Unit 6:
Major Object-Oriented Features

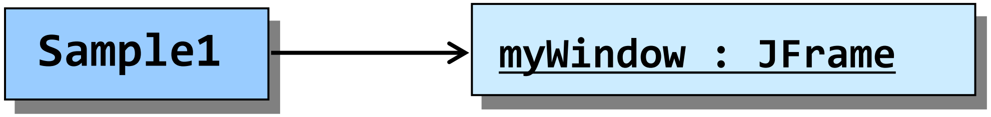
Object-Oriented Programming (OOP)
CCIT 4023, 2025-2026

U6: Major Object-Oriented Features

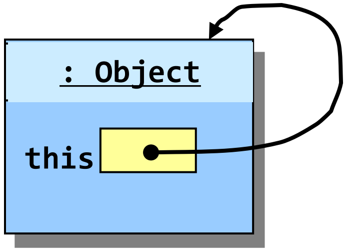
- Self-Accessing: Use of the keyword **`this`**
- Overloading Constructors and Methods
- Encapsulation (Information Hiding)
- Inheritance
  - With Encapsulation, Constructor, and keyword **`super`**
  - Overriding Method
  - Method `toString()`, a Special "Inherited" Method
- Polymorphism

Self-Accessing: Using Keyword this

- We may reference an object from another, e.g.
  - we reference a JFrame object, from Sample1



- Apart from referencing other objects, an object can also reference to itself, with keyword **this**
- The keyword this is called a ***self-referencing pointer*** because it refers to the receiving object of a message from this object's method
- The keyword this can be used in different ways



Case 1: Using this to access Fields or Methods

- We use of this to access a field or call a method of the object itself
- We can have the same name to a *local variable*, *parameter*, or *field*
- The identifier first refers to the local variable or parameter
- In the example below, this must be used to refer to a field age of its own if this is *hidden* by a local variable age

```
public class Person {

  int age; // field

  public void setAge(int age) {// parameter named age
   // below: assign parameter value to field
      this.age = age;
  }
}
```

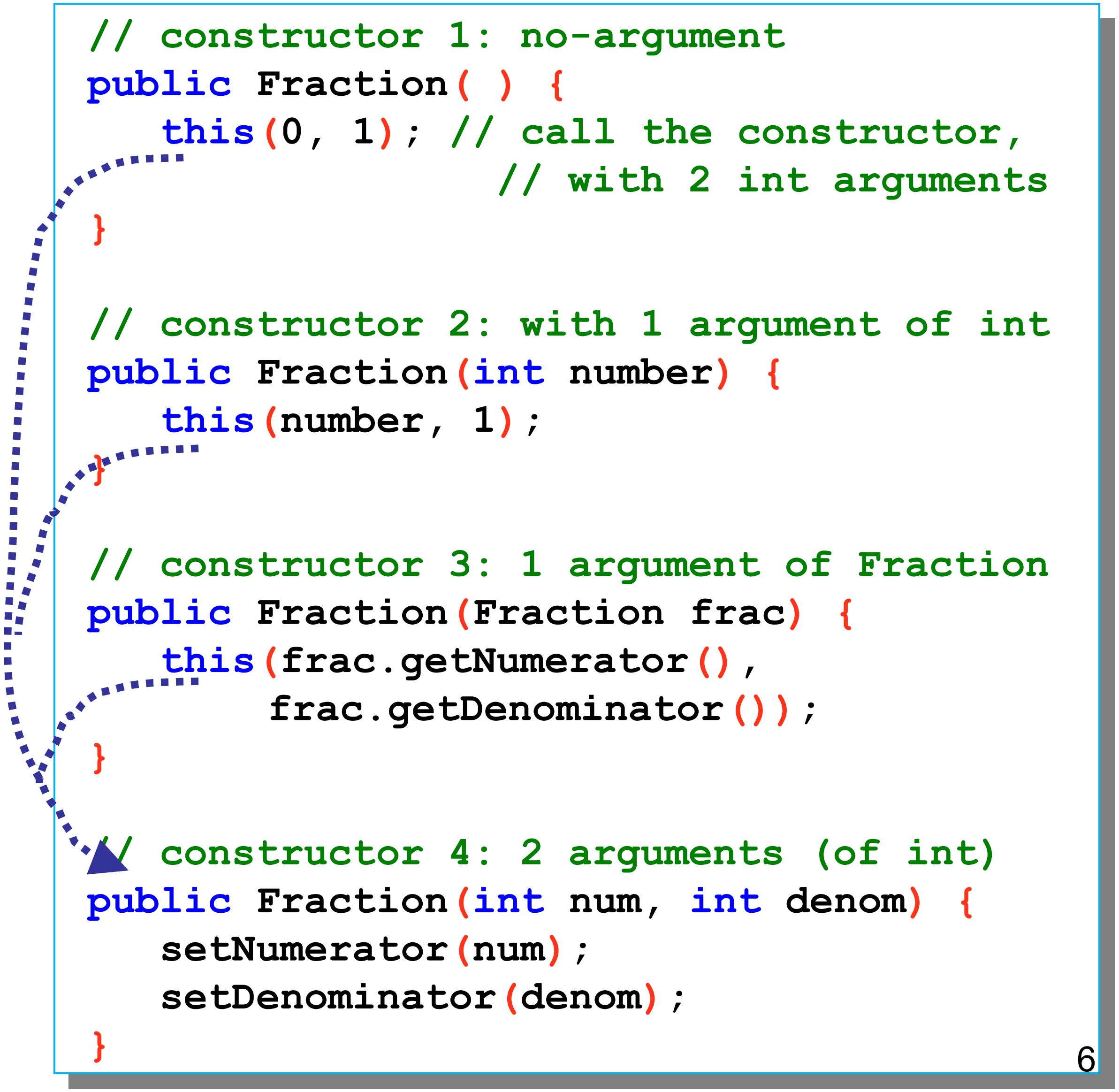
Implicit use of the reserved word `this`

- The use of the reserved word **`this`** is actually *optional* in many cases
- If we do not include it explicitly, the compiler will properly insert the reserved word implicitly
- An example of calling object's own method below:

```
public class Sample {
    public void m1() {
        // ...
    }
    public void m2() {
        // 2 statements below are basically identical
        m1(); // interpreted by the compiler as this.m1();
        this.m1(); // same as above, explicitly using this
    }
}
```

Case 2: Using this for Constructors

- To *call a specific constructor from another constructor of the same class*, we use the reserved word this in a way similar to calling another method with proper parameters
  - as the example, calling this() with 2 parameters of int type would then call the constructor with 2 int parameters.



```
// constructor 1: no-argument
public Fraction( ) {
   this(0, 1); // call the constructor,
                 // with 2 int arguments
}

// constructor 2: with 1 argument of int
public Fraction(int number) {
   this(number, 1);
}

// constructor 3: 1 argument of Fraction
public Fraction(Fraction frac) {
   this(frac.getNumerator(),
        frac.getDenominator());
}

// constructor 4: 2 arguments (of int)
public Fraction(int num, int denom) {
   setNumerator(num);
   setDenominator(denom);
}
```

Overloading Constructors and Methods

- With the **overloading** feature in Java, we may define more than one constructor

- Constructors can share the same name as long as
  - they have a *different number of parameters* (Rule 1) or
  - their parameters are of *different data types* when the number of parameters is the same (Rule 2)

```
   public Person( ) { ... }
   public Person(int age) { ... }
```

Valid ✓

```
   public Pet(int age) { ... }
   public Pet(String name) { ... }
```

Valid ✓

```
   public Fraction(int number) { ... }
   public Fraction(int n) { ... }
```

Invalid ✗

Overloading Constructors and Methods

- The same rules apply to overloading methods
  - This is how we can define more than one method with the same method name in the same class
  - Overloading methods should have different *method signatures*
    - same method name but different parameter lists, the parameter types and their order

```
public void myMethod(int x, int y) { ... }
public void myMethod(int x) { ... }
```

valid ✓

```
public void myMethod(double x) { ... }
public void myMethod(String x) { ... }
```

valid ✓

Major Object-Oriented Features (RECAP)

- **Inheritance** relationship
  - A mechanism designed for different entities (subclasses) that share common features (from a superclass)
- **Encapsulation** of data and methods
  - Internal components are encapsulated from outside (information hiding)
- **Polymorphism**
  - Ability to morph into many (poly) different forms
    - E.g. Same (method / message / name) with different forms of functions / behaviours / results

Encapsulation (Information Hiding)

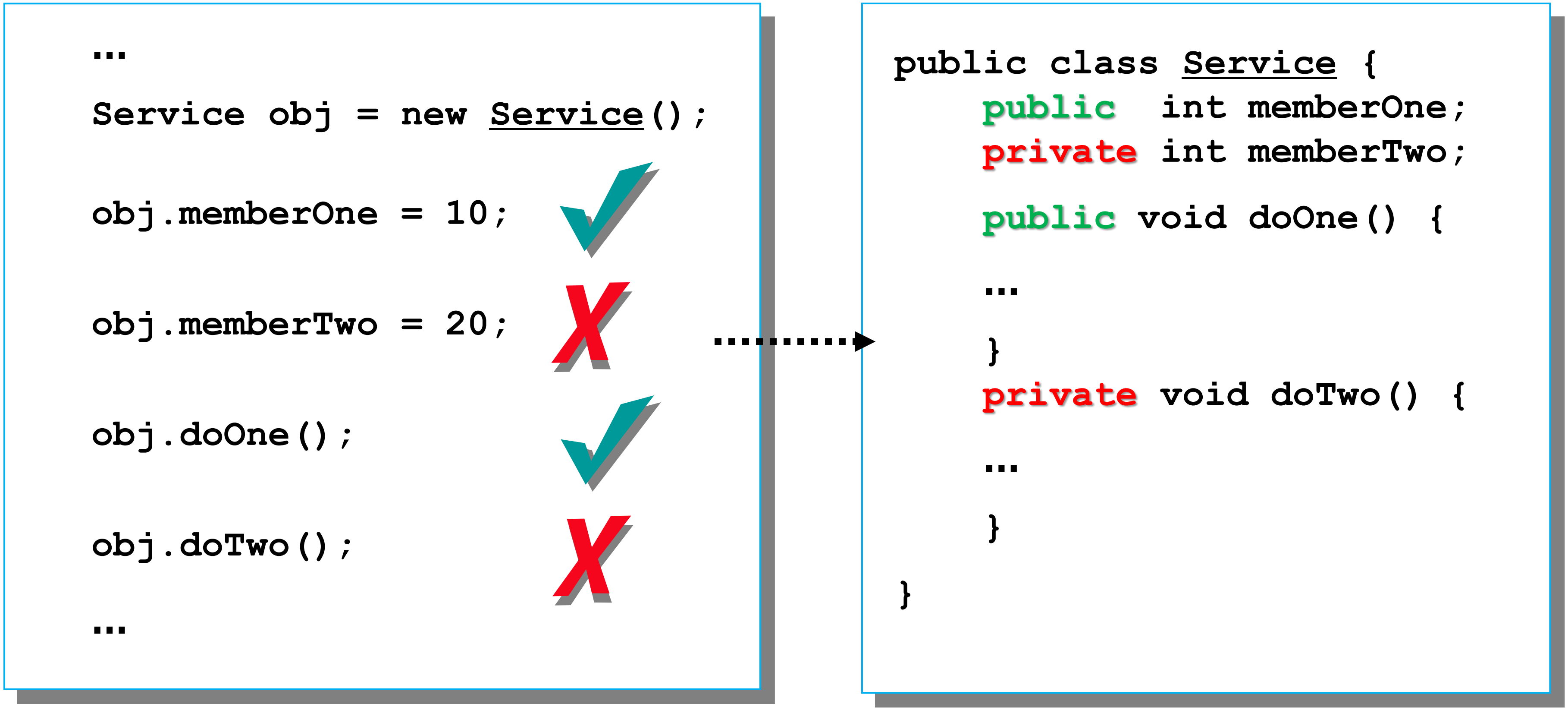
- Internal components of a class are encapsulated, and hidden from the clients
- **Encapsulation** is also called **information hiding**
- Benefits:
  - Components can be replaced/ revised easily
  - Protects the integrity (prevent users from setting the internal data into an invalid or inconsistent state)
  - Reduces complexity and thus increases robustness (limit the interdependencies between software components)

Access (Visibility) Modifiers

- There are different access (visibility) types associated to encapsulate implementation details

- Java includes keywords `public`, `private`, `protected` for four different access types (and a default one without access modifier)
  - `public` fields and methods are accessible to everyone
  - `private` fields and methods are accessible only within the class itself internally
  - `protected` fields and methods access is an intermediate level of access between public and private (intermediate level)
    - Accessible to its subclasses and classes in the same package
  - The fourth type of access level which is without access modifier is also known as *package-private*
    - This default situation is related to the case of “accessible only within the same package”

Accessibility Example



```
...
Service obj = new Service();

obj.memberOne = 10;      ✓

obj.memberTwo = 20;      ✗

obj.doOne();             ✓

obj.doTwo();             ✗

...
```

```
public class Service {
    public  int memberOne;
    private int memberTwo;

    public void doOne() {
        ...
    }
    private void doTwo() {
        ...
    }
}
```

Client

Service

Guidelines of Visibility Modifiers
(Fields Are Usually `private`)

- Typically *fields* are the implementation details of the class attributes/states/fields, so they should be invisible to the clients.
  - Declare them as `private`
  - Exception: Constants can (should) be declared `public` if they are meant to be used directly by the outside methods

- Guidelines in determining the visibility of fields (data members/attributes/variables) and methods:
  - Declare instance variables and class constants `private` *for internal purposes*
  - Declare instance methods `private` if they are used *only* by the other methods *in the same class*
  - Declare the class constants `public` if you want to make their values directly readable by the client programs

Access Private Fields

- Private fields cannot be accessed by an object from outside the class.
- To make a private field accessible, it is common to define a *get* method (getter) to return its value, and a *set* method (setter) to set a new value

```java
public class Service {
    public int memberOne;
    private int memberTwo;
    public int getMemberTwo() {
        return memberTwo;
    }
    public void setMemberTwo(int newMember) {
        memberTwo = newMember;
    }
}
```

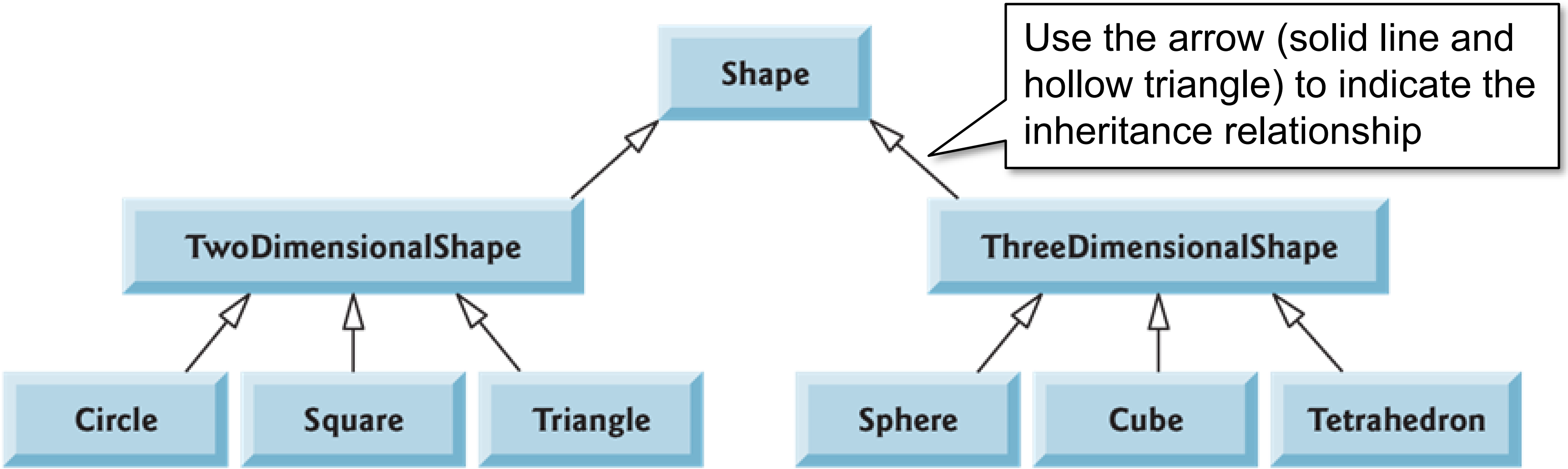
Inheritance

- **Inheritance** is an important feature in OOP to design two or more entities that are different but share common properties and/or behaviors.
  - The common features are defined in a general class that can be shared by other classes

- When creating a class, rather than declaring completely new members, we may designate that the *new specialized class* should inherit the members of an *existing general class*
  - Existing general class is called **superclass**
  - New specialized classes are called **subclasses**
  - In Java, each class is allowed to have ONLY ONE direct superclass, and each superclass may have many direct subclasses

Inheritance Hierarchy

- A superclass exists in a hierarchical relationship with its subclasses
- E.g.: In the hierarchy below, the “root“ class `Shape`, has two subclasses (`TwoDimensionalShape` and `ThreeDimensionalShape`); and each subclass has its own three subclasses, etc.



Use the arrow (solid line and hollow triangle) to indicate the inheritance relationship

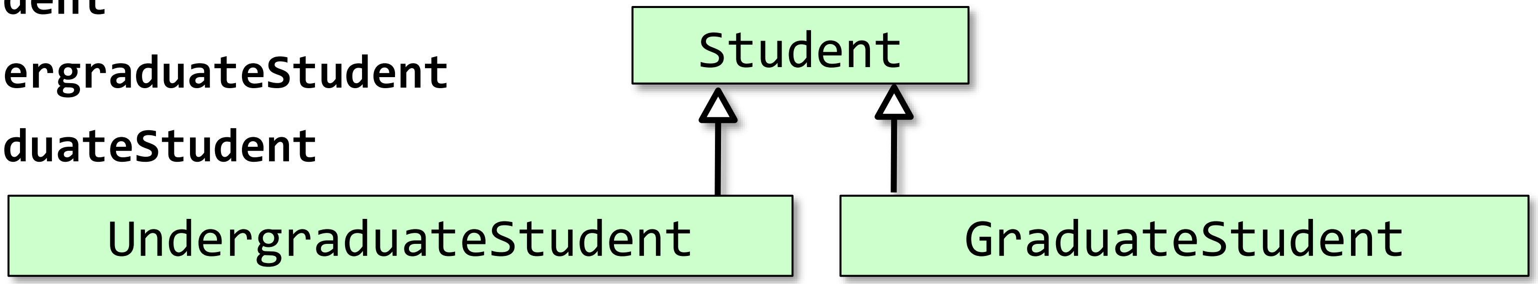
Defining Classes with Inheritance

- Case Study:
  - Suppose we want implement a student record system that contains both **undergraduate** & **graduate** students
  - Each student's record will contain his or her **name**, **three test scores**, and the **final course grade**
  - The **formula for determining the course grade** is different for graduate students than for undergraduate students
  - There are two ways to design the classes to model undergraduate and graduate students:
    1. Define two unrelated classes (*Undergraduates* and *Graduates)*
    2. Model the two types of students by using classes that are related in an **inheritance hierarchy**

Designing Student Record System with Inheritance

- We may design three classes:

  1. **`Student`**
  2. **`UndergraduateStudent`**
  3. **`GraduateStudent`**

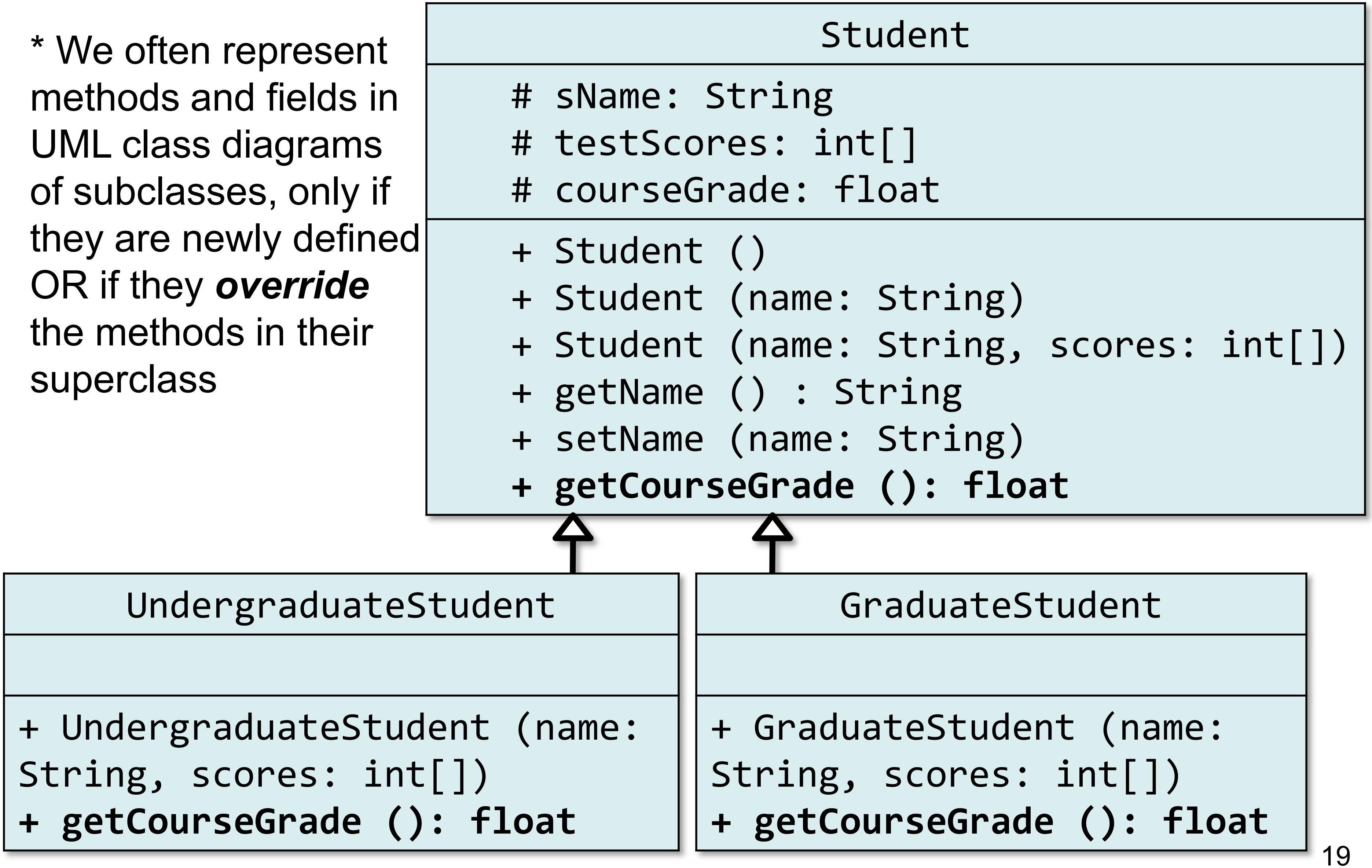


- The *`Student`* class (superclass) will incorporate behavior and data **common** to both *`UndergraduateStudent`* and *`GraduateStudent`* classes (subclasses)

- The *`UndergraduateStudent`* class and the *`GraduateStudent`* class will each contain behaviors and data **specific** to their respective objects

Inheritance (UML Class Diagram)

* We often represent methods and fields in UML class diagrams of subclasses, only if they are newly defined OR if they ***override*** the methods in their superclass



Student

# sName: String
# testScores: int[]
# courseGrade: float

+ Student ()
+ Student (name: String)
+ Student (name: String, scores: int[])
+ getName () : String
+ setName (name: String)
+ getCourseGrade (): float

UndergraduateStudent

+ UndergraduateStudent (name: String, scores: int[])
+ getCourseGrade (): float

GraduateStudent

+ GraduateStudent (name: String, scores: int[])
+ getCourseGrade (): float

Declare a Subclass

- When we extend an existing class, we ***inherit*** the attributes and methods of its superclass (the existing class)
  - This makes programming much easier, and simpler, as we don't re-invent the wheel each time; we extend from the other classes
  - We use the keyword **`extends`** to define the inheritance
  - Example:

```
public class Circle extends GeometricObject {
// Override the getInfo() method defined in GeometricObject
  public String getInfo() {
        // super.getInfo() calls method getInfo() of its superclass
        return super.getInfo() + "\nradius is " + radius;
  }
}
```

Inheritance with Encapsulation
(Public, Private and Protected Access Modifiers)

- `public` fields and methods are accessible to everyone (inside or outside the same package)

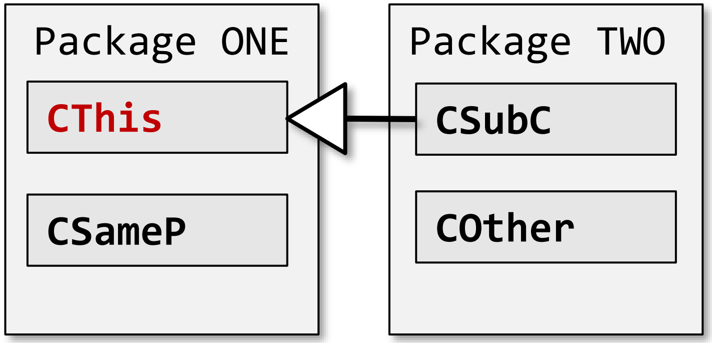
- `private` fields and methods are accessible only within the same class itself

- `protected` access is an intermediate level of access between public and private
  - A class `protected` members can be visible and accessed by the members of
    - its *subclasses* (descendant classes), if any
    - all classes in the same package, including the same class itself

- No modifier: Situation (e.g. fields and methods) without access modifier is also known as *package-private*
  - It is accessible/visible ONLY within its own package

Accessibility Example

- The below figure (left) shows the four classes in this example and how they are related
- The table (right) shows where the *members of the class* **CThis** are visible for each of the access modifiers that can be applied to them
  - Class `CSubC` is a subclass of class `Cthis` (in another package)



Modifier	CThis	CSameP	CSubC	COther
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
**no modifier*	Y	Y	N	N
`private`	Y	N	N	N

* No modifier also known as *package-private.*

Accessibility Summary

- The 1st column indicates if the **class itself** has access to the member defined by the access level
  - A class always has access to its own members
- The 2nd column indicates if classes in the **same package** as the class (regardless of their parentage) have access
- The 3rd column indicates if **subclasses** of the class declared *outside this package* have access
- The 4th column indicates if **all classes** have access to the member

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

Reference: http://download.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Inheritance and Constructors

- Unlike other members (fields and methods) of a superclass, constructors of superclass are ***not*** inherited to its subclasses
  - Thus, we need define a constructor for a class, or use the default constructor added by the compiler
- Every class has a superclass (except root class)
  - If the class declaration does not explicitly designate the superclass with the **`extends`** clause, then the class's superclass is the **`Object`** (the root) class in Java
- Essentially, each (non-abstract) subclass should have at least one constructor, and each constructor should first call the constructor of its superclass
  - *The first execution statement in a constructor body is calling its superclass's constructor*

Inheritance and Constructors
(with keyword super)

- If we do not define any constructor for our class, a default constructor is automatically generated

- If the constructor does not contain an explicit call to a superclass constructor, the compiler adds **super();** to call the superclass no-argument constructor as its first statement, e.g.

```
public class Person { }
```

Similar to (with a **default constructor**):

```
public class Person {
   public Person( ){ // default constructor added
      super(); // also added by the compiler
   }
}
```

* **super()** calls the superclass no-argument constructor.
* Similarly, **super(abc)** calls superclass constructor with 1 argument.

Inheritance and Constructors

```
public class Vehicle { // a class with 1 constructor below
      public Vehicle(String vNum) { }
}
```

Given the class above, the following cases will cause a compilation error:

- Case 1: Not having a matching constructor, e.g.:

```
Vehicle myV = new Vehicle(); // no matched constructor
```

- Case 2: The constructor calls the superclass's constructor with no arguments, but there's no matching constructor in the superclass, e.g.:

```
public class Truck extends Vehicle {
      public Truck(){} // here leads to call the super(),
             // where no matched constructor found
}
```

Inheritance and Constructors
(Brief Summary)

- **Class `Object` is the “root” Java class** at the top of the class inheritance hierarchy, which has no superclass
  - `Object` is a (direct or not) superclass of all classes
- If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass
  - In this case if the super class does not have a no-argument constructor, it causes a compile-time error
- If a subclass constructor invokes a constructor of its superclass, *either explicitly or implicitly*, there will be a whole chain of constructors called, all the way back to the constructor of `Object` in Java
  - It is also called constructor chaining

Overriding Methods

- Often there are situations the implementation of a specific method in the superclass is less useful for the subclass, but the method (name) itself is needed

- E.g. the getting salary method `getSalary()` applies to both superclass `Employee` and subclass `Salesperson`, but the details of calculation (method body implementation) are different (e.g. `Salesperson` is often with certain commission)

- In this case, we would better re-implement the method for the subclass `Salesperson`, and ***override*** the original implementation of the superclass `Employee` method.

Overriding Methods
(Access Overridden Methods with Keyword `super`)

- To **override** a superclass method, a subclass must declare a method with the *same name, same parameter, and same return type* as the superclass method, e.g.

```
// in the superclass Employee
public double getSalary() { return salary; }
```

```
// in the subclass Salesperson
public double getSalary() { // overriding method
       return ( super.getSalary() + commission );
}
```

- To assess the superclass *overridden* method from the subclass, by preceding the superclass method name with keyword `super` and a dot (**.**) separator. E.g.: `super.getName();`

Overloading vs. Overriding

- ***Overloading*** (for constructors and methods) means to define multiple constructors/methods with same name within the same class. E.g. constructors of the class `Student`.

```
// in the class Student
Student (String name){ //..
Student (String name, int grade) { //..
```

- ***Overriding*** method means to provide a new implementation for a method in the subclass, for overriding the same one in the superclass. E.g.

```
// in the superclass Employee
public double getSalary() { return salary; }
```

```
// in the subclass Salesperson
public double getSalary() { // overriding method
       return ( super.getSalary() + commission );
}
```

Example of Superclass / Subclass,
(BEFORE Inheritance)

“Potential” Superclass

```
public class Employee {
    public static int totEmp = 0;
    protected String empName;
    protected double stdSalary;
    protected double otSalary;

    Employee (String name, double sS,
      double oS) {
      //...
    }

    public String getEmpname() {
      return empName;
    }
    public double getSalary() {
      return stdSalary + otSalary;
    }

    public static int getTotEmp()
    { return totEmp; }
}
```

“Potential” Subclass

```
public class Salesperson {
    public static int totEmp = 0;
    protected String empName;
    protected double stdSalary;
    protected double otSalary;
    private double commission;
    Salesperson (String name, double sS,
      double oS, double commission) {
        //...
        this.commission = commission;
    }
    public String  getEmpname() {
      return empName;
    }
    public double getSalary() {
      return stdSalary + otSalary
                + commission ;
    }
    public static int getTotEmp()
    { return totEmp; }
}
```

Example of Superclass / Subclass
(AFTER Inheritance – 1)

Superclass

```
public class Employee {
     public static int totEmp = 0;
     protected String empName;
     protected double stdSalary;
     protected double otSalary;

     Employee (String name, double sS,
       double oS) {
       //...
     }
     public String getEmpname() {
       return empName;
     }
     public double getSalary() {
       return stdSalary + otSalary;
     }

     public static int getTotEmp()
     { return totEmp; }
}
```

Subclass

```
public class Salesperson extends Employee{
     ~~public static int totEmp = 0;~~
     ~~protected String empName;~~
     ~~protected double stdSalary;~~
     ~~protected double otSalary;~~
     private double commission;
     Salesperson (String name, double sS,
       double oS, double commission) {
          super(name, sS, oS);
          this.commission = commission;
     }
     ~~public String  getEmpname() {~~
     ~~  return empName; }~~
     public double getSalary() {
       return super.getSalary()
                    + commission;
     }
     ~~public static int getTotEmp()~~
     ~~{ return totEmp; }~~
}
```

Invoke Explicitly

Overriding Method

* *~~Strikethrough texts~~ are associated to inherited fields and methods.*

Example of Superclass / Subclass
(AFTER Inheritance – 2)

Superclass

```
public class Employee {
     public static int totEmp = 0;
     protected String empName;
     protected double stdSalary;
     protected double otSalary;

     Employee (String name, double sS,
        double oS) {
        //...
     }
     public String getEmpname() {
        return empName;
     }
     public double getSalary() {
        return stdSalary + otSalary;
     }

     public static int getTotEmp()
     { return totEmp; }
}
```

Subclass

```
public class Salesperson extends Employee{
     private double commission;





     Salesperson (String name, double sS,
        double oS, double commission) {
           super(name, sS, oS);
           this.commission = commission;
     }


     public double getSalary() {
        return super.getSalary()
                      + commission;
     }

}
```

* ~~Strikethrough texts~~ *are associated to inherited fields and methods.*

Example of Superclass / Subclass

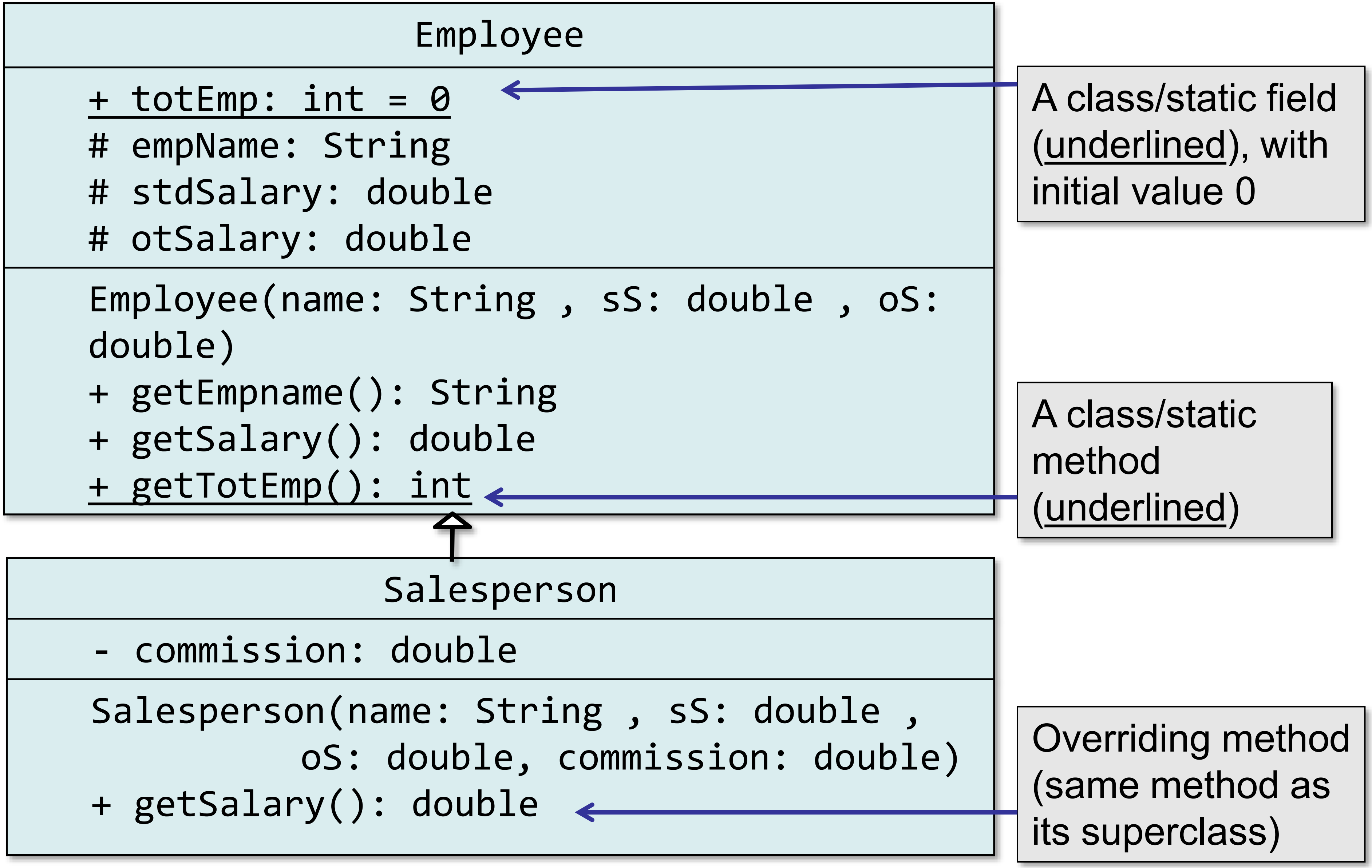
- We can create and access objects of superclass (`Employee`) and subclass (`Salesperson`), as the sample code below:

```
// main(), for self testing
public static void main(String[] args){
    System.out.println("*** For SELF-TESTING ONLY ***");
    Employee aE = new Employee("CHAN Tai Man",
        12345, 120);
    System.out.println("Salary of " + aE.empName
        + " is HK$" + aE.getSalary());
    Salesperson bS = new Salesperson("CHAN Siu Ming",
        12345, 120, 1234);
    System.out.println("Salary of " + bS.empName
        + " is HK$" + bS.getSalary());
}
```

```
*** For SELF-TESTING ONLY ***
Salary of CHAN Tai Man is HK$12465.0
Salary of CHAN Siu Ming is HK$13699.0
```

UML Class Diagram

(Superclass and Subclass, with Inheritance)



Employee

+ totEmp: int = 0
# empName: String
# stdSalary: double
# otSalary: double

Employee(name: String , sS: double , oS: double)
+ getEmpname(): String
+ getSalary(): double
+ getTotEmp(): int

Salesperson

- commission: double

Salesperson(name: String , sS: double ,
oS: double, commission: double)
+ getSalary(): double

A class/static field (underlined), with initial value 0

A class/static method (underlined)

Overriding method (same method as its superclass)

Preventing Inheriting and Overriding

- We may occasionally want to prevent classes from being extended (inherited), or methods from being overridden
  - This can be achieved by using the `final` modifier

```
public final class MyFinalC {
//... This class NOT to be inherited/extended
}
```

```
public class AClass {
   public final void aFinalM() {
      //... This method NOT to be overridden
   }
}
```

Method toString(), a Special "Inherited" Method

- When we directly print an object, we could get a strange information (which is related to the class type and reference address of the object)

```
Student amy = new Student("AU Amy");
System.out.println("Amy object is <" + amy + ">");
```

```
Amy object is <Student@19e0bfd>
```

- We may override and implement a **special method toString()** (inherited from the root class Object) of the class.
  - toString() method returns a string representation of the object.

```
public class Student {
    protected String sName;
    public Student(String name){ sName = name; }
// ..
    public String toString(){ return this.sName; }
```

```
Student amy = new Student("AU Amy");
System.out.println("Amy object is <" + amy + ">");
```

```
Amy object is <AU Amy>
```

Root Class in Java: `Object`

Reference Only

- The Java class `Object` is the root class and is a superclass of all other Java classes
- All class (and array types) inherit the ***methods of root class*** ***`Object`***:
  - `clone` is used to make a duplicate of an object
  - `equals` defines a notion of object equality
  - `finalize` is run just before an object is destroyed
  - `getClass` returns the Class object
  - `hashCode` returns a hash code value for the object
  - `wait`, `notify`, and `notifyAll` are used in concurrent programming
  - **`toString` returns a string representation of the object**

Polymorphism

- **Polymorphism** means ability to morph into many (poly) different forms
  - E.g. Same (method / message / name) with different forms of functions / behaviours / results
- There are various types of Polymorphism
  - Method overloading and method overriding may be regarded as certain forms of polymorphism
- Subtype polymorphism (subtyping) - It enables us to write programs that process objects that share the same superclass in the inheritance hierarchy

Polymorphism

- **Subtype polymorphism (subtyping)** allows a single variable to refer to different objects from subclasses in the same inheritance hierarchy

- For examples, the following codes are valid,
  - if `UndergraduateStudent` and `GraduateStudent` are subclasses of `Student`
  - if `Fruit` and `Meat` are subclasses of `Food`, and `Apple` is a subclass of `Fruit`

```
Student sONE = new UndergraduateStudent();
// ...
Student sTWO = new GraduateStudent();
```

```
Food myFood = new Food();
myFood = new Meat();
// ...
myFood = new Apple();
```

References

- This set of slides is only for educational purpose.
- Part of this slide set is referenced, extracted, and/or modified from the followings:
  - Deitel, P. and Deitel H. (2017) “Java How To Program, Early Objects”, 11ed, Pearson.
  - Liang, Y.D. (2017) “Introduction to Java Programming and Data Structures”, Comprehensive Version, 11ed, Prentice Hall.
  - Wu, C.T. (2010) “An Introduction to Object-Oriented Programming with Java”, 5ed, McGraw Hill.
  - Oracle Corporation, “Java Language and Virtual Machine Specifications” https://docs.oracle.com/javase/specs/
  - Oracle Corporation, “The Java Tutorials” https://docs.oracle.com/javase/tutorial/
  - Wikipedia, Website: https://en.wikipedia.org/