Software Development (cs2500)

Lecture 24: Inheritance

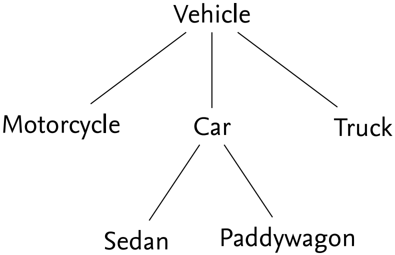
M. R. C. van Dongen

November 18, 2013

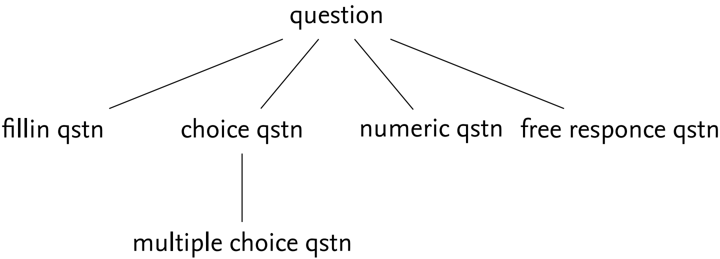
Outline

- We study *inheritance* hierarchies.
- These hierarchies have *superclasses* and *subclasses.*
  - The superclasses are more general.
  - The subclasses are more specific.
- A subclass inherits its superclass's public method behaviour.
- The more general behaviour is in the superclass.
  - This *shares* the implementation of the general behaviour.
- Subclasses may provide more specific behaviour:
  - They may *override* the public methods from their superclass.
  - They may also define new, additional behaviour.
- The JVM always calls a subclass's superclass constructor.
  - By calling `super( )`, the sublcass explicitly calls a constructor.
  - If there's no explicit call, the JVM calls the default constructor.
- A subclass may override a method, `method( )`, in two ways:
  1. It overrides `method` from scratch;
  2. It uses `method`'s inherited behaviour by calling `super.method( )`.
- The *is-a test* helps choosing the sub- and superclass.

Hierarchies are Everywhere



Hierarchies are Everywhere



question

fillin qstn

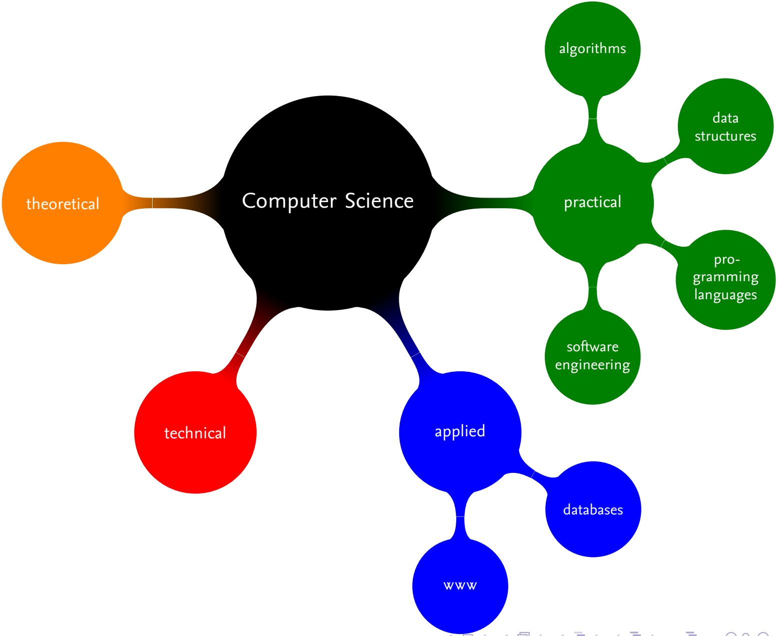
choice qstn

numeric qstn

free responce qstn

multiple choice qstn

Hierarchies are Everywhere



Proper Order

- Hierarchies are drawn as trees.
- The nodes in the tree are the members of the hierarchies.
- The root of the tree is the most general hierarchy member.
  - It corresponds to the name of the hierarchy.
- The lines indicate membership relationships.
  - A node's chidren are more specific.
  - A node's parent (if any) is more general.
- Each node is the parent of a sub-hierarchy.
- Cycles are not allowed.

Java Hierarchies

- Java also has hierarchies.
  - Interface hierarchies;
  - Class hierarchies.
- Defining the hierarchies requires the same keywords.

Java

```java
public interface ParentIterface { ... }

public interface ChildInterface extends ParentInterface { ... }
```

- Makes `ChildInterface` a subinterface of `ParentInterface`.

Java Hierarchies

- Java also has hierarchies.
  - Interface hierarchies;
  - Class hierarchies.
- Defining the hierarchies requires the same keywords.

Java

```
public class ParentClass { ... }

public class ChildClass extends ParentClass { ... }
```

- Makes `ChildClass` a subclass of `ParentClass`.

Substitution Principle

- When you define a subclass of a superclass, you can use subclass instances when the JVM expects an instance of the superclass.
  - (Same for interfaces.)
- This is called the *(Liskov) Substitution Principle*.
- Also works for:
  - Subclass instance of subclass of superclass;
  - Subclass instance of subclass of subclass of superclass;
  - Subclass instance of subclass of subclass of … of superclass;
- Note that the converse is *not* allowed:
  - When a subclass is expected, you cannot use a superclass instance.
- The `Java` compiler is more strict.
  - It only reasons about variable type, not about instance type.

Example

- Let’s assume we have an `Animal` class.
- Let’s assume the `Cat` and `Dog` classes *extend* the `Animal` class.
- Let’s assume the `Dog` class also *extends* the `Animal` class.
- Then `Cat` and `Dog` are subclasses of `Animal`.

Java

```java
Cat cat = new Cat( );
Dog dog = new Dog( );
Animal animal;

animal = cat;
animal = dog;
cat = animal;
dog = animal;
dog = (Dog)animal;
cat = (Cat)animal;
```

Example

- Let's assume we have an `Animal` class.
- Let's assume the `Cat` and `Dog` classes *extend* the `Animal` class.
- Let's assume the `Dog` class also *extends* the `Animal` class.
- Then `Cat` and `Dog` are subclasses of `Animal`.

Java

```
Cat cat = new Cat( );
Dog dog = new Dog( );
Animal animal;

animal = cat;
animal = dog;
cat = animal;
dog = animal;
dog = (Dog)animal;
cat = (Cat)animal;
```

Example

- □ Let’s assume we have an `Animal` class.
- □ Let’s assume the `Cat` and `Dog` classes *extend* the `Animal` class.
- □ Let’s assume the `Dog` class also *extends* the `Animal` class.
- □ Then `Cat` and `Dog` are subclasses of `Animal`.

Java

```
Cat cat = new Cat( );
Dog dog = new Dog( );
Animal animal;

animal = cat;      // grand
animal = dog;
cat = animal;
dog = animal;
dog = (Dog)animal;
cat = (Cat)animal;
```

Example

- □ Let's assume we have an `Animal` class.
- □ Let's assume the `Cat` and `Dog` classes *extend* the `Animal` class.
- □ Let's assume the `Dog` class also *extends* the `Animal` class.
- □ Then `Cat` and `Dog` are subclasses of `Animal`.

Java

```
Cat cat = new Cat( );
Dog dog = new Dog( );
Animal animal;

animal = cat;       // grand
animal = dog;
cat = animal;
dog = animal;
dog = (Dog)animal;
cat = (Cat)animal;
```

Example

- ☐ Let’s assume we have an `Animal` class.
- ☐ Let’s assume the `Cat` and `Dog` classes *extend* the `Animal` class.
- ☐ Let’s assume the `Dog` class also *extends* the `Animal` class.
- ☐ Then `Cat` and `Dog` are subclasses of `Animal`.

Java

```java
Cat cat = new Cat( );
Dog dog = new Dog( );
Animal animal;

animal = cat;       // grand
animal = dog;       // also grand
cat = animal;
dog = animal;
dog = (Dog)animal;
cat = (Cat)animal;
```

Example

- ☐ Let’s assume we have an `Animal` class.
- ☐ Let’s assume the `Cat` and `Dog` classes *extend* the `Animal` class.
- ☐ Let’s assume the `Dog` class also *extends* the `Animal` class.
- ☐ Then `Cat` and `Dog` are subclasses of `Animal`.

Java

```
Cat cat = new Cat( );
Dog dog = new Dog( );
Animal animal;

animal = cat;      // grand
animal = dog;      // also grand
cat = animal;
dog = animal;
dog = (Dog)animal;
cat = (Cat)animal;
```

Example

- Let’s assume we have an `Animal` class.
- Let’s assume the `Cat` and `Dog` classes *extend* the `Animal` class.
- Let’s assume the `Dog` class also *extends* the `Animal` class.
- Then `Cat` and `Dog` are subclasses of `Animal`.

Java

```java
Cat cat = new Cat( );
Dog dog = new Dog( );
Animal animal;

animal = cat;      // grand
animal = dog;      // also grand
cat = animal;      // not allowed by compiler
dog = animal;
dog = (Dog)animal;
cat = (Cat)animal;
```

Example

- Let’s assume we have an `Animal` class.
- Let’s assume the `Cat` and `Dog` classes *extend* the `Animal` class.
- Let’s assume the `Dog` class also *extends* the `Animal` class.
- Then `Cat` and `Dog` are subclasses of `Animal`.

Java

```java
Cat cat = new Cat( );
Dog dog = new Dog( );
Animal animal;

animal = cat;       // grand
animal = dog;       // also grand
cat = animal;       // not allowed by compiler
dog = animal;
dog = (Dog)animal;
cat = (Cat)animal;
```

Example

- □ Let's assume we have an `Animal` class.
- □ Let's assume the `Cat` and `Dog` classes *extend* the `Animal` class.
- □ Let's assume the `Dog` class also *extends* the `Animal` class.
- □ Then `Cat` and `Dog` are subclasses of `Animal`.

Java

```
Cat cat = new Cat( );
Dog dog = new Dog( );
Animal animal;

animal = cat;       // grand
animal = dog;       // also grand
cat = animal;       // not allowed by compiler
dog = animal;       // also not allowed by compiler
dog = (Dog)animal;
cat = (Cat)animal;
```

Example

- Let’s assume we have an `Animal` class.
- Let’s assume the `Cat` and `Dog` classes *extend* the `Animal` class.
- Let’s assume the `Dog` class also *extends* the `Animal` class.
- Then `Cat` and `Dog` are subclasses of `Animal`.

Java

```java
Cat cat = new Cat( );
Dog dog = new Dog( );
Animal animal;

animal = cat;      // grand
animal = dog;      // also grand
cat = animal;      // not allowed by compiler
dog = animal;      // also not allowed by compiler
dog = (Dog)animal;
cat = (Cat)animal;
```

Example

- Let's assume we have an `Animal` class.
- Let's assume the `Cat` and `Dog` classes *extend* the `Animal` class.
- Let's assume the `Dog` class also *extends* the `Animal` class.
- Then `Cat` and `Dog` are subclasses of `Animal`.

Java

```
Cat cat = new Cat( );
Dog dog = new Dog( );
Animal animal;

animal = cat;       // grand
animal = dog;       // also grand
cat = animal;       // not allowed by compiler
dog = animal;       // also not allowed by compiler
dog = (Dog)animal; // allowed and grand
cat = (Cat)animal;
```

Example

- Let’s assume we have an `Animal` class.
- Let’s assume the `Cat` and `Dog` classes *extend* the `Animal` class.
- Let’s assume the `Dog` class also *extends* the `Animal` class.
- Then `Cat` and `Dog` are subclasses of `Animal`.

Java

```java
Cat cat = new Cat( );
Dog dog = new Dog( );
Animal animal;

animal = cat;      // grand
animal = dog;      // also grand
cat = animal;      // not allowed by compiler
dog = animal;      // also not allowed by compiler
dog = (Dog)animal; // allowed and grand
cat = (Cat)animal;
```

Example

- Let’s assume we have an `Animal` class.
- Let’s assume the `Cat` and `Dog` classes *extend* the `Animal` class.
- Let’s assume the `Dog` class also *extends* the `Animal` class.
- Then `Cat` and `Dog` are subclasses of `Animal`.

Java

```
Cat cat = new Cat( );
Dog dog = new Dog( );
Animal animal;

animal = cat;       // grand
animal = dog;       // also grand
cat = animal;       // not allowed by compiler
dog = animal;       // also not allowed by compiler
dog = (Dog)animal; // allowed and grand
cat = (Cat)animal; // compiler allows it but jvm chokes: runtime error
```

Inheritance

- `Java`'s class hierarchy mechanism is very powerful.
- Humans *inherit* from their predecessors:
  - Genes;
  - House/land/money,
- In `Java` a subclass *inherits* from its superclass(es).
- This time *inherits* has a technical meaning.
- It means the subclass instance can:
  - Access the `public` superclass instance attributes;
  - Call the `public` superclass instance methods.
- Superclass can be used as type for polymorphic variable.

Example

Question Superclass

Java

```
public class Question {
    private String text;
    private String answer;

    public Question( ) {
        text = "";
        answer = "";
    }

    public void setText( String text ) {
        this.text = text;
    }

    public void setAnswer( String answer ) {
        this.answer = answer;
    }

    public boolean checkAnswer( String response ) {
        return answer.equals( response );
    }

    public void display( ) {
        System.out.println( text );
    }
}
```

Example (Continued)

`MultipleChoiceQuestion` Subclass

Java

```
public class NumericQuestion extends Question {
    public NumericQuestion( String text, double answer ) {
        setText( text );
        setAnswer( Double.toString( answer ) );
    }
}
```

Example (Continued)

Java

```
public class Exam {
    public static void main( String[] args ) {
        final Question question = new NumericQuestion( "What is the answer?", 42 );
        final Scanner scanner = new Scanner( System.in );

        question.display( );
        final String answer = scanner.next( );
        if (question.checkAnswer( answer )) {
            System.out.println( "Well done." );
        } else {
            System.out.println( "Back to the books." );
        }
    }
}
```

Overriding

- Java subclasses may redefine public superclass methods.
- This is called *overriding* the methods.
- Overriding a method in a subclass only affects the subclass;
    - And its subclasses;
    - And the subclasses of the subclasses;
    - And …;
- Overriding lets subclasses behave in a special, more specific way.

Overriding a public Method

Java

```
public class SuperClass {
    ...

    public void behaviour( ) {
        ...
    }
}

public class SubClass extends SuperClass {
    ...

    @Override
    public void behaviour( ) {
        ...
    }
}
```

Example Continued

Java

```java
public class MultipleChoiceQuestion extends Question {
    private final ArrayList<String> options;

    public MultipleChoiceQuestion( String text,
                                   ArrayList<String> options,
                                   String solution ) {
        setText( text );
        this.options = options;

        int number = 1;
        for (String option : options) {
            if (solution.equals( option )) {
                setAnswer( Integer.toString( number ) );
            }
            number++;
        }
    }

    @Override
    public void display( ) {
        int label = 0;
        // output question's text (omitted)
        for (String option : options) {
            System.out.println( (++label) + ": " + option );
        }
    }
}
```

Example (Continued)

Java

```java
public class Exam {
    public static void main( String[] args ) {
        final String text = "What’s the first month of the year?";
        final String solution = "January";
        final ArrayList<String> options = new ArrayList<String>( );
        options.add( solution );
        options.add( "February" );
        options.add( "March" );
        options.add( "April" );
        final Question question
            = new MultipleChoiceQuestion( text, options, solution );

        final Scanner scanner = new Scanner( System.in );
        question.display( );
        final String answer = scanner.next( );
        if (question.checkAnswer( answer )) {
            System.out.println( "Well done." );
        } else {
            System.out.println( "Back to the books." );
        }
     }
}
```

Running the Example

Unix Session

```
$
```

Running the Example

Unix Session

```
$ javac *.java
```

Running the Example

Unix Session

```
$ javac *.java
$
```

Running the Example

Unix Session

```
$ javac *.java
$ java Exam
```

Running the Example

Unix Session

```
$ javac *.java
$ java Exam
What's the first month of the year?
1: January
2: February
3: March
4: April
```

Running the Example

Unix Session

```
$ javac *.java
$ java Exam
What’s the first month of the year?
1: January
2: February
3: March
4: April
 2
```

Running the Example

Unix Session

```
$ javac *.java
$ java Exam
What's the first month of the year?
1: January
2: February
3: March
4: April
 2
Back to the books.
$
```

Share the Common Code

- Inheritance lets you put the common code in the superclass.
- The code can be shared, so there's no need to copy and paste.
- If there's an error in the code, you only have to fix it once.

Factoring out Common Bahaviour

- When you design a class hierarchy you may make mistakes.
  - E.g. you may not have thought of all the consequences.
- It's very possible you notice unexpected common behaviour.
- If you do, and "if it makes sense," you can factor it out:
  - Identify the common subclass behaviour;
  - Remove the behaviour from the subclasses;
  - Implement it as superclass behaviour (pull it up).

Before Factorization

Java

```
public class SuperClass {
    ...
}

public class SubClass1 extends SuperClass {
    ...

    public void doStuff( ) {
        // do this
    }
}

public class SubClass2 extends SuperClass {
    ...

    public void doStuff( ) {
        // also do this
    }
}
```

After Factorization

Java

```
public class SuperClass {
    ...

    public void doStuff( ) {
        // do this
    }
}

public class SubClass1 extends SuperClass {
    ...
}

public class SubClass2 extends SuperClass {
    ...
}
```

Using Superclass Behaviour

- The subclass constructor always calls the superclass constructor.
- The superclass constructor call is always the first call.
  - You can put in an explict call to the constructor.
  - Leaving it out is equivalent to calling the default constructor.
- To call the constructor, you call super(⟨arguments⟩).

Explicitly Calling Superclass Methods

- You can always call `public` superclass methods.
- You can even do this if you're overriding a method.
- Lets you override superclass methods with superclass methods.
- To call `method( )` in superclass, you write `super.method( )`.

Example

Java

```java
public class MultipleChoiceQuestion {
    ...

    @Override
    public void display( ) {
        int label = 0;
        super.display( );
        for (String option : options) {
            System.out.println( (++label) + ": " + option );
        }
    }
}
```

Using the Superclass Constructor

Java

```
public class Animal {
    private final String species;

    public Animal( final String species ) {
        this.species = species;
    }

    public void eat( ) { }
}

public class Herbivore extends Animal {
    public Herbivore( final String species ) {
        super( species );
    }

    @Override
    public void eat( ) {
        System.out.println( "Eating grass." );
    }
}
```

A Test for Inheritance

- Designing a class hierarchy is an art, more than a science.
- It may be difficult to get things right from the start.
- What classes should you use?
- Which class should go to top, middle, and bottom?
- The *is-a test* provides some help to catch early mistakes.
- If ‘every A *is-a* B’ then A can be a subclass of B.

Examples

- Every `Dog` is-an `Animal`?

Examples

- Every `Dog` is-an `Animal`? ($\surd$)
  - So `Dog` can be a subclass of `Animal`.

Examples

- Every Dog is-an Animal? ($\surd$)
  - So Dog can be a subclass of Animal.
- Every Animal is-a Dog.

Examples

- Every `Dog` is-an `Animal`? (√)
  - So `Dog` can be a subclass of `Animal`.
- Every `Animal` is-a `Dog`.
  - No, so `Animal` cannot be a subclass of `Dog`.

Examples

- Every `Dog` is-an `Animal`? ($\surd$)
  - So `Dog` can be a subclass of `Animal`.
- Every `Animal` is-a `Dog`.
  - No, so `Animal` cannot be a subclass of `Dog`.
- Every `Apple` is-a `Pear`.

Examples

- Every `Dog` is-an `Animal`? (√)
  - So `Dog` can be a subclass of `Animal`.
- Every `Animal` is-a `Dog`.
  - No, so `Animal` cannot be a subclass of `Dog`.
- Every `Apple` is-a `Pear`.
  - No, so `Apple` cannot be a subclass of `Pear`.

Examples

- Every Dog is-an Animal? (√)
  - So Dog can be a subclass of Animal.
- Every Animal is-a Dog.
  - No, so Animal cannot be a subclass of Dog.
- Every Apple is-a Pear.
  - No, so Apple cannot be a subclass of Pear.
- Every Pear is-an Apple.

Examples

- Every `Dog` is-an `Animal`? (√)
  - So `Dog` can be a subclass of `Animal`.
- Every `Animal` is-a `Dog`.
  - No, so `Animal` cannot be a subclass of `Dog`.
- Every `Apple` is-a `Pear`.
  - No, so `Apple` cannot be a subclass of `Pear`.
- Every `Pear` is-an `Apple`.
  - No, so `Pear` also cannot be a subclass of `Apple`.

Examples

- Every `Dog` is-an `Animal`? ($\surd$)
  - So `Dog` can be a subclass of `Animal`.
- Every `Animal` is-a `Dog`.
  - No, so `Animal` cannot be a subclass of `Dog`.
- Every `Apple` is-a `Pear`.
  - No, so `Apple` cannot be a subclass of `Pear`.
- Every `Pear` is-an `Apple`.
  - No, so `Pear` also cannot be a subclass of `Apple`.
- Every `Cat` is-a `Feline`.

Examples

- Every `Dog` is-an `Animal`? (√)
  - So `Dog` can be a subclass of `Animal`.
- Every `Animal` is-a `Dog`.
  - No, so `Animal` cannot be a subclass of `Dog`.
- Every `Apple` is-a `Pear`.
  - No, so `Apple` cannot be a subclass of `Pear`.
- Every `Pear` is-an `Apple`.
  - No, so `Pear` also cannot be a subclass of `Apple`.
- Every `Cat` is-a `Feline`. (√)
  - Yes, so `Cat` can be a subclass of `Feline`.

Examples

- Every `Dog` is-an `Animal`? (√)
  - So `Dog` can be a subclass of `Animal`.
- Every `Animal` is-a `Dog`.
  - No, so `Animal` cannot be a subclass of `Dog`.
- Every `Apple` is-a `Pear`.
  - No, so `Apple` cannot be a subclass of `Pear`.
- Every `Pear` is-an `Apple`.
  - No, so `Pear` also cannot be a subclass of `Apple`.
- Every `Cat` is-a `Feline`. (√)
  - Yes, so `Cat` can be a subclass of `Feline`.
- Every `Feline` is-a `Cat`.

Examples

- Every `Dog` is-an `Animal`? (√)
  - So `Dog` can be a subclass of `Animal`.
- Every `Animal` is-a `Dog`.
  - No, so `Animal` cannot be a subclass of `Dog`.
- Every `Apple` is-a `Pear`.
  - No, so `Apple` cannot be a subclass of `Pear`.
- Every `Pear` is-an `Apple`.
  - No, so `Pear` also cannot be a subclass of `Apple`.
- Every `Cat` is-a `Feline`. (√)
  - Yes, so `Cat` can be a subclass of `Feline`.
- Every `Feline` is-a `Cat`.
  - No, so `Feline` cannot be a subclass of `Cat`.

Other ‘Tests’

The ‘extends test’ is not so robust:

- `Cat` extends `Feline`.

Other ‘Tests’

The ‘extends test’ is not so robust:

- `Cat` extends `Feline`.
  - So `Cat` can be a subclass of `Feline`.

Other ‘Tests’

The ‘extends test’ is not so robust:

- `Cat` extends `Feline`.
    - So `Cat` can be a subclass of `Feline`.
- `Feline` extends `Cat`.

Software Development

M. R. C. van Dongen

Outline

Hierarchies

Overriding Behaviour

Using Superclass Behaviour

The Is-A Test

A Test for Inheritance

An Association Test

For Wednesday

Acknowledgements

About this Document

Other ‘Tests’

The ‘extends test’ is not so robust:

- `Cat` extends `Feline`.
    - So `Cat` can be a subclass of `Feline`.
- `Feline` extends `Cat`.
    - No, so `Feline` cannot be a subclass of `Cat`.

Other ‘Tests’

The ‘extends test’ is not so robust:

- Cat extends Feline.
  - So Cat can be a subclass of Feline.
- Feline extends Cat.
  - No, so Feline cannot be a subclass of Cat.
- Conservatory (Sunroom) extends House.

Other 'Tests'

The 'extends test' is not so robust:

- `Cat` extends `Feline`.
  - So `Cat` can be a subclass of `Feline`.
- `Feline` extends `Cat`.
  - No, so `Feline` cannot be a subclass of `Cat`.
- `Conservatory` (`Sunroom`) extends `House`.
  - Yes, but `Conservatory` cannot be a subclass of `House`.

Other ‘Tests’

The ‘extends test’ is not so robust:

- `Cat` extends `Feline`.
  - So `Cat` can be a subclass of `Feline`.
- `Feline` extends `Cat`.
  - No, so `Feline` cannot be a subclass of `Cat`.
- `Conservatory` (`Sunroom`) extends `House`.
  - Yes, but `Conservatory` cannot be a subclass of `House`.
  - For example:
    - If `Conservatory` extends `House` then it inherits all `House` methods:
    - `Conservatory.ringDoorBell( )`????
    - `Conservatory.lightFireplace( )`????

An Association Test

- The House uses/requires/has access to the Conservatory.
- Still Conservatory cannot extend House.
- However, it makes sense if House class has Conservatory attribute.

Java

```
public class House {
    private Bell doorBell;
    private Window[] groundfloorWindows;
    private Window[] firstFloorWindows;
    private Conservatory conservatory;
    ...
}
```

An Association Test (Continued)

- `MouseCursor` cannot be a subclass of `Window`.
- But, makes sense if `Window` class has `MouseCursor` attribute.

Java

```
public class Window {
    private Position currentPosition;
    private Point lowerLeft;
    private Point upperRight;
    private MouseCursor cursor;
    …
}
```

An Association Test (Continued)

- If a class *A* has a class-*B* attribute then class *A* *uses* *B*.
  - `Window` uses a `MouseCursor`.
  - `House` uses a `Conservatory`.
- The *has-a* test determines when a class uses another class.
- If '*A* has-a *B*' then *A* can have a class-*B* attribute.

Examples

- Every `House` has-a `Conservatory` (possibly `null`).

Examples

- Every `House` has-a `Conservatory` (possibly `null`). ($\surd$)
  - So `House` should have a `Conservatory` attribute.

Examples

- Every `House` has-a `Conservatory` (possibly `null`). ($\surd$)
  - So `House` should have a `Conservatory` attribute.
- Every `Window` has-a `MouseCursor`.

Examples

- Every House has-a Conservatory (possibly null). (√)
  - So House should have a Conservatory attribute.
- Every Window has-a MouseCursor. (√)
  - So Window should have a MouseCursor attribute.

Examples

- Every `House` has-a `Conservatory` (possibly `null`). ($\surd$)
  - So `House` should have a `Conservatory` attribute.
- Every `Window` has-a `MouseCursor`. ($\surd$)
  - So `Window` should have a `MouseCursor` attribute.
- Every `Animal` has-a `Cat`.

Examples

- Every `House` has-a `Conservatory` (possibly `null`). $(\surd)$
  - So `House` should have a `Conservatory` attribute.
- Every `Window` has-a `MouseCursor`. $(\surd)$
  - So `Window` should have a `MouseCursor` attribute.
- Every `Animal` has-a `Cat`.
  - No, so `Animal` shouldn't have a `Cat` attribute.

Examples

- Every House has-a Conservatory (possibly null). (√)
  - So House should have a Conservatory attribute.
- Every Window has-a MouseCursor. (√)
  - So Window should have a MouseCursor attribute.
- Every Animal has-a Cat.
  - No, so Animal shouldn't have a Cat attribute.
- Every Cat has-an Animal.

Examples

- □ Every `House` has-a `Conservatory` (possibly `null`). (√)
  - □ So `House` should have a `Conservatory` attribute.
- □ Every `Window` has-a `MouseCursor`. (√)
  - □ So `Window` should have a `MouseCursor` attribute.
- □ Every `Animal` has-a `Cat`.
  - □ No, so `Animal` shouldn't have a `Cat` attribute.
- □ Every `Cat` has-an `Animal`.
  - □ No, so `Cat` shouldn't have an `Animal` attribute.

Examples

- Every House has-a Conservatory (possibly null). (√)
  - So House should have a Conservatory attribute.
- Every Window has-a MouseCursor. (√)
  - So Window should have a MouseCursor attribute.
- Every Animal has-a Cat.
  - No, so Animal shouldn't have a Cat attribute.
- Every Cat has-an Animal.
  - No, so Cat shouldn't have an Animal attribute.

For Wednesday

- Study [Horstmann 2013, Sections 9.1–9.3].

Acknowledgements

- This lecture corresponds to [Horstmann 2013, Sections 9.1–9.3].
- The cs mindmap picture is from [Tantau 2010].

About this Document

- This document was created with `pdflatex`.
- The LaTeX document class is `beamer`.