

9 More about inheritance

Polymorphism

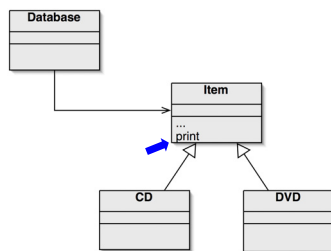
Main concepts to be covered

- method polymorphism
- static and dynamic type
- overriding
- dynamic method lookup
- protected access

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 2

The inheritance hierarchy



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 3

Print method in Item

```
public class Item
{
    ...

    public void print()
    {
        System.out.print("title: " + title +
            " (" + playingTime + " mins)");
        if(gotIt) {
            System.out.println("");
        } else {
            System.out.println();
        }
        System.out.println("    " + comment);
    }
    ...
}
```

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 4

Conflicting output

What we want

```
CD: A Swingin' Affair (64 mins)*
Frank Sinatra
tracks: 16
my favourite Sinatra album

DVD: O Brother, Where Art Thou? (106 mins)
Joel & Ethan Coen
The Coen brothers' best movie!
```

What we have

```
title: A Swingin' Affair (64 mins)*
my favourite Sinatra album

title: O Brother, Where Art Thou? (106 mins)
The Coen brothers' best movie!
```

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 5

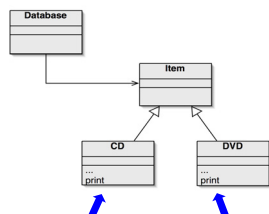
The problem

- The `print` method in `Item` only prints the common fields.
- Inheritance is a one-way street:
 - A subclass inherits the superclass fields.
 - The superclass knows nothing about its subclass's fields.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 6

Attempting to solve the problem



- Place `print` where it has access to the information it needs.
- Each subclass has its own version.
- But `Item`'s fields are private.
- `Database` cannot find a `print` method in `Item`.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 7

Static type and dynamic type

- A more complex type hierarchy requires further concepts to describe it.
- Some new terminology:
 - static type
 - dynamic type
 - method dispatch/lookup

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 8

Static and dynamic type

What is the type of `c1`?

```
Car c1 = new Car();
```

What is the type of `v1`?

```
Vehicle v1 = new Car();
```

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 9

Static and dynamic type

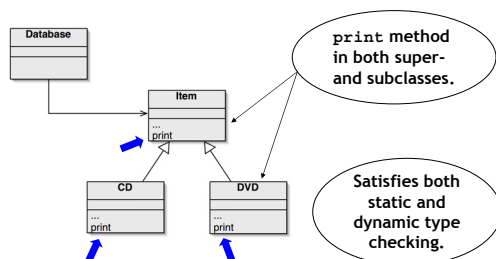
- The declared type of a variable is its *static type*.
- The type of the object a variable refers to is its *dynamic type*.
- The compiler's job is to check for static-type violations.

```
for(Item item : items) {
    item.print(); // Compile-time error.
}
```

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 10

The solution: Overriding



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 11

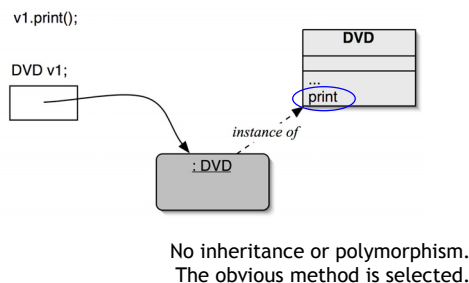
Overriding

- Superclass and subclass define methods with the same signature.
- Each has access to the fields of its class.
- Superclass method satisfies static type checking.
- Subclass method is called at runtime - it *overrides* the superclass version.
- What becomes of the superclass version?

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 12

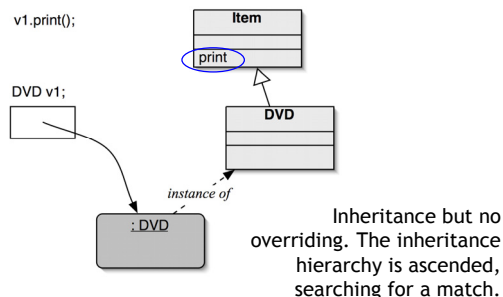
Method lookup



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 13

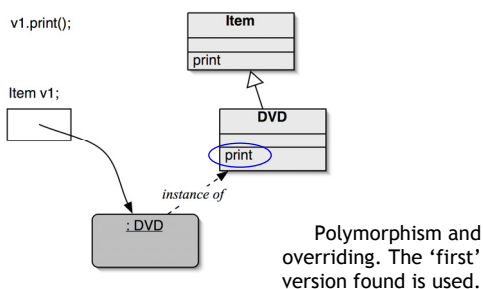
Method lookup



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 14

Method lookup



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 15

Method lookup summary

- The variable is accessed.
- The object stored in the variable is found.
- The class of the object is found.
- The class is searched for a method match.
- If no match is found, the superclass is searched.
- This is repeated until a match is found, or the class hierarchy is exhausted.
- Overriding methods take precedence.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 16

Super call in methods

- Overridden methods are hidden ...
- ... but we often still want to be able to call them.
- An overridden method *can* be called from the method that overrides it.
 - `super.method(...)`
 - Compare with the use of `super` in constructors.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 17

Calling an overridden method

```
public class CD extends Item
{
    ...
    public void print()
    {
        super.print();
        System.out.println("    " + artist);
        System.out.println("    tracks: " +
                           numberOfTracks);
    }
    ...
}
```

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 18

Method polymorphism

- We have been discussing *polymorphic method dispatch*.
- A polymorphic variable can store objects of varying types.
- Method calls are polymorphic.
 - The actual method called depends on the dynamic object type.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 19

The Object class's methods

- Methods in `Object` are inherited by all classes.
- Any of these may be overridden.
- The `toString` method is commonly overridden:
 - `public String toString()`
 - Returns a string representation of the object.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 20

Overriding toString

- Explicit `print` methods can often be omitted from a class:
 - `System.out.println(item.toString());`
- Calls to `println` with just an object automatically result in `toString` being called:

```
for(Item item : items) {  
    System.out.println(item);  
}
```

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 21

Overriding toString in Item

```
public class Item  
{  
    ...  
  
    public String toString()  
    {  
        String line1 = title +  
            " (" + playingTime + " mins)";  
  
        if(gotIt) {  
            return line1 + "\n" + " " +  
                comment + "\n";  
        } else {  
            return line1 + "\n" + " " +  
                comment + "\n";  
        }  
    }  
    ...  
}
```

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 22

Overriding toString in CD

```
public class CD extends Item  
{  
    ...  
  
    public String toString()  
    {  
        return  
            super.toString() + "\n" +  
            " " + artist + "\n" +  
            " " + tracks + "\n";  
    }  
    ...  
}
```

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 23

Protected access

- Private access in the superclass may be too restrictive for a subclass.
- The closer inheritance relationship is supported by *protected access*.
- Protected access is more restricted than public access.
- We still recommend keeping fields private.
 - Define protected accessors and mutators.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 24

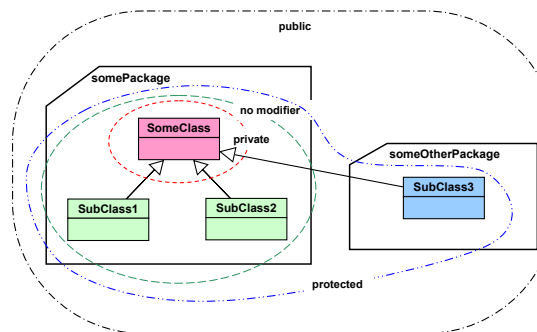
Visibility for class members

Modifier	Class	Package	Subclass	World
<code>public</code>	Yes	Yes	Yes	Yes
<code>protected</code>	Yes	Yes	Yes	No
no modifier (package private)	Yes	Yes	No	No
<code>private</code>	Yes	No	No	No

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 25

Access levels



Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 26

Access levels

Example

A class member (method, constructor, variable) which is declared in `SomeClass` with access level

- `private`
is only visible in `SomeClass`
- no modifier (package private)
is visible in `SomeClass`, `SubClass1`, `SubClass2`
- `protected`
is visible in `SomeClass`, `SubClass1`, `SubClass2`, `SubClass3`
- `public`
is visible everywhere

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 27

Access levels and overriding

- An overriding method in a sub class must be at least as visible as the overridden method in its base class.

Visibility in base class	Allowed visibility in sub class
<code>public</code>	<code>public</code>
<code>protected</code>	<code>protected</code> <code>public</code>
package private	package private <code>protected</code> <code>public</code>

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 28

Overriding and Covariant Return Types

- In Java, the return type of an overriding method is allowed to be a subtype of the return type of the overridden method.
- This is called *covariance*.
- The benefit is type safety
 - Less need for unsafe type casts when calling overridden methods.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 29

Overriding and Covariant Return Types

```
public class A {}
public class B extends A {}

public class Base {
    public A f() { return new A(); }
    public Base g() { return new Base(); }
}

public class Sub extends Base {
    public A f() { return new A(); }
    public A f() { return new B(); }
    public B f() { return new B(); }

    // In particular, covariance can be
    // applied to the class itself:
    public Base g() { return new Base(); }
    public Base g() { return new Sub(); }
    public Sub g() { return new Sub(); }
}
```

Any of these is a correct overriding of f

Any of these is a correct overriding of g

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 30

Overriding and Covariant Return Types

Example (no covariance)

```
public class SomeClass implements Cloneable {
    public Object clone() {
        return super.clone();
    }
}

...

SomeClass x = new SomeClass();
...
SomeClass y = (SomeClass)x.clone();
```

clone overrides
Object.clone

Without covariance we need
an unsafe type cast here

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 31

Overriding and Covariant Return Types

Example (using covariance)

```
public class SomeClass implements Cloneable {
    public SomeClass clone() {
        return (SomeClass)super.clone();
    }
}

...

SomeClass x = new SomeClass();
...
SomeClass y = x.clone();
```

Overriding
Object.clone using
sub type as return
type

No need for type cast

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 32

The @Override annotation

- A simple typing mistake can easily spoil overriding!

Example

```
public class Base {
    public void f() { ... }
    public void f(int x) { ... }
    public void g(float x) { ... }
}

public class Sub extends Base {
    public void f(int x) { ... }
    public void g(int x) { ... }
}
```

Ok, f overrides
Base.f(int)

Sub.g(int) does not override Base.g(float)
- rather, it is a new method!

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 33

The @Override annotation

- The **@Override** annotation indicates to the compiler that a method is intended to override some method in a super class.
- If a method is annotated with **@Override** but does not override a super class method, a compilation error results.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 34

The @Override annotation

```
public class Base {
    public void f() { ... }
    public void f(int x) { ... }
    public void g(float x) { ... }
}

public class Sub extends Base {
    @Override
    public void f(int x) { ... }
    @Override
    public void g(int x) { ... }
}
```

Ok, f overrides
Base.f(int)

The compiler
signals an error

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 35

Review

- The declared type of a variable is its static type.
 - Compilers check static types.
- The type of an object is its dynamic type.
 - Dynamic types are used at runtime.
- Methods may be overridden in a subclass.
- Method lookup starts with the dynamic type.
- Protected access supports inheritance.
- Overriding methods may have covariant return types.
- The **@Override** annotation makes overriding safer.

Object oriented programming, DAT042, D2, 11/12, lp 1

Lecture 9 36