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)
    - Composition
      - “has-a”
    - Inheritance
      - “is-a” or “is-a-type-of” relation

Composition

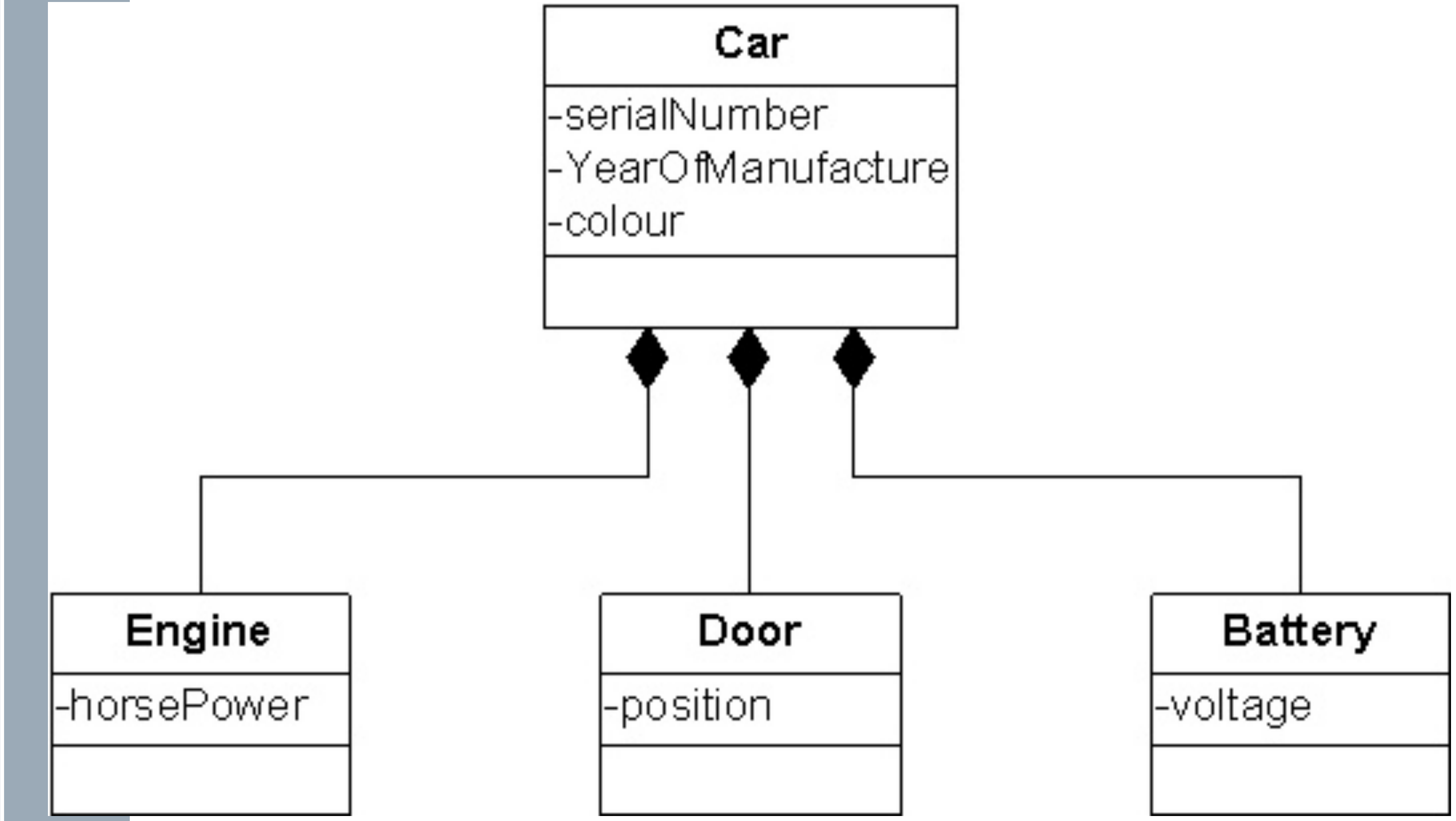
- Composition builds new class by compose existing class(es) as its field(s)

- Require to write methods to make the combined class work together

Composition

What will happen when we have an instance of a Car and call start?

Car aCar = new Car(...);
aCar.start()

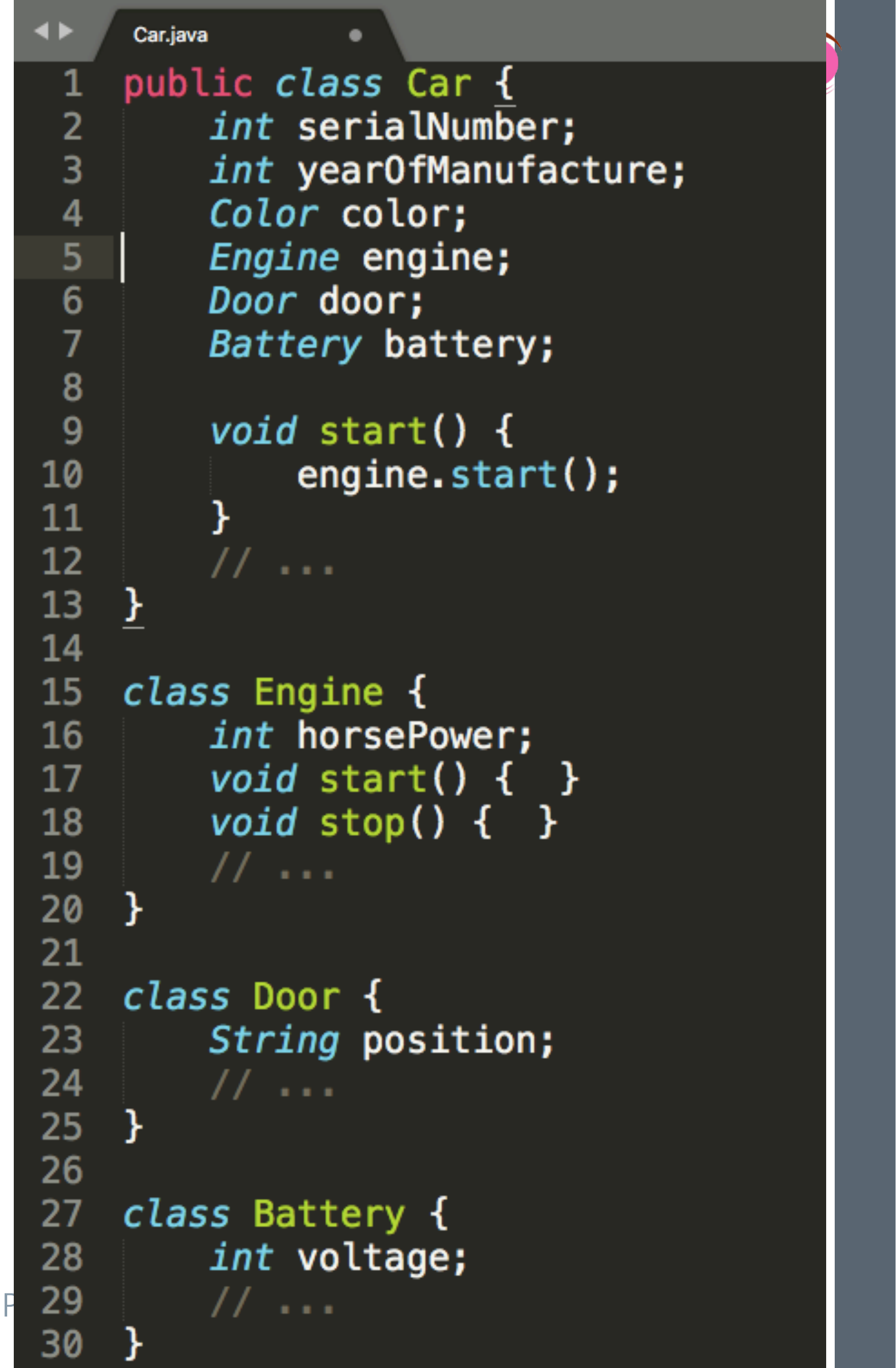


Car
-serialNumber
-YearOfManufacture
-colour

Engine
-horsePower

Door
-position

Battery
-voltage



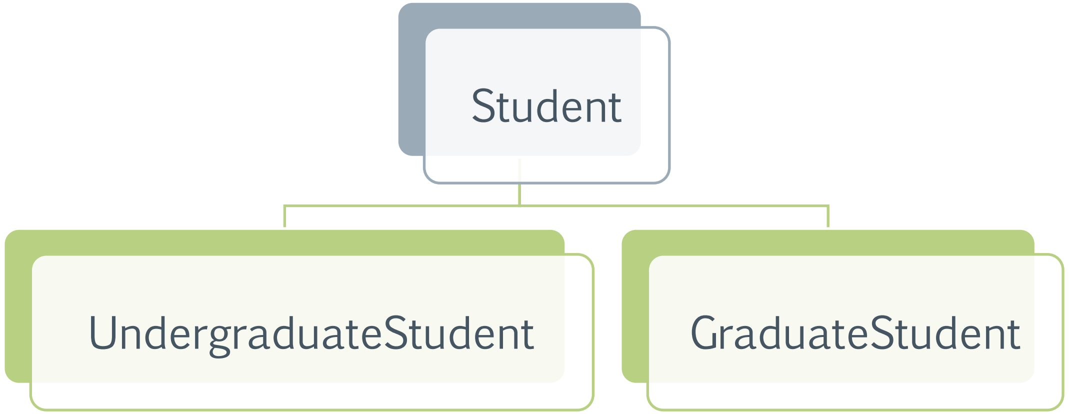
```java
public class Car {
    int serialNumber;
    int yearOfManufacture;
    Color color;
    Engine engine;
    Door door;
    Battery battery;

    void start() {
        engine.start();
    }
    // ...
}

class Engine {
    int horsePower;
    void start() {  }
    void stop() {  }
    // ...
}

class Door {
    String position;
    // ...
}

class Battery {
    int voltage;
    // ...
}
```

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

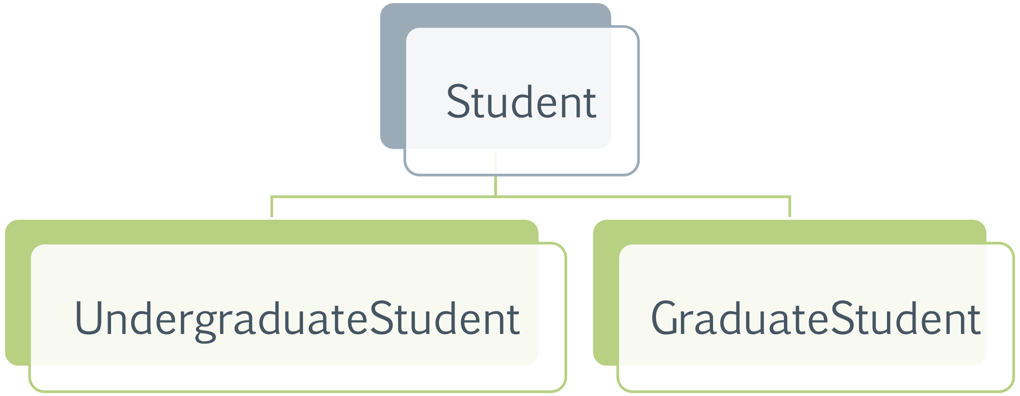
Introduction (cont.)

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



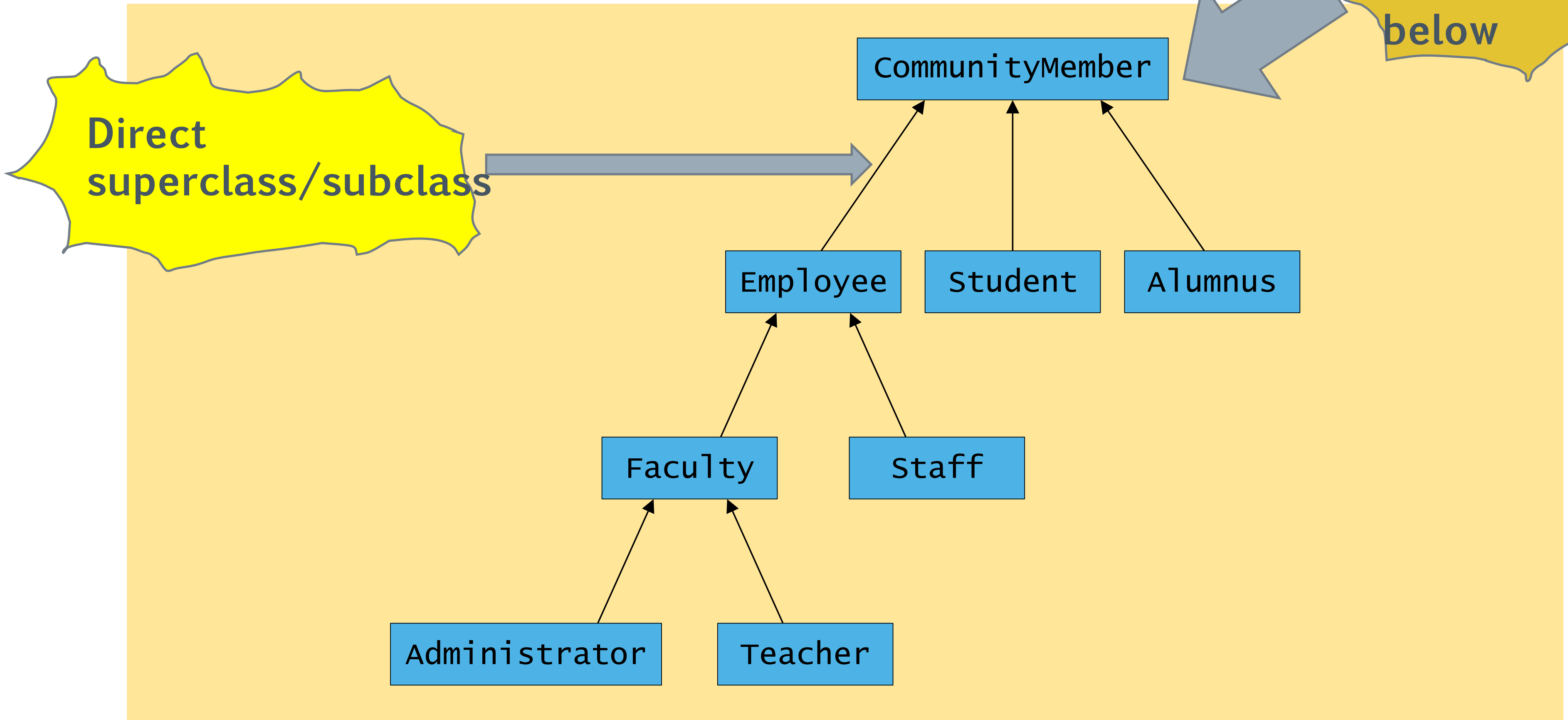
- Derived class
  - **Inherits all non-private members from a base class**
  - Always "is a" case or example of more general base class
  - Also called:
    - Subclass
    - Child class
  - For example:
    - UndergraduateStudent
    - GraduateStudent

Introduction



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

Introduction (cont.): More example

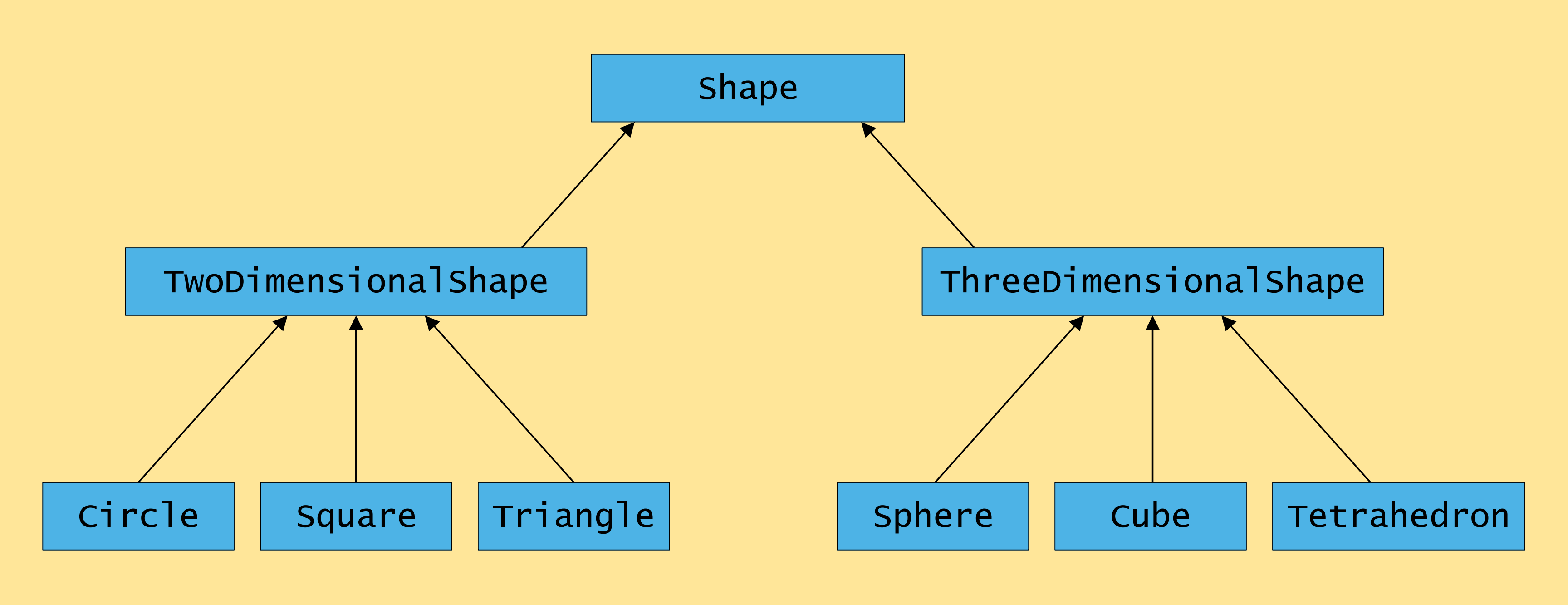


Direct superclass/subclass

Super class of all other classes below

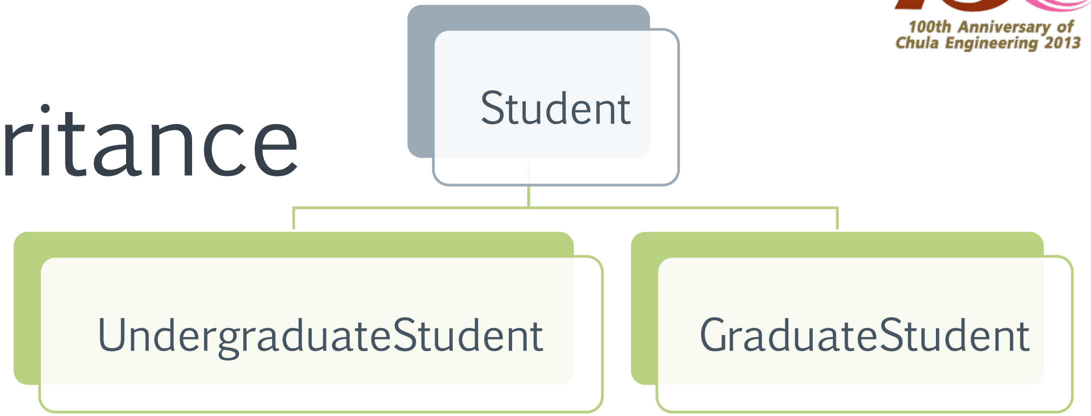
Inheritance hierarchy for university CommunityMembers.

Introduction (cont.): More example



Inheritance hierarchy for Shapes.

Implementation of Inheritance

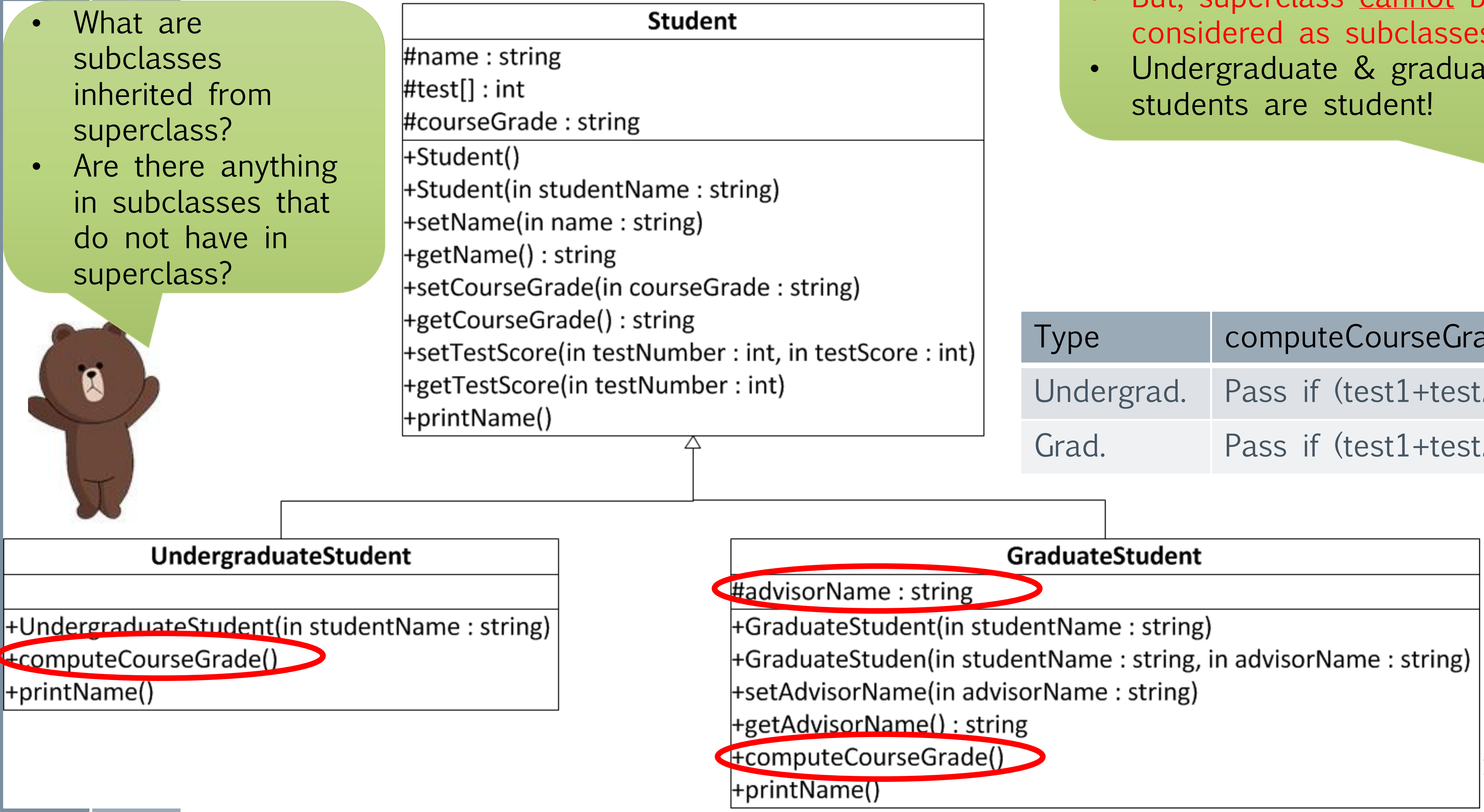


- Keyword **extends**
  - Achieve inheritance in Java
  - Can extends from only one superclass
  - Example:
    - public class UndergraduateStudent extends Student
    - public class GraduateStudent extends Student
- Inheritance one-way proposition
  - Child inherits from parent, not the other way round.

Student Case Study

Package
“Student”

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



Student

#name : string
#test[] : int
#courseGrade : string

+Student()
+Student(in studentName : string)
+setName(in name : string)
+getName() : string
+setCourseGrade(in courseGrade : string)
+getCourseGrade() : string
+setTestScore(in testNumber : int, in testScore : int)
+getTestScore(in testNumber : int)
+printName()

UndergraduateStudent

+UndergraduateStudent(in studentName : string)
+computeCourseGrade()
+printName()

GraduateStudent

#advisorName : string

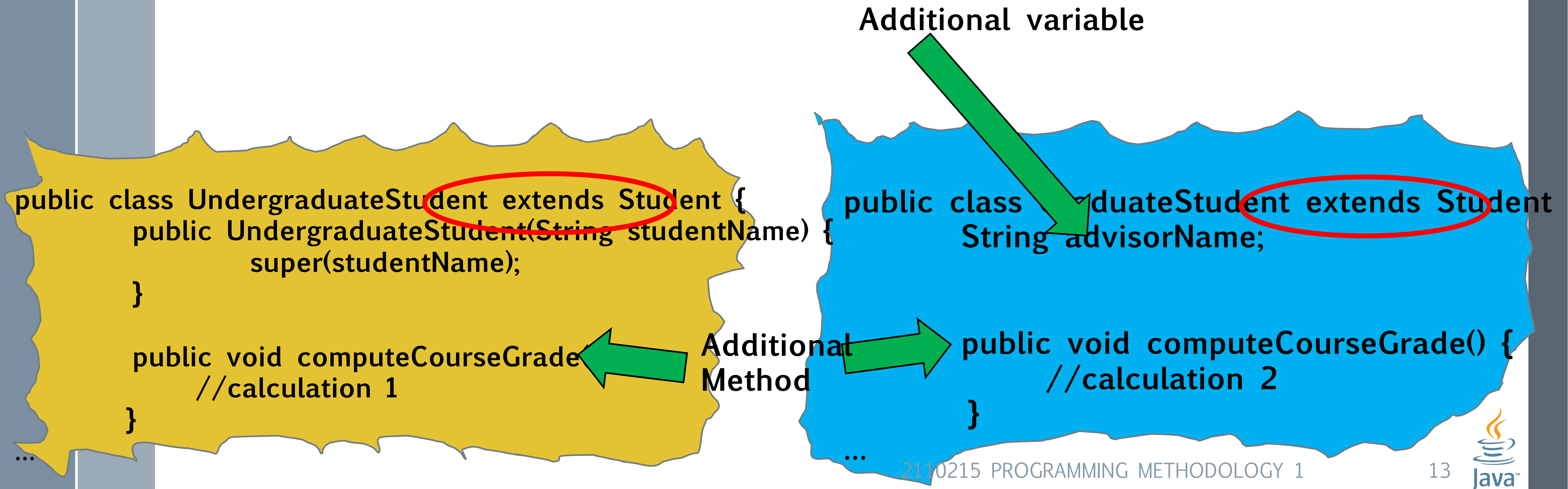
+GraduateStudent(in studentName : string)
+GraduateStuden(in studentName : string, in advisorName : string)
+setAdvisorName(in advisorName : string)
+getAdvisorName() : string
+computeCourseGrade()
+printName()

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;

…
```



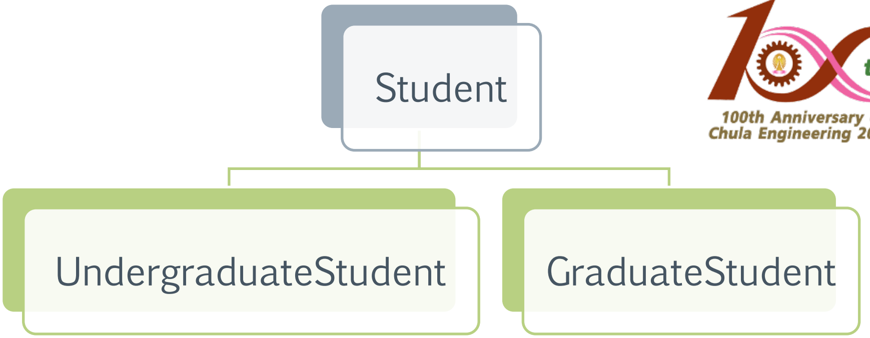
Additional variable

```
public class UndergraduateStudent extends Student {
        public UndergraduateStudent(String studentName) {
                super(studentName);
        }

        public void computeCourseGrade() {
            //calculation 1
        }
…
```

Additional Method

```
public class GraduateStudent extends Student
        String advisorName;


        public void computeCourseGrade() {
            //calculation 2
        }
…
```

Save time & Reduce errors



› Is there anything wrong in the following code?
  - Is it possible to have “ArrayIndexOutOfBound es
  - If yes, should this issue also happen in Student’s subclas Yes
  - How many method should we fix the issu java

Student.java (with ArrayIndexOutOfBound)

```
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 (1)

```
class Student{
…
        public void printName() {
                System.out.println("Student [" + name + 
"]");
        }
…
```

Accidently mistype!!!
Treated as additional method

```
class GraduateStudent extends Student {
…
        public void printname() {
                System.out.println("GraduateStudent [" + name + "]");
        }
…
```

Override method in parent class (2)

```
class Student{
…
        public void printName() {
                System.out.println("Student [" + name +
"]");
        }
…
```

Same signature!!!
And @Override
to prevent error

```
class GraduateStudent extends Student {
…
        @Override
        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]
```

Subtype polymorphism

instanceof

StudentTest2.java

```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.
```

Casting (Type conversion)

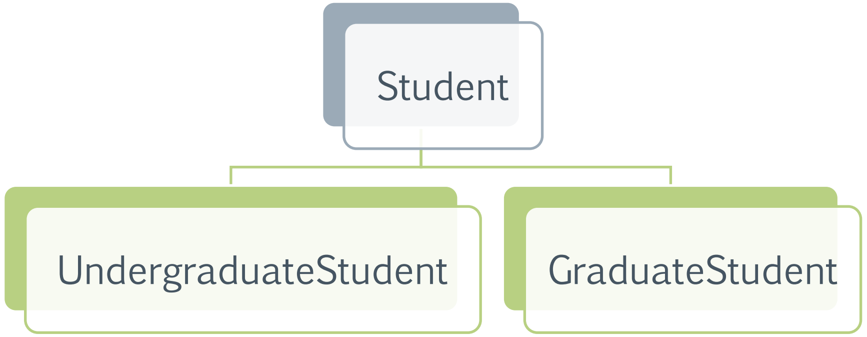
- int n = (int) 5.1534;
- double x = (double) n;   // x = n; is OK

- TypeA varA = (TypeA) varB;
  - (TypeA) is casting operator in Java

Up/down casting

StudentTest3.java

```
public class StudentTest3 {
   public static void main(String[] args) {
      // upcasting (automatically)
      Student s1 = new GraduateStudent("Nat");
      s1.printName();
      // downcasting that is OK
      GraduateStudent g = (GraduateStudent) s1; // OK because s1 is actually a Graduate student
      // 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)
```

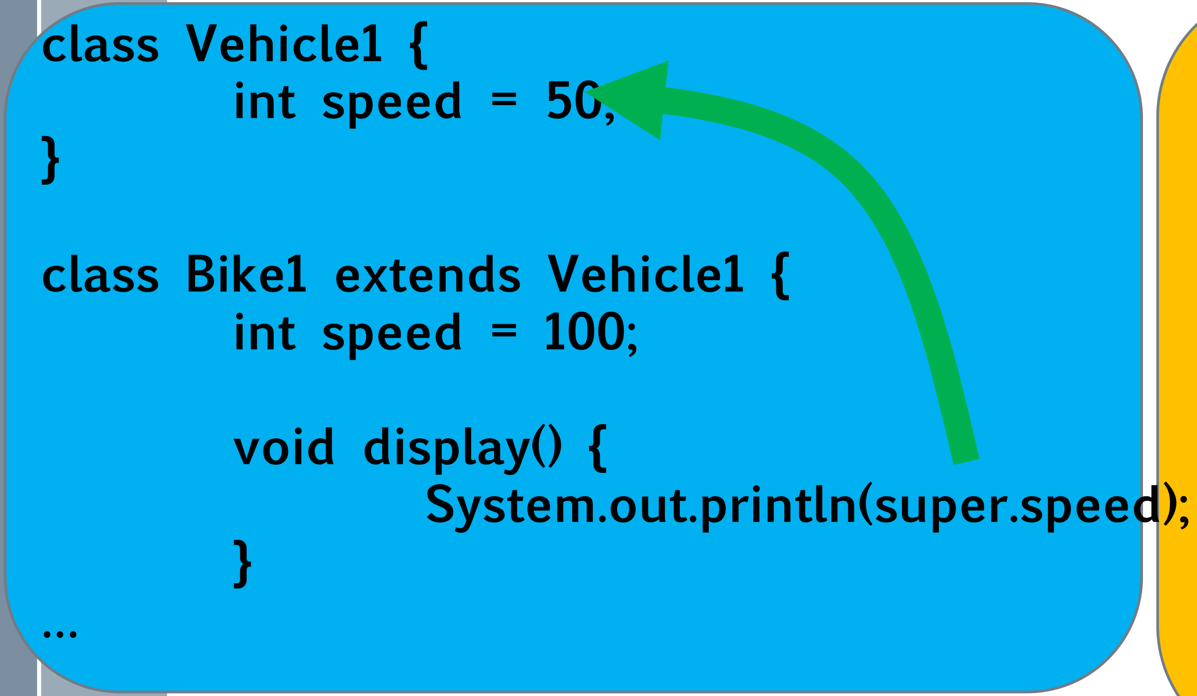
Keyword ‘**super**’

- 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.

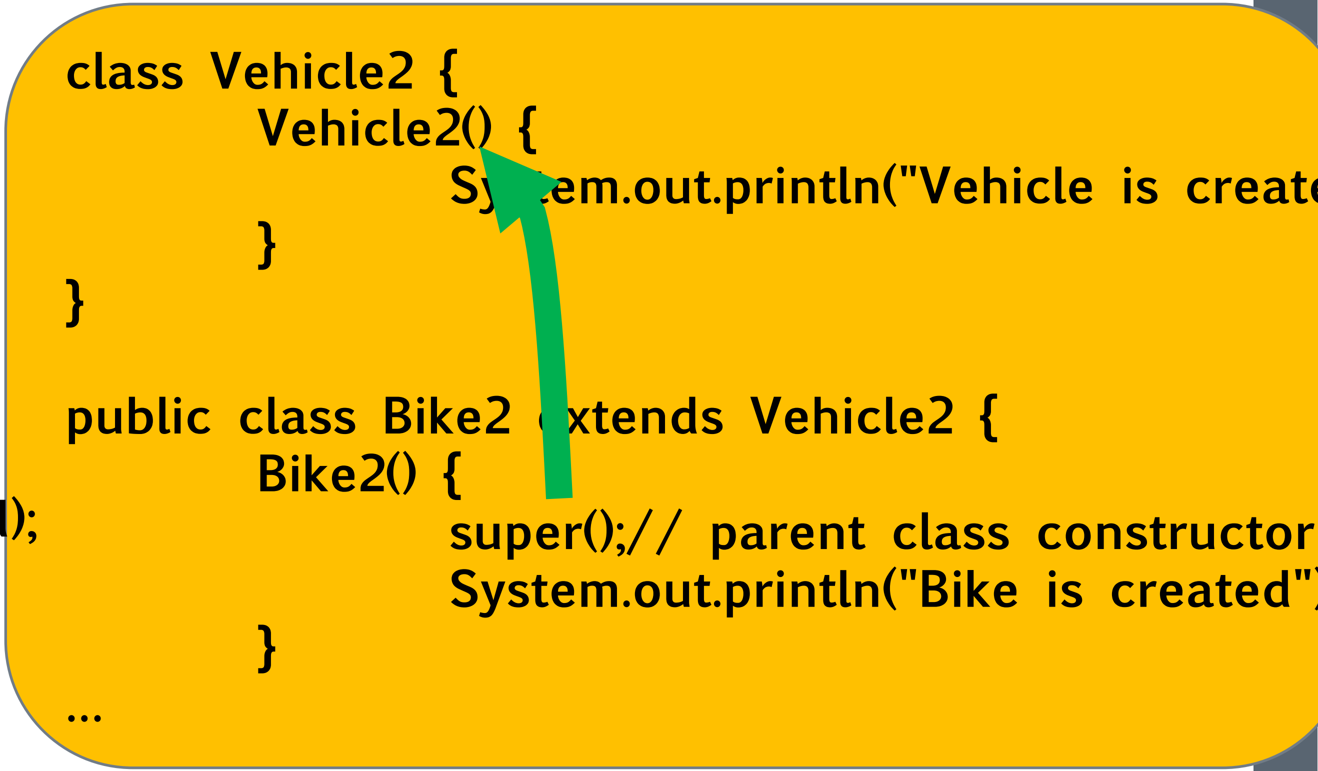
Bike1.java Bike2.java Student1.java Student2.java

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 create
        }
}

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

Result

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

ClassCreation.java

```java
public class ClassCreation {
   public static void main(String[] args){
      C c1 = new C(5);
   }
}

class A {
   A() {
      System.out.println("class A");
   }
}
```

```java
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);
    }
}
```

Instance Creation Mechanism (cont.)

- When superclass contains default constructor
  - Execution of superclass constructor transparent
  - For example, C → B → 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

Error in instantiate. (1)

```
class MyClass {
  int x;  // for a valid state x must be >= 0

  public MyClass(int x) {
    this.x = x;
  }
}

MyClass anObj = new MyClass(-1);
```

Syntax correct but anObj will be in an invalid state. This may cause bugs which is very difficult to find.

Error in instantiate. (2)

```
class MyClass {
  int x;  // for a valid state x must be >= 0

  public MyClass(int x) {
    if (x >= 0)
      this.x = x;
  }
}

MyClass anObj = new MyClass(-1);
```

Syntax correct, anObj is in a valid state (but not the input value)

The user might not aware of this consequence since no ERROR.

This may cause bugs which is very difficult to find.

Error in instantiate. (3)

```
class MyClass {
   int x;   // for a valid state x must be >= 0

   public MyClass(int x) throws Exception {
      if (x >= 0)
         throw new Exception();
   }
}

MyClass anObj = new MyClass(-1); // error
```

Syntax correct

The program will throws the exception make the client class need to handle it.

Error in instantiate. (3)

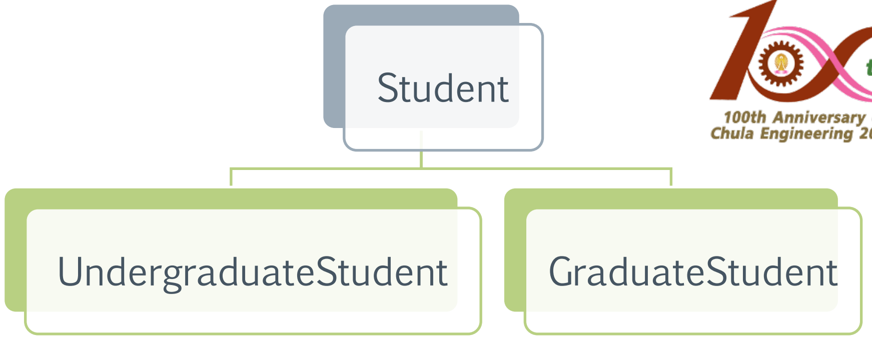
```
class MyClass {
   int x;   // for a valid state x must be >= 0

   public MyClass(int x) throws Exception {
      if (x >= 0)
         throw new Exception(“x must be >= 0”);
   }
}

try {
   MyClass anObj = new MyClass(-1);
} catch (Exception e) {
   // handle here, in this case just stop the program
   System.exit(-1);
}
```

Syntax correct

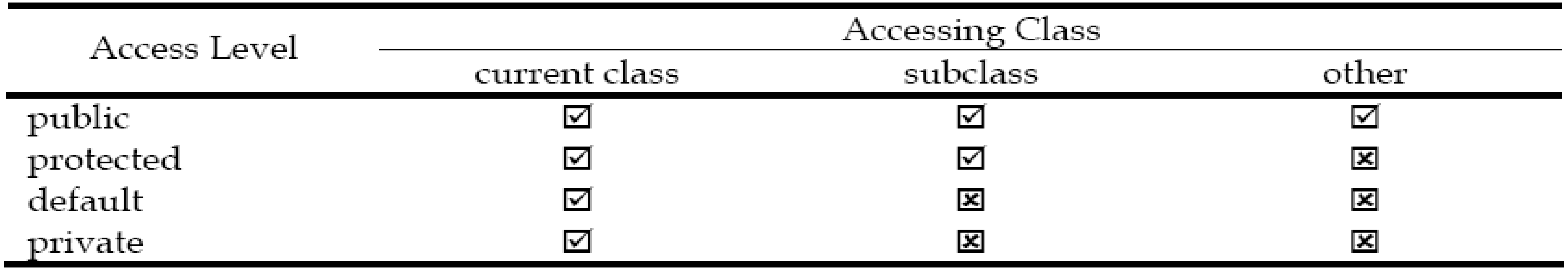
The program will throws the exception make the client class need to handle it.

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
public	☑	☑	☑
protected	☑	☑	☒
default	☑	☒	☒
private	☑	☒	☒

Methods You Cannot Override

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

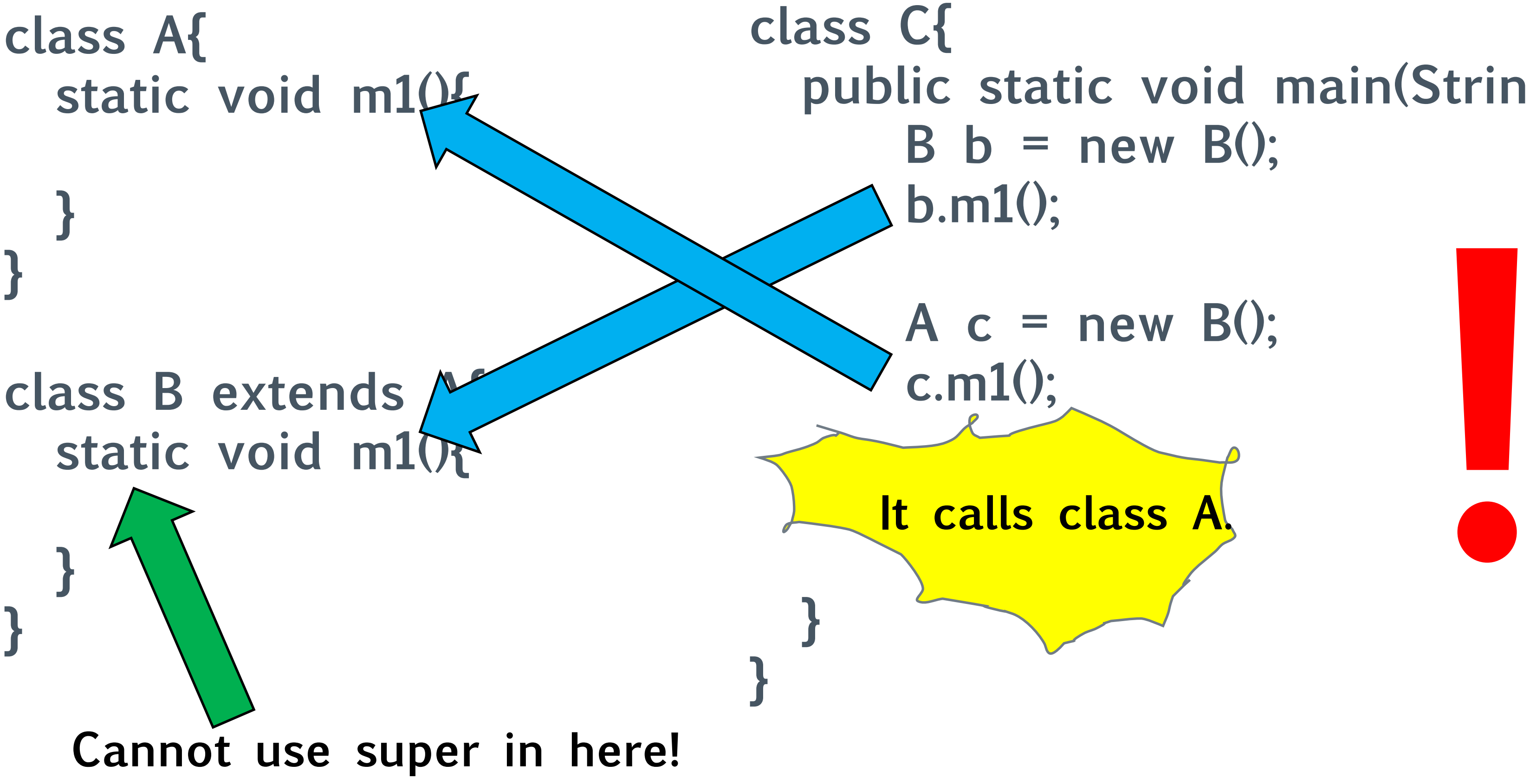
A Subclass Cannot Override `static` Methods in Its Superclass

- 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(){

   }
}
```

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

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


   }
}
```

It calls class A.

!

Cannot use super in here!

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

```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 called at runtime.
    s = g;
    s.printName();
    s = u;
    s.printName();
  }
}
```

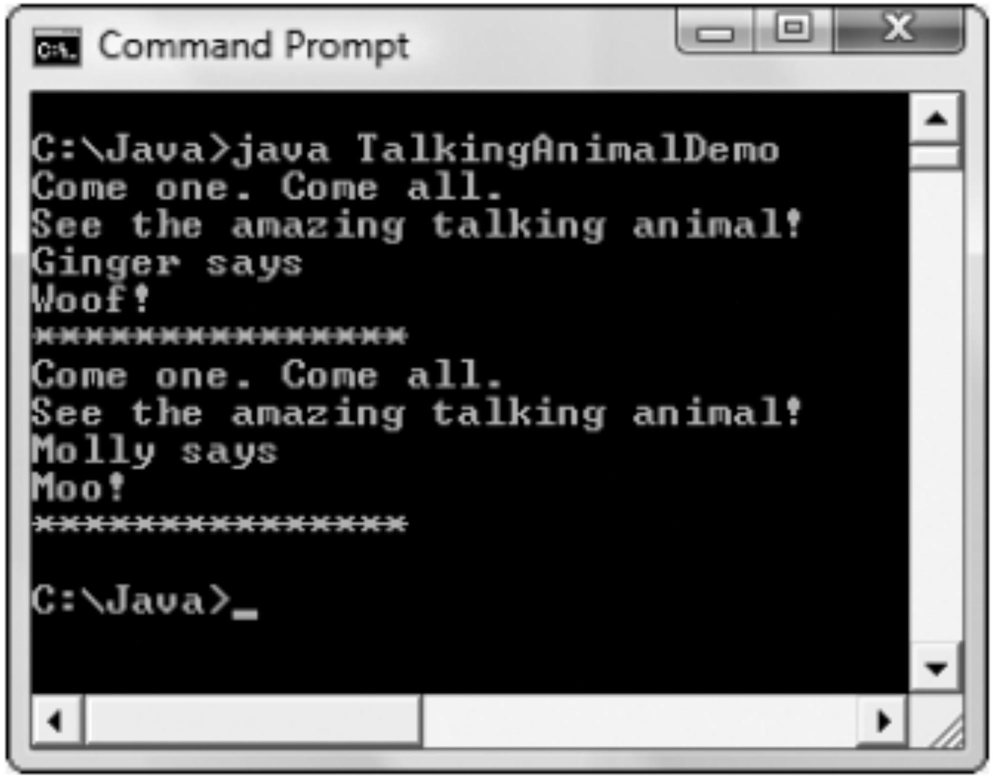
Result

```
GraduateStudent [Nat]
UndergraduateStudent [Toey]
```

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

1

```java
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>_
```

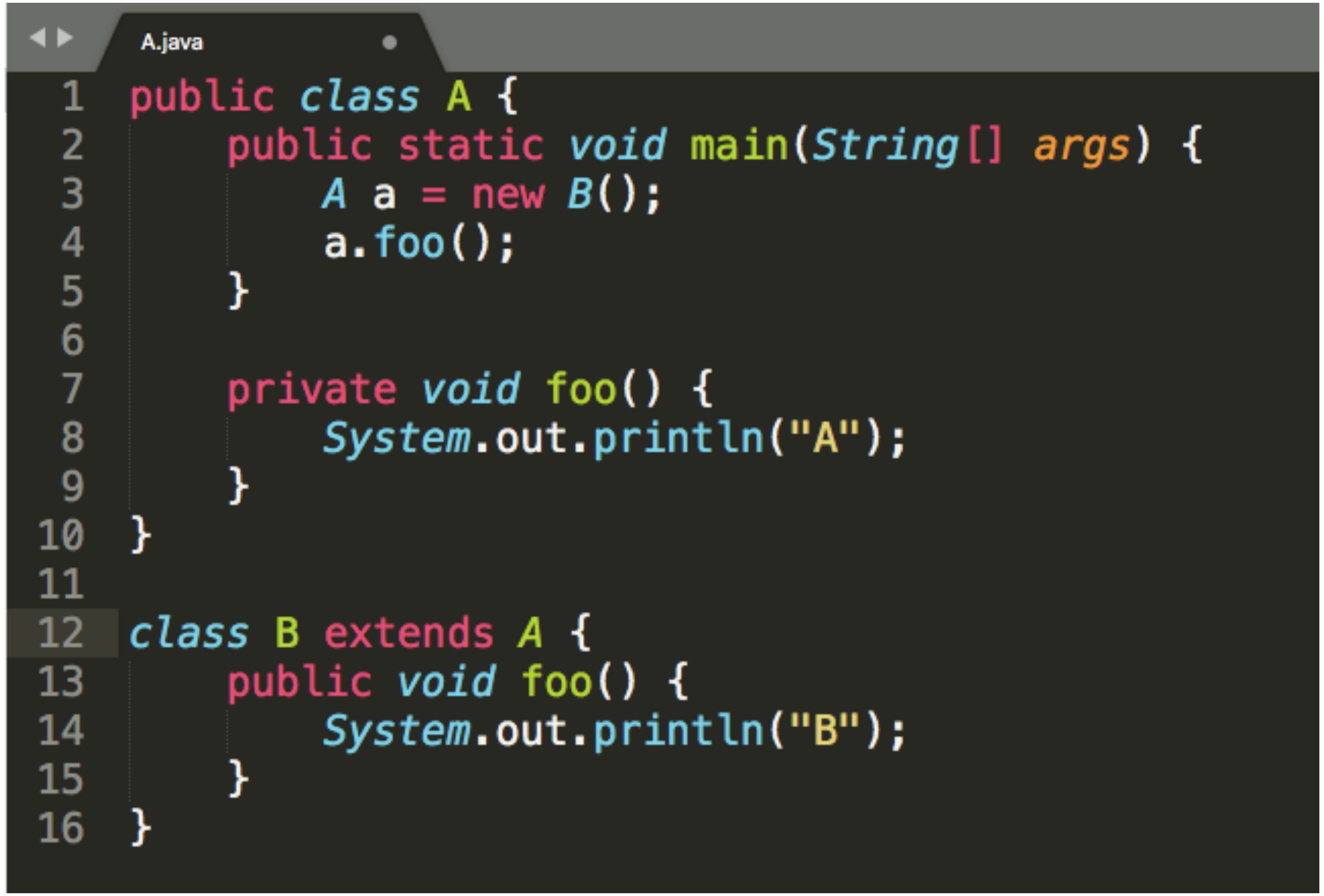
Creating Arrays of Subclass Objects 2

- 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?



```java
public class A {
    public static void main(String[] args) {
        A a = new B();
        a.foo();
    }

    private void foo() {
        System.out.println("A");
    }
}

class B extends A {
    public void foo() {
        System.out.println("B");
    }
}
```

A

B

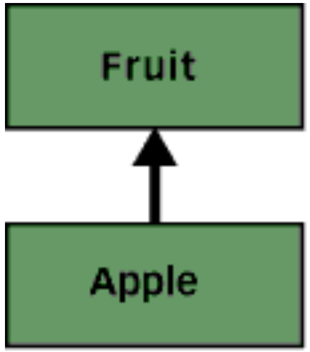
What happens at compile time and/or runtime?

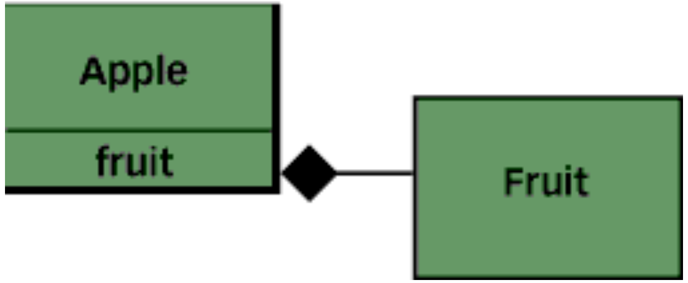
```
public class X {
  MODIFIER int value = 5;
}

For MODIFIER ->
    public, protected, private, unspecified
```

- Y subclasses X and is in the same package;
- Y subclasses X and is in a different package;
- Y does not subclass X and is in the same package;
- Y does not subclass X and is in a different package

Inheritance vs. Composition

https://www.javaworld.com/article/2076814/core-java/inheritance-versus-composition--which-one-should-you-choose-.html



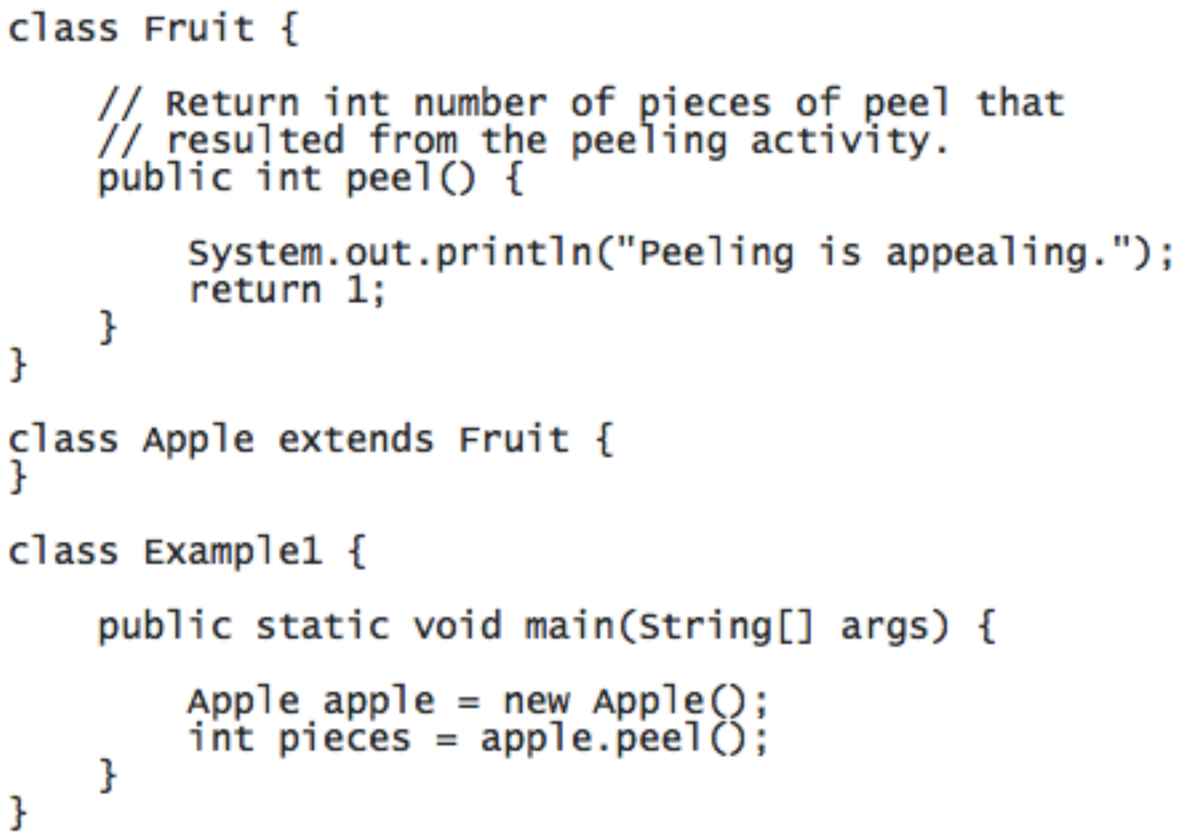


```
class Fruit {
    //...
}
class Apple extends Fruit {
    //...
}
```

Easy to add a new type of Fruit that has is-a relation with Fruit

```
class Fruit {
    //...
}
class Apple {
    private Fruit fruit = new Fruit();
    //...
}
```

Using Inheritance



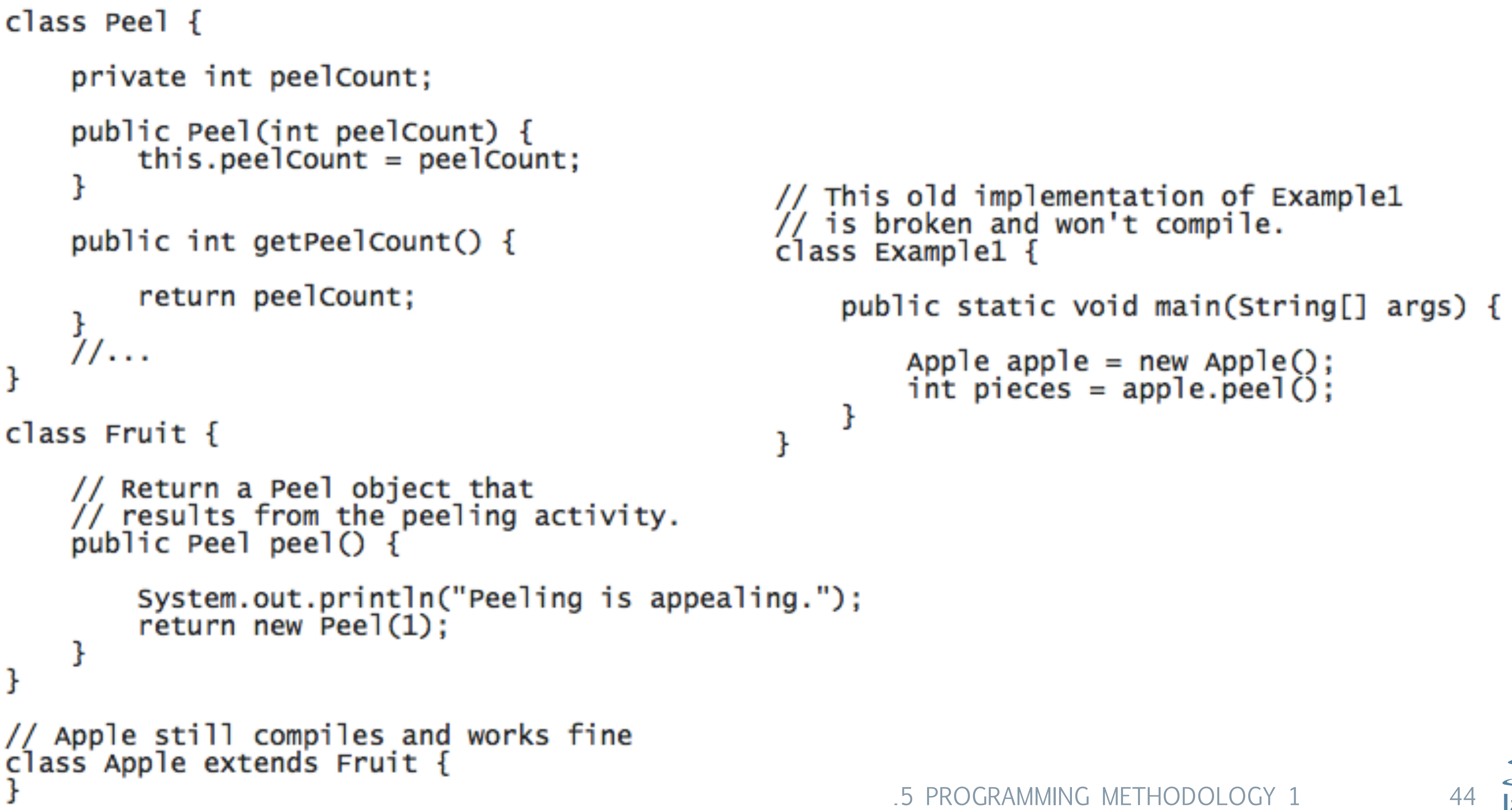
```
class Fruit {

    // Return int number of pieces of peel that
    // resulted from the peeling activity.
    public int peel() {

        System.out.println("Peeling is appealing.");
        return 1;
    }
}

class Apple extends Fruit {
}

class Example1 {

    public static void main(String[] args) {

        Apple apple = new Apple();
        int pieces = apple.peel();
    }
}
```

Problem!!!

What happen if Fruit change the return type of its public method, i.e. peel()?

Old program may fail.

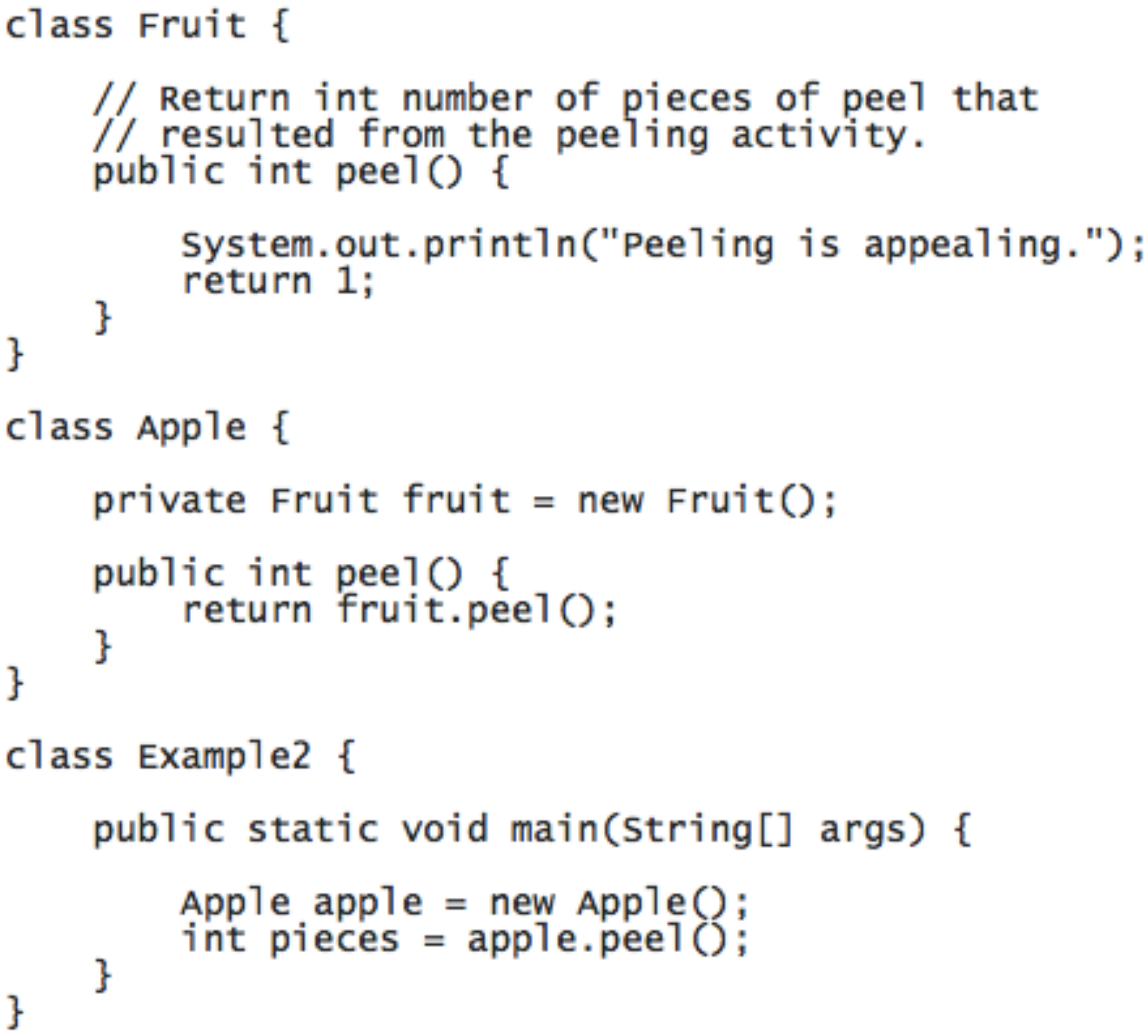
Using Inheritance



```
class Peel {

    private int peelCount;

    public Peel(int peelCount) {
        this.peelCount = peelCount;
    }

    public int getPeelCount() {

        return peelCount;
    }
    //...
}

class Fruit {

    // Return a Peel object that
    // results from the peeling activity.
    public Peel peel() {

        System.out.println("Peeling is appealing.");
        return new Peel(1);
    }
}

// Apple still compiles and works fine
class Apple extends Fruit {
}
```

```
// This old implementation of Example1
// is broken and won't compile.
class Example1 {

    public static void main(String[] args) {

        Apple apple = new Apple();
        int pieces = apple.peel();
    }
}
```

Using Composition



```
class Fruit {

    // Return int number of pieces of peel that
    // resulted from the peeling activity.
    public int peel() {

        System.out.println("Peeling is appealing.");
        return 1;
    }
}

class Apple {

    private Fruit fruit = new Fruit();

    public int peel() {
        return fruit.peel();
    }
}

class Example2 {

    public static void main(String[] args) {

        Apple apple = new Apple();
        int pieces = apple.peel();
    }
}
```

Using Composition

```
class Peel {

    private int peelCount;

    public Peel(int peelCount) {
        this.peelCount = peelCount;
    }

    public int getPeelCount() {

        return peelCount;
    }
    //...
}

class Fruit {

    // Return int number of pieces of peel that
    // resulted from the peeling activity.
    public Peel peel() {

        System.out.println("Peeling is appealing.");
        return new Peel(1);
    }
}
```

```
// Apple must be changed to accomodate
// the change to Fruit
class Apple {

    private Fruit fruit = new Fruit();

    public int peel() {

        Peel peel = fruit.peel();
        return peel.getPeelCount();
    }
}
```

```
// This old implementation of Example2
// still works fine.
class Example1 {

    public static void main(String[] args) {

        Apple apple = new Apple();
        int pieces = apple.peel();
    }
}
```

More readings

- https://stackify.com/oop-concepts-composition/
- https://www.thoughtworks.com/insights/blog/composition-vs-inheritance-how-choose
- https://www.javaworld.com/article/2076814/core-java/inheritance-versus-composition--which-one-should-you-choose-.html

Exercise

yourLOGO
HERE
First M. Lastname
Title
Card Number

PARAGON
EMPORIUM
THE MALL
MCard
7102 0000 0000 0000
CARD HOLDER

world
SCB
ไทยพาณิชย์
BEYOND
5404 9201 9999 9999
GOOD THRU
99/99
THAI PANICH
mastercard

SCB
ไทยพาณิชย์
ATM
0000 0001 9061 3111
GLENN RICHARDSON
FOUNDATION MEMBER

M Card
SCB
M LIVE
4773 7699 9999 9999
99/99
THAI PANICH
VISA
Platinum

KBank
K-DEBIT Card
8888 8888 8888 8888
MONTH/YEAR
EXP 88/88
PRIVILEGED MEMBER
VISA