

Inheritance & Composition





Outlines

BASIC CONCEPTS

- › Introduction
- › Inheritance & Composition
- › Extend classes & instanceof
- › Method overriding
- › Keyword 'super'
- › Creation mechanism
- › Access control
- › Methods you cannot override

ADVANCED CONCEPTS

- › Dynamic binding
 - Create a single method that has one or more parameters that might be one of several types
 - Create a single array of superclass object references but store multiple subclass instances in it.



Introduction

- › Creating new class
 - From scratch
 - From existing class (reuse)
 - › Inheritance
 - “is-a” or “is-a-type-of” relation
 - › Composition
 - “has-a”



Introduction

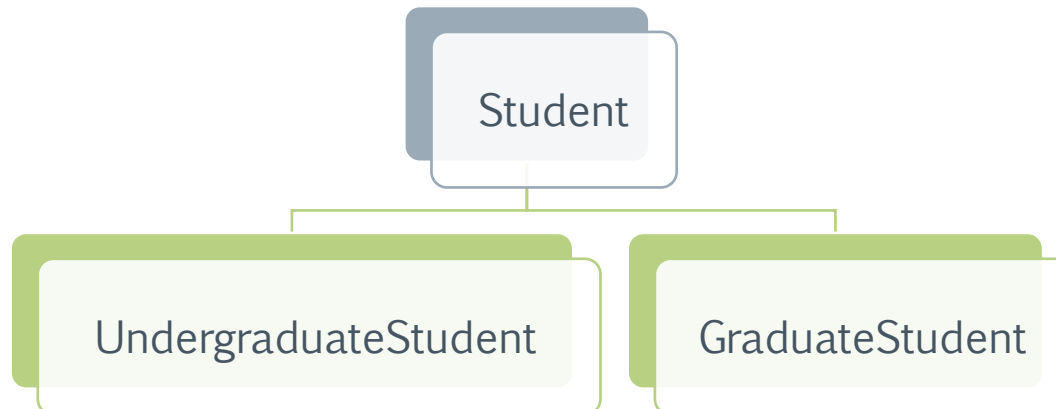
- › Inheritance makes it possible to build new classes from existing classes thus facilitating the reuse of methods and data from one class in another.
- › Inheritance allows data of one type to be treated as data of a more general type.
- › Use inheritance to create derived class
 - Save time
 - Reduce errors
 - Reduce amount of new learning required to use new class



Introduction (cont.)

› Base class

- Used as a basis for inheritance
- Also called:
 - › Superclass
 - › Parent class
- For example:
 - › Student

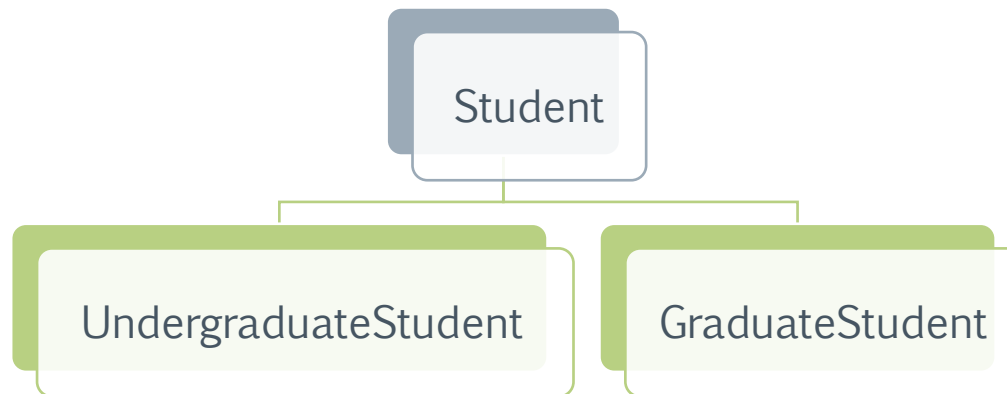


› Derived class

- **Inherits all non-private members from a base class**
- Always “is a” case or example of more general base class
- Also called:
 - › Subclass
 - › Child class
- For example:
 - › UndergraduateStudent
 - › GraduateStudent



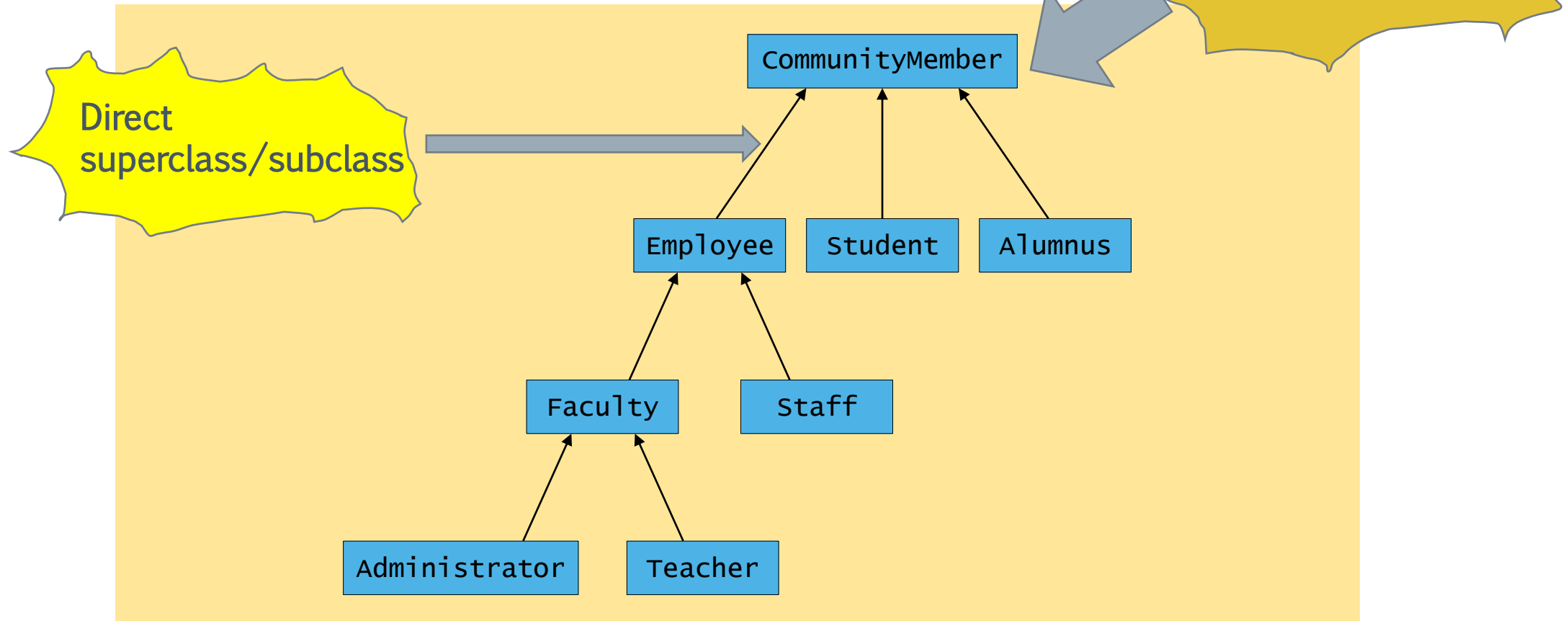
Introduction



- › UndergraduateStudent “is-a” Student
- › GraduateStudent “is-a” Student
- › But not the other way



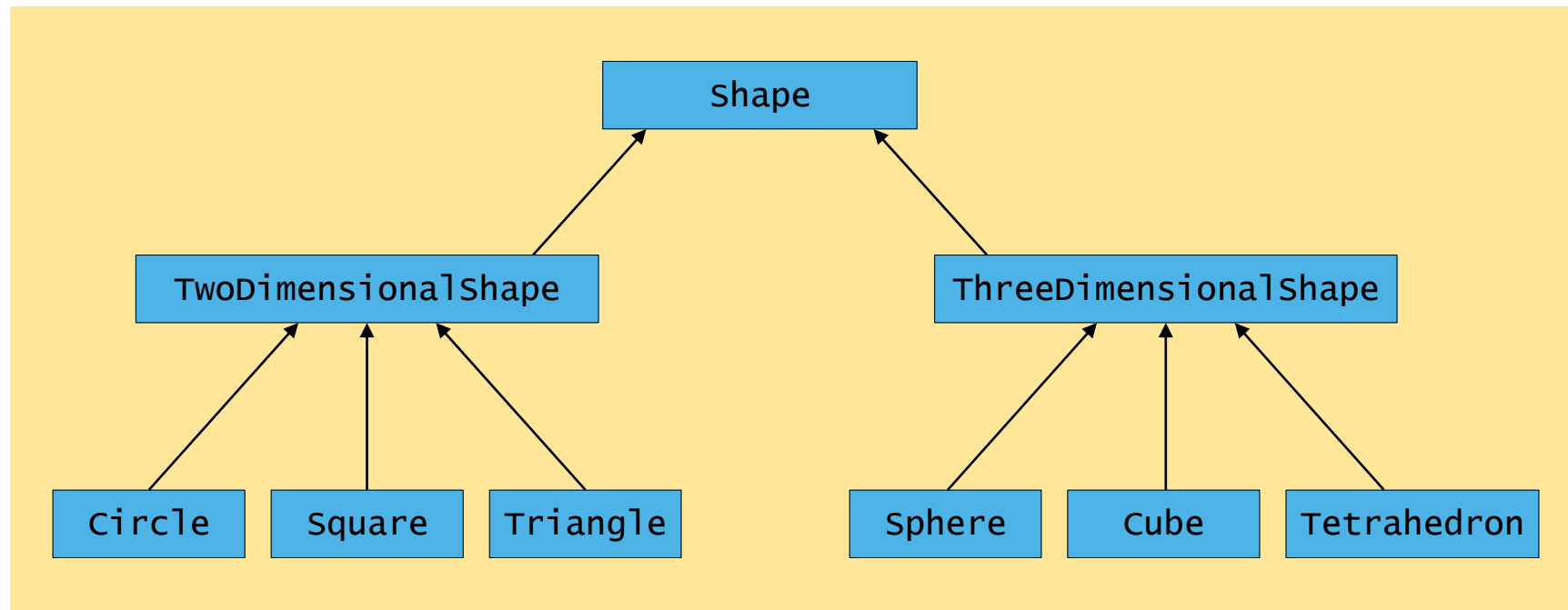
Introduction (cont.): More example



Inheritance hierarchy for university CommunityMembers.



Introduction (cont.): More example



Inheritance hierarchy for Shapes.

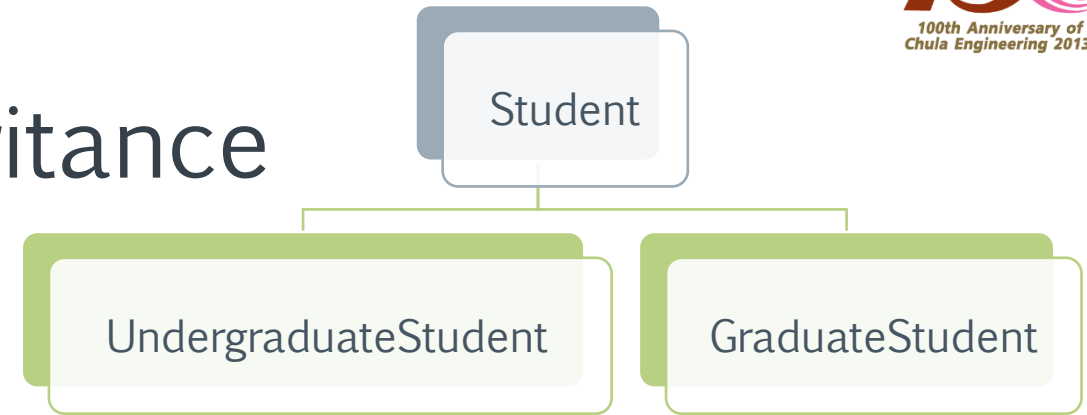


Inheritance Examples





Implementation of Inheritance



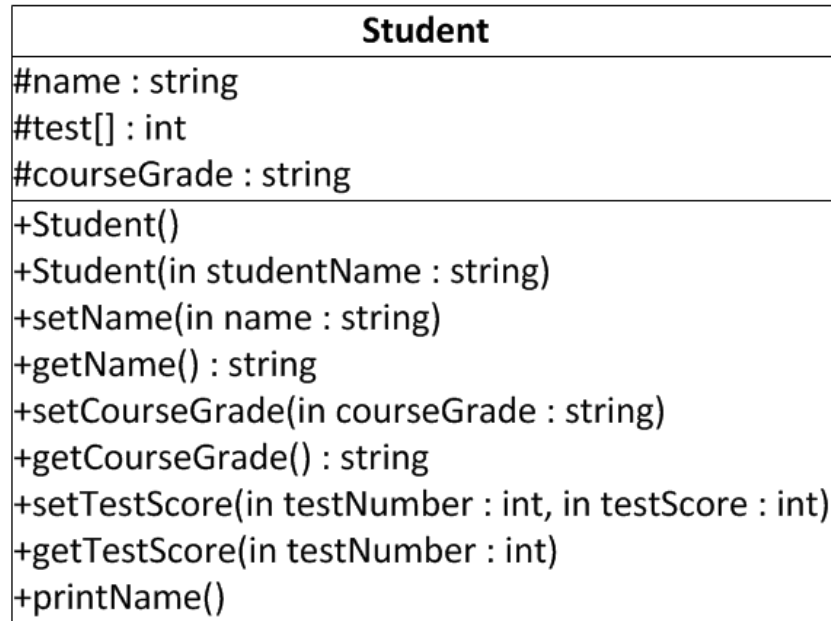
- › Keyword **extends**
 - Achieve inheritance in Java
 - Can extends from only **one** superclass
 - Example:
 - › `public class UndergraduateStudent extends Student`
 - › `public class GraduateStudent extends Student`
- › Inheritance one-way proposition
 - Child inherits from parent, not the other way round.



Student Case Study

Package
"Student"

- What are subclasses inherited from superclass?
- Are there anything in subclasses that do not have in superclass?

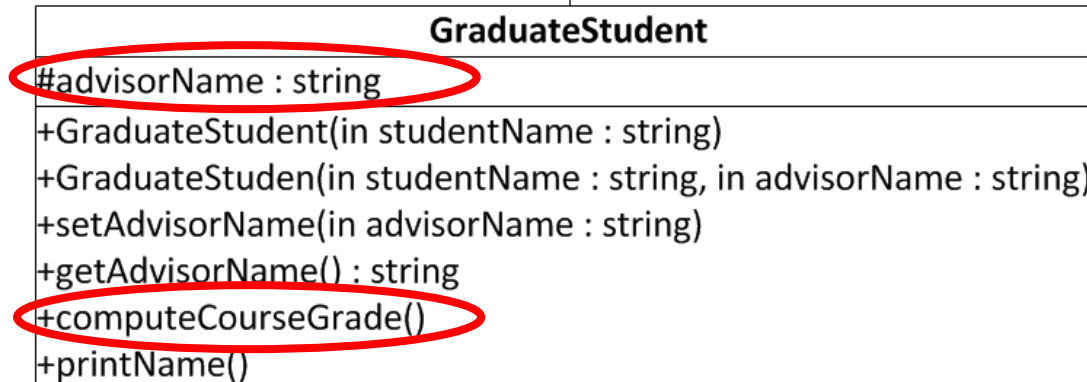
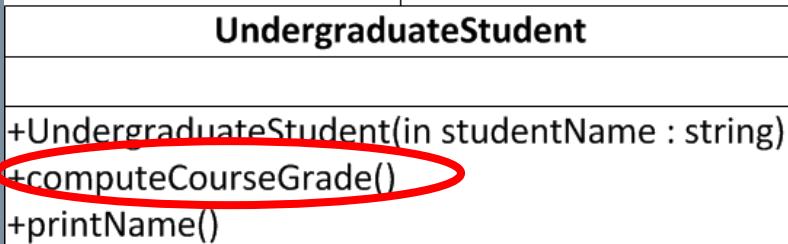


Generalization Concept

- Subclasses can be considered as superclass since they inherit everything from superclass.
- But, superclass cannot be considered as subclasses.
- Undergraduate & graduate students are student!



Type	computeCourseGrade
Undergrad.	Pass if (test1+test2+test3)/3 >= 70
Grad.	Pass if (test1+test2+test3)/3 >= 80





```
public class Student {
```

```
    protected final static int NUM_OF_TESTS = 3;
```

```
    protected String name;
```

```
    protected int[] test;
```

```
    protected String courseGrade;
```

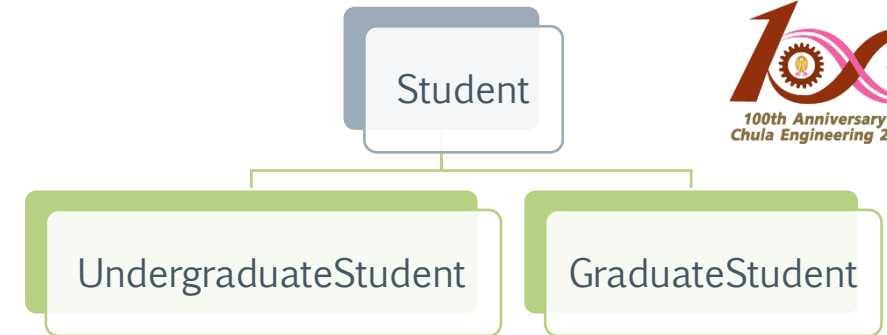
```
    ...
```

```
public class UndergraduateStudent extends Student {  
    public UndergraduateStudent(String studentName) {  
        super(studentName);  
    }  
  
    public void computeCourseGrade() {  
        //calculation 1  
    }  
    ...
```

Additional variable

```
public class GraduateStudent extends Student {  
    String advisorName;  
  
    public void computeCourseGrade() {  
        //calculation 2  
    }  
    ...
```

Additional
Method



Save time & Reduce errors

- › Is there anything wrong in the following code?
 - Is it possible to have “ArrayIndexOutOfBounds”?
 - If yes, should this issue also happen in Student’s subclasses?
 - How many method should we fix the issue?

Student.java (with ArrayIndexOutOfBounds)

```
public int getTestScore(int testNumber) {  
    return test[testNumber - 1];  
}
```

Student.java (no error)

```
public int getTestScore(int testNumber) {  
    return (testNumber <= NUM_OF_TESTS) ? test[testNumber - 1] : test[0];  
}
```



Overriding Superclass Methods

- › Create subclass by extending existing class
 - Subclass contains data and methods defined in original superclass
 - Sometimes superclass data fields and methods not entirely appropriate for subclass objects
- › Polymorphism (in general)
 - Using same method name to indicate different implementations
- › Polymorphism for superclass/subclasses
 - Override method in parent class
 - › Create method in child class that has same name and argument list as method in parent class
 - Subtype polymorphism
 - › Ability of one method name to work appropriately for different subclass objects of same parent class



Override method in parent class

```
class Student{
```

```
...
```

```
    public void printName() {
```

```
        System.out.println("Student [" + name + "]);
```

```
    }
```

```
...
```

Same signature!!!

```
class GraduateStudent extends Student {
```

```
...
```

```
    public void printName() {
```

```
        System.out.println("GraduateStudent [" + name + "]);
```

```
    }
```

```
...
```



Override method in parent class(cont.)

StudentTest1.java

```
public class StudentTest1 {  
    public static void main(String[] args) {  
        Student s1 = new UndergraduateStudent("Toey");  
        Student s2 = new GraduateStudent("Nat");  
        Student s3 = new Student("Jump");  
  
        s1.printName();  
        s2.printName();  
        s3.printName();  
    }  
}
```

Result

```
UndergraduateStudent [Toey]  
GraduateStudent [Nat]  
Student [Jump]
```




instanceof

Subtype polymorphism

StudentTest2.java

```
public class StudentTest2 {  
    public static void main(String[] args) {  
        Student s1 = new UndergraduateStudent("Toey");  
        Student s2 = new GraduateStudent("Nat");  
        checkStatus(s1);  
        checkStatus(s2);  
    }  
    public static void checkStatus(Student s) {  
        if (s instanceof UndergraduateStudent) {  
            System.out.println("You are undergraduate student.");  
        } else if (s instanceof GraduateStudent) {  
            System.out.println("You are graduate student.");  
        }  
    }  
}
```

Result

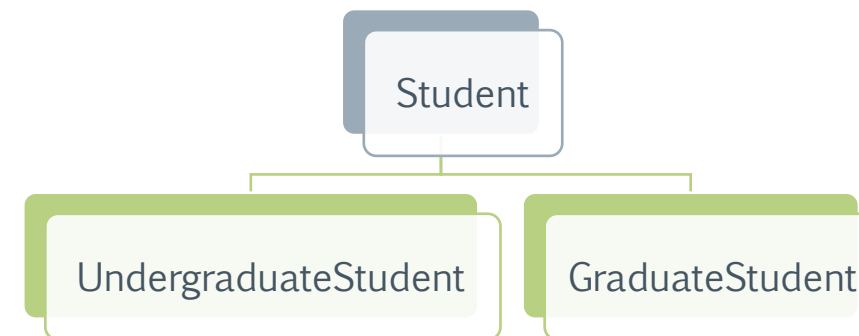
You are undergraduate student.
You are graduate student.



Up/down casting

StudentTest3.java

```
public class StudentTest3 {  
    public static void main(String[] args) {  
        // upcasting (automatically)  
        Student s1 = new GraduateStudent("Nat");  
        s1.printName();  
  
        // downcasting (manually) - may have problem  
        Student s = new Student("Luck");  
        UndergraduateStudent s2 = (UndergraduateStudent) s;  
    }  
}
```



Result

GraduateStudent [Nat]

Exception in thread "main" java.lang.ClassCastException: Student.Student cannot be cast to Student.UndergraduateStudent

at Student.StudentTest3.main(StudentTest3.java:12)



Keyword 'super'

- › The super is a reference variable that is used to refer to parent class object.
- › Whenever you create the instance of subclass, an instance of parent class is created implicitly, i.e. referred by super reference variable.
- › Usage of keyword 'super'
 - super is used to refer to parent class instance variable.
 - super() is used to invoke parent class constructor.
 - super is used to invoke parent class method.



Bike1.java



Bike2.java



Student1.java



Student2.java

Package
'Super'



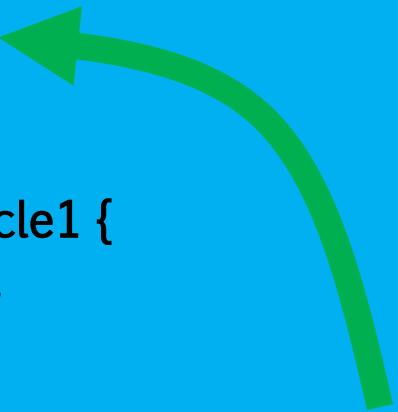
'super' examples

```
class Vehicle1 {
    int speed = 50;
}

class Bike1 extends Vehicle1 {
    int speed = 100;

    void display() {
        System.out.println(super.speed);
    }
}


...
```



```
class Vehicle2 {
    Vehicle2() {
        System.out.println("Vehicle is created");
    }
}

public class Bike2 extends Vehicle2 {
    Bike2() {
        super(); // parent class constructor
        System.out.println("Bike is created");
    }
}

...
```





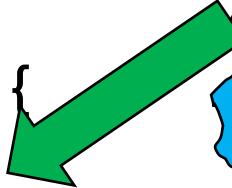
'super' examples (cont).

```
class Person1 {  
    void message() {  
        System.out.println("welcome");  
    }  
}
```

```
public class Student1 extends Person1 {  
    void message() {  
        System.out.println("welcome to java");  
    }  
}
```

```
void display() {  
    message(); // will invoke current class message() method  
    super.message(); // will invoke parent class message() method  
}
```

...



If this method does not exist, a call to message() simply calls message() of the superclass!



Instance Creation Mechanism

ClassCreation.java

```
public class ClassCreation {  
    public static void main(String[] args){  
        C c1 = new C(5);  
    }  
}  
  
class A {  
    A() {  
        System.out.println("class A");  
    }  
}
```

```
class B extends A {  
    B(int val) {  
        // super();  
        System.out.println("class B, value=" + val);  
    }  
}  
  
class C extends B {  
    C(int val) {  
        super(val);  
        System.out.println("class C, value="+ val);  
    }  
}
```

Result

```
class A  
class B, value=5  
class C, value=5
```



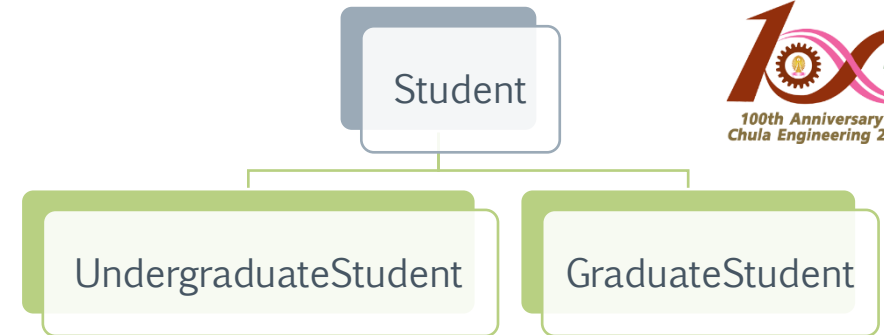
Instance Creation Mechanism (cont.)

- › When superclass contains default constructor
 - Execution of superclass constructor transparent
 - For example, $C \rightarrow B \rightarrow A$

- › Using superclass constructors that require arguments
 - When superclass has default constructor
 - › Can create subclass with or without own constructor (automatically)
 - When there is **no** default constructor in superclass
 - › Must include at least one constructor for each subclass you create
 - › First statement within each constructor must call superclass constructor



Hiding Information

Package
'Student'


› Keyword `protected`

- Provides intermediate level of security between `public` and `private` access
- Can be used within own class or in any classes extended from that class
- Cannot be used by “outside” classes
- In UML, the symbol is “#”.

Access Level	Accessing Class		
	current class	subclass	other
<code>public</code>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<code>protected</code>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<code>default</code>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<code>private</code>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>



Methods You Cannot Override

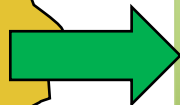
- › `static` methods
- › `final` methods
- › Methods within `final` classes
 - They cannot be superclasses (be extended).



A Subclass Cannot Override `static` Methods in Its Superclass

- › Subclass **cannot** override methods declared `static` in superclass
- › Can **hide** `static` method in superclass
 - By declaring `static` method with same signature as `static` method in superclass
 - Call new `static` method from within subclass or in another class by using subclass object
 - Within `static` method of subclass
 - › Cannot access parent method using `super` object
- › Although child class **cannot** inherit parent's static methods
 - Can access parent's static methods in the same way any other class can -> `SuperclassName.method()`

Experiment!!



- What happen if the method "Student.printName()" is static?
- For the method "GraduateStudent.computeCourseGrade()", if it is static, can we use "super"?





Static method in super class : example

```
class A{  
    static void m1(){  
    }  
}
```

```
class B extends A{  
    static void m1(){  
    }  
}
```

Cannot use super in here!

```
class C{  
    public static void main(String[] a){  
        B b = new B();  
        b.m1();  
    }  
}
```

```
    A c = new B();  
    c.m1();  
}
```

It calls class A.

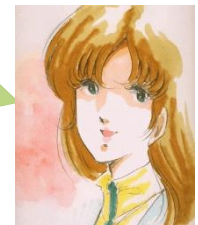




A Subclass Cannot Override `final` Methods in Its Superclass

- › Subclass cannot override methods declared `final` in superclass
- › `final` modifier
 - Does **not allow** method to be overridden
- › Advantage to making method `final`
 - Compiler knows there is only one version of method
 - Compiler knows which method version will be used
 - Can optimize program's performance
 - › By removing calls to final methods
 - › Replacing them with expanded code of their definitions
 - › At each method call location
 - › Called inlining

- What happen if the method "Student.printName()" is final?







Using Dynamic Method Binding

- › Static binding (Early binding) vs. Dynamic binding (Late binding)
 - In static binding, the method or variable version that is going to be called is resolved at **compile time**,
 - While in dynamic binding the **compiler cannot resolve** which version of a method or variable is going to bind.
- › Every subclass object “is a” superclass member
 - Convert subclass objects to superclass objects
 - Can create reference to superclass object
 - › Create variable name to hold memory address
 - › Store concrete subclass object
 - › Example:

```
Animal ref;  
ref = new Cow();
```
- › Dynamic method binding
 - Application’s ability to select correct subclass method
 - Makes programs flexible
- › When application executes
 - Correct method attached to application based on current one



Using Dynamic Method Binding (cont.)

StudentTest4.java

```
public class StudentTest4 {  
    public static void main(String[] args) {  
        Student s;  
        GraduateStudent g = new GraduateStudent("Nat");  
        UndergraduateStudent u = new UndergraduateStudent("Toey");  
  
        // This is called Dynamic binding, as the compiler will never know  
        // which version of printName() is going to be called at runtime.  
  
        s = g;  
        s.printName();  
        s = u;  
        s.printName();  
    }  
}
```

Result


GraduateStudent [Nat]

UndergraduateStudent [Toey]



Using a Superclass as a Method Parameter Type (method argument)

```
public class TalkingAnimalDemo
{
    public static void main(String[] args)
    {
        Dog dog = new Dog();
        Cow cow = new Cow();
        dog.setName("Ginger");
        cow.setName("Molly");
        talkingAnimal(dog);
        talkingAnimal(cow);
    }
    public static void talkingAnimal(Animal animal)
    {
        System.out.println("Come one. Come all.");
        System.out.println
            ("See the amazing talking animal!");
        System.out.println(animal.getName() +
            " says");
        animal.speak();
        System.out.println("*****");
    }
}
```



```
C:\Java>java TalkingAnimalDemo
Come one. Come all.
See the amazing talking animal!
Ginger says
Woof!
*****
Come one. Come all.
See the amazing talking animal!
Molly says
Moo!
*****
C:\Java>_
```




Creating Arrays of Subclass Objects 2

- › Create superclass reference
 - Treat subclass objects as superclass objects
 - › Create array of different objects
 - › Share same ancestry
- › Creates array of three `Animal` references

```
Animal[] ref = new Animal[3];
```

 - Reserve memory for three `Animal` object references



What is the output and why?

```
A.java
1  public class A {
2      public static void main(String[] args) {
3          A a = new B();
4          a.foo();
5      }
6
7      private void foo() {
8          System.out.println("A");
9      }
10 }
11
12 class B extends A {
13     public void foo() {
14         System.out.println("B");
15     }
16 }
```

A

B