Chapter 12

Inheritance

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/`

Permission is hereby granted to use these lecture slides in conjunction with the book.

Modified: 16/2/19

Topics

Key concepts:
- Superclass and subclass
- Generalisation - specialisation
- Inheritance of members
- Inheritance hierarchy

Extending properties and behaviour
- Initialising with `super`() and `this()`
- Client's use of the subclass

Using inherited members
- In the subclass and in clients
- Instance and static members

Superclass references and subclass objects
- Difference between super- and subclass references

Overriding instance methods
- Using an overridden method
- `super` keyword
- Accessing a shadowed field

Final classes and members:
- `final` keyword
- Final classes
- Final methods and variables

Converting references
- Implicit (automatic)
- Conversion operator (*cast*)
- `instanceof` operator
- Superclass - subclass relation (*is-a* relation): `sub` *is-a* `sup`

Example: Soda machine

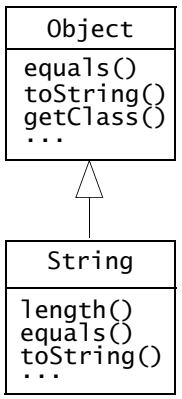
Inheritance: motivation and key concepts

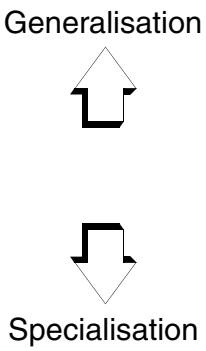
- A useful mechanism for *reuse of code*, by deriving new classes from an existing class.
- The *derived/extended* (sub)class inherits members from the *base* (super)class.
  - All members that are *accessible* from outside the superclass are inherited.
  - Inherited members can be used as if they were defined in the subclass itself.
  - Here we will only declare accessible members (that are inherited).
- Inheritance is used for *extending* a class (*specialisation*):
  - We can add new methods, i.e. new *behaviour*.
  - We can add new (instance)variables, i.e. new *properties*.
  - We can also change behaviour from the superclass by *overriding* a *superclass method*.
- A subclass has one super class, also called *parent class*.
- A subclass can also be parent of other classes, which in turn can be parent of yet other classes. I.e. the inheritance relation is *transitive*.
  - Thus classes can be arranged in an *inheritance hierarchy*, with two or more levels.
- The `Object` class is (implicitly) superclass to all other classes, and thus the root of the inheritance hierarchy.
  - All classes, and thus all objects, inherits common behaviour from the `Object` class.

Inheritance: key concepts

Superclass

Superclass/base class: most general class





Subclass

Subclass/derived class: most specialised class
- inherits properties and behaviour of the superclass
- can define new properties
- can change and/or extend behaviour

Generalisation lead to more abstract classes.

Specialisation leads to more customised classes.

Inheritance is used to implement superclass-subclass ("is-a") class relationships.

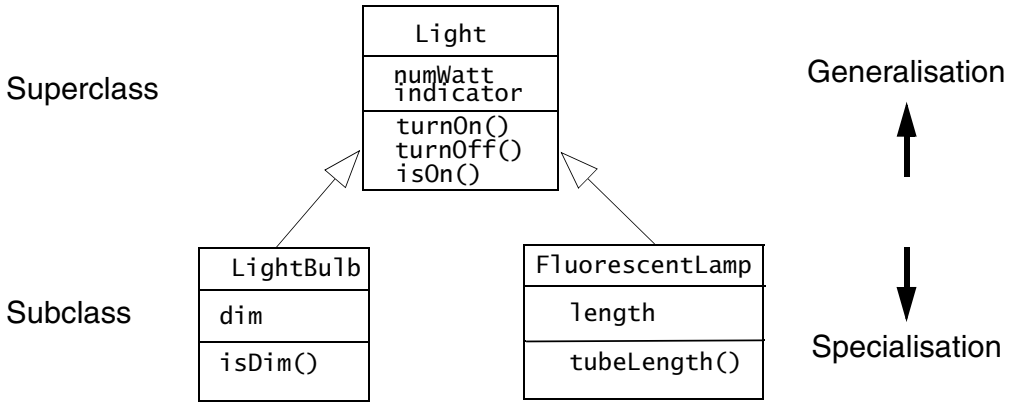
A first example using inheritance

```
public class OOPClient {
  public static void main(String[] args) {
    String str = new String("aha");

    System.out.println(str.getClass());      // inherited from the Object class

    System.out.println(str.length());        // defined in the String class

    System.out.println(str.equals("hihaw")); // redefined (overridden) in String class
  }
}
```

- *Inheritance of members from the superclass*: The subclass `String` inherits the method `getClass()` from the superclass `Object` (but this makes no difference from the `OOPClient` class that can use the `getClass()` method as if it was defined in the `String` class).
- *Extending the subclass*: The subclass `String` has a new method `length()` (that the superclass `Object` does not have).
- *Overriding methods*: The subclass `String` redefines the `equals()` method (more on this later).

Extending properties and behaviour

- The class `Light` is the superclass for more specialised subclasses.
- *Accessible* members are inherited in the subclass.
- Exception: Constructors can *not* be inherited.



Superclass: `Light`

```
public class Light {
  int numWatt;       // wattage (power)
  boolean indicator; // off == false, on == true
  String location;   // where the light is located

  Light(int numWatt, boolean indicator, String location) {
    this.numWatt = numWatt;
    this.indicator  = indicator;
    this.location   = location;
  }
  public void turnOn() { indicator = true; }
  public void turnOff() { indicator = false; }
  public boolean isOn() { return indicator; }
  public String toString() {
    return "[Light: " + numWatt + " Watt, " + "indicator: "
           + indicator + ", location: " + location + "]";
  }
}
```


Extending properties and behaviour (cont.)

- The subclass `LightBulb` extends the superclass `Light`, and is more specialised.

```
public class LightBulb extends Light {
  boolean matt;      // matt if true, clear if false
  LightBulb(int numWatt, boolean indicator, String location, boolean matt) {
    super(numWatt, indicator, location);
    this.matt = matt;
  }
  public String toString() {
    return "[LightBulb: " + numWatt + " Watt, " + "indicator: "
           + indicator + ", location: " + location + " matt: " + matt + "]";
  }
  public boolean isMatt() { return matt; }
}
```

- Each lightbulb *is-a* light.
- The keyword `extends` marks the superclass-subclass relation.
- The subclass constructor uses `super()` to initialise inherited fields.
- The `toString()` method changes the behaviour by overriding the superclass method.
- A new method in the subclass, `isMatt()`, extends the behaviour.

Superclass constructors

- Subclass constructors call the superclass constructor which can be used to initialise inherited instance variables.
- If a subclass constructor does *not* call a superclass constructor explicitly, the *implicit standard superclass constructor*, `super()`, is called automatically.

```
class LightBulb extends Light {
    //...
    LightBulb() { }  // No explicit call to super().
    //...
}
```

The standard constructor in the `LightBulb` class above is equivalent with

```
LightBulb(){ super(); }
```

- If a superclass has *only non-standard* constructors specified, the compiler will give an error message if the *standard superclass constructor* is called from the subclass.
- The call to the superclass constructor is the first statement in the subclass constructor.
- Tip: ALWAYS CALL THE SUPERCLASS CONSTRUCTOR EXPLICITLY!

Using `this()` and `super()` in constructors

- Constructors can be simplified: `this()` calls other constructors in the same class.

```
class Light {
    int numWatt;         // wattage (power)
    boolean indicator;   // off == false, on == true
    String location;     // where the light is located

    Light() {                      // 1. (explicit) standard constructor
        this(0,false,"X");         // 3. non-standard constructor is executed
    }
    Light(int watt, boolean ind) { // 2. non-standard constructor
        this(watt,ind,"X");        // 3. non-standard constructor is executed
    }
    Light(int watt, boolean ind, String loc) { // 3. non-standard constructor
        super(); // Calls the standard superclass constructor.
        numWatt = watt;
        indicator = ind;
        location = new String(loc);
    }
    // ...
}
```

Client's use of subclass objects

- Clients need not bother whether a member is declared in the subclass or inherited from its superclass.

```
public class Lighting {

  public static void main(String[] args) {
    // Create a light bulb.
    LightBulb bulb = new LightBulb(40, false, "living room", true);

    bulb.turnOn(); // inherited method
    System.out.println("The light bulb in the living room is " +
                       bulb.numWatt + " Watt"); //inherited field

    if (bulb.isMatt())                        // subclass method
      System.out.println("The lightbulb is matt.");
    else
      System.out.println("The lightbulb is clear.");

    if (bulb.isOn())                          // inherited method
      System.out.println("The living room light is on.");
    else
      System.out.println("The living room light is off.");
```



```
    System.out.println(bulb);

    // Create an ordinary light.
    Light outLight = new Light(75, true, "outdoor");

    outLight.turnOff();
    System.out.println("The outdoor light is " + outLight.numWatt + " Watts");

    if (outLight.isOn())
      System.out.println("The outdoor light is on.");
    else
      System.out.println("The outdoor light is off.");

    System.out.println(outLight);

  }
}
```

- Output from running the program:

```
The light bulb in the living room is 40 Watt
The lightbulb is matt.
The living room light is on.
[LightBulb: 40 Watt, indicator: true, location: living room matt: true]
The outdoor light is 75 Watts
The outdoor light is off.
[Light: 75 Watt, indicator: false, location: outdoor]
```


Using inherited members

```
public class Rectangle {
  double width;
  double length;

  Rectangle(double width, double length) {
    this.width =  width;
    this.length = length;
  }
  public double calcArea() { return width*length; }
  public String toString() {
    return "[" + getClass() + ": width=" + width +
           ", length=" + length + "]";
  }
}
```

- The class `Rectangle` has only accessible members. These are inherited by a subclass of `Rectangle`, and can thereby be referred by their *simple* name in the subclass.
- The class `Rectangle` is also a subclass (of `Object`) and can call the inherited method `getClass()` directly by using the method name.
- Any class that doesn't explicitly state that it extends another class, is (implicitly) a *direct* subclass of the `Object` class.

Using inherited members inside a subclass

- A subclass can refer *directly* to all inherited members using the member's *simple name*:

```
public class Cube extends Rectangle {
  double height;
  Cube(double width, double length, double height) {
    super(width, length);            // calling superclass constructor
    this.height = height;
  }
  public double calcSurface() {
    return 2 * ((width*length)       // access by simple field name
           +(this.width*height)      // access by this-reference
           +(this.length*height));
  }
  public double calcVolume() {
    return calcArea() * height;      // access by simple method name
  }
  public String toString() {
    return "[" + getClass()          // access by simple method name
           + ": width=" + width + ", length=" + length
           + ", height=" + height + "]";
  }
}
```


Using inherited members in a client

```
public class VolumeCalculation {
  public static void main(String[] args) {
    Cube brick = new Cube(3, 4, 5);
    System.out.println("Area of base: " + brick.calcArea());
    System.out.println("Surface of all sides: " + brick.calcSurface());
    System.out.println("Volume of the brick: " + brick.calcVolume());
    System.out.println("Brick: " + brick);
  }
}
```

- Makes no difference for the client whether the member is inherited (like `calcArea()`) or declared in the subclass itself (e.g. `calcVolume()`).
- The client can also refer to the width and length of the brick using the reference `brick`. e.g. `brick.width`.
- Output from the program:

```
Area of base: 12.0
Surface of all sides: 94.0
Volume of the brick: 60.0
Brick: [class Cube: width=3.0, length=4.0, height=5.0]
```

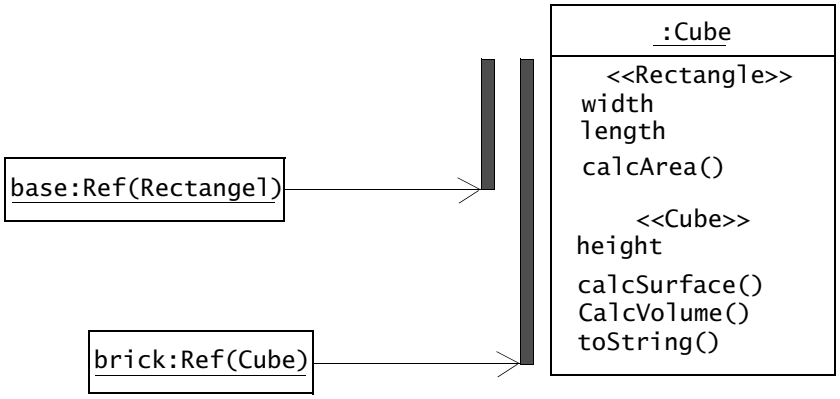
Using inherited members (cont.)

- *Static members* can be accessed through the *class name* (superclass or subclass) or through a *reference of the subclass type*:

```
class EmployeeV0 { // From Program 12.1
    ...
class ManagerV0 extends EmployeeV0 { // From Program 12.2
   ...
class UseInheritance { // From Program 12.4
    public static void main(String[] args) {
        // Creating a manager object
        ManagerV0 manager = new ManagerV0("Jon D.", "Boss", 60.0);
        ...
        System.out.printf("Normal work week is %.1f hours%n",
                          manager.NORMAL_WORKWEEK);            // (1)
        if (numOfHoursWorked > ManagerV0.NORMAL_WORKWEEK)      // (2)
            System.out.printf("Overtime: %.1f hours%n",
                       (numOfHoursWorked-EmployeeV0.NORMAL_WORKWEEK));// (3)
        else
            System.out.println("No overtime.");
    }
}
```


Superclass references and subclass objects

- A superclass reference can refer to an object of a subclass -- but this reference can only be used to refer to *inherited* members.




```
Cube brick = new Cube(3, 4, 5);
Rectangle base = brick;  // aliases
System.out.println("Width: " + base.width);  // ok
base.calcArea();         // ok
base.calcSurface();      // Not allowed -the method is declared in the subclass
```

Overriding instance methods

- Overriding means redefining an *instance method* from the superclass.
  - The *method signatur*e must be *identical*.
  - The *return type* can be a a subtype of superclass method.
  - *Accessibilty* can be *widen*, but *not narrowed*.
  - *Cannot* throw *new* checked exceptions (see Section 10.4)
- The method declaration in the subclass *overrides* the method declaration in the superclass.
- Access from the subclass to the overridden method declaration in the superclass is through the `super` keyword, as shown in line marked (5) below.

```
class EmployeeV0 { // From Program 12.1
    double calculateSalary(double numHours) {
        ...
    }
    ...
}
class ManagerV2 extends EmployeeV0 { // From Program 12.9

    @Override            // Annotation to ensure that override is  correct.
    double calculateSalary(double numHours) {                       // (4)
        double fixedSalary = super.calculateSalary(numHours);       // (5)
        ...
    }
}
```


Hidden fields

- Declaring a field in the subclass with the same name as in the superclass will *hide* the field in the superclass.
  - Use the keyword `super` to access the hidden field.

```
public class LowEnergyBulb extends Light {
  int numWatt;                          // hides field from the superclass
  LowEnergyBulb(int numWatt, boolean indicator, String location) {
    super(0, indicator, location);
    this.numWatt = numWatt;             // subclass field
  }
  public String toString() {
    return "[Low-energy bulb: " + numWatt + " Watts, corrsponding to " + super.numWatt
           + " Watts, " + "indicator: " + indicator + ", location: " + location + "]";
  }
  public double convertWattage() {
    switch(numWatt) {                    // subclass field
      case  7: super.numWatt = 40; break; // superclass field
      case 11: super.numWatt = 60; break;
      case 15: super.numWatt = 75; break;
      default: super.numWatt =  0; break;
    }
    return super.numWatt;                // superclass field
  }
}
```

Final classes and members (`final`)

- Final *classes*
  - Classes that cannot be specialised, i.e. that cannot be extended.

```
public final class Finito { ... }

public final class String { ... } // in the java.lang package

public final class UnitCube extends Cube {
    UnitCube() { super(1, 1, 1); }    // using the superclass constructor
    public double calcSurface() { return 6; } // 2*((1*1)+(1*1)+(1*1))
    public double calcVolume()  { return 1; } // 1*1*1
}
```

  - Note that it suffices to designate the class `final`.
  - None of the methods needs to be final.

Final classes and members (`final`) (cont.)

- Final *methods*
  - Instance methods that cannot be overridden or static methods that cannot be hidden (but that can be overloaded).
  - *The compiler can generate more efficient code for final methods.*

```
class Pension { public final double calculatePension(){...} ... }
class Fee { private static final double collectFee(){...}... }
class Greenhouse { public final void startAlarm(){...} ...}
```

- Final member variables or *class constants*
  - Such member variables cannot be modified after initialisation.

```
class Pension { private final int RETIREMENT_AGE = 67; ... }

class Fee  { public static final double VAT = 23.5; ... }
```

Reference Conversion

- Conversion of references from subclass to superclass is done *implicitly*, for instance.

```
Light bulb = new LowEnergyBulb(7, true, "hall"); // when LowEnergyBulb extends Light
```

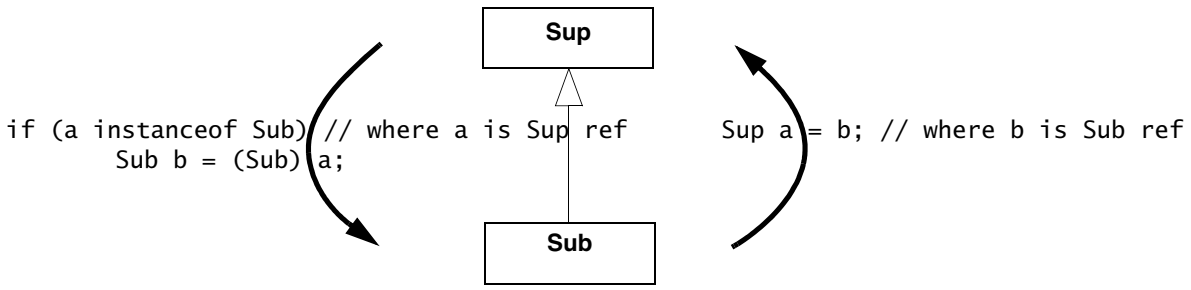
- Conversion the opposite way, i.e. from superclass to subclass, must be done *explicitly* using *the cast operator* `(type)`, e.g.

```
LowEnergyBulb newBulb = (LowEnergyBulb) bulb; // when bulb is of reference type Light
```

- Such conversions may however cause problems.
  - how can the program know that it is referring to a subclass object of the correct type that is referred to by the superclass reference?
- The solution is to use the operator `instanceof` to check that the object being referred to has the correct type:

```
if (bulb instanceof LowEnergyBulb) {
    LowEnergyBulb newBulb = (LowEnergyBulb) bulb;
    System.out.println("A low-energy bulb of " + numWatt + " Watts corresponds to a "
        + " common lightbulb of " + convertWattage() + " Watts.");
}
```


Sub *is-a* Sup



A subclass object can behave like a superclass object, and can therefore be used as if it was a superclass object. Consequently, the compiler allows *upcasting*:

```
Sup a = b; // where b is Sub ref
```

However, there is no guarantee that a superclass reference denotes a subclass object of the correct type. As a result, we must always check using `instanceof` that there indeed exists a subclass object of the correct type before performing *downcasting*:

```
if (a instanceof Sub) // where a is Sup ref
    Sub b = (Sub) a;
```

Example: Soda Machine

- Super- and subclass relation
  - `VendingMachine` - a simple machine that can dispense different items with a common price
  - `SodaMachine` - a more advanced machine that handles individual prices for different items
- Calling `super()`
- Overriding methods
- Hidding fields
- Extending behaviour
- Running the program: `java -ea`
  - We use `assert` statements to verify program behaviour.
- Clients use both superclass and subclass objects.
  - Clients can use all accessible methods.
  - If a subclass overrides a method in the superclass, the clients that use subclass objects can only call the *overridden* method in the subclass.

Example: Soda Machine -- Class `Item` (cont.)

```
public class Item {
  String itemName;   // name of the item that can be purchased
  int inventory;     // number of items in stock
  Item(String itemName, int inventory) {
    assert inventory >= 0 :
      "The inventory must be more than 0 items.";
    this.itemName = itemName;
    this.inventory = inventory;
  }
  public String getItemName() { return itemName; }
  public int getInventory() { return inventory; }
  public void setInventory(int inventory) {
    assert inventory >= 0 :
      "The inventory must be more than 0 items.";
    this.inventory = inventory;
  }
  public String toString() { // overrides toString() from the Object class
    return "\nItem name: " + itemName +
    "\tInventory: " + inventory;
  }
}
```

Example: Soda Machine -- Class `VendingMachine` (cont.)

```
public class VendingMachine {
  Item[] items;          // the items that can be bought
  double price;          // common price for all items
  double receivedAmount; // the amount of money entered into the machine
  double change;         // the amount of money the machine has left to give change
  double till;           // money that cannot be given back as change
  private Display display;  // display window where information is shown

  VendingMachine(Item[] items, double price, double change) {
    this.items = items;
    this.price = price;
    this.change = change;
    receivedAmount = 0.0;
    till = 0.0;
    display = new Display();
  }

  public void showMessage(String message) {
    display.showMessage(message);
  }

  public double findPrice(int itemSelected) {
    return price;
  }
```



```
  public void acceptCash(double amount) {
    receivedAmount += amount;
    showMessage("Accepted: " + formatMoney(receivedAmount) + " GBP");
  }

  public boolean itemAvailable(int selectedItem) {
    assert selectedItem >=0 && selectedItem < items.length :
      "Item number must be between 0 and " + (items.length-1);
    return (items[selectedItem].getInventory() > 0);
  }

  private void deliverItem(int selectedItem) {
    // here the vending machine will place chosen item in the receiver door
    showMessage("The item " + items[selectedItem].getItemName() + " is dispensed.");
  }

  private void giveChange(double itemChange) {
    // here the machine will give change.
    showMessage("Change " + formatMoney(itemChange) + " given back.");
    change -= itemChange;
  }


  private void updateTill(double amount) { till += amount; }
```

```
public void selectItem(int selectedItem) {
  assert selectedItem >=0 && selectedItem < items.length :
    "Item number must be between 0 and " + (items.length-1);
  if (itemAvailable(selectedItem)) {
    double price = findPrice(selectedItem);
    if (receivedAmount >= price) {
      deliverItem(selectedItem);
      items[selectedItem].setInventory(items[selectedItem].getInventory() - 1);
      giveChange(receivedAmount-price);
      updateTill(receivedAmount);
      receivedAmount = 0.0;
      showMessage("Thank you for buying our product - Welcome back!");
    }
    else { // received amount is less than its price
      showMessage("Not enough money inserted. " +
          items[selectedItem].getItemName() + " costs " + formatMoney(price) + " GBP");
    }
  }
  else { // out of stock
    showMessage("Sorry, but we are out of " + items[selectedItem].getItemName()
                + ". Money is returned.");
  }
}
```



```
public void cancel() {
  giveChange(receivedAmount);
  showMessage("Item purchase is cancelled.");
  receivedAmount = 0.0;
}

public void showSelection() {
  String str = "The vending machine contains these items:";
  str += this;
  str += "\nEach item costs: " + formatMoney(price) + " GBP";
  str += "\nChange available: " + formatMoney(change) + " GBP";
  str += "\nTill contains: " + formatMoney(till) + " GBP";
  str += "\nInserted amount: " + formatMoney(receivedAmount) + " GBP";
  System.out.println(str);
}

public String toString() { // overrides toString() from the Object class
  String str = "";
  for (int i=0; i<items.length; i++) {
    str += "\nItem number: " + i;
    str += items[i];
  }
  return str;
}
```

```
  protected String formatMoney(double amount) {
    return String.format("%.2f", amount);
  }
}
```

Example: Soda Machine -- Class `Display` (cont.)

```
public class Display {
  // this class uses System.out to display messages, and therefore doesn't
  // need any field variables
  public void showMessage(String message) {
    System.out.println(message);
  }
}
```


Example I: Soda Machine (cont.)

- Client with simple test of the vending machine:

```
public class VendingMachineClient {
  public static void main(String[] args) {
    Item[] soda = {
        new Item("Coca cola", 10),
        new Item("Pepsi Max", 10),
        new Item("Pepsi Light", 2),
        new Item("Sparkling water", 5) };
    VendingMachine automat = new VendingMachine(soda, 2.50, 100.0);
    automat.showSelections();
    automat.acceptCash(6.0);
    automat.selectItem(1);
    automat.acceptCash(10.0);
    automat.selectItem(2);
    automat.acceptCash(5.0);
    automat.selectItem(2);
    automat.acceptCash(3.0);
    automat.selectItem(2);
    automat.acceptCash(4.5);
    automat.selectItem(0);
    automat.cancel();
    automat.showSelections();
  }}
```

Example I: Soda Machine (cont.)

- Running the `VendingMachineClient` class:

```
The vending machine contains these items:
Item number: 0
Item name: Coca cola Inventory: 10
Item number: 1
Item name: Pepsi Max Inventory: 10
Item number: 2
Item name: Pepsi Light Inventory: 2
Item number: 3
Item name: Sparkling water Inventory: 5
Each item costs: 2,50 GBP
Change available: 100,00 GBP
Till contains: 0,00 GBP
Inserted amount: 0,00 GBP
Accepted: 6,00 GBP
The item Pepsi Max is dispensed.
Change 3,50 GBP given back.
Thank you for buying our product - Welcome back!
Accepted: 10,00 GBP
The item Pepsi Light is dispensed.
Change 7,50 GBP given back.
```



```
Thank you for buying our product - Welcome back!
Accepted: 5,00 GBP
The item Pepsi Light is dispensed.
Change 2,50 GBP given back.
Thank you for buying our product - Welcome back!
Accepted: 3,00 GBP
Sorry, but we are out of Pepsi Light. Money is returned.
Accepted: 7,50 GBP
The item Coca cola is dispensed.
Change 5,00 GBP given back.
Thank you for buying our product - Welcome back!
Change 0,00 GBP given back.
Item purchase is cancelled.
The vending machine contains these items:
Item number: 0
Item name: Coca cola Inventory: 9
Item number: 1
Item name: Pepsi Max Inventory: 9
Item number: 2
Item name: Pepsi Light Inventory: 0
Item number: 3
Item name: Sparkling water Inventory: 5
Each item costs: 2,50 GBP
```

```
Change available: 81,50 GBP
Till contains: 28,50 GBP
Inserted amount: 0,00 GBP
```


Example II: Soda Machine (cont.)

- Extends `Item` class that handles price per unit:

```
public class ItemWithUnitPrice extends Item {
  double unitPrice;
  ItemWithUnitPrice(String itemName, int inventory, double price) {
    super(itemName, inventory);
    unitPrice = price;
  }
  public double getUnitPrice() { return unitPrice; }
  @Override public String toString() {
    return super.toString() + "\tUnit price: "
           + String.format("%.2f", unitPrice);
  }
}
```

  - extends properties (new field `unitPrice`)
  - extends behaviour (new instance method `getUnitPrice()`)
  - modifies behaviour (overrides instance method `toString()` from the `Item` class)
  - inherits all members from the `Item` class
  - uses the superclass constructor
  - uses the superclass `toString()` method through the keyword `super`

Example II: Soda Machine (cont.)

- A subclass for vending machines allowing items with different prices:

```
public class SodaMachine extends VendingMachine {
  SodaMachine(ItemWithUnitPrice[] items, double change) {
    super(items, 0.0, change);
  }
  @Override
  public double findPrice(int selectedItem) {
    return ((ItemWithUnitPrice)items[selectedItem]).getUnitPrice();
  }
  @Override
  public void showSelections() {
    String str = "The vending machine contains:";
    str += super.toString();
    str += "\nChange available: " + formatMoney(super.change) + " GBP";
    str += "\nTill contains: " + formatMoney(super.till) + " GBP";
    str += "\nInserted amount: " + formatMoney(super.receivedAmount) + " GBP";
    System.out.println(str);
  }
}
```


Example II: Soda Machine (cont.)

- A client that uses `ItemWithUnitPrice` and `SodaMachine` subclasses:

```
import java.util.Scanner;
public class SodaMachineClient {
  public static void main(String[] args) {
    ItemWithUnitPrice[] items = {
        new ItemWithUnitPrice("Coca cola 0.5l", 10, 4.00),
        new ItemWithUnitPrice("Pepsi Max 0.5l", 10, 4.00),
        new ItemWithUnitPrice("Pepsi Light 0.5l", 5, 4.50),
        new ItemWithUnitPrice("Sparkling water 0.5l", 5, 3.00),
        new ItemWithUnitPrice("Coca cola 0.3l", 10, 2.50),
        new ItemWithUnitPrice("Pepsi Max 0.3l", 5, 2.50) };
    SodaMachine automat = new SodaMachine(items, 100.00);
    Scanner keyboard = new Scanner(System.in);
    String answer;
    do {
      automat.showSelections();
      System.out.print("Enter amount: ");
      automat.acceptCash(keyboard.nextDouble());
      System.out.print("Select soda number: ");
      automat.selectItem(keyboard.nextInt());
      System.out.print("Do you want to purchase another soda (y/n)? ");
```

```
      answer = keyboard.next();
    } while (answer.charAt(0) == 'y' || answer.charAt(0) == 'Y');
    System.out.println("Status for the automat:");
    automat.showSelections();
  }
}
```

- Output from the running the `SodaMachineClient` class:

```
The vending machine contains:
Item number: 0
Item name: Coca cola 0.5l Inventory: 10 Unit price: 4,00
Item number: 1
Item name: Pepsi Max 0.5l Inventory: 10 Unit price: 4,00
Item number: 2
Item name: Pepsi Light 0.5l Inventory: 5 Unit price: 4,50
Item number: 3
Item name: Sparkling water 0.5l Inventory: 5 Unit price: 3,00
Item number: 4
Item name: Coca cola 0.3l Inventory: 10 Unit price: 2,50
Item number: 5
Item name: Pepsi Max 0.3l Inventory: 5 Unit price: 2,50
Change available: 100,00 GBP
Till contains: 0,00 GBP
```



```
Inserted amount: 0,00 GBP
Enter amount: 3,50
Accepted: 3,50 GBP
Select soda number: 0
Not enough money inserted. Coca cola 0.5l costs 4,00 GBP
Do you want to purchase another soda (y/n)? y
The vending machine contains:
Item number: 0
Item name: Coca cola 0.5l Inventory: 10 Unit price: 4,00
Item number: 1
Item name: Pepsi Max 0.5l Inventory: 10 Unit price: 4,00
Item number: 2
Item name: Pepsi Light 0.5l Inventory: 5 Unit price: 4,50
Item number: 3
Item name: Sparkling water 0.5l Inventory: 5 Unit price: 3,00
Item number: 4
Item name: Coca cola 0.3l Inventory: 10 Unit price: 2,50
Item number: 5
Item name: Pepsi Max 0.3l Inventory: 5 Unit price: 2,50
Change available: 100,00 GBP
Till contains: 0,00 GBP
Inserted amount: 3,50 GBP
Enter amount: 2,00
```

```
Accepted: 5,50 GBP
Select soda number: 0
The item Coca cola 0.5l is dispensed.
Change 1,50 GBP given back.
Thank you for buying our product - Welcome back!
Do you want to purchase another soda (y/n)? y
The vending machine contains:
Item number: 0
Item name: Coca cola 0.5l Inventory: 9 Unit price: 4,00
Item number: 1
Item name: Pepsi Max 0.5l Inventory: 10 Unit price: 4,00
Item number: 2
Item name: Pepsi Light 0.5l Inventory: 5 Unit price: 4,50
Item number: 3
Item name: Sparkling water 0.5l Inventory: 5 Unit price: 3,00
Item number: 4
Item name: Coca cola 0.3l Inventory: 10 Unit price: 2,50
Item number: 5
Item name: Pepsi Max 0.3l Inventory: 5 Unit price: 2,50
Change available: 98,50 GBP
Till contains: 5,50 GBP
Inserted amount: 0,00 GBP
Enter amount: 5,00
```



```
Accepted: 5,00 GBP
Select soda number: 1
The item Pepsi Max 0.5l is dispensed.
Change 1,00 GBP given back.
Thank you for buying our product - Welcome back!
Do you want to purchase another soda (y/n)? n
Status for the automat:
The vending machine contains:
Item number: 0
Item name: Coca cola 0.5l Inventory: 9 Unit price: 4,00
Item number: 1
Item name: Pepsi Max 0.5l Inventory: 9 Unit price: 4,00
Item number: 2
Item name: Pepsi Light 0.5l Inventory: 5 Unit price: 4,50
Item number: 3
Item name: Sparkling water 0.5l Inventory: 5 Unit price: 3,00
Item number: 4
Item name: Coca cola 0.3l Inventory: 10 Unit price: 2,50
Item number: 5
Item name: Pepsi Max 0.3l Inventory: 5 Unit price: 2,50
Change available: 97,50 GBP
Till contains: 10,50 GBP
Inserted amount: 0,00 GBP
```

Example II: Soda Machine (cont.)

- Output from a run when assertions are enabled:

```
The vending machine contains:
Item number: 0
Item name: Coca cola 0.5l Inventory: 10 Unit price: 4,00
Item number: 1
Item name: Pepsi Max 0.5l Inventory: 10 Unit price: 4,00
Item number: 2
Item name: Pepsi Light 0.5l Inventory: 5 Unit price: 4,50
Item number: 3
Item name: Sparkling water 0.5l Inventory: 5 Unit price: 3,00
Item number: 4
Item name: Coca cola 0.3l Inventory: 10 Unit price: 2,50
Item number: 5
Item name: Pepsi Max 0.3l Inventory: 5 Unit price: 2,50
Change available: 100,00 GBP
Till contains: 0,00 GBP
Inserted amount: 0,00 GBP
Enter amount: 5,00
Accepted: 5,00 GBP
Select soda number: 6
```



```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 6
 at VendingMachine.itemAvailable(VendingMachine.java:34)
 at VendingMachine.selectItem(VendingMachine.java:55)
 at SodaMachineClient.main(SodaMachineClient.java:19)
```

OOP (Object-oriented programming)

OOP is an implementation method,

where programs are organised as collections of cooperating objects,

where each object is an instance of a class, and

where all classes are members of a hierarchy of classes related through inheritance relationships.

If the last point is not fulfilled, the programming method is called *OBP* (*Object-based programming*).

Methods + data = objects

- One of the biggest advantages of OOP is *reuse, i.e. that classes can be used again to make new programs.*
  - Reuse of classes saves development time and reduces the probability of errors since the classes have been tested earlier.
  - The challenge is to create *reusable classes!*