Introductory Programming
Inheritance, sections 7.0-7.4

Anne Haxthausen, IMM, DTU

1. Class hierarchies: superclasses and subclasses (sections 7.0, 7.2)
2. The `Object` class: is automatically superclass for all classes (self study: section 7.2)
3. Abstract classes: serves as placeholders in class hierarchies (self study: section 7.2)
4. Inheritance: a subclass inherits fields and methods from its superclass (section 7.0)
5. Constructors are not inherited (section 7.0)
6. Visibility modifiers (section 7.0)
7. Type conversion and check (section 7.4)
8. Overriding of methods: redefining an inherited method (section 7.1)
9. Polymorphism: the class of an object decides which method is invoked (section 7.4)
10. Single versus multiple inheritance (section 7.0)

a. Parts of this material are inspired by/originate from a course at ITU developed by Niels Hallenberg and Peter Sestoft on the basis of a course at KVL developed by Morten Larsen and Peter Sestoft.

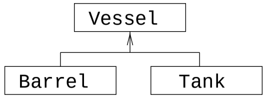
Concept hierarchies

Examples of *concepts* are: animal, person, vessel, . . .

Related concepts can be arranged in a *hierarchy* (according to how general they are).

Example 1: ‘animals’ can be divided into ‘mammals’ (’pattedyr’), ‘birds’, ‘fish’

Example 2: ‘vessels’ (‘beholdere’) can be divided into ‘barrels’ (‘tønder’), ‘tanks’, . . .



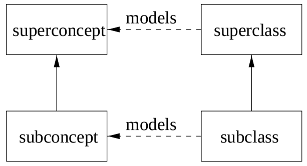
The concept ‘vessel’ is more general than ‘tank’, as one can say ‘a tank *is a* vessel’.

As every concept has some properties, one can also explain the hierarchy by saying that a concept B is a subconcept of another concept A, if the subconcept B has (inherited/arvet) all the properties of A and probably has some more properties.

Concept hierarchies are often used to describe the world around us.

Modelling concept hierarchies as class hierarchies

In Java and many other object-oriented programming languages concept hierarchies are modelled by class hierarchies. (Classes represent concepts, as you know.)



A superclass models a general concept (e.g. a vessel), a subclass models a more special concept (e.g. a tank).

Inheritance:

A subconcept has all the properties of its superconcept.

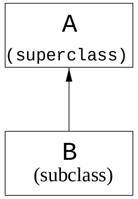
Therefore, a subclass has all fields and methods of its superclass. Often the subclass is made more specific than the superclass by defining more fields and methods.

Creation of class hierarchies in Java

Java has a language construct for making class hierarchies:

A class B can be defined as an *extension* of an existing class A so that A becomes a *superclass* of B , and B a *subclass* of A .

```
class B extends A {
  new fields and methods
  redefined methods
  constructors
}
```



Example:

```
class Tank extends Vessel {
  ...
}
```

Inheritance

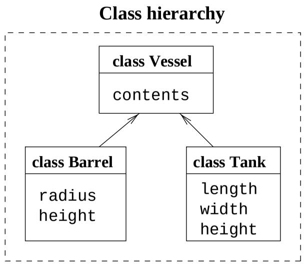
A subclass *inherits (arver)* methods and fields, but not constructors, from its superclass.

They can be used in the subclass as if they were defined in the subclass.

In addition to that, a subclass can define new fields and methods, and/or override (overskrive) methods that would otherwise have been inherited.

A subtlety of private fields and methods will be explained later.

Example: class hierarchy for vessels



Class `Vessel` should represent what is common for all kinds of vessels.

Class `Barrel` should represent barrel formed vessels and `Tank` tank formed vessels.

`Barrel` and `Tank` should inherit properties from `Vessel`.

`Barrel` and `Tank` should each define properties that are special for barrels and tanks, respectively.

Implementation of vessel hierarchy in Java

```
class Vessel {
  double contents; //in litre (= cubic decimetre, dm^3)
}

class Tank extends Vessel {
  double length, width, height; //in decimetre, 1 dm = 10 cm

  Tank(double length, double width, double height)
  { this.length = length; this.width = width; this.height = height; }
}

class Barrel extends Vessel {
  double radius, height; //in decimetre, 1 dm = 10 cm

  Barrel(double radius, double height)
  { this.radius = radius; this.height = height; }
}
```

Example: use of objects from the vessel hierarchy

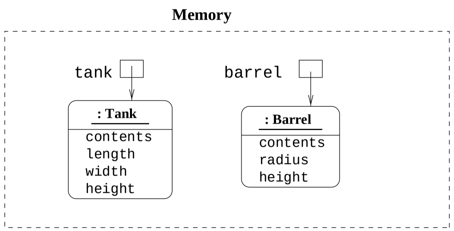
```
public class Vessel1 {
  public static void main(String[] args) {
    Tank tank = new Tank(15, 9, 12);
    Barrel barrel = new Barrel(2.5, 8);

    tank.contents = 0; barrel.contents = 1.5;
    System.out.println("Contents of tank = " + tank.contents);
    System.out.println("Width of tank  = " + tank.width);

  }
}
```

All `Tank`- and `Barrel`-objects have a `contents` field, inherited from class `Vessel`.

Objects in `Vessel1.java`



Exercise: Is it legal to write `System.out.println(barrel.width)` ?

`System.out.println(barrel.height)` ?

`System.out.println(tank.height)` ?

Constructors are not inherited

However, the first a subclass constructor does, is to invoke a constructor for its superclass.

This can be done explicitly with an invocation of the form **super**(...).

If this is not done explicitly, then Java automatically makes an invocation of **super**(), i.e. of a constructor with no parameters of the superclass. (Such a constructor exists automatically if you have not defined one.)

Example: invocation of super class constructors

```
class Vessel {
  double contents;

  Vessel() { contents = 0.0; }
  Vessel(double contents) { this.contents = contents; }
}

class Tank extends Vessel {
  double length, width, height;

  Tank(double length, double width, double height)
  { this.length = length; this.width = width; this.height = height; }
}

class Barrel extends Vessel {
  double radius, height;

  Barrel(double contents, double radius, double height)
  { super(contents); this.radius = radius; this.height = height; }
}
```

```
public class Vessel2 {
  public static void main(String[] args) {
    Tank tank = new Tank(15, 9, 12);
    Barrel barrel = new Barrel(1.5, 2.5, 8);
    System.out.println("Contents of tank = " + tank.contents);
    System.out.println("Contents of barrel = " + barrel.contents);
  }
}
```

Execution of this program gives the following output:

```
Contents of tank = 0.0
Contents of barrel = 1.5
```

Subtlety of private methods and fields

If a field (or a method) is **private** in a superclass A, then you cannot explicity refer to it in subclasses of A.

However, the field exists in subclass objects.

Default visibility

The visibility properties for a field or a method that has no visibility modifier are like **public** in classes (and subclasses) in the same package, and like **private** for classes (and subclasses) in other packages.

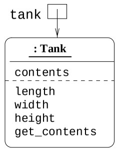
Protected methods and fields

The visibility properties for a field or a method declared as **protected** are like **public** in classes (and subclasses) in the same package and in all subclasses in other packages, but like **private** for non subclasses in other packages.

Example: private methods and fields

```
class Vessel {
  private double contents; //in litre (= cubic decimetre, dm^3)
  Vessel() { contents = 0.0; }
  double getcontents() { return contents; }
}
class Tank extends Vessel {  ...  } // contents unknown name in Tank

public class Vessel6 {
  public static void main(String[] args) {
    Tank tank = new Tank(15, 9, 12);
    System.out.println("Contents of tank = " + tank.getcontents());
    //System.out.println("Contents of tank = " + tank.contents);  is illegal
} }
```



Type conversions

Implicit widening conversion from subclass to superclass

A variable of type T can refer to objects belonging to class T *and its subclasses*.

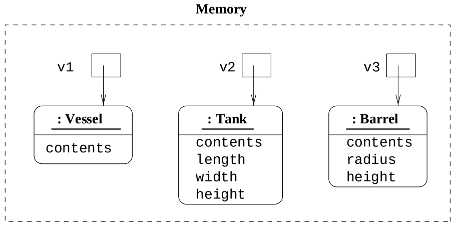
A variable that can refer to objects of different classes is named a *polymorphic* reference.

Example: (assume given the declarations on page 7 11)

```
Vessel v1 = new Vessel();
Vessel v2 = new Tank(15, 9, 12);
Vessel v3 = new Barrel(1.5, 2.5, 8);
```

A variable (like `v1, v2, v3`) of type `Vessel` can refer to an object of class `Vessel`, `Tank` or `Barrel`.

Memory for the example



Type conversions

Explicit narrowing conversion with cast from superclass to subclass

Example:

```
  Vessel v2 = new Tank(15, 9, 12);
  Vessel v3 = new Barrel(1.5, 2.5, 8);
  Tank tank1  = v2;           //illegal (gives compilation error)
  Tank tank2  = (Tank) v2;    //an explicit cast is needed
  Tank tank3  = (Tank) v3;    //illegal (gives runtime error)
```

The type conversion `(Tank)v3` will give rise to a runtime error when the program is executed as `v3` does not refer to a `Tank` object.

A variable has a type, an object has a class

Important distinction:

- A variable has a declared *type*, e.g. the variable `v2` of the example above has type `Vessel`.
- An object has (i.e. belongs to) a specific *class*. Which one, is determined by the constructor that was used to create the object.

  E.g. an object created with **new** `Tank(15, 9, 12)` has class `Tank`.

Check of access to the fields and methods of an object

Rule:

Field access `o.f` and method invocation `o.m(...)` are checked wrt the declared type `T` of the variable `o`:

`o.f` is legal, if `T` has a field `f` with appropriate visibility properties (behaves like **public** and not like **private** at the given place). Similarly for `o.m(...)`.

Example: `Vessel v2 = new Tank(15, 9, 12);`

The variable `v2` has type `Vessel`, and `Vessel` has a field `contents`.

So the expression `v2.contents` is accepted by the Java compiler.

But `Vessel` does not have a field named `width`, so the expression `v2.width` is rejected by the Java compiler, although `v2` actually refers to a `Tank` object that has a `width` field.

Overriding of methods

Overloading

When a class defines several methods with the same name, but distinct parameter types, it is called *overloading*, cf. overhead collection 4.

Overriding

When a subclass (re-)defines a method with the same name `m`, result type and parameter types as the superclass does, it is called *overriding* (Danish: overskrivning). Then the subclass does not inherit the superclass method `m`, but has its own version of `m`. However, the version of the superclass can be accessed via **super**`.m(...)`.

This flexibility is good, because related classes can use the same name conventions for methods that do “the same”.

If you declare a method to be **final** then you *cannot* override it.

Example of overriding of a volume method in the vessel hierarchy

```
class Vessel {
  double contents;
  ...
  double volume() { return 0; }
}

class Tank extends Vessel {
  double length, width, height;
  ...
  double volume() { return length * width * height; }
}

class Barrel extends Vessel {
  double radius, height;
  ...
  double volume() { return height * Math.PI * radius * radius; }
}
```

The subclasses *override* (overskriver, omdefinerer) the `volume` method from the superclass.

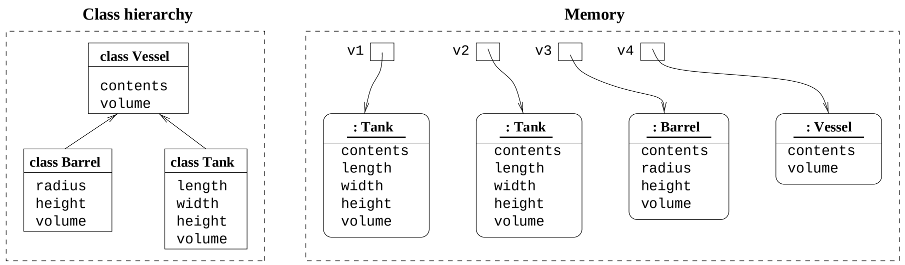
```
public class Vessel4 {
  public static void main(String[] args) {
    Vessel v1 = new Tank(15, 9, 12);
    Vessel v2 = new Tank(0.7, 0.7, 2.05);
    Vessel v3 = new Barrel(1.5, 2.5, 8);
    Vessel v4 = new Vessel();

    System.out.println("Volume of v1 = " + v1.volume());
    System.out.println("Volume of v2 = " + v2.volume());
    System.out.println("Volume of v3 = " + v3.volume());
    System.out.println("Volume of v4 = " + v4.volume());
  }
}
```

Output when executing Vessel4:

```
Volume of v1 = 1620.0
Volume of v2 = 1.0044999999999997
Volume of v3 = 157.07963267948966
Volume of v4 = 0.0
```

Class hierarchy and memory for Vessel4.java



Exercise: Which of the three `volume` methods are invoked by `v1.volume()`, `v2.volume()`, `v3.volume()` and `v4.volume()` ?

Overriding and polymorphism

`v1` is a polymorphic reference that can refer to `Vessel`, `Tank` and `Barrel` objects. As each of these have a `volume` method, there are potentially 3 possibilities for, which method is invoked by `v1.volume()`.

Which one, is determined by the class of that object `v1` is referring to.

Hence, the declared type for `v1` — which is `Vessel` — does not determine which method is invoked. However, in `Vessel` there *must* be a method with the given name, otherwise the Java compiler rejects the program (cf. the rule on page 19).

Rule:

Which version of an overridden method `m`, that is invoked with `o.m(...)`, depends on the class of the object that `o` refers to, not the type of `o`.

Example of use of super to invoke an overridden method

```
class Vessel {
  ...
  public String toString() { return "contents: " + contents +  " l"; }
}

class Tank extends Vessel {
  ...
  public String toString() {
    return "Tank with volume: " + volume() + " l and " + super.toString();
  }
}

public class Vessel5 {
  public static void main(String[] args) {
    Vessel v1 = new Tank(15, 9, 12);
    System.out.println("v1: " + v1.toString());
}
```

Output: `v1: Tank with volume: 1620.0 l and contents: 0.0 l`

Shadowing fields

If you in a subclass (re-)declare a field with the same name `f` as a field in its superclass, then you get two fields.

The name of the field of the subclass is just `f`.
The field from the superclass can be accessed in the subclass as **super**`.f` (but not if it is **private**).

Redeclaring fields usually results in errors and confusion. **Only override methods, not fields!**

Summary about the super reference

super is a reference like **this**.

1. Constructors from a superclass can be invoked from a subclass with **super**(...).
2. Methods m from a superclass can be invoked from a subclass with **super.**m(...).
3. Fields f from a superclass can be accessed with **super.**f.

2 and 3 do not hold, when m and f are **private**.

Single and multiple inheritance

In some object-oriented languages, a class can have several superclasses.

Multiple inheritance is useful when you have two different concept hierarchies at the same time.

E.g. vessels (Vessel, Tank, Barrel) and colors (Plain, Colored).

- A colored barrel is colored (Colored) as well as a vessel (Vessel).
  So an object should could be an instance of Colored and Vessel at the same time.
- A barrel is a Vessel, but not a Colored.
  So Vessel can not be a subclass of Colored.
- A colored piece of paper is Colored, but not a Vessel.
  So Colored can not be a subclass of Vessel.

Single and multiple inheritance, continued

Java only supports single inheritance: a class can only have one immediate superclass.

This is because multiple inheritance leads to theoretical and practical problems.

Example: In which order should the constructors of the superclasses be invoked?

Example: If two methods with same signature are inherited from two different superclasses, which one should then be used?

Inheritance in Java: summary

- Classes can be ordered in hierarchies that reflect concept hierarchies.
- A subclass inherits fields and methods from its superclass, i.e. they can be used as if they were defined in the subclass. Exceptions:
  - Constructors are not inherited. However, they can be used in a subclass as **super**(...).
  - Private fields and methods are not inherited, but exist and can be accessed indirectly.
- A subclass can define new fields and methods.
- A subclass can redefine (overskrive, ‘override’) existing methods `m` (that are not **final**). In this case the subclass can access `m` of the superclass using the name **super**.`m` .
- Which version of `m` that is invoked with `o.m(...)`, depends on the class of the object to which `o` refers, not on the type `T` of the variable (`o`) .
- The subclass can redeclare a field, but it is not recommended.
- A variable `o` of type `T` can refer to objects of class `T` and all its subclasses.
- Field access `o.f` and method invocation `o.m(...)` are checked with respect to the declared type `T` of the variable `o`, not with respect to the class of the referenced object.
- You can explicitly type convert (‘cast’) an expression of type `T` to a subclass of `T`.

Advantages of using inheritance

- class hierarchies can explicitly reflect concept hierarchies of the problem domain
- code can be re-used (code is faster to write and easier to maintain)