Section 7

Inheritance & Ploymorphism

Learning Outcomes

- After this section you should be able to:
  - Define reusable classes based on inheritance.
  - Differentiate between Inheritance and Composition
  - Override methods
  - Differentiate between overridding methods & overlaoding methods
  - Write programs that are easily extensible and modifiable by applying polymorphism in program design.
  - ***Reading*** and ***studying*** recommended text is essential to improve your understanding of the above

Inheritance

- A form of software reuse in which a new class is created by absorbing an existing class's members and embellishing them with new or modified capabilities.
- Can save time during program development by basing new classes on existing proven and debugged high-quality software.
- Increases the likelihood that a system will be implemented and maintained effectively.

Inheritance (Cont.)

- When creating a class, rather than declaring completely new members, you can designate that the new class should inherit the members of an existing class.
  - Existing class is the superclass
  - New class is the subclass
- Each subclass can be a superclass of future subclasses.
- A subclass can add its own fields and methods.
- A subclass is more specific than its superclass and represents a more specialized group of objects.
- The subclass exhibits the behaviors of its superclass and can add behaviors that are specific to the subclass.
  - This is why inheritance is sometimes referred to as specialisation.

Inheritance (Cont.)

- The direct superclass is the superclass from which the subclass explicitly inherits.
- An indirect superclass is any class above the direct superclass in the class hierarchy.
- The Java class hierarchy begins with class Object (in package java.lang)
  - Every class in Java directly or indirectly extends (or "inherits from") Object.
- Java supports only single inheritance, in which each class is derived from exactly one direct superclass.

Inheritance (Cont.)

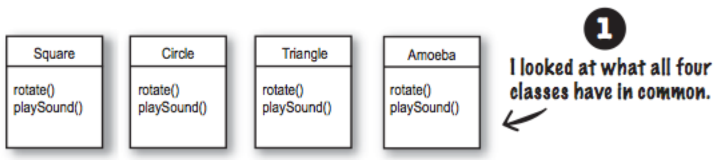
- We distinguish between the is-a relationship and the has-a relationship
- Is-a represents inheritance
  - In an is-a relationship, an object of a subclass can also be treated as an object of its superclass
  - i.e. Lecturer is a Faculty member
- Has-a represents composition
  - In a has-a relationship, an object contains as members references to other objects
  - Traffic Light has a colour (where the field colour has a reference type Colour, which is a class)

Superclasses and Subclasses

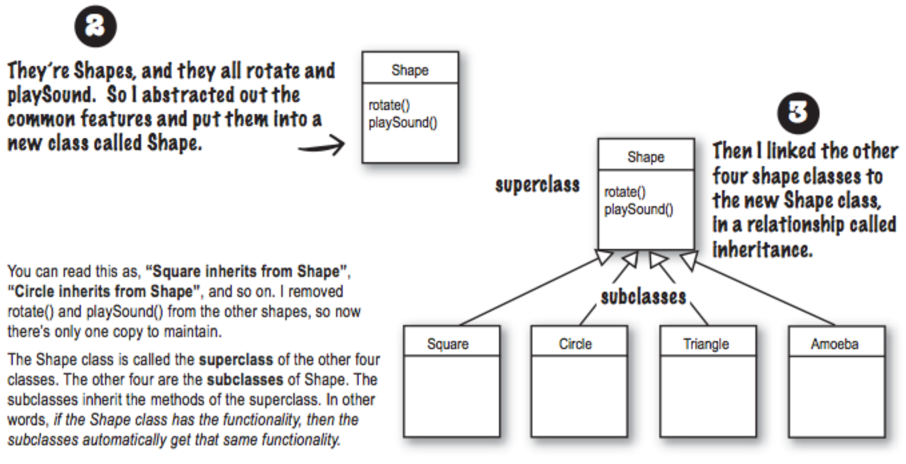
- Objects of all classes that extend a common superclass can be treated as objects of that superclass.
  - Commonality expressed in the members of the superclass.
- Inheritance issue
  - A subclass can inherit methods that it does not need or should not have.
  - Even when a superclass method is appropriate for a subclass, that subclass often needs a customized version of the method.
  - The subclass can override (redefine) the superclass method with an appropriate implementation.

Chair Wars Revisited

- Duplicate code in each class!
- Illustration of how inheritance works



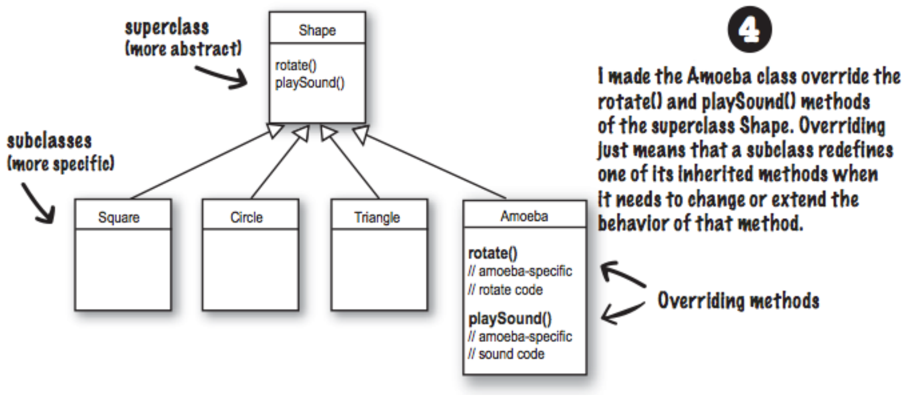
Inheritance


2
They're Shapes, and they all rotate and playSound. So I abstracted out the common features and put them into a new class called Shape.
Shape
rotate()
playSound()
superclass
3
Then I linked the other four shape classes to the new Shape class, in a relationship called inheritance.
subclasses
Square
Circle
Triangle
Amoeba

You can read this as, **"Square inherits from Shape"**, **"Circle inherits from Shape"**, and so on. I removed rotate() and playSound() from the other shapes, so now there's only one copy to maintain.

The Shape class is called the **superclass** of the other four classes. The other four are the **subclasses** of Shape. The subclasses inherit the methods of the superclass. In other words, *if the Shape class has the functionality, then the subclasses automatically get that same functionality.*

Inheritance


superclass (more abstract)
Shape
rotate()
playSound()
subclasses (more specific)
Square
Circle
Triangle
Amoeba
rotate()
// amoeba-specific
// rotate code
playSound()
// amoeba-specific
// sound code
4
I made the Amoeba class override the rotate() and playSound() methods of the superclass Shape. Overriding just means that a subclass redefines one of its inherited methods when it needs to change or extend the behavior of that method.
Overriding methods

Understanding Inheritance

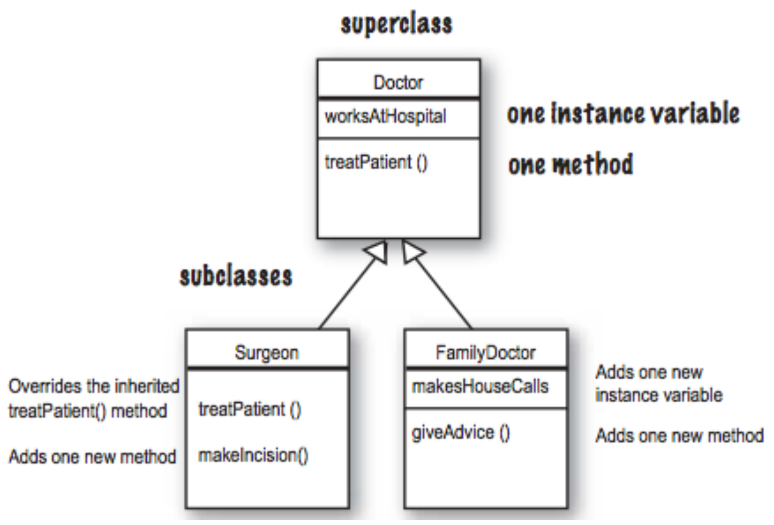
- Put common code in a class
- Other more specific subclasses inherit code from the more abstract superclass
- subclass extends the superclass
- Subclass inherits the members (instance variables & methods) of the superclass

```
public class Doctor {

    boolean worksAtHospital;

    void treatPatient() {
        // perform a checkup
    }
}
```

Inheritance Example

```
public class Surgeon extends Doctor {

    void treatPatient() {
        // perform surgery
    }

    void makeIncison() {
        // make incision
    }
}
public class FamilyDoctor extends Doctor {

    boolean makesHouseCalls;

    void giveAdvice() {
        // give advice to patient
    }

}
```

Inheritance Example (cont.)

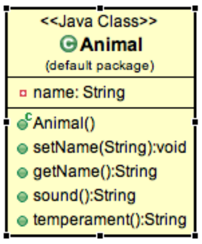


Designing an Inheritance Tree

- Look for objects that have common behaviour which can be abstracted out
- Design classes that represent the common state and behaviour
- Decide if a subclass needs behaviours (method implementations) that are specific to that particular subclass type.
- Look for opportunities to use abstraction, by finding two or more subclasses that might need common behaviour.

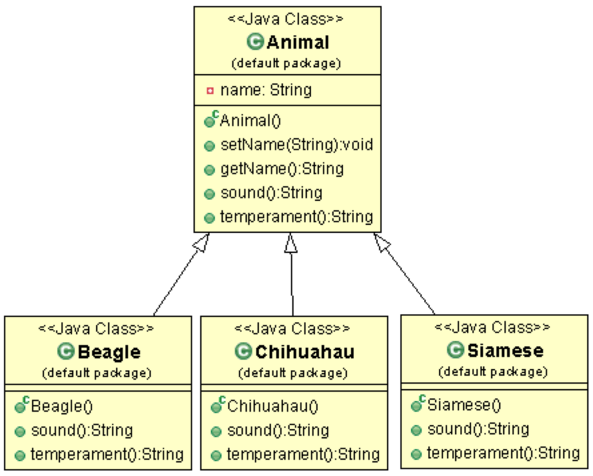
Animal Class

- Instance variable
  - *name*
- Methods
  - *setName()*
  - *getName()*
  - *sound()*
  - *temperament()*



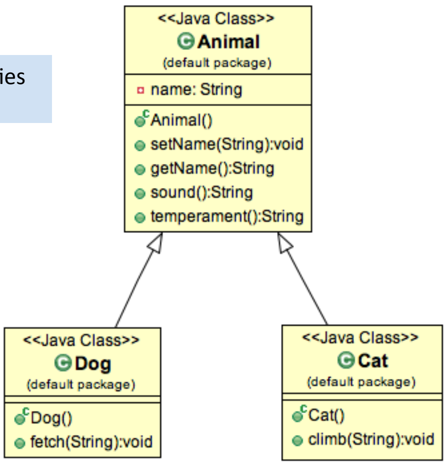
Animal Subclasses

Do subclasses need behaviours that are specific to that particular subclass? - override

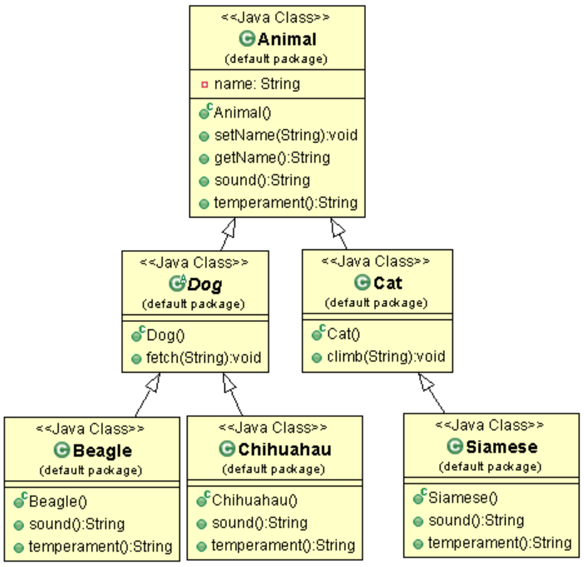


Dog Class & Cat Class

More opportunities to abstract?



Inheritance Hierarchy



Which Method is called?

- *In class exercise - Write an outline of the code required to create classes for the previous inheritance hierarchy*.

- Beagle has five methods, two inherited from Animal, one inherited from Dog and two overridded in the Beagle class.
- Once a Beagle object is created and assigned to a variable, you can use the dot operator on that reference variable to invoke all five methods.
- Which version of the methods gets called?

Which Method is called?

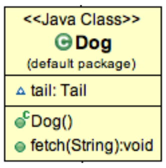
- Make a new Beagle object
  - `Beagle b = new Beagle();`
- Call the version in Beagle
  - `b.temperament(); & b.sound();`
- Calls the version in Dog
  - `b.fetch("ball");`
- Calls the version in Animal
  - `b.setName("Tiny"); b.getName();`
- When you call a method on an object reference, you are calling the most specific version (lowest in the tree). The JVM looks first in the Beagle class and then works it's way up.

IS-A and HAS-A

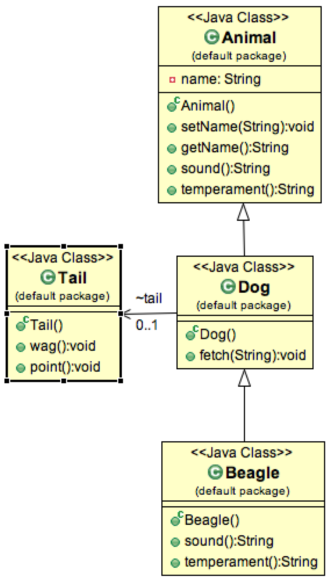
- When a class inherits from an other class the subclass extends the superclass (*IS-A* relationship - Inheritance)
- To know if a class should extend another class apply the IS-A test - Beagle IS-A Dog
- A class can be related to an other class, but not by inheritance i.e. Dog HAS-A Tail.
- Dog will have a Tail instance variable, that is Dog has a reference to a Tail but does not extend Tail.
- *HAS-A* relationship -Composition
  - strong has-a – Composition,
  - weak has-a – Aggregation)

Composition

- If You assume all Dogs have a tail



- Write the code to represent the classes illustrated in the hierarchy



Composition (cont.)

- Aggregation differs from ordinary composition in that it does not imply ownership.
- In composition, when the owning object is destroyed, so are the contained objects. In aggregation, this is not necessarily true.
- Composition is strong relationship: A Composes B, means A contains and owns B i.e. when A is destroyed B will also be destroyed.
- Aggregration between A and B means, A contains but not owns B i.e. when A is destroyed B remains intact.

Composition (cont.)

- Composition
  - A “university” has several “disciplines”. Without existence of “university” there is no chance for the “disciplines” to exist. Hence “university” and “disciplines” are strongly associated and this strong association is known as composition.
- Aggregation
  - A “discipline” has several “lecturers”. Without existence of “discipline” there is good chance for the “lecturers” to exist. Hence “lectures” and “discipline” are loosely associated and this loose association is known as Aggregation.

Composition (cont.)

```
import java.util.ArrayList;

class Lecturer{
     ArrayList<Lecturer> lecturerList = new ArrayList<Lecturer>();
     …
}

class Discipline{
    // Aggregation: ArrayList of Lecturer objects living outside the Discipline
    ArrayList<Lecturer> lecturerList;
     …
}

class University
{
    ArrayList<Discipline> discipline = new ArrayList<Discipline>();

    // constructor
    University()  {

          // Composition: Disciplines exist as long as the University exists
          Discipline arts = new Discipline();
          Discipline informationTechnology = new Discipline();
          discipline.add(arts);
          discipline.add(informationTechnology);

    }
}
```

Inheritance Summary

- With Inheritance
  - A subclass extends a superclass.
  - A subclass inherits all *public* instance variables and methods of the superclass, but does **not** inherit the *private* instance variables and methods of the superclass.
  - Inherited methods can be overridden; instance variables cannot be overridden (although they can be redefined in the subclass – very rare).
  - Use the IS-A test to verify that your inheritance hierarchy is valid.
  - The IS-A relationship works in only one direction.
  - When a method is overridden in a subclass and that method is invoked on an instance of the subclass, the overridden version of the method is called. (The lowest one wins.)

Polymorphism

- *The dictionary definition of polymorphism refers to a principle in biology in which an organism or species can have many different forms or stages. This principle can also be applied to object-oriented programming and languages like the Java language. Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.* (Source: Oracle)
- When you define a supertype for a group of classes, any subclass of that supertype can be substituted where the supertype is expected.
- Refer to a subclass object using a reference declared as the supertype.

Declare & Create a Reference Object (Revisited)

```
Beagle beagle = new Beagle();
```

- Declare a reference variable (`Beagle beagle`)
  - JVM allocates space for a reference variable.
- Create an Object (`new Beagle();`)
  - JVM allocates space for a new object on the garbage collectable heap.
- Link the object and the reference (`=`)
  - Assigns the new object to the reference variable.

Polymorphism

- The reference type can be a superclass to the actual object type.
- Polymorphic Arrays
  - `Animal[] animals = new Animal[5];`
- Call one of the Animal-class methods and every object carries out the behaviour associated with that object.
  - `animals[i].sound();`
- When `animals[i]` is a Beagle, the Beagles sound() method is called, when `animals[i]` is a Siamese, the Siamese sound() method is called and so on …

Polymorphic Arguments & return Types

- If you write your code using polymorphic arguments, when you declare the method parameter as a superclass type, you can pass any subclass object at run time.
  - `public void detectType(Animal a)`
- Animal parameter can take any Animal type as an argument

```
 public void makeNoise(Animal a){
     a.sound();
 }
// Beagle sound
 public void sound(){
     System.out.println("woof woof");
 }
 // Chihuahau sound
 public void sound(){
     System.out.println("yip woof");
 }
```

Overloading & Overridding

- When you override a method from a super class, the arguments and return types of your overridding method must look to the outside world exactly like the overridden method in the superclass.

```
public String toString(){
    return " a string type ";
}
```

- An overloaded method is just a different method that happens to have the same method name. It has nothing to do with inheritance and polymorphism.
- An overloaded method is not the same as an overridden method.

Rules for Overriding & Overloading

- Overridding
  - Arguments must be the same and return types must be compatible
  - The method can't be less accessible (public – private)
- Overloading
  - The return types can be different
  - You can't change only the return type – you **must** change the argument list.

```
public int addNums(int a, int b){
    return a+b;
}
public double addNums(double a, double b){
    return a+b;
}
```

  - You can vary the access levels in any direction.
  - Has nothing to do with inheritance and ploymorphism.