Programmieren II

Polymorphism

Alexander Fraser

fraser@cl.uni-heidelberg.de

(Based on material from T. Bögel)

June 4, 2014

Outline

1. Recap - Collections

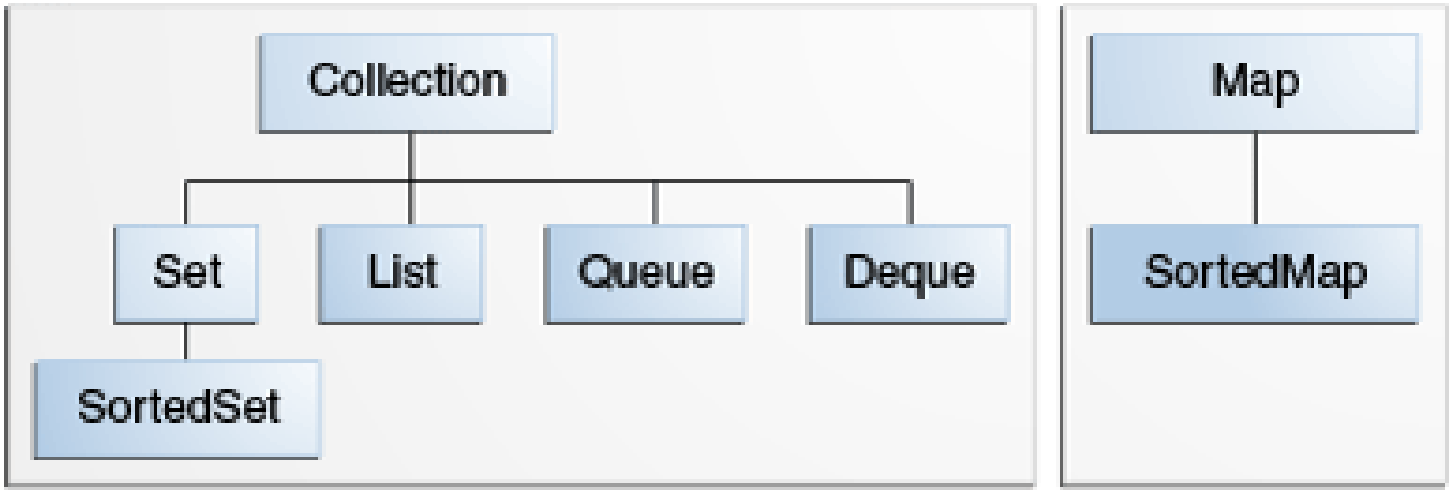
2. Advanced OOP: Polymorphism
   - Polymorphism
   - Interfaces
   - Abstract classes
   - Interfaces vs. Abstract Classes

Outline

Collections

- Containers for `Objects`
- Convenient for storing, accessing and manipulating elements
- Learn the Java Core Collections hierarchy by heart!
- You should know common *wrapper types* that capture primitive data types as objects

Core Collections hierarchy



source: `http://docs.oracle.com/javase/tutorial/collections/interfaces/index.html`

- `Set` is special kind of `Collection` etc.
- `Map` is separate from `Collection`
- **All** collection interfaces are **generic**: `ArrayList<E> list = new ArrayList<E>;`
- You should specify type of objects within a collection

Concrete implementations

- `List`:
  - `ArrayList` *(insertion-order, allows duplicates)*
  - `LinkedList` *(insertion-order, allows duplicates)*
- `Set`:
  - `HashSet` *(unordered, no duplicates)*
  - `LinkedHashSet` *(insertion-order, no duplicates)*
  - Later: `TreeSet` (ordered, no duplicates)
- `Map`:
  - `HashMap` *(unordered keys, no duplicate keys)*
  - `LinkedHashMap` *(insertion-ordered keys, no duplicate keys)*
  - Later: `TreeMap` (ordered keys, no duplicate keys)

Concrete implementations

- `List`:
  - `ArrayList` *(insertion-order, allows duplicates)*
  - `LinkedList` *(insertion-order, allows duplicates)*
- `Set`:
  - `HashSet` *(unordered, no duplicates)*
  - `LinkedHashSet` *(insertion-order, no duplicates)*
  - Later: `TreeSet` (ordered, no duplicates)
- `Map`:
  - `HashMap` *(unordered keys, no duplicate keys)*
  - `LinkedHashMap` *(insertion-ordered keys, no duplicate keys)*
  - Later: `TreeMap` (ordered keys, no duplicate keys)

Concrete implementations

- `List`:
  - `ArrayList` *(insertion-order, allows duplicates)*
  - `LinkedList` *(insertion-order, allows duplicates)*
- `Set`:
  - `HashSet` *(unordered, no duplicates)*
  - `LinkedHashSet` *(insertion-order, no duplicates)*
  - Later: `TreeSet` (ordered, no duplicates)
- `Map`:
  - `HashMap` *(unordered keys, no duplicate keys)*
  - `LinkedHashMap` *(insertion-ordered keys, no duplicate keys)*
  - Later: `TreeMap` (ordered keys, no duplicate keys)

Concrete implementations

- `List`:
  - `ArrayList` *(insertion-order, allows duplicates)*
  - `LinkedList` *(insertion-order, allows duplicates)*
- `Set`:
  - `HashSet` *(unordered, no duplicates)*
  - `LinkedHashSet` *(insertion-order, no duplicates)*
  - Later: `TreeSet` (ordered, no duplicates)
- `Map`:
  - `HashMap` *(unordered keys, no duplicate keys)*
  - `LinkedHashMap` *(insertion-ordered keys, no duplicate keys)*
  - Later: `TreeMap` (ordered keys, no duplicate keys)

Concrete implementations

- `List`:
  - `ArrayList` *(insertion-order, allows duplicates)*
  - `LinkedList` *(insertion-order, allows duplicates)*
- `Set`:
  - `HashSet` *(unordered, no duplicates)*
  - `LinkedHashSet` *(insertion-order, no duplicates)*
  - Later: `TreeSet` (ordered, no duplicates)
- `Map`:
  - `HashMap` *(unordered keys, no duplicate keys)*
  - `LinkedHashMap` *(insertion-ordered keys, no duplicate keys)*
  - Later: `TreeMap` (ordered keys, no duplicate keys)

Concrete implementations

- `List`:
  - `ArrayList` *(insertion-order, allows duplicates)*
  - `LinkedList` *(insertion-order, allows duplicates)*
- `Set`:
  - `HashSet` *(unordered, no duplicates)*
  - `LinkedHashSet` *(insertion-order, no duplicates)*
  - Later: `TreeSet` (ordered, no duplicates)
- `Map`:
  - `HashMap` *(unordered keys, no duplicate keys)*
  - `LinkedHashMap` *(insertion-ordered keys, no duplicate keys)*
  - Later: `TreeMap` (ordered keys, no duplicate keys)

Concrete implementations

- `List`:
  - `ArrayList` *(insertion-order, allows duplicates)*
  - `LinkedList` *(insertion-order, allows duplicates)*
- `Set`:
  - `HashSet` *(unordered, no duplicates)*
  - `LinkedHashSet` *(insertion-order, no duplicates)*
  - Later: `TreeSet` (ordered, no duplicates)
- `Map`:
  - `HashMap` *(unordered keys, no duplicate keys)*
  - `LinkedHashMap` *(insertion-ordered keys, no duplicate keys)*
  - Later: `TreeMap` (ordered keys, no duplicate keys)

Concrete implementations

- `List`:
  - `ArrayList` *(insertion-order, allows duplicates)*
  - `LinkedList` *(insertion-order, allows duplicates)*
- `Set`:
  - `HashSet` *(unordered, no duplicates)*
  - `LinkedHashSet` *(insertion-order, no duplicates)*
  - Later: `TreeSet` (ordered, no duplicates)
- Map:
  - `HashMap` *(unordered keys, no duplicate keys)*
  - `LinkedHashMap` *(insertion-ordered keys, no duplicate keys)*
  - Later: `TreeMap` (ordered keys, no duplicate keys)

Concrete implementations

- `List`:
  - `ArrayList` *(insertion-order, allows duplicates)*
  - `LinkedList` *(insertion-order, allows duplicates)*
- `Set`:
  - `HashSet` *(unordered, no duplicates)*
  - `LinkedHashSet` *(insertion-order, no duplicates)*
  - Later: `TreeSet` (ordered, no duplicates)
- `Map`:
  - `HashMap` *(unordered keys, no duplicate keys)*
  - `LinkedHashMap` *(insertion-ordered keys, no duplicate keys)*
  - Later: `TreeMap` (ordered keys, no duplicate keys)

Concrete implementations

- `List`:
  - `ArrayList` *(insertion-order, allows duplicates)*
  - `LinkedList` *(insertion-order, allows duplicates)*
- `Set`:
  - `HashSet` *(unordered, no duplicates)*
  - `LinkedHashSet` *(insertion-order, no duplicates)*
  - Later: `TreeSet` (ordered, no duplicates)
- `Map`:
  - `HashMap` *(unordered keys, no duplicate keys)*
  - `LinkedHashMap` *(insertion-ordered keys, no duplicate keys)*
  - Later: `TreeMap` (ordered keys, no duplicate keys)

Concrete implementations

- `List`:
  - `ArrayList` *(insertion-order, allows duplicates)*
  - `LinkedList` *(insertion-order, allows duplicates)*
- `Set`:
  - `HashSet` *(unordered, no duplicates)*
  - `LinkedHashSet` *(insertion-order, no duplicates)*
  - Later: `TreeSet` (ordered, no duplicates)
- `Map`:
  - `HashMap` *(unordered keys, no duplicate keys)*
  - `LinkedHashMap` *(insertion-ordered keys, no duplicate keys)*
  - Later: `TreeMap` (ordered keys, no duplicate keys)

Concrete implementations

- `List`:
  - `ArrayList` *(insertion-order, allows duplicates)*
  - `LinkedList` *(insertion-order, allows duplicates)*
- `Set`:
  - `HashSet` *(unordered, no duplicates)*
  - `LinkedHashSet` *(insertion-order, no duplicates)*
  - Later: `TreeSet` (ordered, no duplicates)
- `Map`:
  - `HashMap` *(unordered keys, no duplicate keys)*
  - `LinkedHashMap` *(insertion-ordered keys, no duplicate keys)*
  - Later: `TreeMap` (ordered keys, no duplicate keys)

Concrete implementations

- `List`:
  - `ArrayList` *(insertion-order, allows duplicates)*
  - `LinkedList` *(insertion-order, allows duplicates)*
- `Set`:
  - `HashSet` *(unordered, no duplicates)*
  - `LinkedHashSet` *(insertion-order, no duplicates)*
  - Later: `TreeSet` (ordered, no duplicates)
- `Map`:
  - `HashMap` *(unordered keys, no duplicate keys)*
  - `LinkedHashMap` *(insertion-ordered keys, no duplicate keys)*
  - Later: `TreeMap` (ordered keys, no duplicate keys)

Description of core collection interfaces

Collection

- Root of collection hierarchy
- Collection contains *elements*

Methods for each `Collection`

- `size(): int` Number of elements in a collection
- `isEmpty(): boolean`
- `contains(Object element): boolean`
- `add(E element): boolean`
- `remove(Object element): boolean`
- `iterator(): Iterator<E>`
- `toArray(): Object[]`
- `equals(): boolean`

Collection bulk operations

- Methods that operate on *entire* collection

Methods

- `containsAll`: does the collection contain all elements specified in another collection?
- `addAll`: add all elements of one collection to another collection
- `removeAll`: remove all elements that are elements of a second collection
- `clear`: removes all elements from the collection

Additional collections

Interface	**implementation/description**
`List`	`Stack` (LIFO)
`Queue`	holding elements prior to processing
`Dequeue`	supports element insertion and removal at both ends

Outline

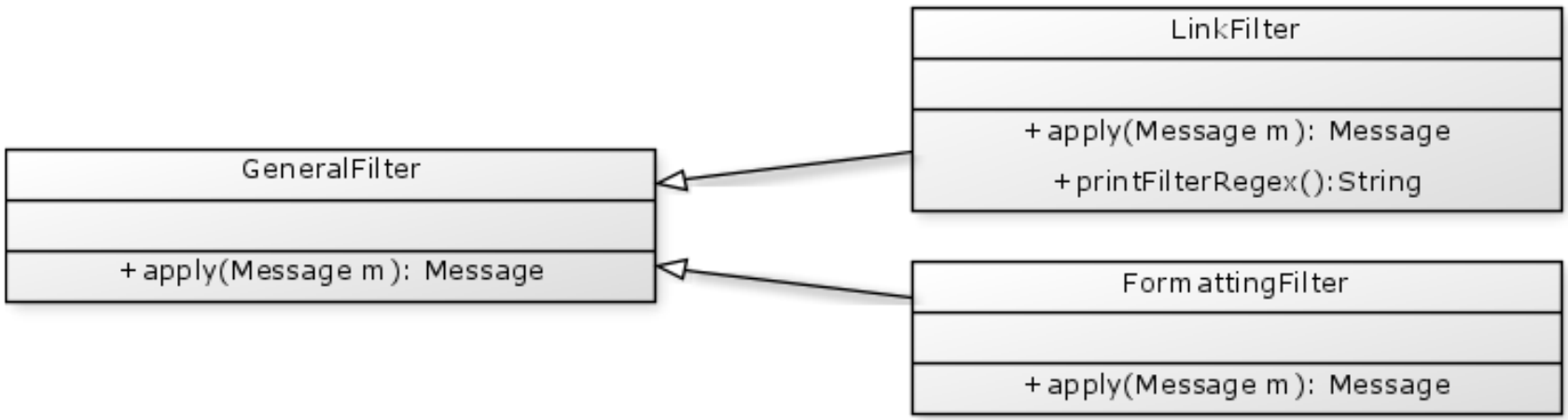
Outline

Motivation: Polymorphism

- Imagine we need to filter Twitter messages
- We define a Message class (think of this as a string for now)
- Then we can define different types of filters which operate on messages
  - We will chain these filters (apply them in a sequence)

Example: Filter

Each Filter filters...



- Sub classes *inherit* methods of super class
- This allows for **Polymorphism**

Polymorphism I

- Objects of a concrete *sub class* can be used where *super classes* are expected
- All sub classes have complete functionality of super class
- **But:** special functionality implemented in the sub class cannot be accessed via super class

Polymorphism II

Example

```
public Message filterMessage(Message m, GeneralFilter f) {
        f.apply(m);
        // f.printFilterRegex() would not work
}
...
public void runFiltering(Message m) {
        LinkFilter f = new LinkFilter();
        this.filterMessage(m,f);
}
```

- `filterMessage()` expects `GeneralFilter`
- `LinkFilter` **is** also a `GeneralFilter`
- Each sub class of `GeneralFilter` has a `apply()` method
- `filterMessage()` does not need to know which filter's method it is calling!

Polymorphism: Arrays and Collections I

Polymorphic Collections

```
List<GeneralFilter> filters = new ArrayList<GeneralFilter>();
filters.add(new LinkFilter());
filters.add(new GeneralFilter());
```

- Each sub-class of `GeneralFilter` is also a `GeneralFilter`
- Collections can be filled with sub-classes
- We can only access methods in `GeneralFilter`
- `List` → `ArrayList` itself is polymorphic

Polymorphism: Arrays and Collections II

Polymorphic Arrays

```
GeneralFilter[] filters = new GeneralFilter[2];
filters[0] = new LinkFilter();
filters[1] = new GeneralFilter();
```

- Each sub-class of `GeneralFilter` is also a `GeneralFilter`
- Arrays can be filled with sub-classes

Polymorphism: Arrays and Collections III

Accessing array elements

```
GeneralFilter f = filters[0].filter(m);
```

- For each element of the array, **all functionality of the super class** can be used

Polymorphic Methods

- The return type of a method can also be polymorphic

```
public GeneralFilter returnFilter() {
    FormattingFilter f = new FormattingFilter();
    return f;
}
```

- Method returns `GeneralFilter`
- Each sub-class of `GeneralFilter` is also a `GeneralFilter`

Polymorphism

Polymorphism

- Using Polymorphism allows extension of code
- New sub-classes do not require changes in the client code
- But: if a class overrides inherited methods, which method is called?
- Answer: the **most specific** one is called

Polymorphism

Polymorphism

- Using Polymorphism allows extension of code
- New sub-classes do not require changes in the client code
- But: if a class overrides inherited methods, which method is called?
- Answer: the **most specific** one is called

Dynamic Method Lookup

Dynamic Method Lookup

- If a class overrides inherited methods, which method is called?
- polymorphic classes should keep their *specific properties*, even if they seem to be objects of the super class
- Decision which method to call is made during runtime
- The fact that the method is looked up at runtime is called Dynamic Method Lookup

Outline

Motivation for Interfaces I

- Multiple developers: need for programming *contract*
- People should be able to write code independently
- Knowledge about behavior of classes should be known early and without knowledge about implementation

Example

- Future: automatic driving
- Automobile manufacturers write software to operate an automobile
- GPS company writes code to use GPS to drive the car
- Manufacturer needs to explicitly state specification about car operation
- Which methods+parameters does a car have (that can be used by the GPS company)?
- GPS company not interested *how* operation methods are implemented

Interfaces in Java

- Interface: reference type (similar to classes)
- Specifies **only** *constants* and *method signatures*
- Does **not** contain method bodies
- Interfaces cannot be instantiated (i.e. no `new Interface`)
- Interfaces can be *extended*
- Using an interface: `implements` keyword

Interface for an automobile

```
public interface OperateCar {
    // constant declarations, if any
    // method signatures
    int turn(Direction direction,
             double radius,
             double startSpeed,
             double endSpeed);
    int changeLanes(Direction direction,
                    double startSpeed,
                    double endSpeed);
    int signalTurn(Direction direction,
                   boolean signalOn);
    // more method signatures
}
```

Concrete car implementation I

```
public class OperateBMW760i implements OperateCar {

    // the OperateCar method signatures, with implementation --
    // for example:
    int signalTurn(Direction direction, boolean signalOn) {
        // code to turn BMW's LEFT turn indicator lights on
        // code to turn BMW's LEFT turn indicator lights off
        // code to turn BMW's RIGHT turn indicator lights on
        // code to turn BMW's RIGHT turn indicator lights off
    }

    // other members, as needed -- for example, helper classes
        not
    // visible to clients of the interface
}
```

Concrete car implementation II

Implementing an interface

- `OperateBMW760i implements` the `OperateCar` interface
- All methods specified in the interface **need to be** implemented
- Each car manufacturer can individually implement all methods
- GPS company receives concrete implementation of different companies
- GPS company is able to invoke `OperateCar` methods without knowing about their implementation
- By implementing an interface, you specify that your class has certain **functionality**

General Interface definition

```
public interface Interface extends Interface1, Interface2,
    Interface3 {

    // constant declarations

    // base of natural logarithms
    double E = 2.718282;

    // method signatures
    void doSomething (int i, double x);
    int doSomethingElse(String s);
}
```

- Interfaces can extend multiple interfaces
- All methods in an interface are public
- All constant values are public, static and final

Interfaces as types

Using interfaces as types

- Interface is a reference type
- Interface name can be used just like any other data type
- Reference variable with interface type must **always point to instance that implements interface**

Example: Relatable interface

- Interface that provides a method to determine the size of two Relatable interfaces
- Example: Rectangle implements Relatable

```
public interface Relatable {
    // this (object calling isLargerThan)
    // and other must be instances of
    // the same class returns 1, 0, -1
    // if this is greater than, equal
    // to, or less than other
    public int isLargerThan(Relatable other);
}
```

Example: Interfaces as types I

- Goal: find largest object in a pair of objects
- Works for *any* objects that implement `Relatable`

```
public Object findLargest(Object object1, Object object2) {
   Relatable obj1 = (Relatable)object1;
   Relatable obj2 = (Relatable)object2;
   if (obj1.isLargerThan(obj2) > 0)
      return object1;
   else
      return object2;
}
```

- `Object` is casted to `Relatable` $\rightarrow$
- `isLargerThan` can be called
- Concrete implementation is irrelevant

Example: Interfaces as types II

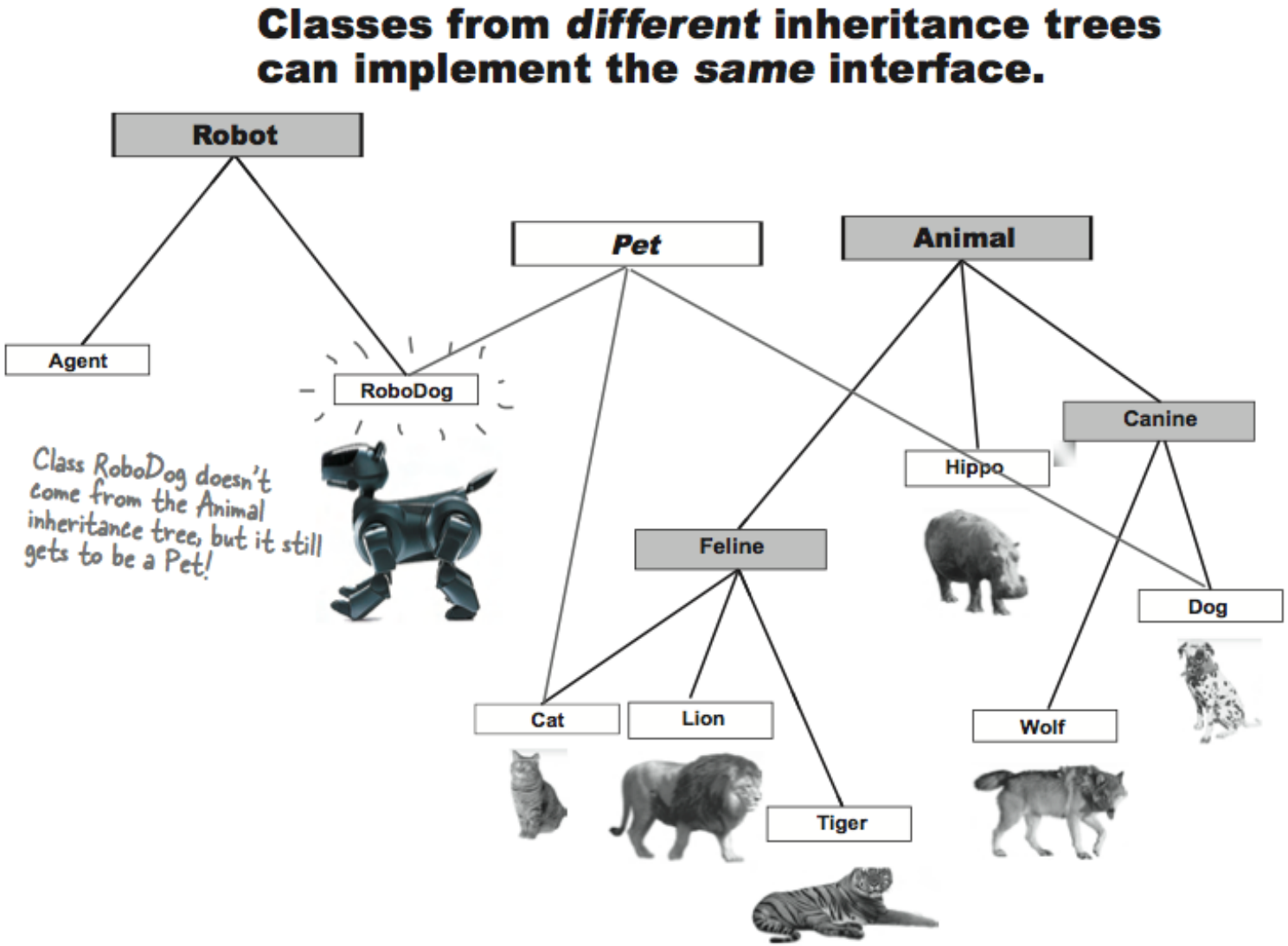
- Similar to multiple inheritance: each object implementing `Relatable` is simultaneously an `Object` and `Relatable`

Checking the type of a class

`isInstance`

- Access to the underlying `class` for an instance: `.class` property
- `Class` is an object
- Method `.isInstance(Object o)` checks whether an object is an instance of the class
- `Class` provides other useful methods that allow querying information about an object's class

Substitution for multiple inheritance



source: HFJ, p. 226

How you implement the `Relatable` interface

- This depends on the relationship of your classes
- For instance, suppose we can only compare the size of Rectangles.

Implementing the Relatable interface

```
public int isLargerThan(Relatable other) {
    Rectangle otherRect
        = (Rectangle)other;
    if (this.getArea() < otherRect.getArea())
        return -1;
    else if (this.getArea() > otherRect.getArea())
        return 1;
    else
        return 0;
}
```

- This works fine for Squares
- It throws a casting exception if other is a non-rectangle
- How should we change this so we can also compare with, e.g., Circles?

Summary

Summary: Interfaces

- Interfaces define protocols for communication between objects
- Interface declarations only contain method signatures & constants, no implementation
- A class implementing an interface must implement all of its methods
- Interfaces can be used just like other (reference) types

Outline

Abstract classes

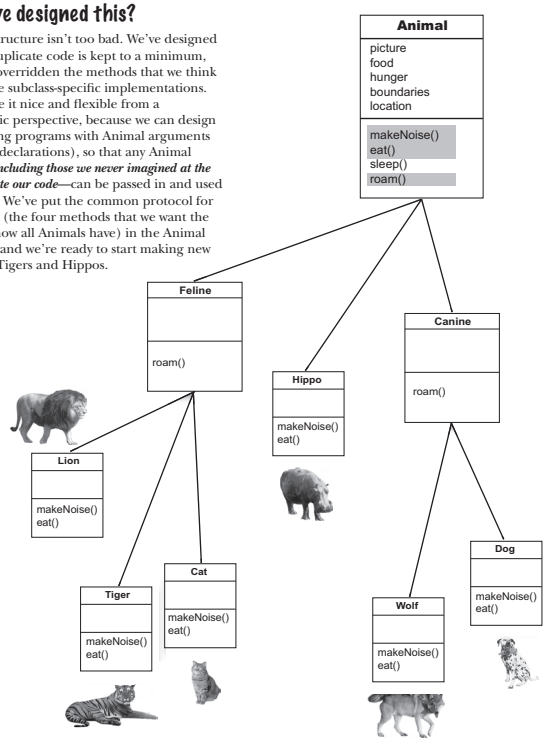
Motivation

- Super classes represent an *abstraction* of sub classes
- Sometimes, however, instantiating the super class does not make sense
- Examples:
  - Animal
  - Shape
  - Person
- University library software knows about two kinds of `Persons`: `Student` and `Teacher`
- Instantiating `Person` would be strange

Motivation: HFJ, pp. 198–199

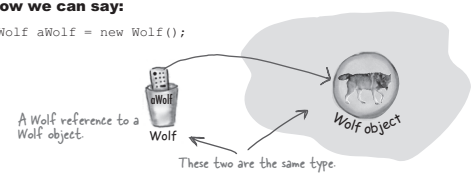
Did we forget about something when we designed this?

The class structure isn't too bad. We've designed it so that duplicate code is kept to a minimum, and we've overridden the methods that we think should have subclass-specific implementations. We've made it nice and flexible from a polymorphic perspective, because we can design Animal-using programs with Animal arguments (and array declarations), so that any Animal subtype—***including those we never imagined at the time we wrote our code***—can be passed in and used at runtime. We've put the common protocol for all Animals (the four methods that we want the world to know all Animals have) in the Animal superclass, and we're ready to start making new Lions and Tigers and Hippos.



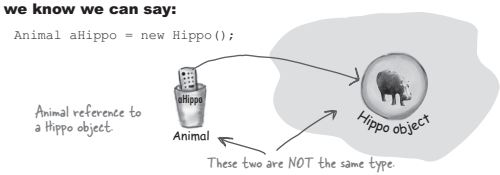
We know we can say:

```
Wolf aWolf = new Wolf();
```



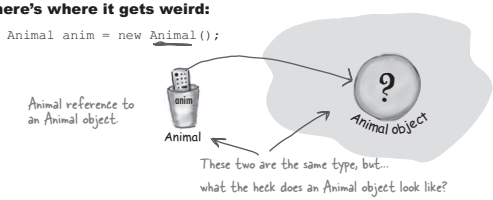
And we know we can say:

```
Animal aHippo = new Hippo();
```



But here's where it gets weird:

```
Animal anim = new Animal();
```



Abstract classes I

Abstract classes

- Keyword `abstract`
- Declare commonalities in super class, enforce implementation in sub-class
- `abstract` classes cannot be instantiated (`new`)
- Sub-class needs to implement (i.e. override) all **abstract** methods of the abstract class
- . . . **or** it needs to be `abstract` itself

Abstract classes

Rules for abstract classes

- Each class with `abstract` methods needs to be `abstract`
- `abstract` classes cannot be instantiated (`new`)!
- `abstract` class can contain `abstract` methods and implemented methods
- Methods that are `private`, `static` or `final` cannot be `abstract` as they can't be overridden

Example: GeometricShape

```
public abstract class GeometricShape {
    public abstract double getArea();
    ...
}
```

- GeometricShapes provide a `getArea()` method (implementation hidden)
- Concrete implementation in `GeometricShape` not possible in this case
- `abstract` methods $\rightarrow$ `abstract` class
- `abstract` methods define signature only

Concrete implementation

```
public class Circle extends GeometricShape {
    public static final double PI = 3.1415926536;
    private double r;
    public Circle( double r ) { this.r = r; }
    public double getArea() { return PI*r*r; }
    ...
}

public class Rectangle extends GeometricShape {
   ...
}
```

- $\Rightarrow$ Concrete implementation of abstract super class
- $\Rightarrow$ Implementation of abstract methods in `GeometricShape`
- $\Rightarrow$ Additional elements (specific to `Circle`)

Abstract classes and Polymorphism

- `new GeometricShape()` is not allowed
- Nevertheless, `GeometricShape` can be used as reference type (**polymorphism**)
- `GeometricShape` can be used just like any other data type

Example

```
GeometricShape s = new Circle( 1.0 );
GeometricShape[] shapeArray = new GeometricShape[1];
shapeArray[0] = s;
```

Outline

Abstract classes vs. Interfaces

Flexibility vs. reusability

- Interfaces allow more *flexibility* by multiple inheritance
- But: code duplication very likely if multiple classes implement the same interface
- Abstract class: possibility to partially implement common methods

Abstract classes vs. Interfaces

Compatibility

- Adding new methods to an interface: all implementing classes need to be changed
- Abstract class can also add non-abstract methods that are automatically inherited by sub-classes

Combining Interfaces and abstract classes

Combination of Interfaces & abstract classes

- Usually: Interface + implementing abstract class (skeleton implementation)
- Concrete class can implement interface or extend abstract skeleton class
- Example: Java Collections

Literature


Sierra, K. & Bates, B.
Head First Java. (end of Chapter 7, Chapter 8)
O'Reilly Media, 2005.


Ullenboom, Ch.
Java ist auch eine Insel. (Sections 5.11, 5.12 & 5.13)
Galileo Computing, 2012.


The Java tutorials
`http://docs.oracle.com/javase/tutorial/java/concepts`


Eckel, B. (For Reference)
Thinking in Java. (Ch. 7 & 8)
Prentice Hall, 2006.