Chapter 13

Polymorphism and Interfaces

Lecture slides for:

Java Actually: A Comprehensive Primer in Programming

Khalid Azim Mughal, Torill Hamre, Rolf W. Rasmussen

Cengage Learning, 2008.

ISBN: 978-1-844480-933-2

`http://www.ii.uib.no/~khalid/jac/`

Modified: 16/2/19

Topics

Programming using inheritance:

- One superclass with several subclasses
- Polymorphism and dynamic method lookup
- Using polymorphic references
- Consequences of polymorphism

Interfaces in Java

- Abstract methods
- Interface types
- Multiple inheritance for interfaces
- Interfaces and polymorphic references
- Super- and subinterfaces

Abstract classes:

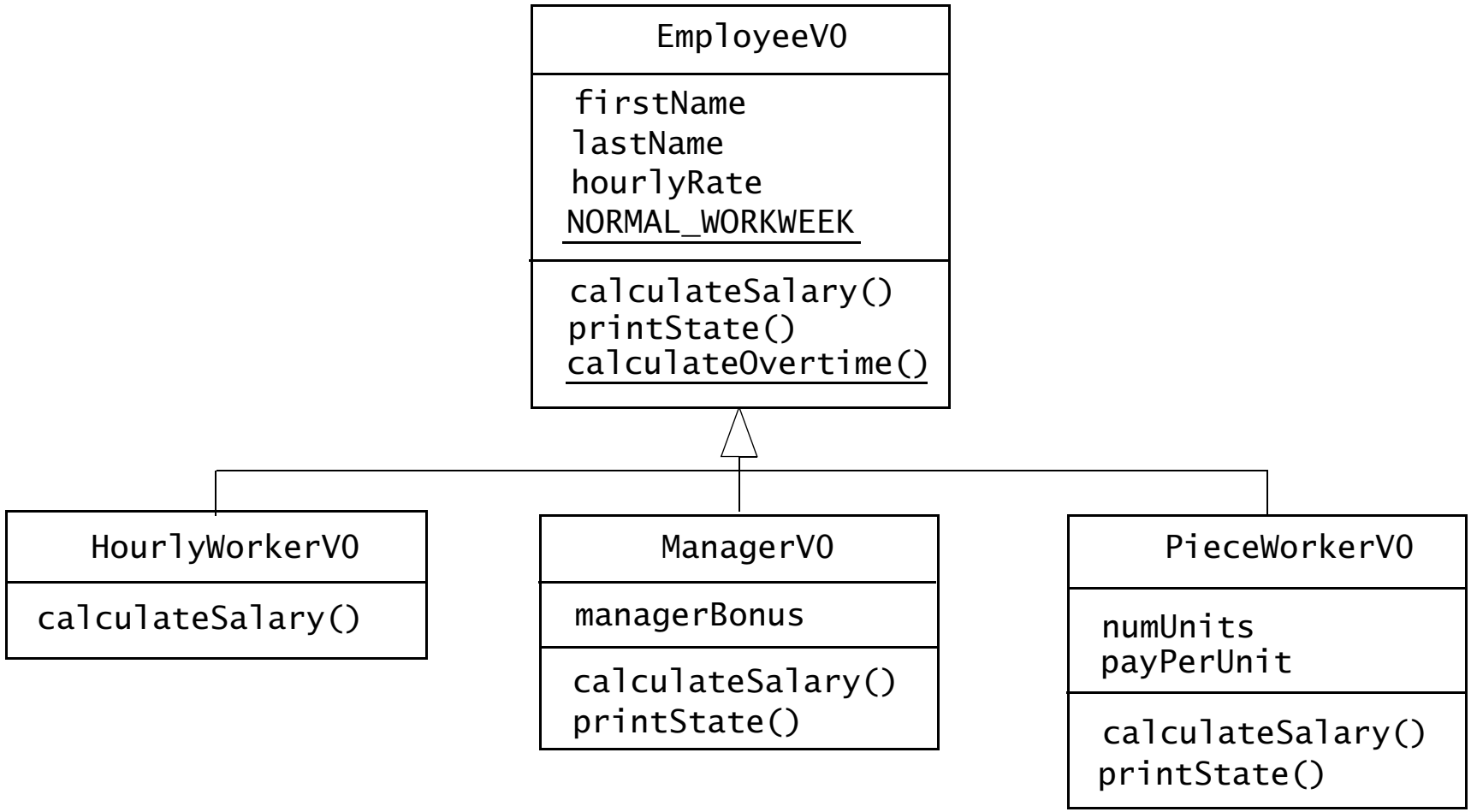
- Instantiation and inheritance
- Method overriding
- Concrete classes

Inheritance versus aggregation

Substitution principle - Liskov

Programming using inheritance

- A superclass often has multiple subclasses, which can declare new fields and hide inherited fields, and override instance methods to implement the abstraction of the subclass.



Polymorphic references

- Which object a reference denotes during program execution cannot always be determined *during compilation*.

```
...
EmployeeV0 employee;
Scanner keyboard = new Scanner(System.in);
System.out.println("Choose a category " +
                   "(1=employee, 2=hourly worker, 3=manager, 4=piece worker):");
int selection = keyboard.nextInt();
if (selection == 1) employee = new EmployeeV0("John", "Doe", 30.0);
else if (selection == 2) employee = new HourlyWorkerV0("Mary", "Smith", 35.0);
else if (selection == 3) employee = new ManagerV0("John D.", "Boss", 60.0, 105.0);
else if (selection == 4) employee = new PieceWorkerV0("Ken", "Jones", 12.5, 75);
else employee = new EmployeeV0("Ken", "Harris", 25.50);
// Which object does the employee reference denote after the if statement?
...
```

- The compiler cannot determine which object the reference `employee` will denote after the `if` statement above.
- A *polymorphic reference* is a reference that can refer to objects of different classes (types) at different times.
  - A *reference to a superclass* is polymorphic since it can refer to objects of all subclasses of the superclass during program execution.

Dynamic method lookup

- When a method is called for an object, it is the *class of the object (i.e. object type)* and the *method signature* that determines which *method definition* is executed.
  - *The type of the reference (i.e. declared type)* that denotes the object is *not* relevant.

```
...
// Which object does employee denote at the end of the if statement?
System.out.println("Weekly salary="
    + employee.calculateSalary()); // Which calculateSalary() method is executed?

employee.printState();       // and which printState() method is executed here?
...
```

- *Dynamic method lookup* is the process that determines which *method definition* a method signature denotes during execution, based on the class of the object.
  - This process *can lead to searching the inheritance hierarchy upwards to find the method definition*.
  - For the first call above, the `calculateSalary()` method in the class that the object `employee` refers to will be called. For the second call, the `printState()` method in the `EmployeeV0` class will be executed, if the object referred to `employee` belongs to the `EmployeeV0` or `HourlyWorkerV0` class. Otherwise, the `printState()` method in one of the two other subclasses will be executed, dependent on whether `employee` denotes a manager or a piece worker.

Using polymorphic references

```
class EmployeeArray { // From Program 13.4
  static EmployeeV0[] empArray = {                          // (1)
      new EmployeeV0("John", "Doe", 30.0),
      new HourlyWorkerV0("Mary","Smith", 35.0),
      new PieceWorkerV0("Ken", "Jones",  12.5, 75),
      new ManagerV0("John D.", "Boss", 60.0, 105.0),
  };
  static double[] empHours = { 37.5, 25.0, 30.0, 45.0 };    // (2)

  public static void main(String[] args) {
    // Traverses the employee array and prints selected properties for each
    // employee in the array.
    for (int i=0; i< empArray.length; i++){
      System.out.printf("Employee no. %d: %s %s" +
                        " has a weekly salary of %.2f GBP%n",
          i+1, empArray[i].firstName,                       // (3)
          empArray[i].lastName,                             // (4)
          empArray[i].calculateSalary(empHours[i]));        // (5)
    }
  }
}
```

- Program output:

```
Employee no. 1: John Doe has a weekly salary of 1125,00 GBP
Employee no. 2: Mary Smith has a weekly salary of 1312,50 GBP
Employee no. 3: Ken Jones has a weekly salary of 937,50 GBP
Employee no. 4: John D. Boss has a weekly salary of 3150,00 GBP
```

Consequences of polymorphism

- Polymorphism allows us to denote different types of objects with a *common (polymorphic) reference*, as long as these objects belongs to one of the subclasses or the superclass in an *inheritance hierarchy*.
- Dynamic method lookup causes the same call in the source code to result in *different method implementations being executed during program execution*.
  - *The actual type* of the object decides which method is called.
- Clients can treat superclass and subclass objects in the same manner, and there is no need for changing the client even if new subclasses are added.
  - This makes development and maintenance of clients easier.

Interfaces in Java

- An *interface* in Java defines a set of services - but *without providing an implementation*.
- The interface contains a set of *abstract methods.*
- A *abstract method* is comprised of the method name, parameter list and return type — but no implementation.
- Example:

```
interface ISportsClubMember {
    double calculateFee();   // abstract method
}
```

- Classes that want to *implement* an interface have to specify this using the keyword `implements`, for instance:

```
class Student implements ISportsClubMember {
    double calculateFee() { return 100.0; }
    // ... other methods and fields
}
```

  – *and* provide an implementation of the abstract methods in the interface.

Interfaces in Java (cont.)

- An interface is *not* a class - so we cannot create objects of an interface.
- However, an interface defines a type - an *interface type* - and we can declare references of this type, e.g.

```
ISportsClubMember student;
```

- A reference of an interface type can denote objects of *all* classes that implements the interface:

```
class EmployeeV1 implements ISportsClubMember {
    ...
    double calculateFee() { ... }
}
...
ISportsClubMember member;
member = new EmployeeV1("Al", "Hansen", 325.0);
...
member = new Student("Peter", "Jablonski", 35);
```

- With an interface type reference we can therefore get access to all services that the interface defines.

Interfaces in Java (cont.)

- An interface can also be used to import *shared constants*.

```
class ImportantClass implements ImportantConstants {
  public int reply() {
    if (isDone())
      return TERMINATE; // Interface name is not necessary due to implements.
    else
      return CONTINUE;
  }
  boolean isDone() { ... }
}
```

- If the class `ImportantClass` had not declared that it implemented the interface `ImportantConstants`, it would have to use the dot notation, e.g. `ImportantConstants.TERMINATE`, to access the constants.

Interfaces in Java (cont.)

- A class can implement multiple Java interfaces. This is called *multiple inheritance for interfaces*.

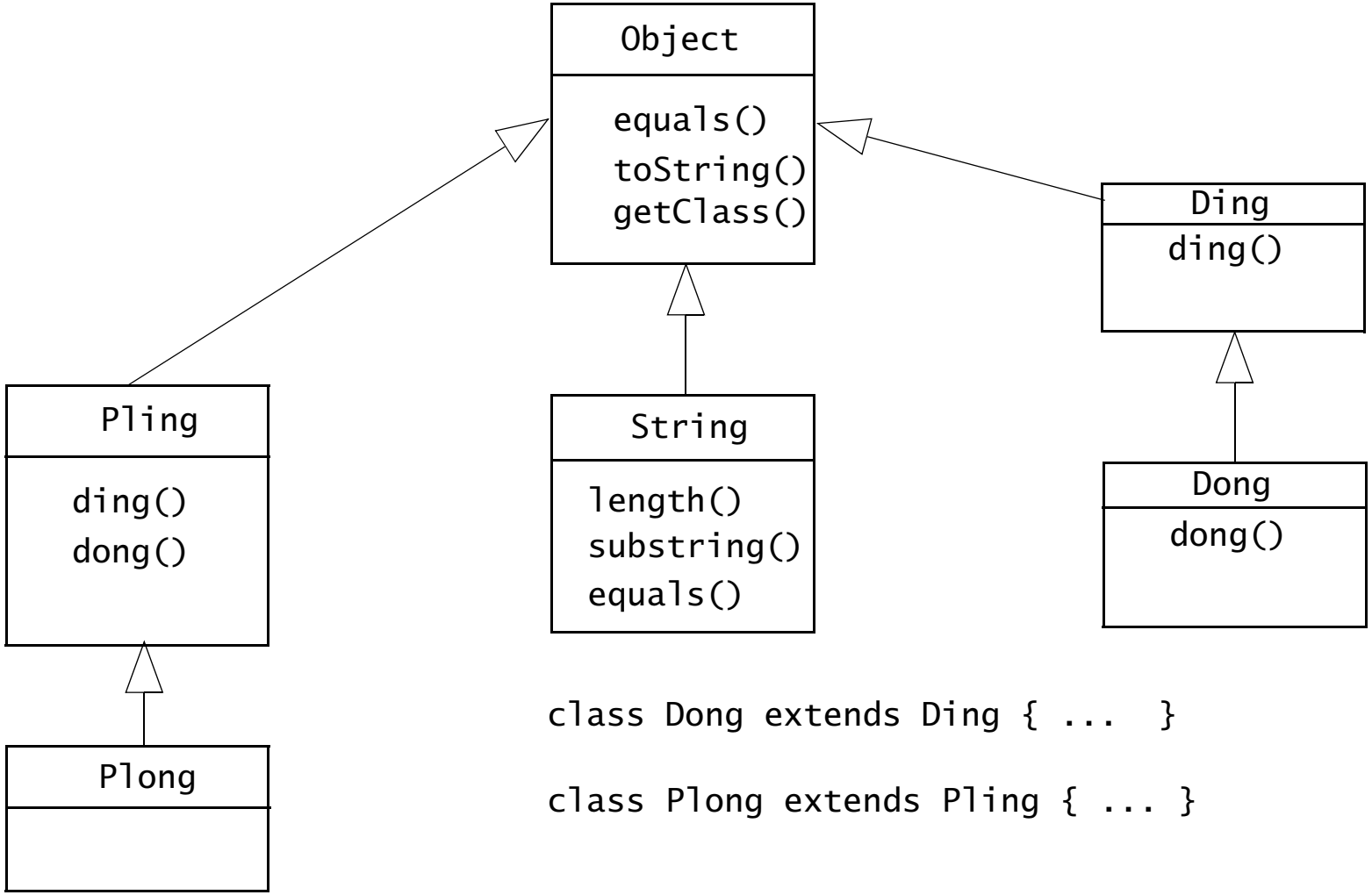
```
class ManagerV1 extends EmloyeeV1
                    implements ISportsClubMember, IRepresentative {
    ...
}
```

  – The class has to implement all abstract methods from every interface.

- A interface `B` can extend another interface `A`. Then the interface `A` is called a *superinterface* and `B` a *subinterface*.
- A class that implements `B` must implement the methods from *both* `A` and `B`.

```
interface ITick { void tick(); }
interface ITickTock extends ITick { void tock(); }

class Watch implements ITickTock {
    void tick() { ... } // from ITick (superinterface)
    void tock() { ... } // from ITickTock (subinterface)
}
```

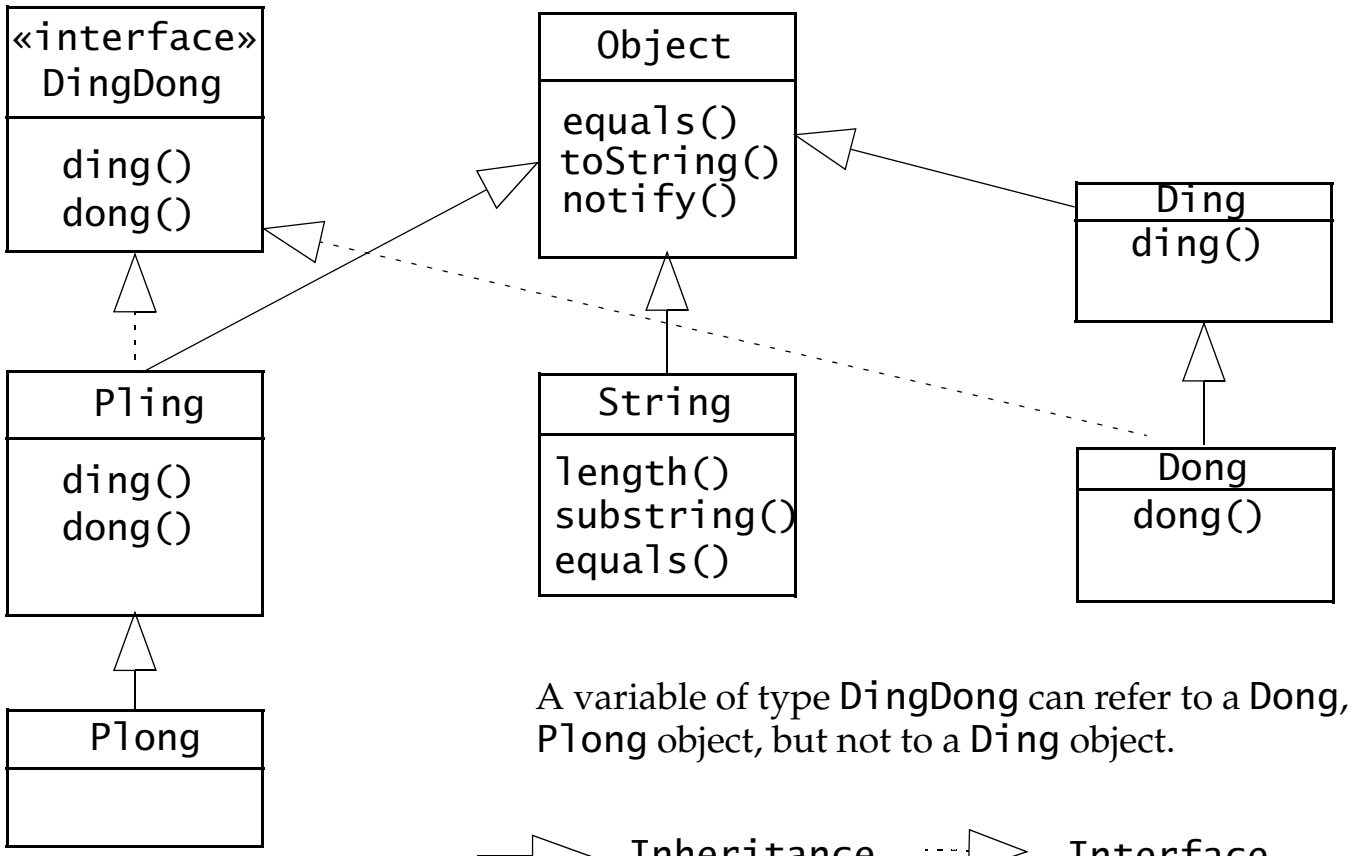
Inheritance hierarchy



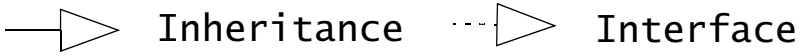

```
class Dong extends Ding { ...  }

class Plong extends Pling { ... }
```

Using interfaces to handle objects of multiple types

- Create a *reference type* that can denote things with both `ding()` and `dong()` methods.



A variable of type `DingDong` can refer to a `Dong`, a `Pling` and a `Plong` object, but not to a `Ding` object.



Using interfaces to handle objects of multiple types (cont.)

```
interface DingDong {
    public void ding();
    public void dong();
}

class Ding {
    public void ding() { System.out.println("ding from Ding"); }
}

class Dong extends Ding implements DingDong {
    public void dong() { System.out.println("dong from Dong"); }
}

class Pling implements DingDong{
    public void dong() { System.out.println("dong from Pling"); }
    public void ding() { System.out.println("ding from Pling"); }
}

class Plong extends Pling {
    public void plong() { System.out.println("plong fra Plong"); }
}
```

```
public class TestInterface {
    public static void main(String[] args) {
        DingDong dingdongs[] = new DingDong[3];   // Create array of references
        dingdongs[0] = new Dong();                // Initialising references
        dingdongs[1] = new Pling();
        dingdongs[2] = new Plong();
        for (int i = 0; i < dingdongs.length; i++)
            dingdongs[i].ding(); // Polymorphic references + dynamic method lookup
    }
}
```

- Program output:

```
ding from Ding
ding from Pling
ding from Pling
```

Summing up the use of interfaces in Java

- A class must explicitly declare that it implements an interface (`implements` keyword).
- The class (or its subclasses) must implement methods for all abstract methods that are specified in the interface.
- If a class implements an interface, all of its subclasses also implement the interface automatically as a result of inheritance.
  - Inheritance also applies for interfaces: super- and subinterfaces.
- Interfaces define a new reference type, and references of an interface type are polymorphic.
- A class can implement *multiple interfaces*.
- A class that implements a subinterface must also implement all methods from its superinterface.
- Two variants of inheritance:
  - *Design inheritance*: Multiple interface inheritance where objects can belong to multiple types (using interfaces in Java).
  - *Implementation inheritance*: Simple implementation inheritance where method and variable lookup is simpler (using subclasses in Java).

Almost all advantages of general multiple inheritance without all implementation difficulties.

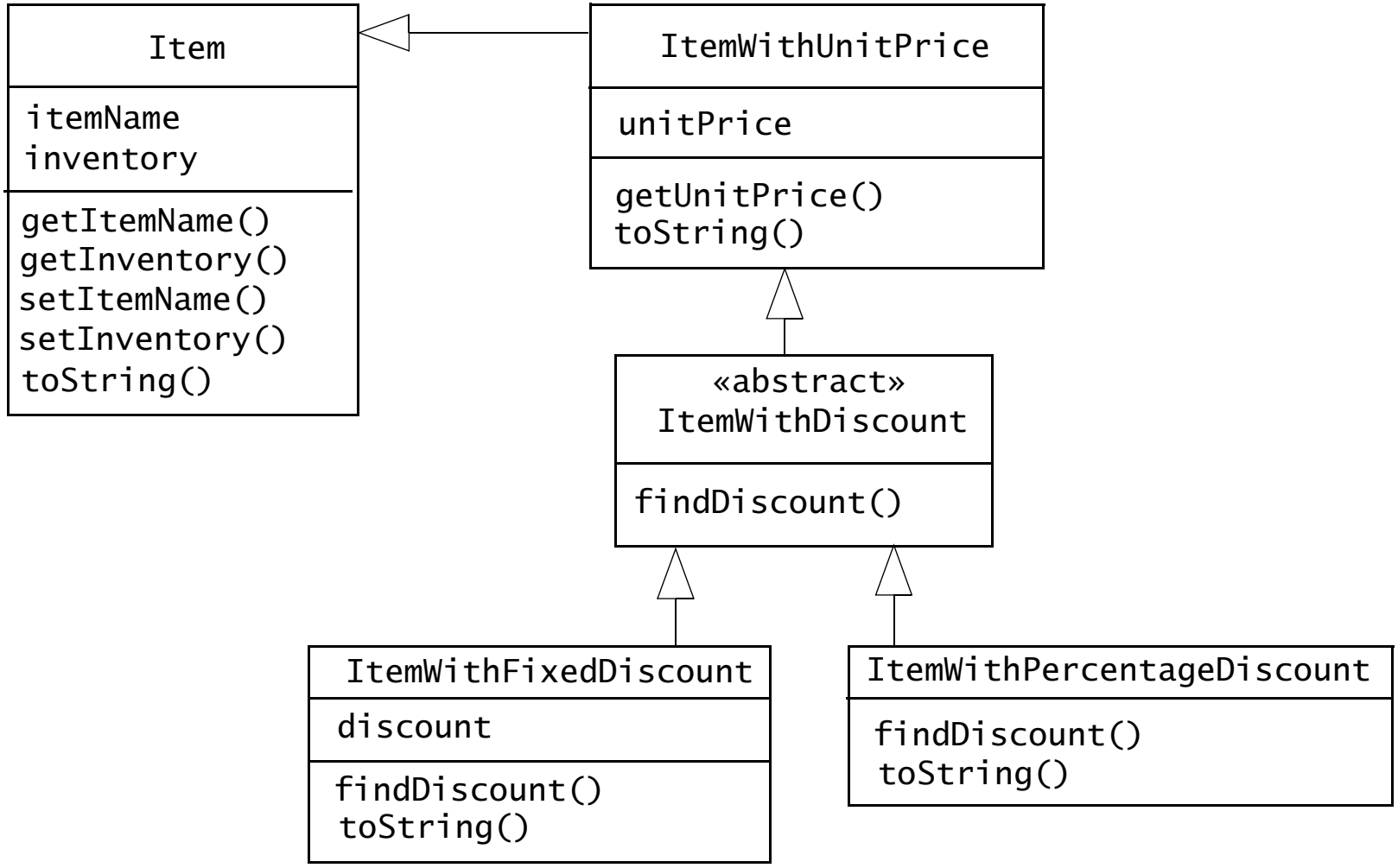
Abstract classes

- An *abstract class* is a class that cannot be instantiated.
- Abstract classes can be used as a superclass, which through inheritance enforces all subclasses to fulfill a common contract.
  - An abstract class *can* implement parts of (or the whole) contract defined by a Java interface.
- A subclass of an abstract class must implement the abstract methods of the superclass, otherwise it should be declared `abstract`.
- Clients can use *references of the abstract class* to denote subclass objects, and by means of polymorphism the correct instance methods will be executed.

- Example: We want to offer discounted items. The class `ItemWithDiscount` is an *abstract superclass* for all discounted items. The classes `Item` and `ItemWithUnitPrice` are reused.

- Example: Use of the *Template Method* design pattern - an abstract class implements a common framework for behaviour; subclasses must implement the parts that are specific for them.

Abstract classes (cont.)



Abstract classes (cony.)

```
public class Item {    // ... as before
}
public class ItemWithUnitPrice extends Item {    // ... as before
}

public abstract class ItemWithDiscount extends ItemWithUnitPrice {
  ItemWithDiscount(String itemName, int inventory, double price) {
    super(itemName, inventory, price); // calls the superclass constructor
  }
  abstract double findDiscount(double priceWithoutDiscount, int numOrdered);
}

public class ItemWithFixedDiscount extends ItemWithDiscount {
  private double discount;      // fixed discount in GBP
  ItemWithFixedDiscount(String itemName, int inventory, double price, double discount) {
    super(itemName, inventory, price); // calling the superclass constructor
    assert discount >= 0.0:
      "Item discount must be greater than or equal to 0.0 GBP.";
    this.discount = discount;
  }
  public double findDiscount(double priceWithoutDiscount, int numOrdered) {
    return discount;
  }
```

```
  public String toString() { // overrides toString() from the ItemWithUnitPrice class
    return super.toString() + "\tDiscount: " + discount;
  }
}

public class ItemWithFixedDiscount extends ItemWithDiscount {
  private double discount;      // fixed discount in GBP
  ItemWithFixedDiscount(String itemName, int inventory, double price, double discount) {
    super(itemName, inventory, price); // calling the superclass constructor
    assert discount >= 0.0:
      "Item discount must be greater than or equal to 0.0 GBP.";
    this.discount = discount;
  }
  public double findDiscount(double priceWithoutDiscount, int numOrdered) {
    return discount;
  }
  public String toString() { // overrides toString() from the ItemWithUnitPrice class
    return super.toString() + "\tDiscount: " + discount;
  }
}
```

```
public class ItemWithPercentageDiscount extends ItemWithDiscount {
  int discountInterval;  // number of items that must be sold to get a discount
  double percentage;     // percentage the discount is increased by for each interval
  double maxPercentage;  // maximum percentage for discount
  ItemWithPercentageDiscount(String itemName, int inventory, double price,
      int discountInterval, double prosentSats, double maksimalSats) {
    super(itemName, inventory, price); // kaller superklassens konstruktør
    assert discountInterval > 0 : "Discount interval must be > 0";
    assert percentage >= 0.0 : "Discount msut be >= 0% per interval";
    assert maxPercentage >= 0.0 : "Maximum discount must be >= 0%";
    this.discountInterval = discountInterval;
    this.percentage = percentage;
    this.maxPercentage = maxPercentage;
  }
  public double findDiscount(double priceWithoutDiscount, int numOrdered) {
    int numIntervals = numOrdered / discountInterval;
    double discount = numIntervals * percentage;
    if (discount > maxPercentage) discount = maxPercentage;
    assert discount >= 0.0 : "Given discount must be >= 0%";
    return discount/100.0*priceWithoutDiscount;
  }
  public String toString() { // overrides toString() from the ItemwithUnitPrice class
    return super.toString() + "\tDiscount interval: " + discountInterval
    + "\tPercentage: " + percentage + "\tMaximum percentage: " + maxPercentage;
  }}
```

```
public class Sales {
  public static void main(String[] args) {
    ItemWithUnitPrice[] stock = {
        new ItemWithUnitPrice("Coca cola 0.5l", 150, 3.50),
        new ItemWithFixedDiscount("Pepsi Max 0.5l", 100, 3.50, 0.50),
        new ItemWithUnitPrice("Pepsi Light 0.5l", 50, 3.00),
        new ItemWithFixedDiscount("Sparkling water 0.5l", 50, 2.00, 0.75),
        new ItemWithPercentageDiscount("Coca cola 0.33l", 500, 1.50,
            10, 0.5, 10.0) };
    int[] order = { 50, 50, 10, 20, 250 };
    double total = 0;
    double discount  = 0;
    double itemPrice, itemDiscount;
    for (int i=0; i<stock.length; i++) {
      itemPrice = stock[i].getUnitPrice();
      itemDiscount = 0;
      if (stock[i] instanceof ItemWithDiscount) {
        ItemWithDiscount item = (ItemWithDiscount) stock[i];
        itemDiscount = item.findDiscount(itemPrice, order[i]);
        itemPrice -= itemDiscount;
        itemDiscount *= order[i]; // for the whole order
      }
      itemPrice *= order[i]; // for the whole order
      System.out.println("Price for " + order[i] + " pieces of item '"
          + stock[i].getItemName() +"' is " + itemPrice + " GBP.");
```

```
      total += itemPrice;
      discount += itemDiscount;
    }
    System.out.println("Total amount: " + total+ " GBP.");
    System.out.println("Discount of: " + discount + " GBP has been subtracted.");
  }
}
```

- Program output:

```
Price for 50 pieces of item 'Coca cola 0.5l' is 175.0 GBP.
Price for 50 pieces of item 'Pepsi Max 0.5l' is 150.0 GBP.
Price for 10 pieces of item 'Pepsi Light 0.5l' is 30.0 GBP.
Price for 20 pieces of item 'Sparkling water 0.5l' is 25.0 GBP.
Price for 250 pieces of item 'Coca cola 0.33l' is 375.0 GBP.
Total amount: 755.0 GBP.
Discount of: 40.0 GBP has been subtracted.
```

Inheritance versus aggregation

- Inheritance offers reuse of classes by *allowing new classes access to fields and methods from the superclass* (and all other superclasses further up in the inheritance hierarchy).
- Aggregation offers reuse of classes by *allowing new classes to declare composite objects* where objects of existing classes are included as *parts* (a.k.a. *constituent objects*).

- Use *inheritance* when:
  - the new class is a *logical specialisation* of an existing class.
  - the existing class can be extended (i.e. is not a `final` class).
- Use *aggregation* when:
  - the new class represents composite objects, and classes have already been declared for the part-objects.
  - an existing class cannot be extended (i.e. is a `final` class).

Note that aggregation if offered by default in Java, while for inheritance we need to use the keyword `extends`.

Substitution principle - Liskov

- A reference type can be:
  - a class name
  - an interface name
  - an array type
- Subtypes and supertypes:
  - In the inheritance hierarchy, a reference type can have one or more *supertypes*. The reference type is a *subtype* of each of its supertypes.
- Substitution principle:

  The behaviour of subtypes is in accordance with the specification of their supertypes, meaning that a subtype object can be substituted for a supertype object.