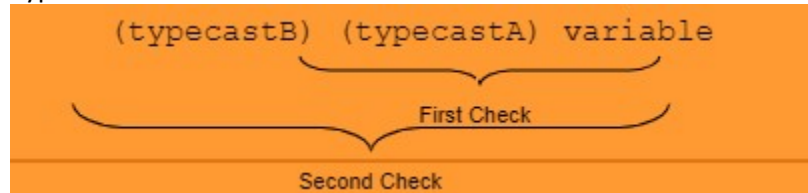
- The last assignment failed compilation as the LHS is not a subtype of RHS.
- At compile time, type is checked using declared type.

- Btw, LHS = left hand side and RHS = right
- 2 types of type casting:
- Downcasting (Narrowing): Conversion of a supertype to one of its subtypes, narrowing of reference types req explicit casts
- Upcasting (Widening): Conversion of a subtype to one of its supertypes, a reference to an object of class X can be implicitly converted to a reference to an object of one of the superclasses of X whenever necessary
- Explicit case necessary:

- `gs = (Graduate) s2; // downcasting`

#### ○ Types Casting: Compile Time Check

- Type casting syntax: (type) variable
- Simple rule used during compilation:
  - If the class of variable is either an ancestor class or a subclass of type, then its valid
  - Simpler way to remember: as long as u can trace a straight line from the class of var to the type in the class hierarchy, then its valid
- Typecast checked from the innermost lvl:

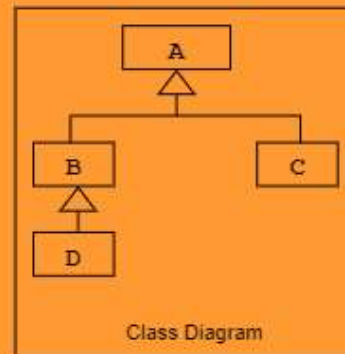


#### ○ Type Casting: Compile Time Check

Are the following valid **during compilation** given the class diagram?

```
A a = new A( );
B b = new B( );
C c = new C( );
D d = new D( );
```

```
(D) a; //valid?
(A) d; //valid?
(C) d; //valid?
(C) (A) d; //valid?
(D) (A) (C) (A) b; //valid?
(B) (C) (A) d; //valid?
(A) (B) d; //valid?
```



#### ○ Type Casting: Runtime Check

- `gs = (Graduate) s2; // compilation ok`

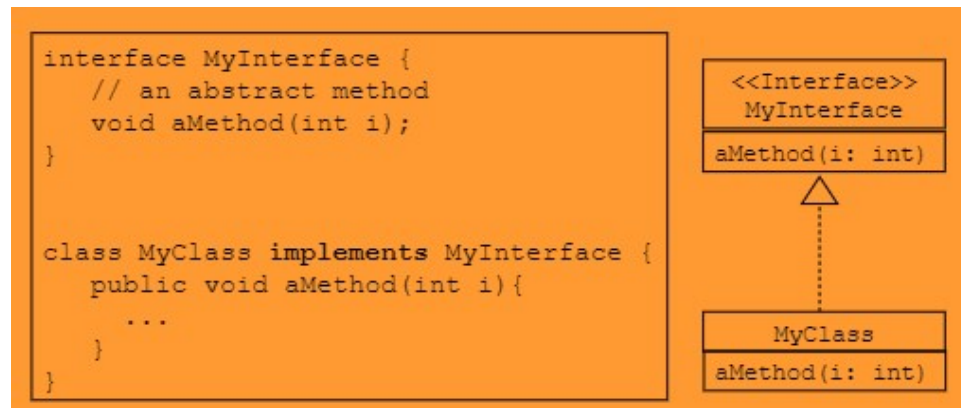
- Validity of explicit cast is checked at run-time
  - Run-time check will be performed to determine whether s2 actually holds an object that is an instance of Graduate or its subclasses

- `gs = (Graduate) s1; // compilation ok`

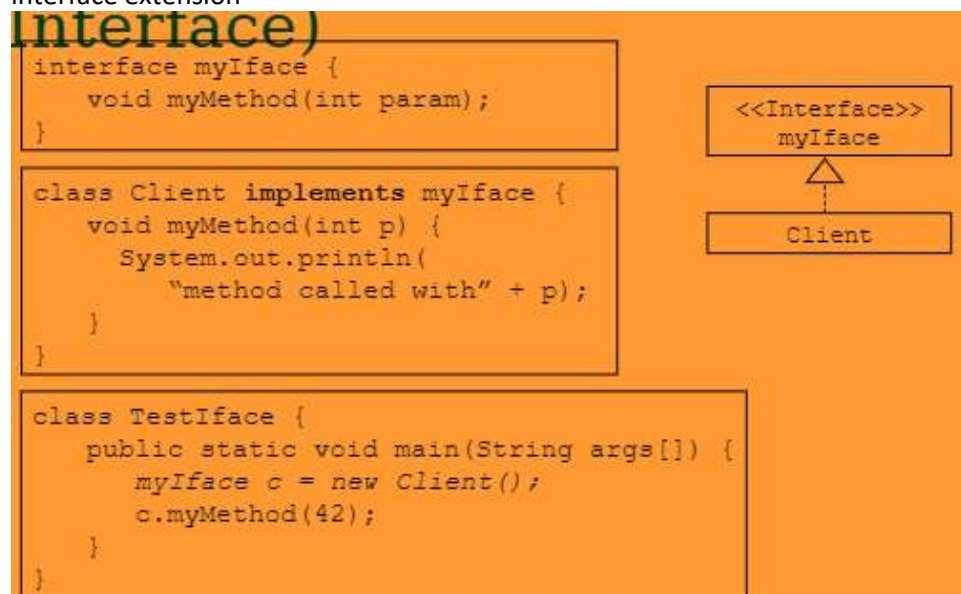
- The statement will throw a run-time exception as s1 actually holds an instance of Undergraduate (not a subtype of graduate)

#### ○ Interfaces

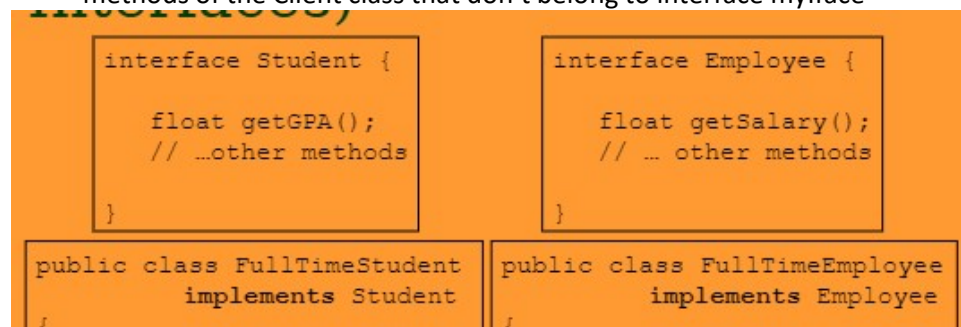


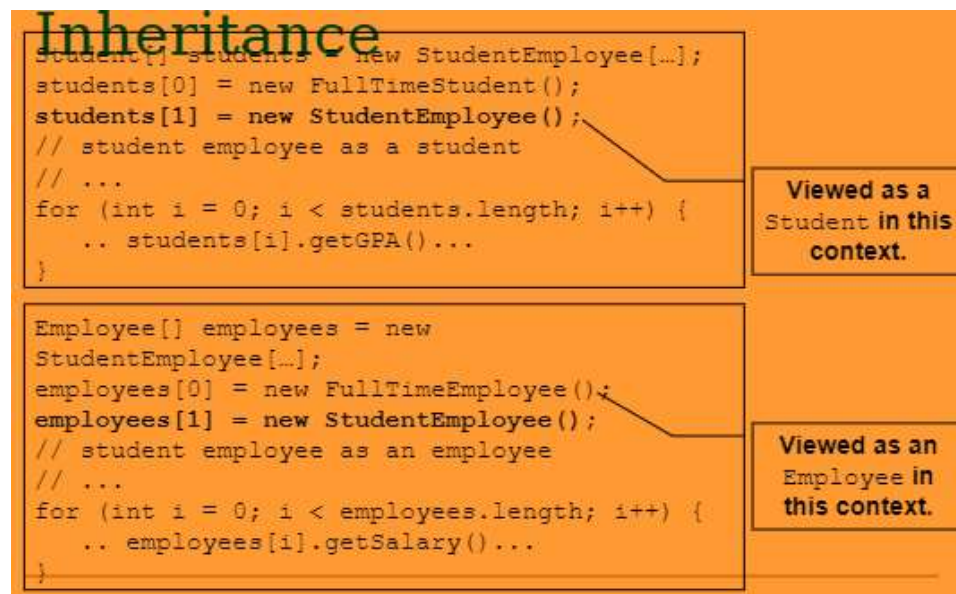
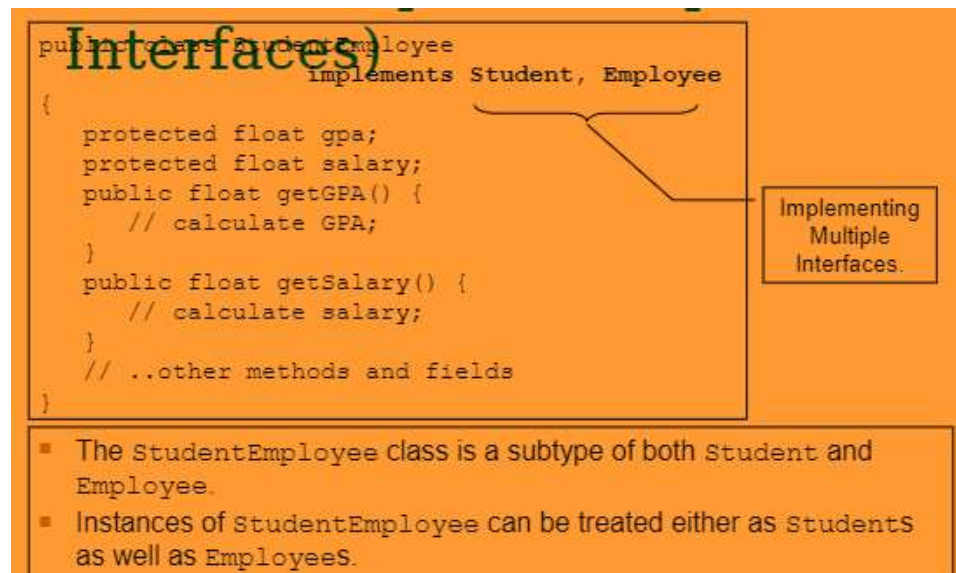
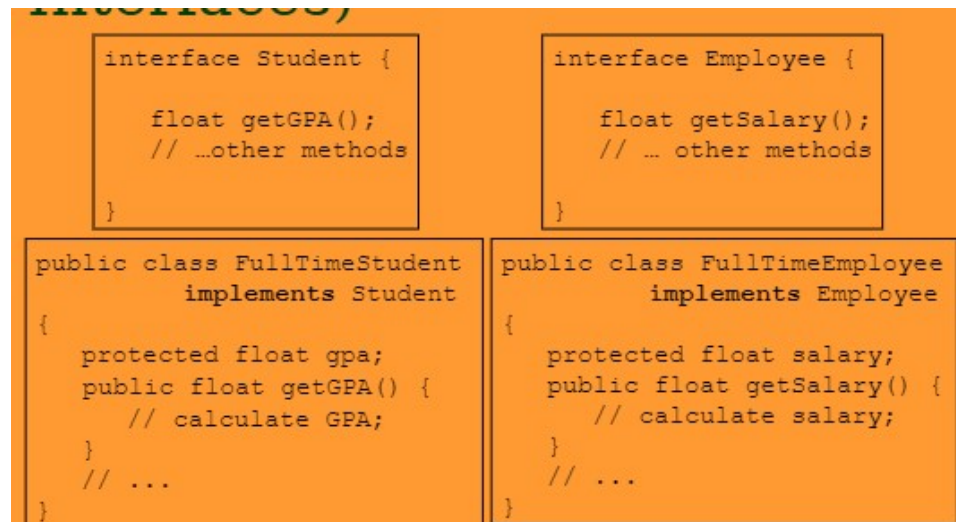


- Interface encapsulates abstract methods and constants
- Interface gives no implementation
- Interface methods cannot be static (an interface accepts static attributes/data)
- An interface can extend other interfaces
- Classes that implement an interface should provide implementation for all methods declared in the interface
- Java allows only single inheritance for class extension, and multiple inheritance for interface extension



- Can declare vars as object references which use an interface as the type rather than a class
- Any instance of any class which implements the declared interface may be stored in such a var:
  - Variable c was declared to be of interface myIface, but it was assigned an instance of Client
  - This way, c can only be used to access the myMethod() and not any of the other methods of the Client class that don't belong to interface myIface





### ○ Name Conflicts

- When implementing multiple interfaces, name conflict reps a prob
  - 2/more of interfaces define methods w/ same name



## ■ Example:

```
interface X {
    void method1(int i);
    void method2(int i);
    void method3(int i);
}
```

```
interface Y {
    void method1(double d);
    void method2(int i);
    int method3(int i);
}
```

### ○ Resolving Name Conflicts

- Rules:
- Methods w/ diff type signature: overloaded, both versions must be implemented
- Methods w/ same type signature, and same returned type: considered as a single method, only 1 implementation is req
- Methods w/ same type signature but diff return type: compilation error

### ○ Constants in Interfaces

- Unlike UML interface, a Java interface lets def of constants other than method headers
- Similar prob when implementing 2 interfaces w/ constant(s) pf the same name

```
interface X
{
    static int a = 1;
}
```

```
interface Y
{
    static double a = 2.0;
}
```

```
public class MyClass
    implements X, Y
{
    void aMethod() {
        // How to access a in X?
        // How to access a in Y?
    }
}
```

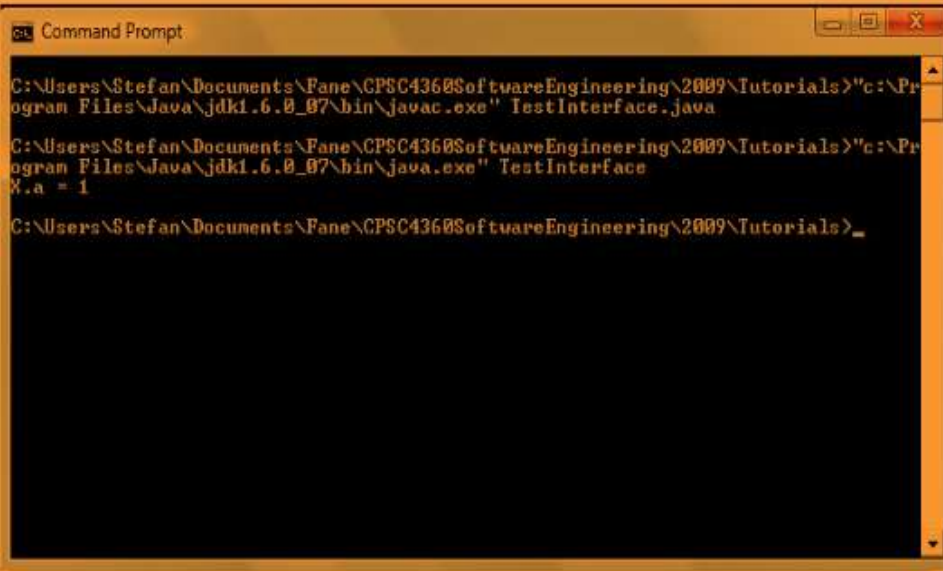
## Interface: Example

```
interface X {
    static int a = 1;
    void myMethod();
}

class Client implements X {
    public void myMethod() {
        System.out.println("X.a = " + X.a);
    }
}

public class TestInterface {
    public static void main(String args[]) {
        X c = new Client();
        c.myMethod();
    }
}
```

## Interface: Example - cont



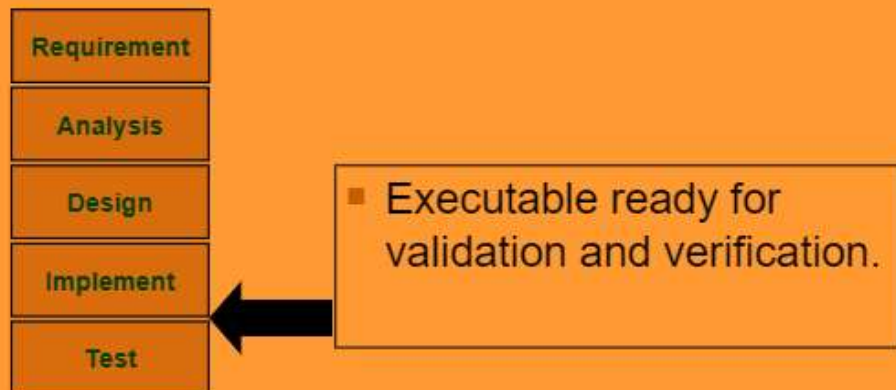
```
Command Prompt

C:\Users\Stefan\Documents\Fane\CPSC4360SoftwareEngineering\2009\Tutorials>"c:\Program Files\Java\jdk1.6.0_07\bin\javac.exe" TestInterface.java

C:\Users\Stefan\Documents\Fane\CPSC4360SoftwareEngineering\2009\Tutorials>"c:\Program Files\Java\jdk1.6.0_07\bin\java.exe" TestInterface
X.a = 1

C:\Users\Stefan\Documents\Fane\CPSC4360SoftwareEngineering\2009\Tutorials>_
```

## Where are we now?



## Summary

- Implementing a Statechart
- Review of Java programming concepts
  - Inheritance
  - Polymorphism
  - Type Casting
  - Interface

○  
○

■

# L11 Assessment

Thursday, March 30, 2023 1:24 PM

## Question 1

### 10 Points

Use the State pattern to model the following specification. Write skeletal code for the classes and methods to describe learn, work, and change in student's status.

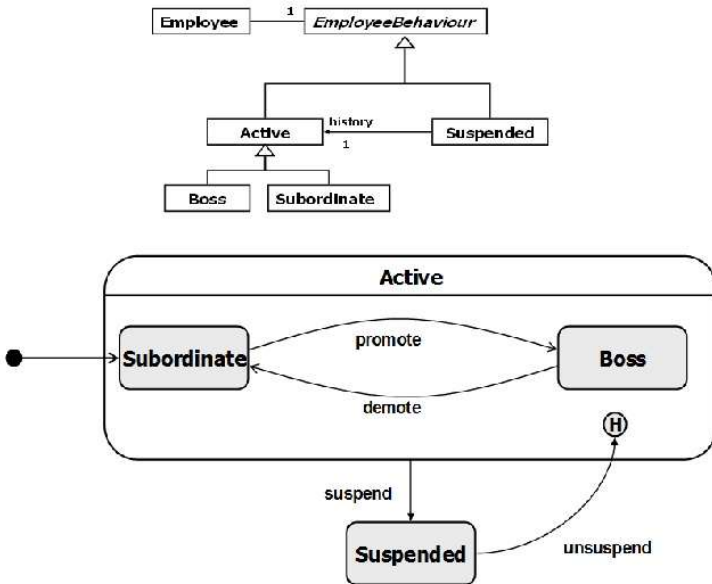
"A student-worker exhibits student behavior as well as worker behavior in his/her life. When the student learns, a student acquires a learning behavior. He/she continually learns and when he/she is about to start working, he/she changes the behavior to worker behavior."

```
public class Life{
    private final int student = 0;
    private boolean learning = true;
    private final int working = 2;
    //^thos are our states
    private int historyState;
    private int state;
    //these are other states
    public Life() {
        state = student;
    }
    /*
     * code that might change learning
     */
    public void readyToChange() {
        switch (state) {
            case student:
                if(!learning) { //so if the student is not learning
                    state = working;
                }
            case working:
                if(learning) { //so if worker is learning
                    state = student;
                }
        }
    }
}
```

## Question 2

### 10 Points

Let us consider an inheritance relation to model composite states for an Employee class. Use the State pattern to implement the following state-chart for the Employee class.



```

class Employee {
    EmployeeBehaviour behaviour = new EmployeeBehaviour();
    //other stuff
}

class EmployeeBehaviour {
    //code
}

class Active extends EmployeeBehaviour {
    private final int Boss = 0;
    private final int Subordinates = 1;

    private final int Suspend = 2;

    public int historyState;
    public int state;

    public Active() {
        state = Subordinates;
    }

    public void promote() {
        switch (state) {
            case Subordinates:
                state = Boss;
            case Boss:
                break;
        }
    }
}

```



```

    }

    public void demote() {
        switch (state) {
            case Subordinates:
            case Boss:
                state = Subordinates;
                break;
        }
    }

    public void suspend() {
        switch(state) {
            case Subordinates:
                historyState = state;
                state = Suspend;
                break;
            //other case
        }
    }

    public void unsuspend() {
        switch(state) {
            case Suspend:
                state = historyState;
                break;
            //other case
        }
    }
}

```

### Question 3

#### 10 Points

Analyze the following code, and find output at invocations of tryMe() in main() as well the justification of the output. Also, find out if each of the commented statements below will result in a compile time error or a run time error, and why?

```

class Animal{
    public void tryMe(){
        System.out.println("####");
    }
}

class Cat extends Animal {
    public void tryMe(){
        System.out.println("Meow !");
    }
}

class Dog extends Animal{
    public void tryMe(){
        System.out.println("Bark !");
    }
}

public class AnimalTest {
    public static void main(String args[]) {

```

```
Animal a = new Animal();
Animal c = new Cat();
Dog d = new Dog();
a.tryMe();
c.tryMe();
d.tryMe();
```

```
Animal anotherA = ((Animal)d);
anotherA.tryMe();
Dog anotherD = ((Dog)(Animal)d);
anotherD.tryMe();
```

```
// anotherD = ((Dog)a); //will result in run time error
// anotherA = ((Cat)(Animal)d); //will result in run time error
// anotherD = ((Cat)(Animal)d); //will result in compile time error
// Cat anotherC = ((Cat)(Animal)d); //will result in run time error
}
}
```

anotherD will result in a runtime error because this is like an unambiguous down casting. The Dog class is a type of Animal, so it is not right to type case an Animal object into a Dog object.

anotherA will result in a runtime error because of a similar reason to anotherD. The d object can be upcasted to an Animal object, but it should not be down cast to a Cat object when it is initially read as a Dog object.

The second anotherD will result in a compile time error because the type Cat cannot be passed as a Dog object.

AnotherC is of the type Cat, so a Dog object should not be passed in, otherwise it would result in a runtime error.

```
//q3
class Animal {
    public void tryMe() {
        System.out.println("####");
    }
}
class Cat extends Animal {
    public void tryMe() {
        System.out.println("Meow !");
    }
}
class Dog extends Animal {
    public void tryMe() {
        System.out.println("Bark !");
    }
}
public class Ch11Hw {
    public static void main(String args[]) {
        Animal a = new Animal();
        Animal c = new Cat();
        Dog d = new Dog();
        a.tryMe();
        c.tryMe();
        d.tryMe();
        Animal anotherA = ((Animal)d);
        anotherA.tryMe();
        Dog anotherD = ((Dog)(Animal)d);
        anotherD.tryMe();

        // anotherD = ((Dog)a);
        // anotherA = ((Cat)(Animal)d);
        // anotherD = ((Cat)(Animal)d);
        // Cat anotherC = ((Cat)(Animal)d);
    }
}
```

```
####
Meow !
Bark !
Bark !
Bark !
```

#### Question 4

10 Points

Analyze the following code. What will be printed when this is executed? Justify each of the output.

```
class Parent {
    void overload (Parent p) {
        System.out.println("Parent");
    }
}
```

```
}  
} // end Parent
```

```
public class Child extends Parent {  
    void overload (Child c) {  
        System.out.println("Child overload");  
    }  
    void overload (Parent p) {  
        System.out.println("Child override");  
    }  
    void testOverload () {  
        Parent p = new Child();  
        Child c = new Child();  
        overload(p);  
        overload(c);  
        p.overload(p);  
        c.overload(c);  
        p.overload(c);  
        c.overload(p);  
    } // end testOverload
```

```
public static void main(String[] args) {  
    Child c = new Child();  
    c.testOverload();  
} // end main  
} // end Child
```

```
//q4
class Parent {
    void overload(Parent p) {
        System.out.println("Parent");
    }
} // end Parent
public class Ch11Hw extends Parent {
    void overload(Ch11Hw c) {
        System.out.println("Child overload");
    }
    void overload(Parent p) {
        System.out.println("Child override");
    }
    void testOverload() {
        Parent p = new Ch11Hw();
        Ch11Hw c = new Ch11Hw();
        overload(p);
        overload(c);
        p.overload(p);
        c.overload(c);
        p.overload(c);
        c.overload(p);
    } // end testOverload
    public static void main(String[] args) {
        Ch11Hw c = new Ch11Hw();
        c.testOverload();
    } // end main
} // end Child
```

```
Child override
Child overload
Child override
Child overload
Child override
Child override
```

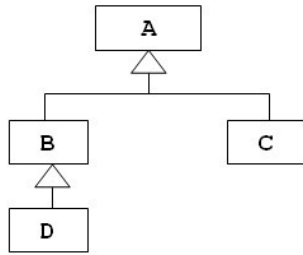
This is because (labeling them 1-6):

1. The overload method is passed in a parent type, resulting in "Child override"
2. The overload method is passed in a child type, resulting in "Child overload"
3. The overload method is passed in a parent type, and also uses an object reference of parent. This results in "Child override"
4. The overload method is passed in a child type, and also uses an object reference of child. This results in "Child overload"
5. The overload method is passed in a child type, and also uses an object reference of parent. Since the parent is broader (upcasting), the result is "Child override"
6. The overload method is passed in a parent type, and also uses an object reference of child. Since the parent is broader (upcasting), the result is "Child override"

**Question 5**  
**10 Points**



Let us consider the below UML class diagram and the below Java definition for six Java objects.



```
A a = new A();
A a2 = new B();
B b = new B();
B b2 = new D();
C c = new C();
D d = new D();
```

Identify which conversions are valid at compile-time and at run-time.

- (B) a
- (C) a
- (D) a2
- (A)(C) a2
- (B)(A)(D) b
- (C)(D)(A) b
- (A)(C)(B) b2
- (D)(C)(B)(A) b2
- (C)(A)(B)(D)(B) c
- (A)(D)(B) c
- (C)(D)(A) d
- (D)(B)(C) d

1. (B) a – this works because A inherits B
2. (C) a – this works because A inherits C
3. (D) a2 – this works because A inherits B, which inherits D
4. (A)(C) a2 – this does not work because, though A inherits C, it only has accessibility to B
5. (B)(A)(D) b – this does not work because b can only access B

6. (C)(D)(A) b – same as reasons before
7. (A)(C)(B) b2 – this does not work because D does not inherit from A, C, or B
8. (D)(C)(B)(A) b2 – same as above
9. (C)(A)(B)(D)(B) c – same as above
10. (A)(D)(B) c – same as above
11. (C)(D)(A) d – same as above
12. (D)(B)(C) d – same as above