Subclasses & Interfaces

Lecture 11 of TDA 540
Object-Oriented
Programming

Jesper Cockx

Fall 2018

Chalmers University of Technology — Gothenburg University

Last lecture: classes and objects

An object consists of a private state and a public interface.

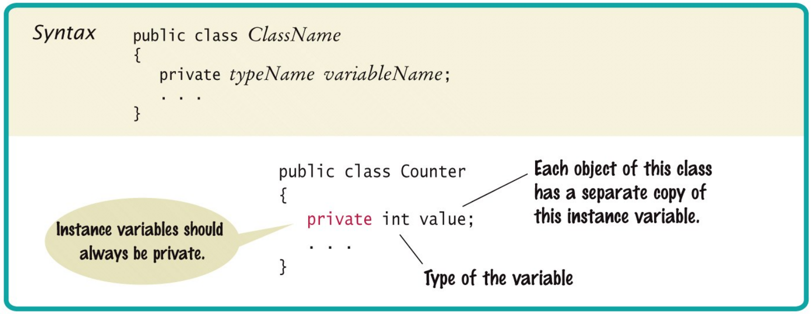
A class describes a collection of objects with a common structure:

Attributes describe how objects' state is represented in memory.

Methods describe how objects can be observed and modified.

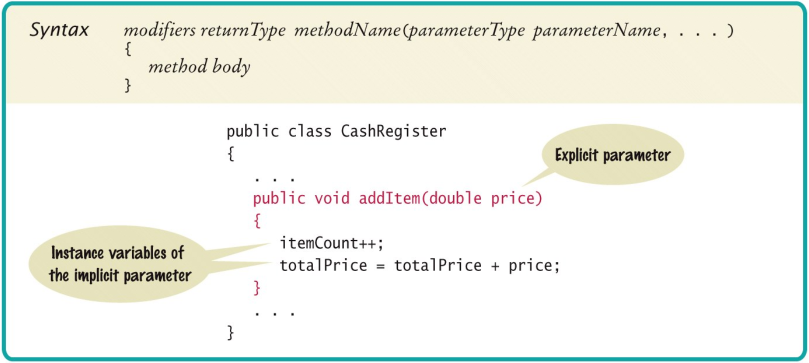
Constructors describe how to construct new objects of the class.

Last lecture: attributes



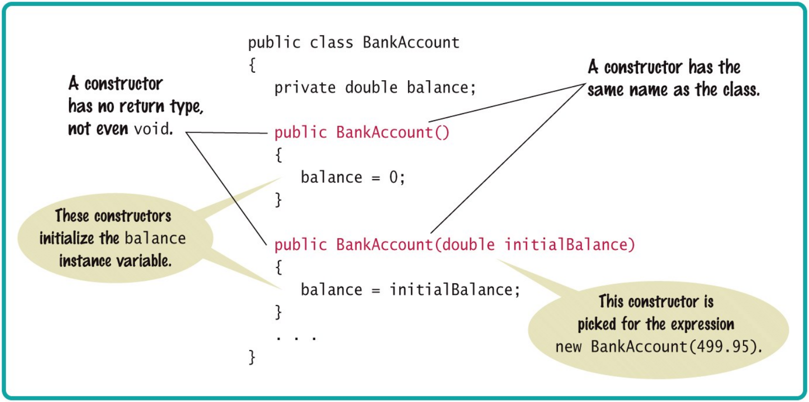
Syntax 8.1

Last lecture: methods



Syntax 8.2

Last lecture: constructors



Syntax 8.3

Inheritance

Inheritance

Inheritance = relation between a general class (the superclass) and a more specific one (the subclass)

Example: a car is a vehicle
$\Rightarrow$ `Car` is a subclass of `Vehicle`

In Java:

```
class Vehicle { ... }
class Car extends Vehicle { ... }
```

All members (attributes and methods) of `Vehicle` are automatically also members of `Car`.

Inheritance example

```
class Account {
  int balance;
  void deposit(int amount) {
    balance += amount;
  }
}

class CheckingAccount
  extends Account {
   void withdraw(int amount) {
     deposit(-amount);
   }
   void close() {
     balance = 0;
   }
}
```

Using `CheckingAccount`:

```
CheckingAccount a =
  new CheckingAccount();
a.deposit(1000);
a.withdraw(500);
a.close();
```

Warning: do not redefine attributes

If a class has an attribute with the same name as one of the superclass' attributes, it gets two copies of the attribute:

```
class Account {
  int balance;
  int getBalance() {
    return balance;
  }
}
```

```
class CheckingAccount
  extends Account {
  int balance;
  void withdraw(int amount) {
    balance -= amount;
  }
}
```

Calling `withdraw()` does not change the result of `getBalance()`!

Overriding

Overriding = redefine a method from the superclass

```java
class Account {
  int balance;

  void withdraw(int amount) {
    balance -= amount;
  }
}
```

```java
class NoOverdrawnAccount
  extends Account {
   // redefinition of withdraw
   @Override
   void withdraw(int amount) {
     if (amount <= balance)
       balance -= amount;
   }
}
```

super: referencing the superclass

The keyword super denotes a reference to the current object as an instance of the superclass.

```java
class Account {
  int balance;
  void withdraw(int amount) {
    balance -= amount;
  }
}
```

```java
class NoOverdrawnAccount
  extends Account {
   @Override
   void withdraw(int amount) {
     if (amount <= balance)
       // call withdraw
       // from Account
       super.withdraw(amount);
     }
}
```

Inheritance and constructors

You can call the constructor of the superclass using `super(...)`. This must be the first statement in the constructor of the subclass.

```
class Account {
  int balance;
  Account(int balance) {
    this.balance =
      balance;
}
```

```
class LimitedAccount
extends Account {
  int maxOverdraw;
  LimitedAccount(int balance,
      int max) {
    // calls Account(balance);
    super(balance);
    this.maxOverDraw = max;
  }
}
```

`final` methods and classes

The keyword `final` can also be used to restrict inheritance:

- a method marked as `final` cannot be overridden.
- a class marked as `final` cannot be inherited from.

Example: `String` is `final`, so we cannot create new subclasses of `String`.

Abstract classes and methods

An `abstract` method has a signature but no implementation.

Only `abstract` classes can have `abstract` methods. Abstract classes cannot be instantiated.

Non-abstract subclasses must override all abstract methods.

Abstract class example

```java
// Partial implementation
abstract class Account {
  int balance;
  abstract void addInterest();
}

class CheckingAccount
extends Account {
  @Override
  void addInterest() {
    return;
  }
}
```

```java
class SavingAccount
extends Account
  static INTEREST = 0.001;
  @Override
  void addInterest() {
    balance += balance*INTEREST;
  }
}
```

Developing a class hierarchy

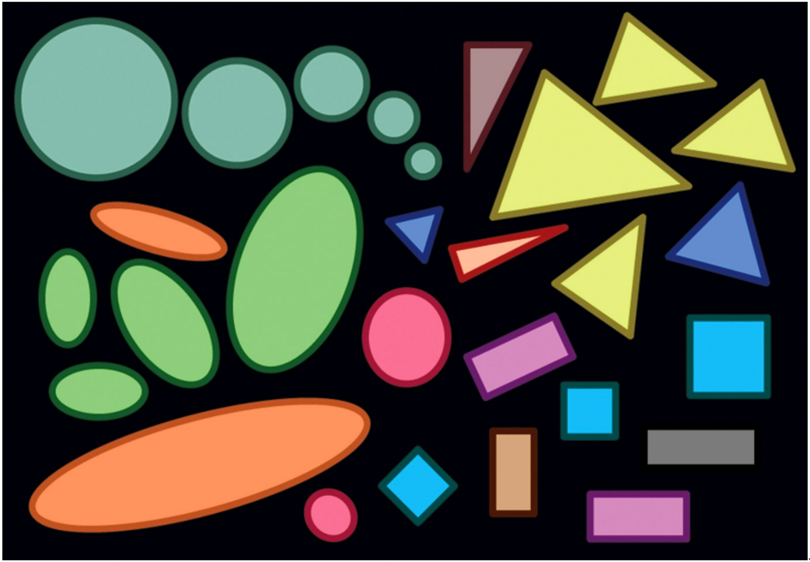
1. List the classes that are part of the hierarchy
2. Organize the classes according to the 'is a' relation
3. Determine responsibilities of each class, starting at the top of the hierarchy
4. Implement each class
   4.1 Declare the public interface
   4.2 Identify instance variables
   4.3 Implement constructors and methods
5. Test the whole hierarchy

When to create a new subclass?

Not every kind of object needs its own subclass:

- If objects vary in their behaviour
  $\Rightarrow$ different subclasses
- If objects only vary in some values
  $\Rightarrow$ one class is enough

Live coding:
designing a class hierarchy of shapes



Inheritance and types

Every class `C` corresponds to a `type`.

If `C` is a subclass of another class `B`, then `C` is a subtype of `B`: an object of type `C` can be used as an object of type `B`.

```
class Car {
  void openDoor()
  { /* ... */ }
}
```

```
class Convertible
extends Car {
  void openTop()
  { /* ... */ }
}
```

A convertible is a car!

Liskov's substitution principle

A program that expects an object of a superclass should also work when given an object of a subclass instead.

e.g. a program that works with a `Vehicle` should also work for a `Car`.

- Subclass can only add new attributes and methods, never remove them
- Return types of methods can only become more specific
- Argument types can only become more general

15 min. break

Interfaces

Interfaces

An interface is a list of abstract operations describing the *public interface* (API) of a class.

```java
public interface IGlass {
  double getCurrentVolume();
  void addWater(double amount);
  void removeWater(double amount);
}
```

All methods are automatically `public` and `abstract`.

No attributes[1] or constructors.

[1]Except for `static final` attributes

Interfaces and classes

A class can implement one or more interfaces:

- it must override all methods of the interfaces (no need for `@Override`)
- it can also introduce other members (private or public) without restrictions

```
class Glass implements IGlass {
  private double volume;
  int getCurrentVolume() {
    return volume;
  }
  // ... other implementations ...
}
```

Interfaces and inheritance

An interface also can inherit from one or more interfaces (but not from classes), by providing additional public methods (or constants).

```
interface IAccount {
  void deposit(long amount);
  // ...
}

interface ISavingAccount
  extends IAccount {
    static final double INTEREST = 0.001;
    void addInterest();
}
```

Interfaces and types

Every interface `I` also corresponds to a type.

Types of interfaces and classes are related by inheritance:

- If a class `C` implements an interface `I`, then `C` is a subtype of `I`.
- If an interface `J` extends another interface `I`, then `J` is a subtype of `I`.

A spectrum of abstraction

Classes and interfaces are two opposite endpoints on a spectrum of abstraction:

(CONCRETE) CLASS	INTERFACE
complete implementation	no implementation
must have constructor	cannot have constructors
can be instantiated	cannot be instantiated
all visibilities	only `public` visibility
completely concrete	completely abstract

A spectrum of abstraction

Classes and interfaces are two opposite endpoints on a spectrum of abstraction:

(CONCRETE) CLASS	ABSTRACT CLASS	INTERFACE
complete implementation	partial implementation	no implementation
must have constructor	may have constructor	cannot have constructors
can be instantiated	cannot be instantiated	cannot be instantiated
all visibilities	all visibilities	only `public` visibility
completely concrete	partially abstract	completely abstract

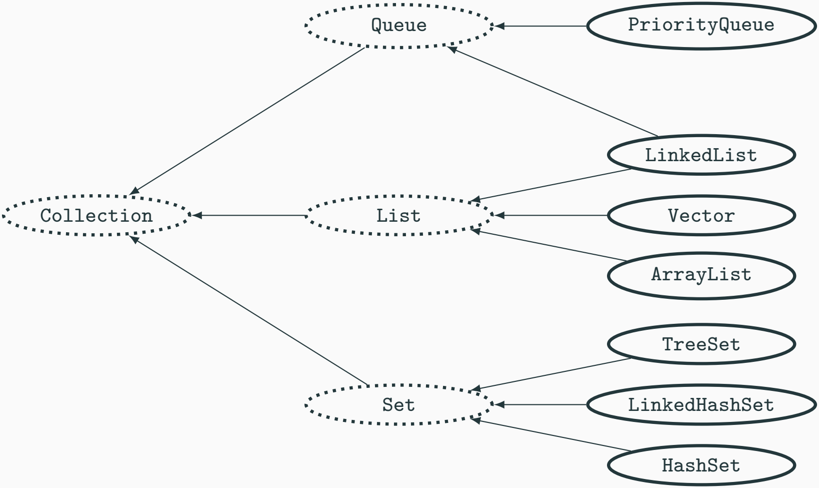
The collections framework

The Collections framework

Java's Collections framework is a part of the standard library containing commonly used data structures such as `ArrayList`.

- Interfaces are separated from their concrete implementations.
- Multiple different implementations of each interface: user can choose best one based on the situation.
- All interfaces are generic: they can store objects of arbitrary classes (e.g. `ArrayList<String>`, `ArrayList<Integer>`, `ArrayList<Object>`, ...).

Overview of the Collections framework



Using the Collections framework

Official documentation:

`https://docs.oracle.com/javase/8/docs/technotes/guides/collections/`

1. Select the interface that provides the operations your application needs
2. Select one implementation class of the interface that offers efficient implementation of those operations

In most cases, you do not have to worry too much about the implementation details.

The `List` interface

A list is an *ordered* collection of elements

```
interface List<E> {
  void add(int index, E element);

  E get(int index);

  E remove(int index);

  int size();

  // ... several more methods are available ...
}
```

Implementations of the `List` interface

Two implementations of the `List` interface:

- `ArrayList` uses an array to store data
  - `get` is very fast, `add` and `remove` are slower
- `LinkedList` stores data in a sequence of nodes referencing each other
  - `add` and `remove` are fast, `get` is slower

Both perform automatic resizing: they grow as we add more elements.

`ArrayList` is a good default choice.

The Set interface

A set is an *unordered* collection with *no duplicates*

```
interface Set<E> {
  void add(E element);

  boolean contains(Object o);

  boolean remove(Object o);

  int size();

  // ... several more methods are available ...
}
```

Implementations of the `Set` interface

Two implementations of the `Set` interface:

- `HashSet` stores elements in buckets according to their `hashCode`
  - all operations are very fast, takes more memory
- `TreeSet` stores elements in a tree structure
  - all operations are quite fast, takes less memory

`HashSet` is a good default choice.

The `Map` interface

A map is a data structure associating *keys* to *values*

```
interface Map<K,V> {
  void put(K key, V value);

  V get(Object key);

  V remove(Object key);

  boolean containsKey(Object o);

  int size();
  // ... several more methods are available ...
}
```

Implementations of the `Map` interface

Two implementations of the `Map` interface:

- `HashMap` stores elements in buckets according to the `hashCode` of the key
  - all operations are very fast, takes more memory
- `TreeMap` stores elements in a tree structure
  - all operations are quite fast, takes less memory

`HashMap` is a good default choice.

The `Queue` interface

A queue is an *ordered* collection (like `List`) meant to store a sequence of objects that await processing

```
interface Queue<E> {
  boolean offer(E e); // add element to the queue

  E remove();  // get first element and remove it

  E element(); // get first element (do not remove)

  // ... several more methods are available ...
}
```

Implementations of the `Queue` interface

Two implementations of the `Queue` interface:

- `LinkedList` keeps elements in the order they were added
  - `element` returns element that was added first (FIFO: first-in, first-out)
- `PriorityQueue` assigns a priority to each element
  - `element` returns element with highest priority

Generic classes and interfaces

A generic class or interface has one or more parameters written as `<E>`.

```
interface Set<E>
```

We can instantiate the parameter to any type[2]:

```
Set<String> names = new HashSet<String>();
```

Once the parameter is instantiated, it is fixed:

```
HashSet<Integer> intSet;
intSet = new HashSet<String>(); // type error
```

[2] except primitive types, use wrapper types instead

Subtyping and generic classes

Warning: subtyping does *not* extend through generic types.

- `Car` is a subtype of `Vehicle`
- `Set<Car>` is not a subtype of `Set<Vehicle>`

```
Set<Car> cars = new HashSet<Car>();
Vehicle myBike = new Bike();
cars.add(myBike); // type error
```

A `Vehicle` is not necessarily a `Car`!

Polymorphism

Polymorphism: we can switch between different concrete implementations of an interface without changing anything else in the program!

```java
interface List<E> {
  E get(int index);
  void add(int index, E e);
  int size();
}
```

```java
List<String> l;
l = // choose any List implementation
l.add(0, "hej");
l.add(1, " då");
if (l.size() >= 2)
  String s = l.get(0) + l.get(1);
  System.out.println(s);
```

Polymorphism: formal definition

If `S` is a subtype of `T`, an expression of type `S` can be used wherever *an expression of type `T` is expected.*

The class/interface `S` is a specialization of class/interface `T` (a `Convertible` is a `Car`!), so all types are still consistent.

Polymorphism

Advantages of using polymorphism:

Decoupling You can think about (and use!) an interface without worrying about the implementation.

Cohesion If you know how to use one implementation of `List`, you know how to use all of them.

Component-based design You can switch out one part of the code for another without changing the overall behaviour.

Polymorphism on client-side: example

```
class CreditCard {
  IBankAccount account;
  List<Transaction> transactions;

  void setPayments(IBankAccount ba) {
    account = ba;
  }

  void pay(int nt) {
     Transaction tr = transactions.get(nt);
     if (tr != null) {
        account.withdraw(tr.amount());
        transactions.remove(nt);
     }
  }
}
```

Live coding!

Let us design and implement a stack data structure.

What's next?

Next lecture: **Graphical interfaces & event-driven programming**.

To do:

- Read the book:
  - Today: chapter 9
  - Next lecture: chapter 10
- Hand in lab #5
- Start on lab #6