Programming Paradigms
159.272
Inheritance and Polymorphism

Amjed Tahir
a.tahir@massey.ac.nz

Original author: Jens Dietrich

Readings

1. Java Tutorial, trail: Learning the Java Language
   http://docs.oracle.com/javase/tutorial/java/index.html
   Lesson Interfaces and Inheritance

Overview

- inheritance
- overriding methods
- references to super
- abstract classes
- interfaces
- polymorphism
- method lookup and dispatch

The Evils of Duplications

- programs require constant **maintenance**
- this begins during development - code is continuously changed as the understanding of the program increases
- once the software is deployed, there is continuous **change pressure**: bugfixes, adding features, adapting to changing environments (OS and VM versions, regulations, other systems, etc)

The Evils of Duplications ctd

- copy & paste (c&p) programming makes it easy to duplicate code quickly (i.e., save time)
- if code changes, all of its copies must be detected and must be consistently changed as well

- **this makes maintenance very expensive and error-prone (i.e., possible to increase errors)**

Duplication in Code

```
public class Student {
        public Date dob = null;
        public String name = null;
        public String firstName = null;
        public Address address = null;
    public Course course = null;
     public int studentId = null;
     public String getFullName() {
         return firstName + " " +name;
     }
}
```

```
public class StaffMember {
        public Date dob = null;
        public String name = null;
        public String firstName = null;
        public Address address = null;
    public Department department = null;
    public int staffId = null;
    public String getFullName() {
        return firstName + " " +name;
    }
}
```

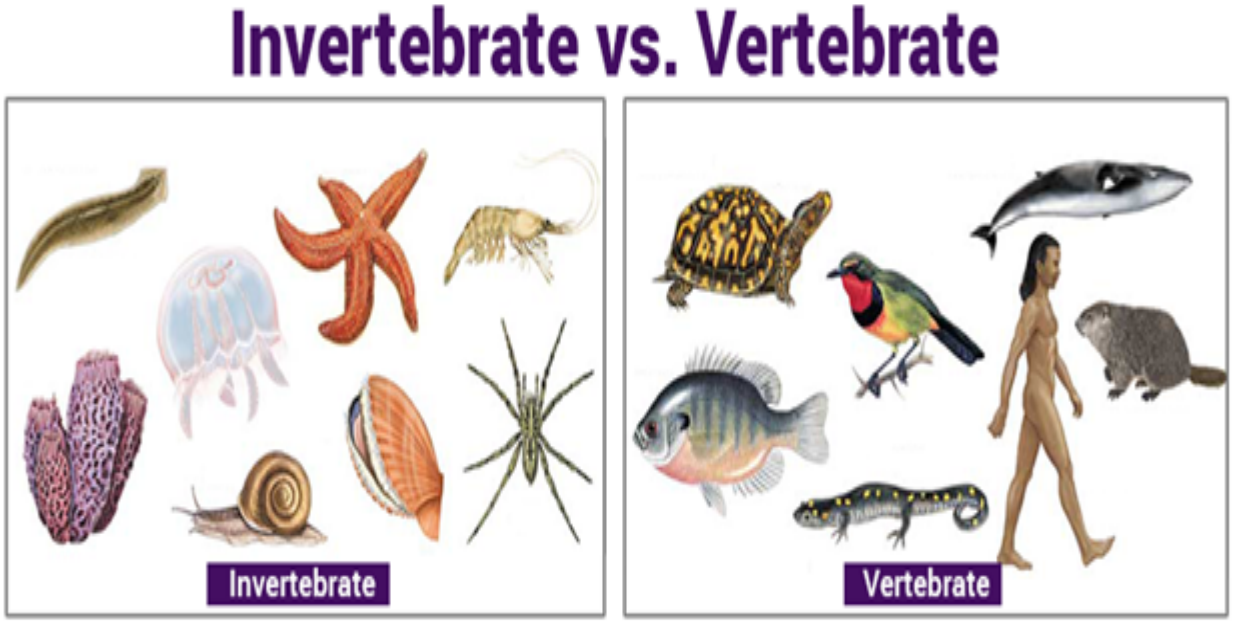
duplicated state

duplicated behaviour

Taxonomies

- the (English) language already has a solution for this
- when concepts are organised in taxonomies, **common** properties of types are grouped together in more general types
- in the `Student`/`StaffMember` example, a common term `Person` would be used to describe what is common
- the relation between `Student` and `Person` is **"isA"**, i.e., each (instance of) Student **is also** an (instance of) Person

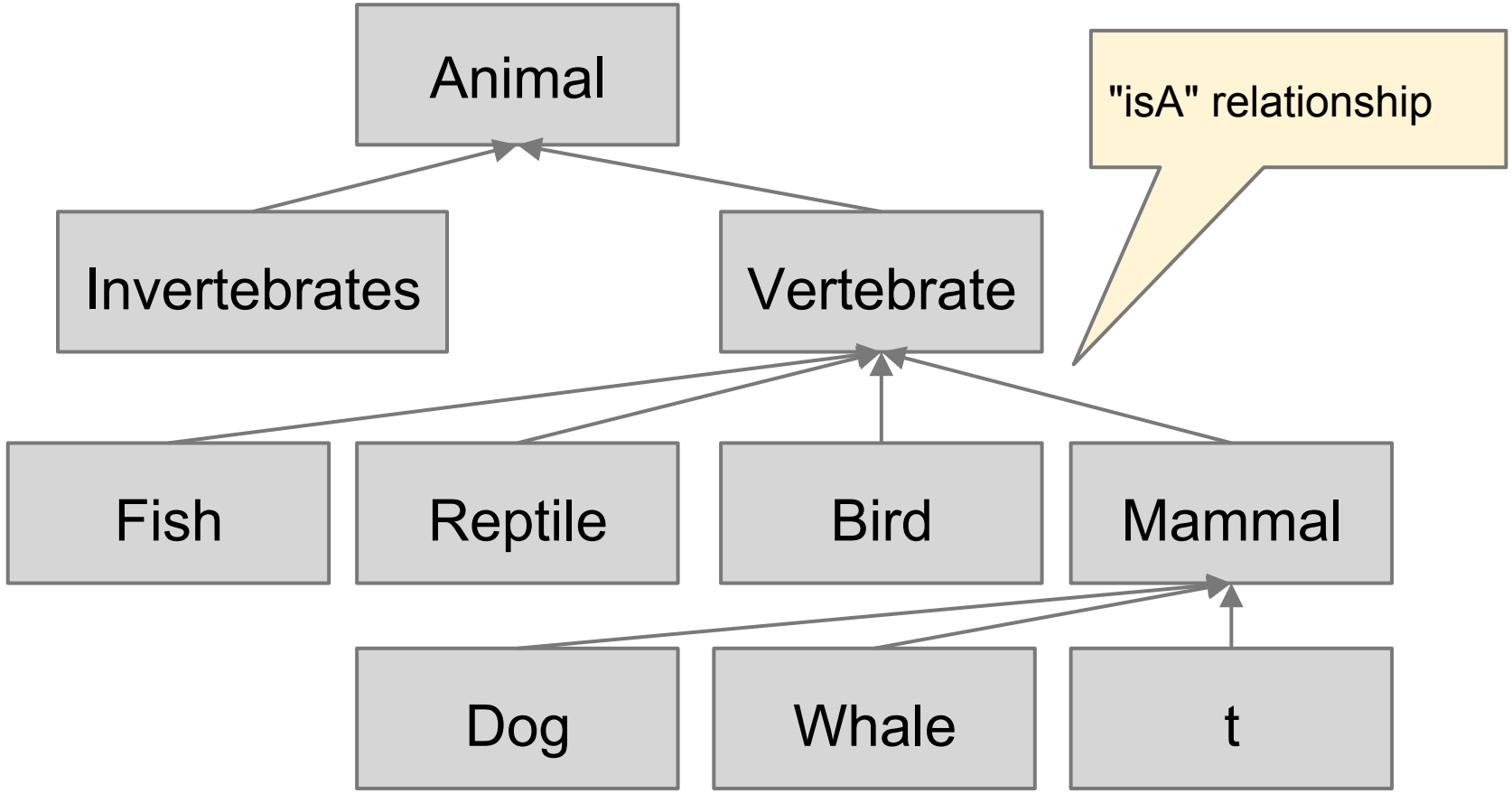
Taxonomy used in Biology



Don’t have a spinal column or a backbone

have a spinal column or a backbone

Taxonomy used in Biology



The Idea of Inheritance

```
public class Person {
        public Date dob = null;
        public String name = null;
        public String firstName = null;
        public Address address = null;

    public String getFullName() {
        return firstName + " " +name;
    }
}
```

common features are defined here

"isA" relationship

```
public class StaffMember {
    public Department department = null;
    public int staffId = null;
}
```

```
public class Student {
    public Course course = null;
    public int studentId = null;
}
```

Inheritance in Java

```
public class Student extends Person  {
    public Course course = null;
    public int studentId = null;
}
```

declare a class to be a **subclass** of another class

- this is called **inheritance**
- classes **inherit** features (state and behaviour) from other classes
- `Student` is a (direct) **subclass** of `Person`
- `Person` is the (direct) **superclass** of `Student`
- inheritance is **transitive**: `Student` also (indirectly) inherits from the superclass of `Person`

Single vs Multiple Inheritance

- **single inheritance**: classes can have only one direct superclass
- **multiple inheritance**: classes can have multiple direct superclasses
- Java supports only single inheritance
- this results in safer, more predictable code
- C++ is an example of a language that supports multiple inheritance

The Problem with Multiple Inheritance

```
public class B {
        public void whoAmI() {

        System.out.println("B");
    }
}
```

```
public class C {
        public void whoAmI() {

        System.out.println("C");
    }
}
```

remember: this is **not** supported in Java !

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

What happens when `new A().whoAmI()` is invoked ?

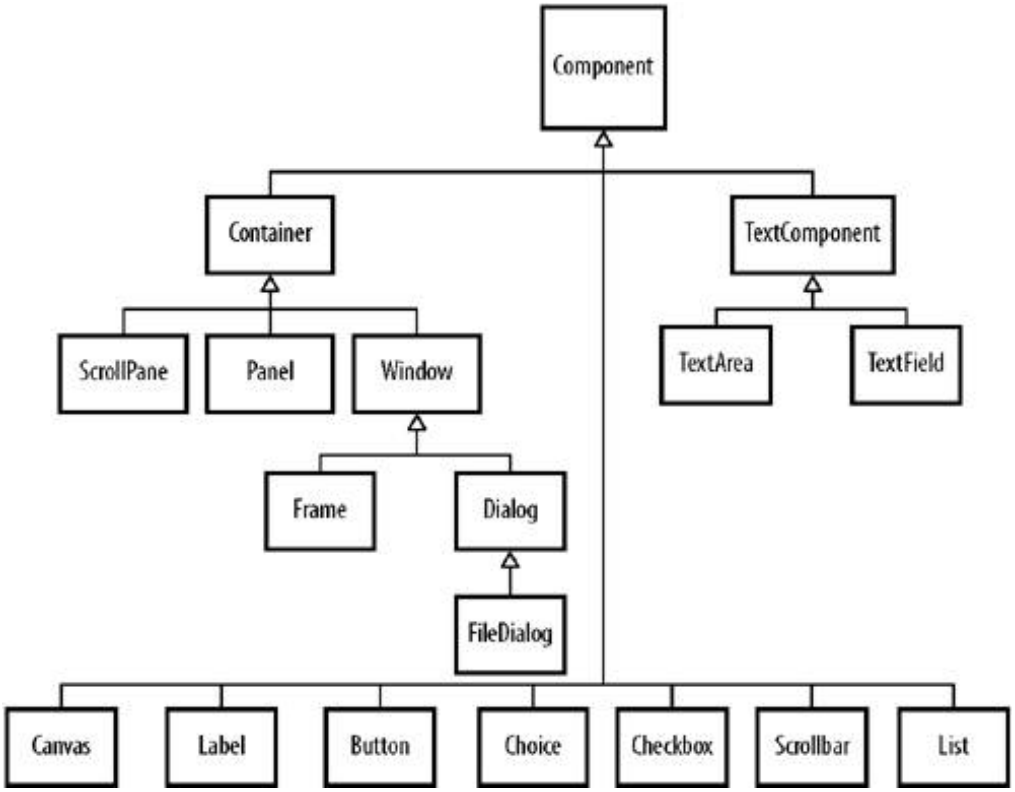
The Problem with Multiple Inheritance

- there are now two **conflicting** implementations of `whoAmI()` to chose from
- simple strategy: first one declared wins (i.e., use `whoAmI()` implemented in `B`, because `B` comes first in the `extends` clause)
- this is less obvious if we indirectly inherit `whoAmI()`, and there is a large number of superclasses
- it then becomes difficult to predict which version of `whoAmI()` is executed!

The Inheritance Tree

- all classes inherit from the **root class** `java.lang.Object`
- all classes except `java.lang.Object` have exactly one super class
- if the super class is not declared in the definition of the class, then `java.lang.Object` is the super class
- this implies that classes form a large tree - the class hierarchy, with `java.lang.Object` at the top

Example: AWT Inheritance Tree



- inheritance tree of classes in the `java.awt` package (incomplete)
- "top" is missing: `Component` is a subclass of `Object`
- see also: http://docs.oracle.com/javase/1.4.2/docs/api/java/awt/package-tree.html

Inheritance and Type Rules 1

```
Person person = new Student();
```

- assume that `Student` is a subclass of `Person`

Inheritance and Type Rules 1

```
Person person = new Student();
```

- this is **ok** (= accepted by the compiler)
- an instance of `Student` is also an instance of `Person`
- note that now we have declared a variable with two types:
  - the **declared type**: `Person`
  - the **actual type** used for instantiation: `Student`

Inheritance and Type Rules 2

```
Student student = new Person();
```

- assume that `Student` is a subclass of `Person`

Inheritance and Type Rules 2

```
Student student = new Person();
```

- the compiler will **reject** this
- not every person is a student!
- i.e., more general types cannot be used to instantiate their subtypes

Inheritance and Type Rules 3

```
Person person = new Student();
Student student = person;
```

- assume that `Student` is a subclass of `Person`

Inheritance and Type Rules 3

```
Person person = new Student();
Student student = person;
```

- the first assignment is ok (see rule 1)
- the second assignment is rejected by the compiler because person could also be a non-student `StaffMember` (see rule 2)
- but: **we** know that person is actually a student
- can we tell the compiler to accept this anyway?

Casts

```
Person person = new Student();
Student student = (Student) person;
```

Casts

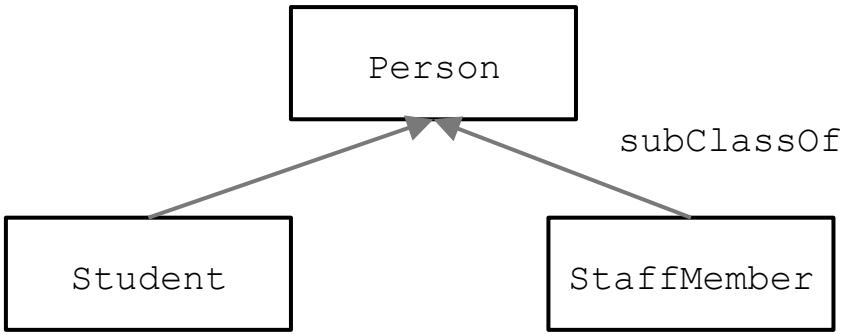
```
Person person = new Student();
Student student = (Student) person;
```

- if the programmer knows that the actual (not the declared) type of an object, a (down-)**cast** can be used
- a cast converts a reference to an instance of a type (`Person`) to a reference to an instance of a subtype (`Student`)
- the compiler will accept this

Cast Example 1

```
Person person = new StaffMember();
Student student = (Student) person;
```

- assume that `Student` and `StaffMember` are both subclasses of `Person`



Cast Example 1

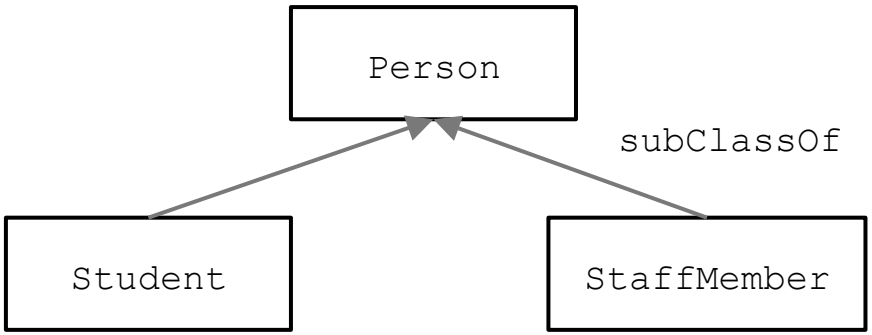
```
Person person = new StaffMember();
Student student = (Student) person;
```

- from the compilers point of view, this is ok
- but a `StaffMember` is **not** a kind of `Student`
- the cast is incorrect: the programmer got it wrong!
- this will therefore create a **runtime problem**: a special exception (`java.lang.ClassCastException`) is created, and if not handled, the program will crash

Cast Example 2

```
StaffMember person = new StaffMember();
Student student = (Student) person;
```

- assume that `Student` and `StaffMember` are **both** subclasses of `Person`



Cast Example 2

```
StaffMember person = new StaffMember();
Student student = (Student) person;
```

- the compiler will **reject** this
- a `StaffMember` can never be a `Student`
- note that the compiler only reasons based on the compatibility of declared types, not of actual types

Guarding Casts

```
Person person = ..
if (person instanceof Student) {
   Student student = (Student) person;
   ..
}
```

- it is easy to write guard conditions to prevent class cast exceptions
- there are several ways to do this, the easiest way is to use the built-in `instanceof` operator
- with this operator, the **actual** type of an object can be checked

Overriding Methods

- classes can replace methods inherited from superclasses
- this is called **overriding**
- exception: final methods cannot be overridden

Overriding Methods Example 1

```
public class Mammal {
        public boolean hasStripes() {return false;}
}
public class Zebra extends Mammal {
        public boolean hasStripes() {return true;}
}
...
Zebra zebra = new Zebra();
System.out.println(zebra.hasStripes());
```

Overriding Methods Example 1

```
public class Mammal {
	public boolean hasStripes() {return false;}
}
public class Zebra extends Mammal {
	public boolean hasStripes() {return true;}
}
...
Zebra zebra = new Zebra();
System.out.println(zebra.hasStripes());
```

- this prints **true**
- the implementation in `Zebra` is used
- why?
- is it because `Zebra` is the declared or the actual type?

Overriding Methods Example 2

```
public class Mammal {
        public boolean hasStripes() {return false;}
}
public class Zebra extends Mammal {
        public boolean hasStripes() {return true;}
}
...
Mammal zebra = new Zebra();
System.out.println(zebra.hasStripes());
```

Overriding Methods Example 2

```
public class Mammal {
        public boolean hasStripes() {return false;}
}
public class Zebra extends Mammal {
        public boolean hasStripes() {return true;}
}
...
Mammal zebra = new Zebra();
System.out.println(zebra.hasStripes());
```

- this prints still **true**
- this means: the actual type matters
- this is called **dynamic method lookup**

Dynamic Method Lookup and Dispatch

- in dynamic method lookup, the **actual type of the callee** is used to decide which method to invoke
- this means that the decision is made at **runtime**, not by the compiler
- this is also called **single dispatch**
- some older languages (Delphi) use static lookup based on the declaration type - this results in counterintuitive behaviour

Multiple Dispatch

- but note that for parameter types, the declared type is used
- that means that Java does not support **multiple dispatch**
- this is outside the scope of this paper, an example can be found here: https://bitbucket.org/jensdietrich/oop-examples/src/e13dc6b3212fca04088b6b3012f503848e4383a8/dispatch/src/nz/ac/massey/cs/asdc/?at=master

Overloading Methods

```
public class Graphic {
       // set the colour using an instance of the colour class
       public void setColour(java.awt.Color c) {..}
       // set the colour using RGB values
       public void setColour(int red, int green, int blue) {..}
       // set the colour using a string-encoded rgb colour code
    public void setColour(String rgbCode) {..}
}
```

- methods can be **overloaded**
- that is, a class can define different methods with the same name, but different (sets of) parameter types

The @Override Annotation

```
public class Mammal {
	public boolean hasStripes() {return false;}
}

public class Zebra extends Mammal {
	@Override
	public boolean hasStripes() {return true;}
}
```

- a method overriding another method can be tagged with `@Override`
- this is an **annotation** - a Java construct used to add metadata to classes, methods or fields
- the compiler will check whether this method is actually overriding another method, and create an error if not

Commonly Overridden Methods

- most methods inherited from `Object` are usually overridden
- `String toString()` - used to convert objects to strings, these strings are used when objects are printed to the console (`System.out.println()`), or when a debugger is used
- `boolean equals(Object)` - used to compare this object with other objects
- `int hashCode()` - computes a number that can be used to index objects - this is used in some data structures (to be discussed later)

Extending Methods: The super Reference

- so far there are two options to deal with inherited methods:
- use (inherit) them
- discard them completely and replace them by a new implementation (overriding)
- often, only minor modifications (in particular additions) are required

Example: Implementing equals

```
public class Student extends Person {
   public int studentId = 0;
   ..
   public boolean equals(Object obj) {
      boolean same = super.equals(obj);
      if (!same) return false;

      if (obj instanceof Student) {
            Student s = (Student)obj;
            return this.studentId==s.studentId;
      }
      return false;
   }
}
```

this compares name, firstName etc - the state defined in Person!
"the Person part of Student"

then the "Student-specific part" is compared,
note that the results is false if the other object is not a Student ("apples and oranges"

Extending Methods: The super Reference ctd

- note that `super` **does not** reference another object - it still references the callee
- i.e., `super` and `this` reference the same object
- but with `super`, the method lookup (dispatch) is different
- `this` - use the **first** implementation of a method found when walking up the inheritance hierarchy
- `super` - use the **second** implementation of a method found when walking up the inheritance hierarchy

Polymorphism

- overriding methods is an example of **polymorphism** in Java
- i.e., the behaviour of a particular method depends on the actual callee
- example: all objects understand `toString()`, but what this actually does depends on the actual object

Abstract Methods

- in the previous example, the class `Mammal` implemented `hasStripes()` as follows:
  ```
  public boolean hasStripes() {return false;}
  ```
- i.e., in this class a useful **default behaviour** is implemented: most mammals don't have stripes
- there is another point of view to model this: we know that this behaviour is there (`hasStripes()` exists) but we cannot make any further assumptions about this behaviour
- it is possible to implement this strategy by making methods abstract:
  ```
  public abstract boolean hasStripes();
  ```

Abstract Classes

- abstract methods do not have method bodies (implementations), they are declarations only
- this is not a problem as long as they are only used in types used for declaration, but not in types actually instantiated
- this kind of class (to be used for declaration only) is called an **abstract class**, and is defined using the `abstract` keyword
- abstract classes cannot be instantiated (but their subclasses can)
- only abstract classes can have abstract methods

Abstract Class Example

```
public abstract class Mammal {
       public abstract boolean hasStripes();
}
public class Zebra extends Mammal {
    @override
       public boolean hasStripes() {return true;}
}
...
Mammal mammal = ...;
mammal.hasStripes();
```

- this works, even though `Mammal` does not define (but it declares) `hasStripes()`
- `mammal` can only be instantiated with some non-abstract subclass of `Mammal` that does define `hasStripes()`
- there are no "Mammals as such"

Abstract Class Example ctd

```
public abstract class Mammal {
       public abstract boolean hasStripes();
}
public class Zebra extends Mammal {
    @override
       public boolean hasStripes() {return true;}
}
...
Mammal mammal = new Zebra();
mammal.hasStripes();
```

in `Zebra`, `hasStripes()` is defined !

- `mammal` can only be instantiated with some non-abstract subclass of `Mammal` that does define `hasStripes()`

Interfaces

- abstract classes are usually a combination of methods that are only declared and methods that are implemented
- but abstract classes are still classes and only support single inheritance
- note that we could get rid of the problems of multiple inheritance if we had only "**pure abstract classes**" without any implemented methods
- this is possible in Java, and it is called **interfaces**

Interfaces cnt'd

- An interface is a reference type (similar to a class)
- collection of abstract methods
- a typical class will implement an interface, and inherits all the abstract methods of this interface.
- Things to know:
  - You cannot instantiate an interface
  - All methods MUST be abstract
  - Cannot contain instance fields (fields can be declared only as **static** and **final**)

Example: Sorting Arrays

- sorting arrays of objects has two aspects:
  - which sort algorithm to use? by default, merge sort is used
  - how to compare objects
- comparing strings:
  - obvious, use alphabetical order
- comparing student instances: less obvious, could try:
  - by name
  - by id
  - by first name, then name
  - by year or enrollments, then major etc
- how can we define a sorting function that can **abstract** from all these options ?

java.lang.Comparable

```
interface Comparable {
   int compareTo(Object o);
}
```

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
The implementor must ensure sgn(x.compareTo(y)) == -sgn(y.compareTo(x)) for all x and y. (This implies thatx.compareTo(y) must throw an exception iff y.compareTo(x) throws an exception.)
The implementor must also ensure that the relation is transitive: (x.compareTo(y)>0 && y.compareTo(z)>0) implies x.compareTo(z)>0.

from
http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html

Example: Sorting Arrays 1

```
public class Student  {
        public int id = 0;
        public String firstName = "";
        public String lastName = "";
        public Student(int id, String firstName, String lastName) {
                super();
                this.id = id;
                this.firstName = firstName;
                this.lastName = lastName;
        }
        @Override public String toString() {
        return "Student [id=" + id + ", firstName=" + firstName
            + ", lastName=" + lastName + "]";
        }
}
```

implements `toString()` for readable console output

Example: Sorting Arrays 1

```
Student[] students = {
    new Student(10101,"James","Smith"),
        new Student(10102,"James","Taylor"),
        new Student(11221,"Albert","Taylor"),
        new Student(12395,"Ronald","MacDonald"),
        new Student(12396,"Sarah","Anderson")
};
Arrays.sort(students);
for (Student student:students) {
        System.out.println(student);
}
```

in this method it is expected that the objects in the array know how to compare themselves to other objects

- creates a `ClassCastException`
- `Arrays.sort` does not know how to compares students, tries to cast students to `Comparable`

Example: Sorting Arrays 2

```
public class Student implements Comparable {
        ..
    @Override public int compareTo(Object o) {
                Student s = (Student)o;
                int diff = this.id - s.id;
                if (this.id < s.id) return -1;
                else if (this.id > s.id) return 1;
                diff = this.firstName.compareTo(s.firstName);
                if (diff<0) return -1;
                else if (diff>0) return 1;
                diff = this.lastName.compareTo(s.lastName);
                if (diff<0) return -1;
                else if (diff>0) return 1;
                return 0;
}       }
```

first compare id, then first name, then name

all properties have equal values, objects must be equal

Example: Sorting Arrays 2

```
Arrays.sort(students);
for (Student student:students) {
	System.out.println(student);
}

// console output:
Student [id=10101, firstName=James, lastName=Smith]
Student [id=10102, firstName=James, lastName=Taylor]
Student [id=11221, firstName=Albert, lastName=Taylor]
Student [id=12395, firstName=Ronald, lastName=MacDonald]
Student [id=12396, firstName=Sarah, lastName=Anderson]
```

- objects are sorted by id first

Example: Sorting Arrays 3

```
public class Student implements Comparable {
        ..
    @Override public int compareTo(Object o) {
                Student s = (Student)o;
                int diff = this.firstName.compareTo(s.firstName);
                if (diff<0) return -1;
                else if (diff>0) return 1;
        diff = this.lastName.compareTo(s.lastName);
                if (diff<0) return -1;
                else if (diff>0) return 1;
        diff = this.id - s.id;
                if (this.id < s.id) return -1;
                else if (this.id > s.id) return 1;
                return 0;
}       }
```

first compare first name, then name and then id

Example: Sorting Arrays 3

```
Arrays.sort(students);
for (Student student:students) {
	System.out.println(student);
}

// console output:
Student [id=11221, firstName=Albert, lastName=Taylor]
Student [id=10101, firstName=James, lastName=Smith]
Student [id=10102, firstName=James, lastName=Taylor]
Student [id=12395, firstName=Ronald, lastName=MacDonald]
Student [id=12396, firstName=Sarah, lastName=Anderson]
```

- objects are sorted by first name, then last name

Sorting Arrays ctd

- with interfaces it is possible to code different **strategies** to compare objects instantiating the respective classes (like `Student`)
- the `sort` methods in `Arrays` only needs to know that there is **some way** to compare objects
- the interface is used to **decouple** (keep separate) classes like `Student` and utilities like `Arrays.sort()`
- decoupling is an important OO design principle

Sorting Arrays ctd

- there is still a problem: one particular way of comparing students is **hardcoded** in `Student`
- solution: use another interface `java.util.Comparator` to separate the comparison of objects from the actual object classes
- then it is possible to create different comparator classes, like `CompareStudentsById`, `CompareStudentsByName` etc, and use `Arrays.sort(array,comparator)` to sort students (or any other type of object)
- another shortcoming in the `compare` method is the cast - this can be addressed by using generic parameter types (to be discussed later)

Comparable vs comparator

“To summarize, if sorting of objects needs to be based on natural order then use Comparable whereas if you sorting needs to be done on attributes of different objects, then use Comparator in Java.”

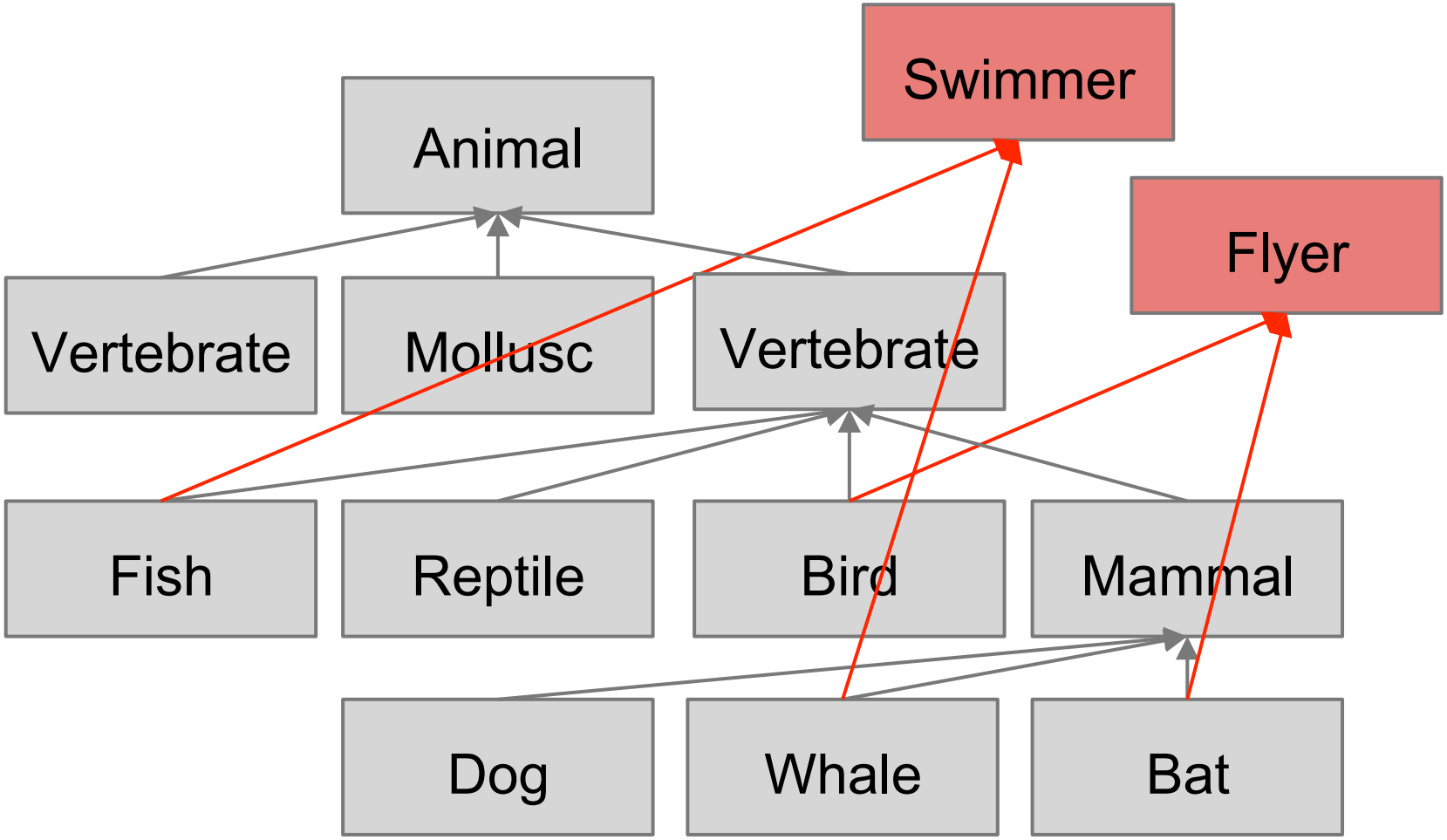
See here:
https://www.geeksforgeeks.org/comparable-vs-comparator-in-java/

Interfaces ctd

- classes **implement** interfaces: this is a **contract** where the class promises to implement all methods declared in the interface
- classes can implement multiple interfaces
- interfaces usually have only methods, no fields
- exceptions are `public static final` fields, i.e. **constants**
- all methods in an interface are abstract
- interfaces can extend multiple other interfaces, i.e. **multiple inheritance for interfaces** is supported
- interfaces can be used as declaration types:

```
Comparable comp = new Student();
```

Interface like Concepts used in Biology



The Semantics Of Subtyping

- so far, only syntax rules have been discussed
- this is sometimes not enough in practise to ensure that inheritance is safe and works as expected
- additional semantic rules are needed, in particular **Liskov's Substitution Principle** (LSP)
- this will be discussed later in the **semantics section**