BOOK2 - OBJECT-ORIENTED PROGRAMMING - YEAR 2 OO JAVA

by

Dr. Matt Smith

https://github.com/dr-matt-smith

Acknowledgements

Thanks to ...

Table of Contents

1

Enumerations

1.1 Enumerations: Java keyword: `enum`

To enumerate...

How Do I Love Thee? (Sonnet 43)

Elizabeth Barrett Browning - 1806-1861

~~enumerate~~

How do I love thee? Let me count the ways.
I love thee to the depth and breadth and height
My soul can reach, when feeling out of sight
For the ends of being and ideal grace.
I love thee to the level of every day's
Most quiet need, by sun and candle-light.
I love thee freely, as men strive for right.
I love thee purely, as they turn from praise.
I love thee with the passion put to use
In my old griefs, and with my childhood's faith.
I love thee with a love I seemed to lose
With my lost saints. I love thee with the breath,
Smiles, tears, of all my life; and, if God choose,
I shall but love thee better after death.

To **enumerate** is to list out every possibility, one by one:

```
{
```

```
    VALUE1, VALUE2 ...
}
```

1.2 Simple concept – list all permitted values

We can define (enumerate) the full set of allowed values for a variable

- Allows us to create value names that are **meaningful** in terms of the real-world software system

```
RegistrationType  { FULL_TIME, PART_TIME, GRADUATE, SUSPENDED }
FoodStatus  { FRESH, SELL_TODAY, OUT_OF_DATE, LASTS_FOREVER }
TeachingEvent  { LECTURE, LAB, EXAM, TUTORIAL, FIELD_TRIP }
```

1.3 Better than DIY codes

(the Do-It-Yourself approach to enumerations…)

In the past, we might have used an integer variable, and allocated codes using constants. E.g.

```
public static final int NOT_SET = 0;
public static final int PART_TIME = 1;
public static final int FULL_TIME = 2;
public static final int SINGLE_MODULE = 3;
// and so on ...

private int registrationType  = NOT_SET;
```

But there many problems with this approach, including:

- validation - integers allow negative numbers, they allow values not matching one of these constants
- programmers may just use the numbers rather than the meaningful constants, e.g. `if(registrationType ==0){ ... send registration reminder ...}`
  - so we lose the chance to write meaningful code, rather than code with special **sentinal** values, that we have look up elsewhere in the source files …

1.4 Declaring an enum

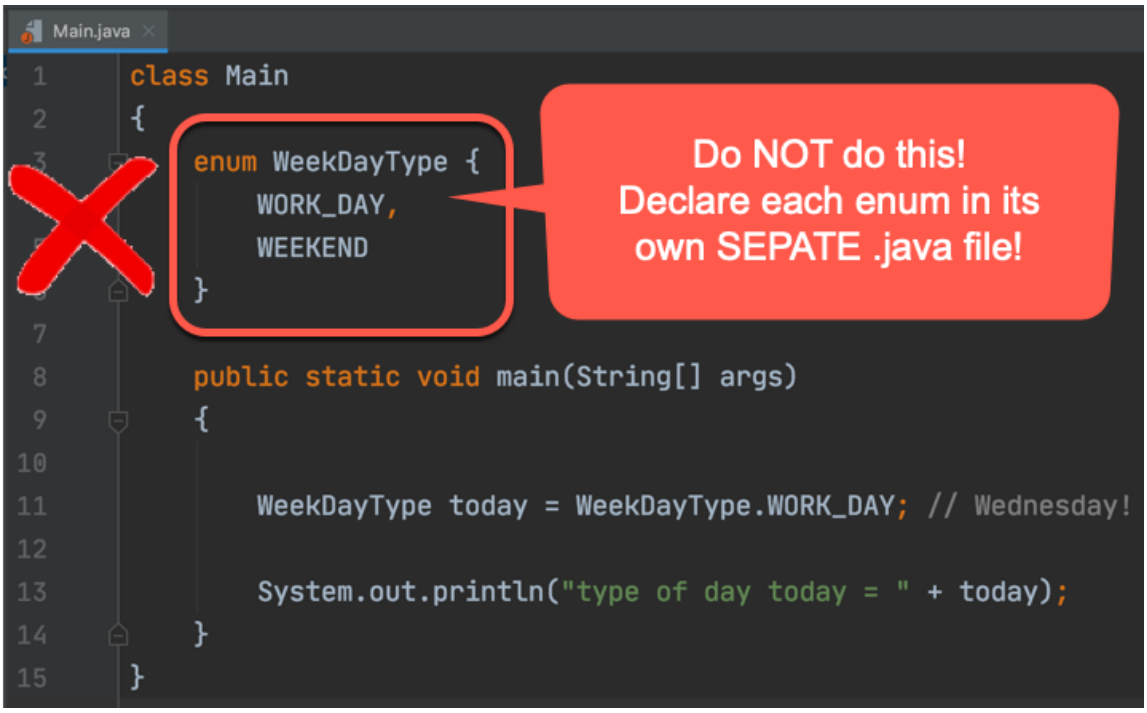
- Can be `private` , but usually `public` ,since we are declaring a named set of constants
- No data type since we are defining set of all possible names (values)
  - Internally, will be stored as 0, 1, 2 etc. (small integers)
- So in most cases we write:

- `public enum { <CONSTANT_VALUES> }`
- The set of CONSTANTS are comma-separated, e.g.

```
public enum RegistrationType {
  FULL_TIME,
  PART_TIME,
  GRADUATE,
  SUSPENDED
}
```

1.5 Where to declare enumerations

We can use enums in general Java class (e.g. Main)- but best practice is to have a separate, public enum class for each enum declaration e.g. `WeekDayType.java`



This is **NOT** a good idea

Please do NOT declare enums inside other classes …

Give them their own enum class: `OperatingSystemType.java`

OperatingSystemType.java

```
public enum OperatingSystemType {
    APPLE,
    ANDROID,
    OTHER
}
```

Phone.java

```
class Phone
{
    private OperatingSystemType operatingSystem = OperatingSystemType.ANDROID;

    public OperatingSystemType getOperatingSystem()
    {
        return this.operatingSystem;
    }

    public void setOperatingSystem(OperatingSystemType newOperatingSystem)
    {
        this.operatingSystem = newOperatingSystem;
    }
}
```

Main.java

```
class Main
{
    public static void main(String[] args)
    {
        Phone phone1 = new Phone();
        phone1.setOperatingSystem(OperatingSystemType.APPLE);
        System.out.println("type of OS = " + phone1.getOperatingSystem());
    }
}
```

1.6 Need to refer to class and value, when using enums from other classes

Since enum declared inside an enum class then from other classes we need to tell Java what class to look inside to find enum declaration

`enumFoodType` declared in enum class `FoodType`

To use values in other classes, e.g. `Main, Dessert` ,we must prefix enum value with enum class name. E.g.

`FoodType.VEGATABLE`

We are referring to an enum CONSTANT declared in an enum class

So just like referring to a CONSTANT. E.g.

```
DrivingLicence.MIN_AGE
Math.PI
```

1.7 Summary & Conclusions

- Enumerations Java keyword: `enum`
  - For use-defined enumerated types
  - Where programmer can enumerate set of possible values
  - `public enum ModuleGrade {A, B, C, D, F}`
- Declare it its own enum class file, e.g. `ModuleGrade.java`
- Refer to enum values in other classes using enum class prefix
- `ModuleGrade.A`

1.8 Java Docs

Enums

- https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

ADVANCED ENUM STUFF – not needed for this module …

- properties and methods in enums
  - http://tutorials.jenkov.com/java/enums.html

(ADVANCED) For highly memory-speed efficient enums (bit sets) seeEnumSets…

- https://docs.oracle.com/javase/1.5.0/docs/api/java/util/EnumSet.html

1.9 Exercise - follow a guided solution to enumerations in Java

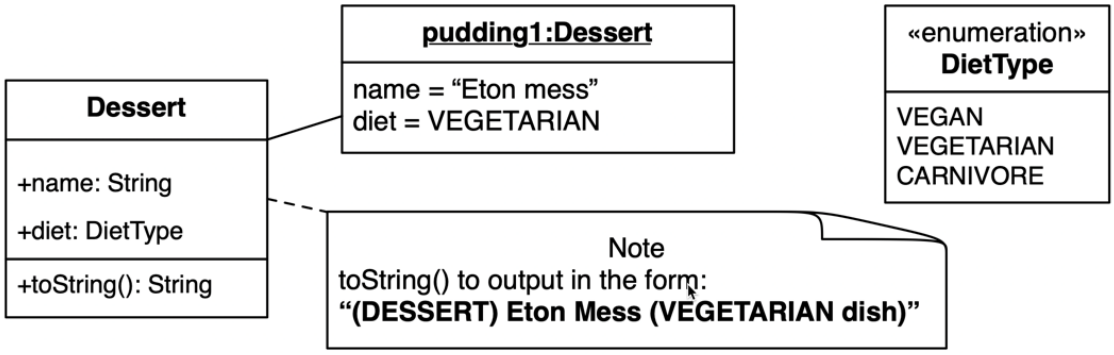


Figure 1.1: Class-Object diagram for project with an Enumeration.

AIM:

- see how to create and use an enum in Java, using guided steps.

See the diagram, it shows:

- an **enumeration** `DietType` with 3 values `{VEGAN, VEGETARIAN, CARNIORE}`
- class `Dessert`
  - with public String `name` property
  - with public `DietType diet` property

Here is the code for class `DietType`, which declares the `enum` values (this is called an enumeration class):

```
// File: DietType.java
public enum DietType {
    VEGAN,
    VEGETARIAN,
    CARNIVORE
}
```

Here is the code for class `Dessert`, which makes use of the `enum` declared above:

```
// File: Dessert.java
class Dessert
{
    public String name;
    public DietType diet;
```

```
    public String toString() {
        return "(DESSERT) "
                + this.name
                + " (" + this.diet + " disk)";
    }
}
```

Here is a `Main` class to ake use of the `Dessert` class:

```
// File: Main.java
class Main
{
    static public void main(String[] args)
    {
        Dessert pudding1 = new Dessert();
        pudding1.diet = DietType.VEGETARIAN;
        pudding1.name = "Eton Mess";

        System.out.println(pudding1);
    }
}
```

OUTPUT:

```
$ java Main
(DESSERT) Eton Mess (VEGETARIAN disk)
```

1.10 Exercise - create a solution using an enumeration class

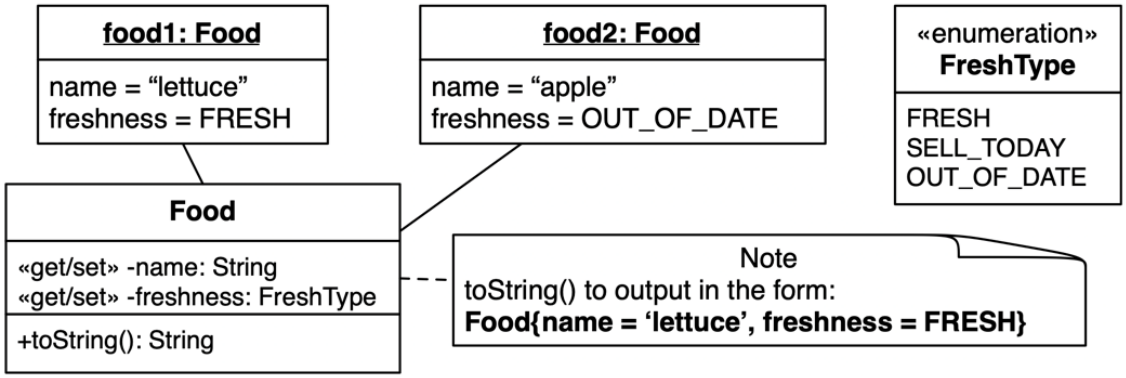


Figure 1.2: Class-Object diagram for Food-FreshType enum.

AIM:

- explore use of enums for a specified set of values for a variable

ACTION:

- enumeration class `FreshType` (file: `FreshType.java`)
  - declare an enumeration class named `FreshType` with 3 values `{FRESH, SELL_TODAY, OUT_OF_DATE}`
- class `Food` (file: `Food.java`)
  - declare a variable `freshness` which stores a `FreshType` value
  - public get and set methods for both variables
  - with public `toString()` method to return a String summary of the object's state (as shown in the diagram)
- class `Main` (file: `Main.java`)
  - Create an instance of `Food` named `food1`, which is still fresh lettuce
  - Create an instance of `Food` named `food2`, which is an out of date apple
  - print out each object's state via its `toString()` method, i.e.

```
System.out.println(food1);
System.out.println(food2);
```

- compile (`javac *.java`) and run your program

OUTPUT:

```
$ java Main
Food{name='lettuce', freshness=FRESH}
Food{name='apple', freshness=OUT_OF_DATE}
```

2

Protected visibility modifier (acceess for subclasses)

2.1 Protected visibility, only for class and its subclasses

There are limitations to `private` visibility, once we start having an inheritance hierarchy of related classes.

The methods in subclasses cannot access `private` members of any superclass

2.2 Java inheritance: Java keywords: `protected`

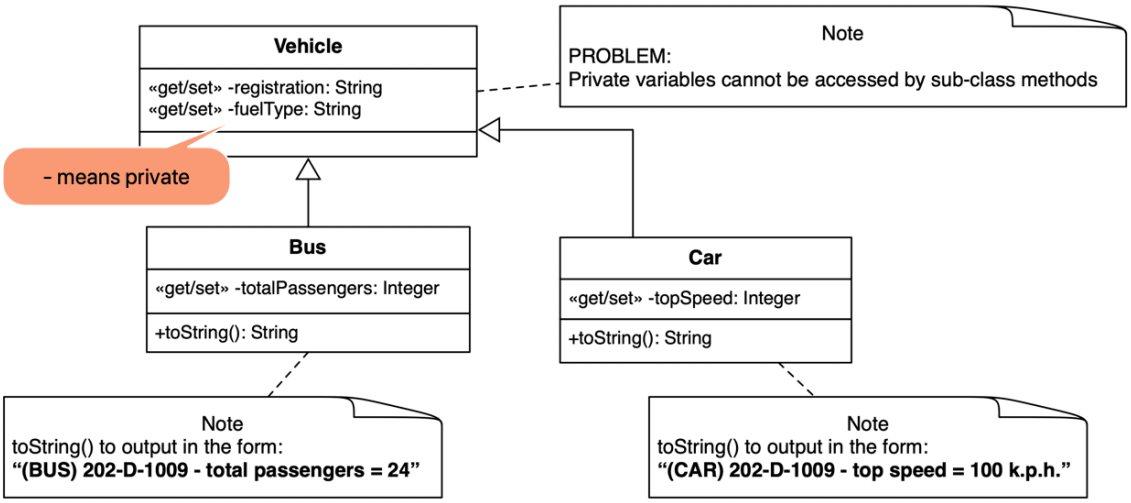
We can solve the issue with `protected` visibility…

2.3 `private` visibility

`private` visibility means:

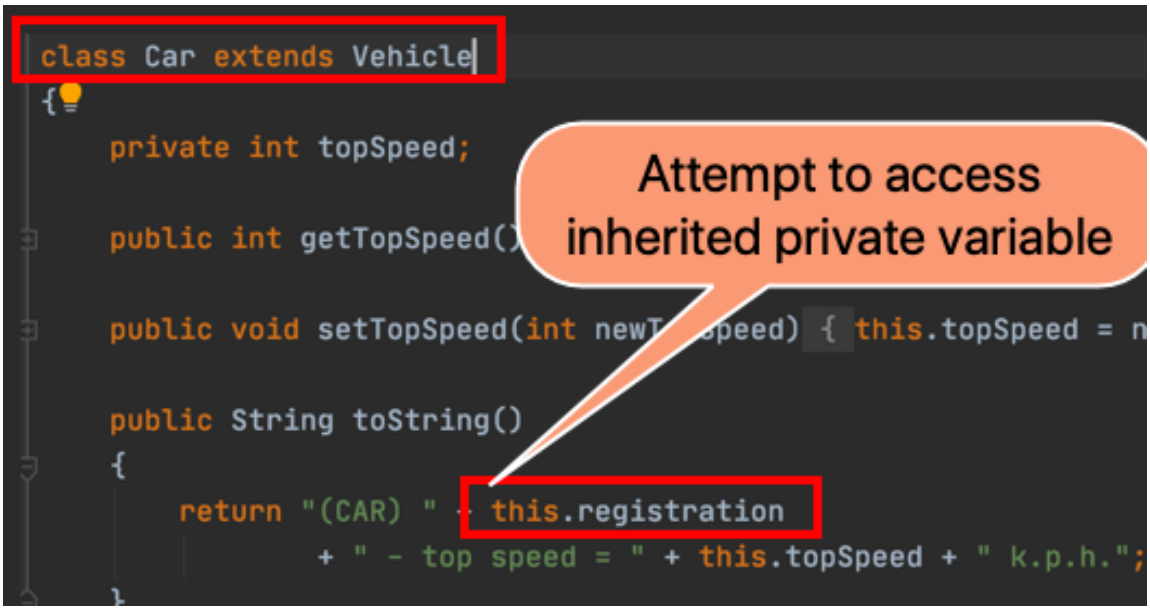
- Variable/method cannot be accessed from outside of object
  - Usually what we want
  - But also means variable/method cannot be used by `subclasses`
    - Not always what we want…

2.4 Example



Vehicle.java

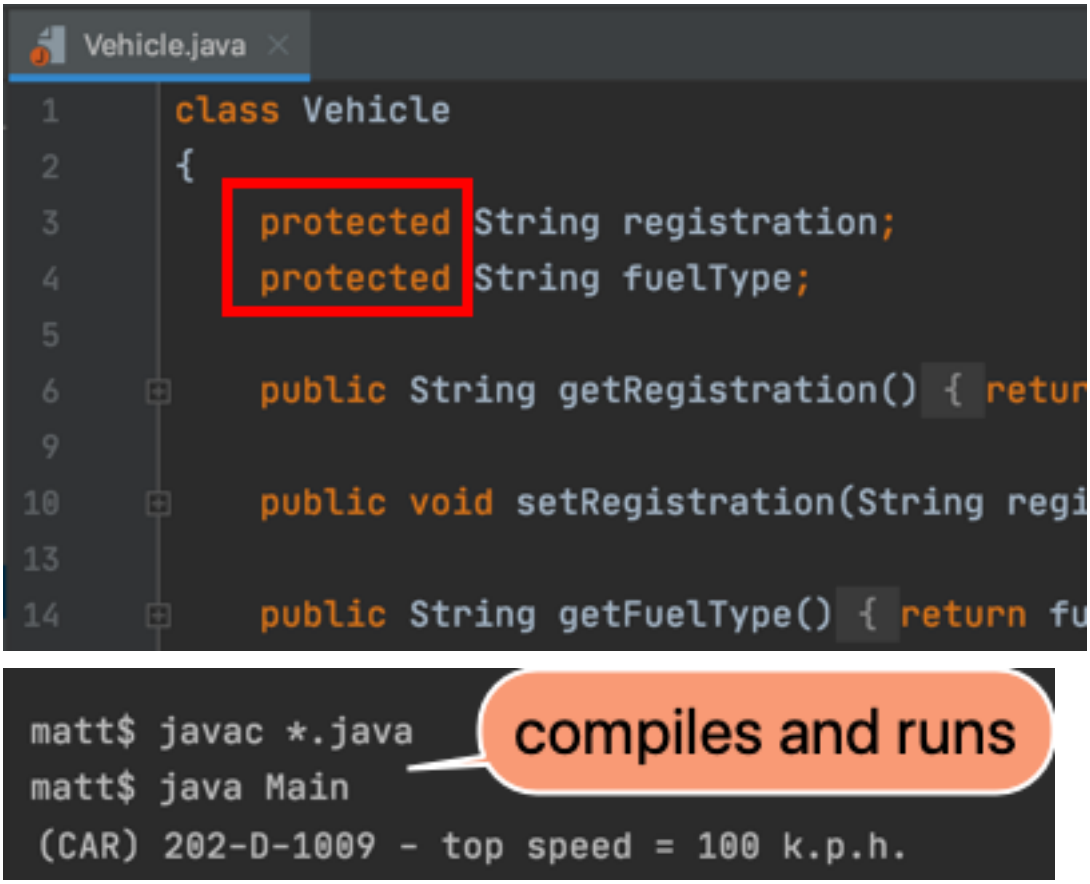
```
1   class Vehicle
2   {
3       private String registration;
4       private String fuelType;
5
6       public String getRegistration() {
7           return registration;
8       }
9
10      public void setRegistration(String registration) {
11          this.registration = registration;
12      }
13
14      public String getFuelType() {
```




```
matt$ javac *.java
Car.java:15: error: registration has private access in Vehicle
        return "(CAR) " + this.registration
                                ^
1 error
```

2.5 Here are 3 potential solutions

1. make variables `public` ✗
   - Not good, since lose encapsulation / validation setters etc.
2. Use public accessor methods to get/set variable values ✓
   - Okay – but in many cases subclass methods should be allowed to `directly access` inherited variables/methods
   - but public methods available ANYWERE in software system
     - so may not be what we wish
3. Make variables `protected` ✓
   - Same as private BUT also allows `subclasses` to access inherited protected properties and methods



2.6 UML class diagram notation for `protected`

The **#** (hash) symbol indicates protected visibility in UML class diagrams. See Figure 2.1.

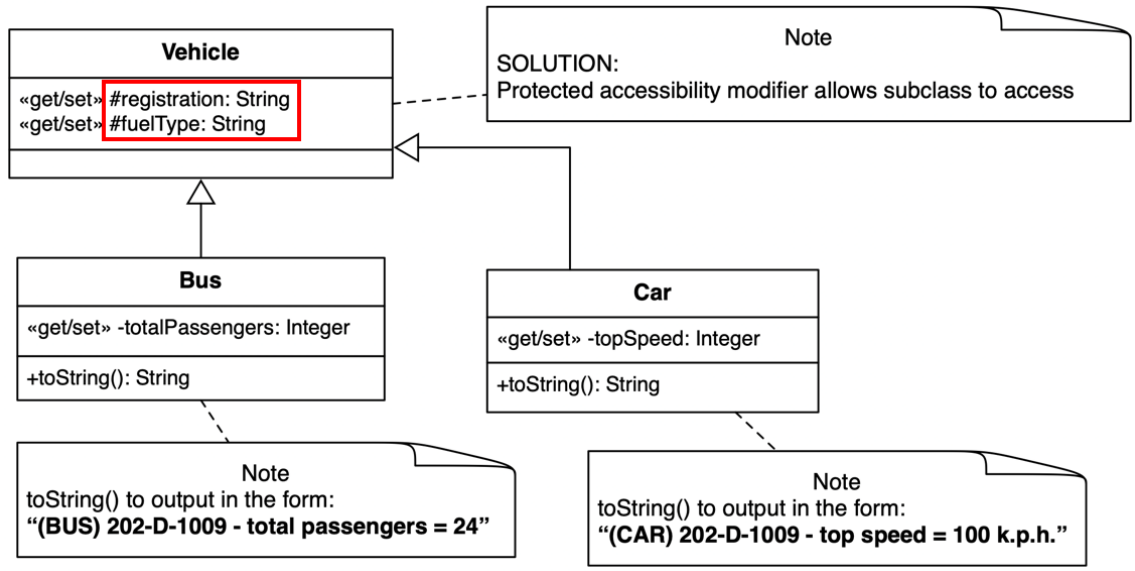


Figure 2.1: Class diagram showing protected members with # symbol.

2.7 Summary & Conclusions

`protected`

- Subclasses cannot access `private` variables or methods
- But they can access `protected`
- much better than making everything `public` just to allow access from methods of subclass

2.8 Exercise - use public accessor methods in subclass code

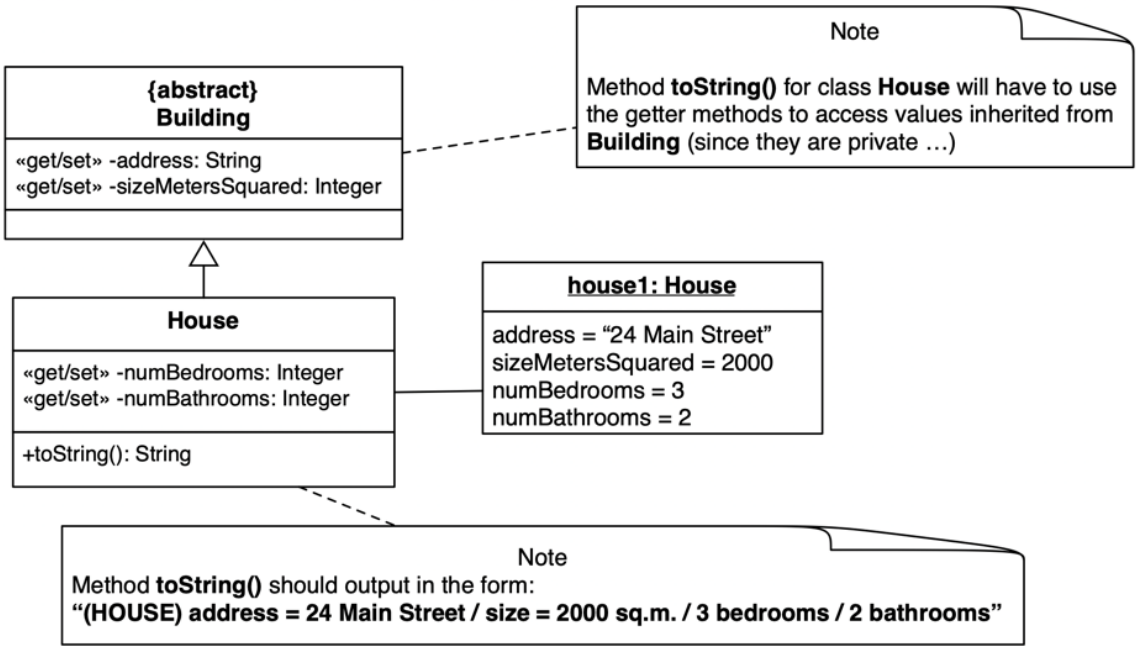


Figure 2.2: Class-Object diagram for `toString()` method of class `Building`.

AIM:

- reflect on when to use the public getter/setter methods to overcome `private` variables inherited from a superclass

ACTION:

- duplicate your folder for the previous exercise (abstract `Building`)
- class `House` (File: `House.java`)
  - add a `toString()` method to your `House` class, that outputs in the following form:

    `(HOUSE) address = 24 Main Street / size = 2000 sq.m. / 3 bedrooms / 2 bathrooms`
- class `Main` (File: `Main.java`)
  - method `main()`:
    - use the setter methods to create `house1`, and instance-object of class `House` with values:
      - address = `24 Main Street` / size = `2000` sq.m. / `3` bedrooms / `2` bathrooms
    - use `System.out.println()` to output the details of `house1`

OUTPUT:

```
$ java Main
(HOUSE) address = 24 Main Street / size = 2000 sq.m. / 3 bedrooms / 2 bathrooms
```

HINT: Since the properties in `Building` are `private`, you'll have to use the public `get<>()` methods in the `toString()` method of class `House`

2.9 Exercise - use `protected` visibility to allow subclass methods to directly access inherited properties

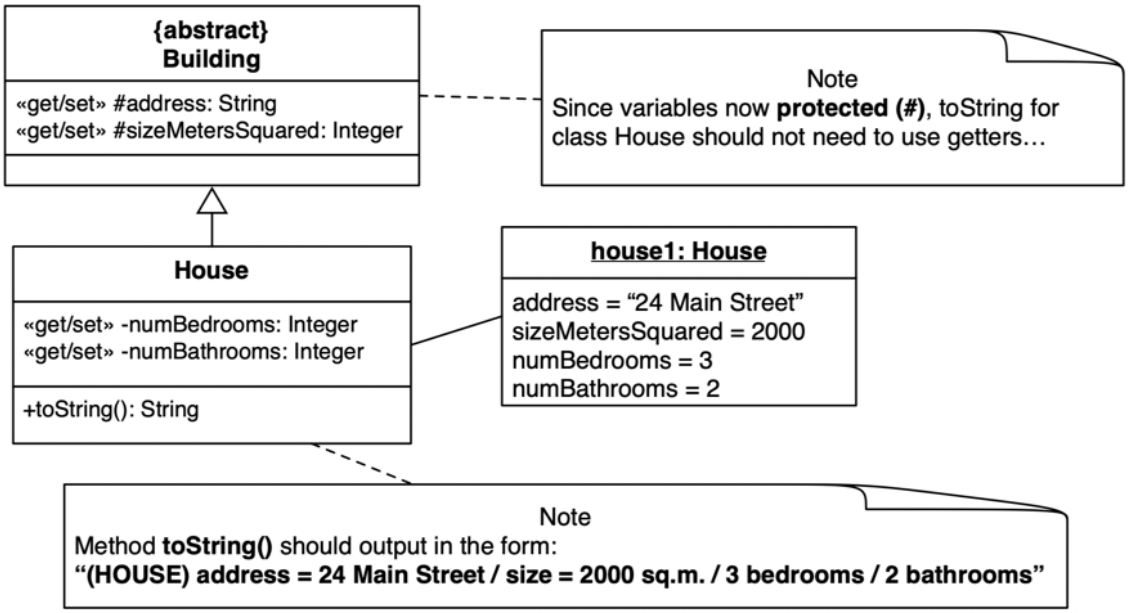


Figure 2.3: Class-Object diagram for `toString()` method of class `Building`.

AIM:

- reflect on when to use the `protected` accessibility modifier to allow methods in subclasses to directly access inherited variables

Do the following:

- duplicate your folder for the previous question (abstract `Building` using get/set methods)
- class `Building` (File: `Building.java`)
  - change the visibility of the variables for class `Building` to `protected`
- class `House` (File: `House.java`)
  - improve the `toString()` method of class `House`, to directly access the inherited `protected` variables for `address` and `sizeMetersSquared`
    - so now your code should **not** need to use any getter methods, since it has direct access to the inherited variables
- the output should be the same…

3

Java Packages

3.1 Keep in mind how folders/directories work in computer filesystem

In a computer filesystem we know:

- the name of a file
- the location (path) of a file

The combination of path+filename allows us to uniquely locate a file.

It is possible to have 2 or more files with the same name, in different locations on the computer disk.

This is how packages/namespaces work with classes:

- we can uniquely reference a class through its package+classname
- so we can have 2 classes with the same name (identifier), as long as they are declared in different packages (namespaces)

3.2 Java keywords: `package`, `import`

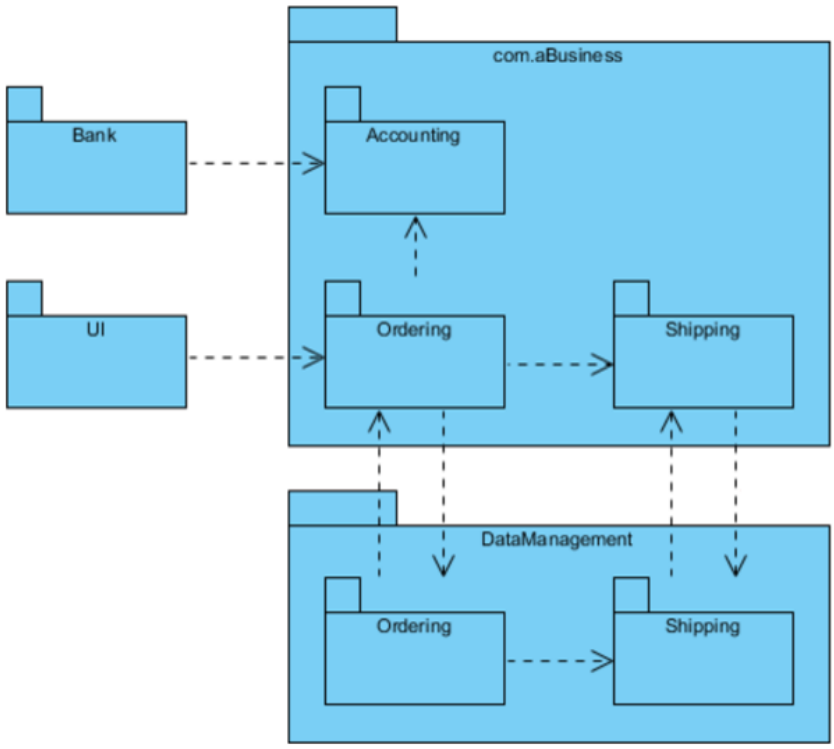
(and `public`)

3.3 System - Subsystems

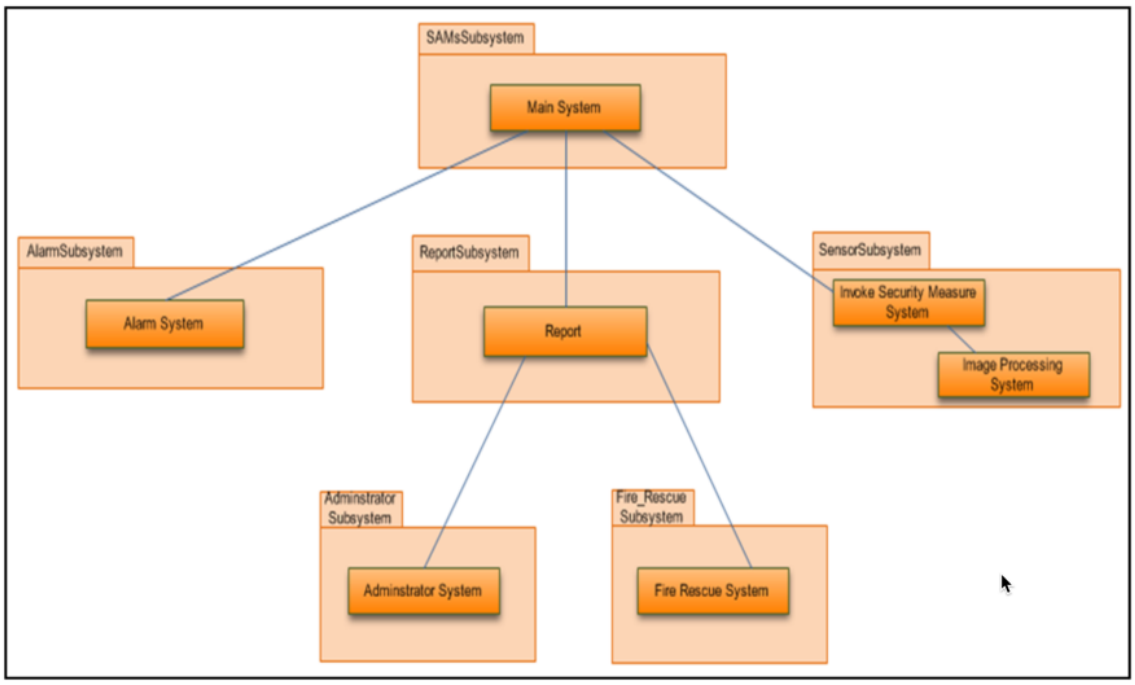
Any complex system can be decomposed into *subsystems*

- each subsystem can be modeled as a *package*

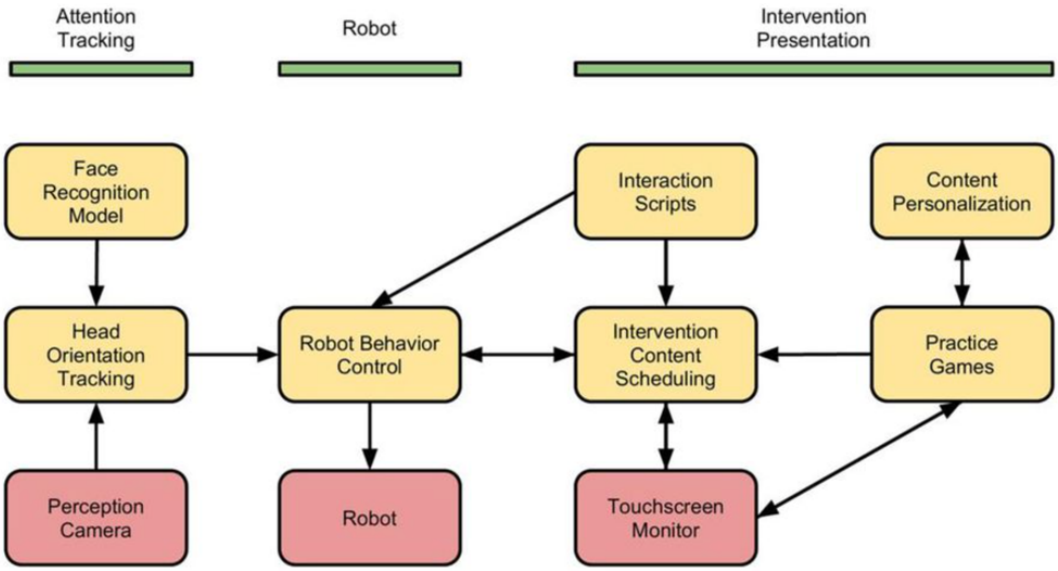
3.4 Sales-Shipping-Finance system sub-systems



3.5 Business system sub-systems



3.6 Robotics system sub-systems



3.7 UML concept of "namespaces"

Quite simply, think of namespaces (and Java packages) as 'virtual folders' for classes

- so a several related classes can be collected together as a library
  - and these related classes can have more access to each other's members, than code outside the packate ...
- so we can have 2 or more classes of the same name, and be able to distinguish between them

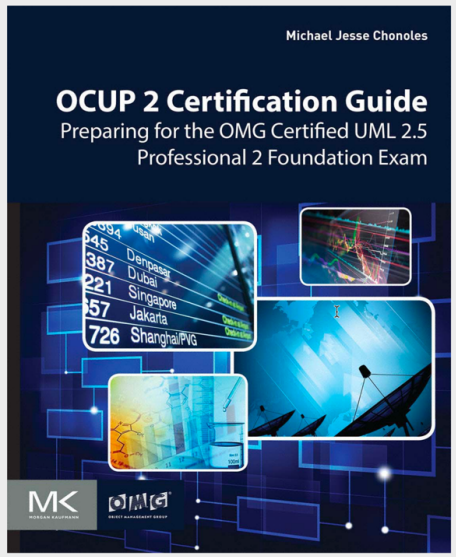
4.6.3.4 Namespace

What makes the given <kind> values the possible values? These are all namespaces in UML that are capable of holding other elements. A namespace is a software development concept. It is a set of unique names that identify entities. A namespace does not allow duplicate names. If a name appears more than once, it must refer to the same thing.

A package is a namespace. The symbol for a package is rectangle body with a tab in the upper left, which usually contains the name of the package, see Fig. 4.11. The elements inside the package cannot have clashing names. A class is also a namespace. It has a set of attributes (see the class symbol, **Book**, inside the figure), each with a name, and no duplicate names are allowed for the attributes.

A package is primarily an organization feature for a UML model. Consider it similar to a folder or directory in an operating system. These are also namespaces. The operating system will not allow you to have two files with identical names within the same directory/folder. A package will not allow two elements (of the same type) with the same name inside the package. If an element appears twice in a package, they are just representations of the same thing.

From OCUP 2 UML 2.5 Certification Guide book:

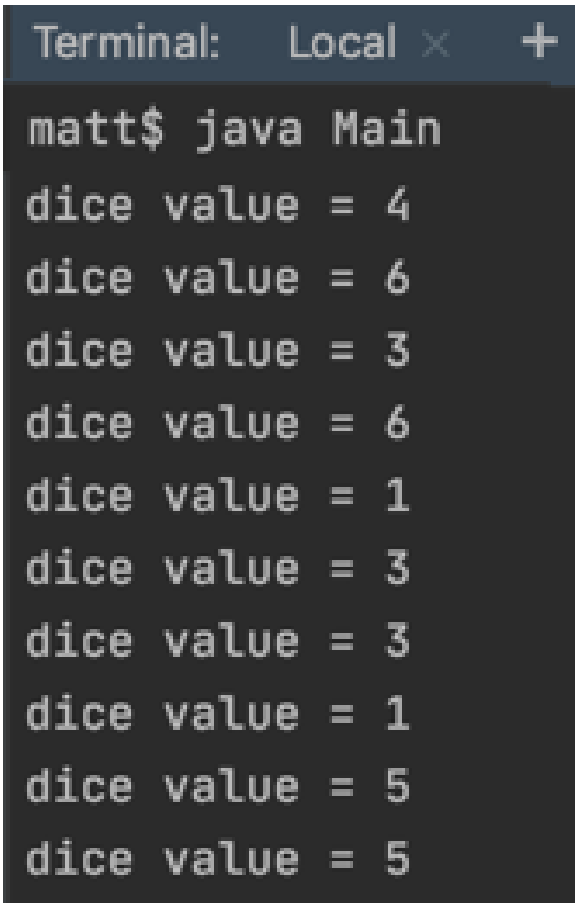


3.8 Packages - a tidy way to store-share-name classes

If you've written a Java program with an `import` statement,you've used packages

Here's one, using class `Random` from package `java.util`

For simulating dice rolls...




```
import java.util.Random;

class Main
{
    public static void main(String[] args)
    {
        int numRolls = 10;
        for(int i = 0; i < numRolls; i++){
            int randomZeroToFive = new Random().nextInt( bound: 6);
            int diceRoll = 1 + randomZeroToFive;
            System.out.println("dice value = " + diceRoll);
        }
```

3.9 3 ways to use a class defined in a package

There are 3 methods to get Java to let you use a class declared in a package

1. Refer to the `fully qualified class` – i.e. with package path before class name
    - `r = new java.util.Random();`
2. Import `just the class` from the package
    - `import java.util.Random;`
    - `... r = new Random();`
3. Import `ALL the classes` from a package, using*
    - `import java.util.*;`
    - `... r = new Random();`

3.9.1 Method 1. Refer to the fully qualified class

```
{
    ...new java.util.Random()
    ...
}
```

```
class Main
{
    public static void main(String[] args)
    {
        int numRolls = 10;
        for(int i = 0; i < numRolls; i++){
            int randomZeroToFive = new java.util.Random().nextInt( bound: 6);
            int diceRoll = 1 + randomZeroToFive;
            System.out.println("dice value = " + diceRoll);
        }
    }
}
```

3.9.2 Method 2. Import just the class from the package

```
import java.util.Random;
...
{
    ... new Random()
    ...
}
```

```java
import java.util.Random;

class Main
{
    public static void main(String[] args)
    {
        int numRolls = 10;
        for(int i = 0; i < numRolls; i++){
            int randomZeroToFive = new Random().nextInt( bound: 6);
            int diceRoll = 1 + randomZeroToFive;
            System.out.println("dice value = " + diceRoll);
        }
    }
}
```

3.9.3 Method 3. Import all the classes from a package

```java
import java.util.*;
...
    {
        ... new Random()
        ...
    }
```

```java
import java.util.*;

class Main
{
    public static void main(String[] args)
    {
        int numRolls = 10;
        for(int i = 0; i < numRolls; i++){
            int randomZeroToFive = new Random().nextInt( bound: 6);
            int diceRoll = 1 + randomZeroToFive;
            System.out.println("dice value = " + diceRoll);
        }
    }
}
```

3.10 Creating your own packages

3.10.1 How to declare a package

Choose package name – always in `lower case`:

- unique (reverse web domains are common choice) – but a bit long-winded...

  - `ie.tudublin.informatics.example`
  - `com.amazon.cart.tax`

- We can use single word examples for now (we're not publishing globally…) e.g.
  - `tudublin`
  - `year2`
  - `mattsmith`

Write `package <name>` before declaring class

```
// File: Student.java
package tudublin;

public class Student
{
    ...
}
```

Create class in `folder` matching package name

Compile `*.java` for `all` folders containing your code

- (or be careful about ensuring all *changed* files are re-compiled

3.11 NOTE: A class can only be in one package

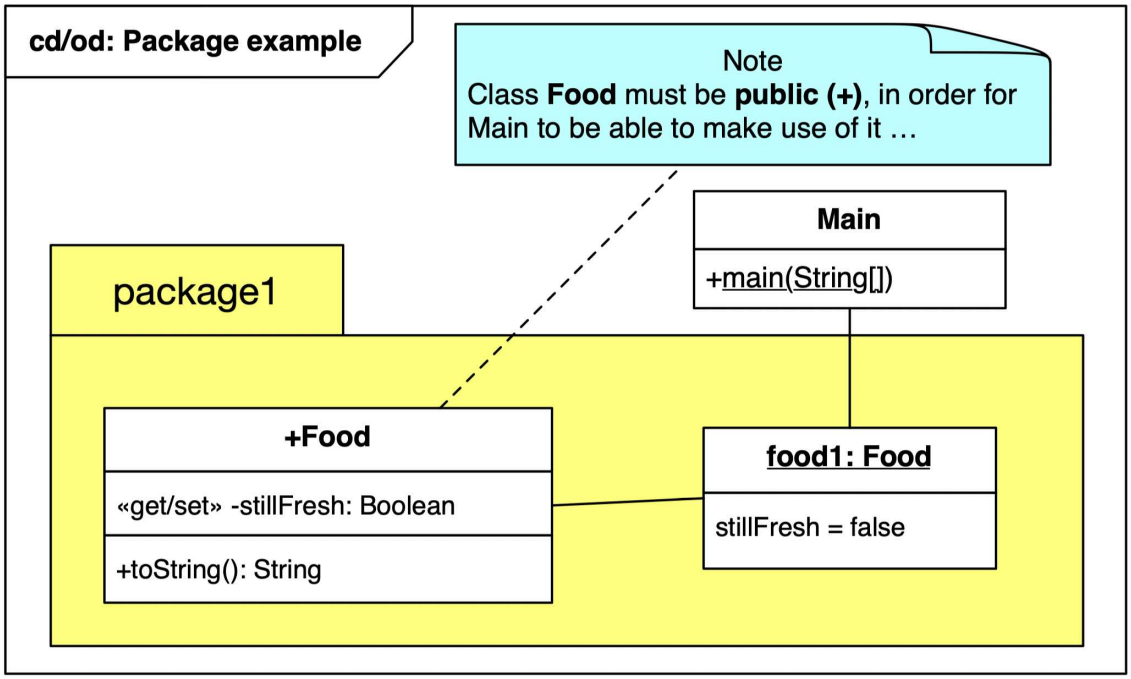
Just as a file cannot be in more than one folder/directory, a class can only be in **ONE** package

3.11.1 Example 1: "Food" in "package1"

A `Main` class (not in any package) using a public `Food` class in `package1`.

NOTE:

Class `Food` must be `public` if it is to be used from outside the package (i.e. from class `Main`)

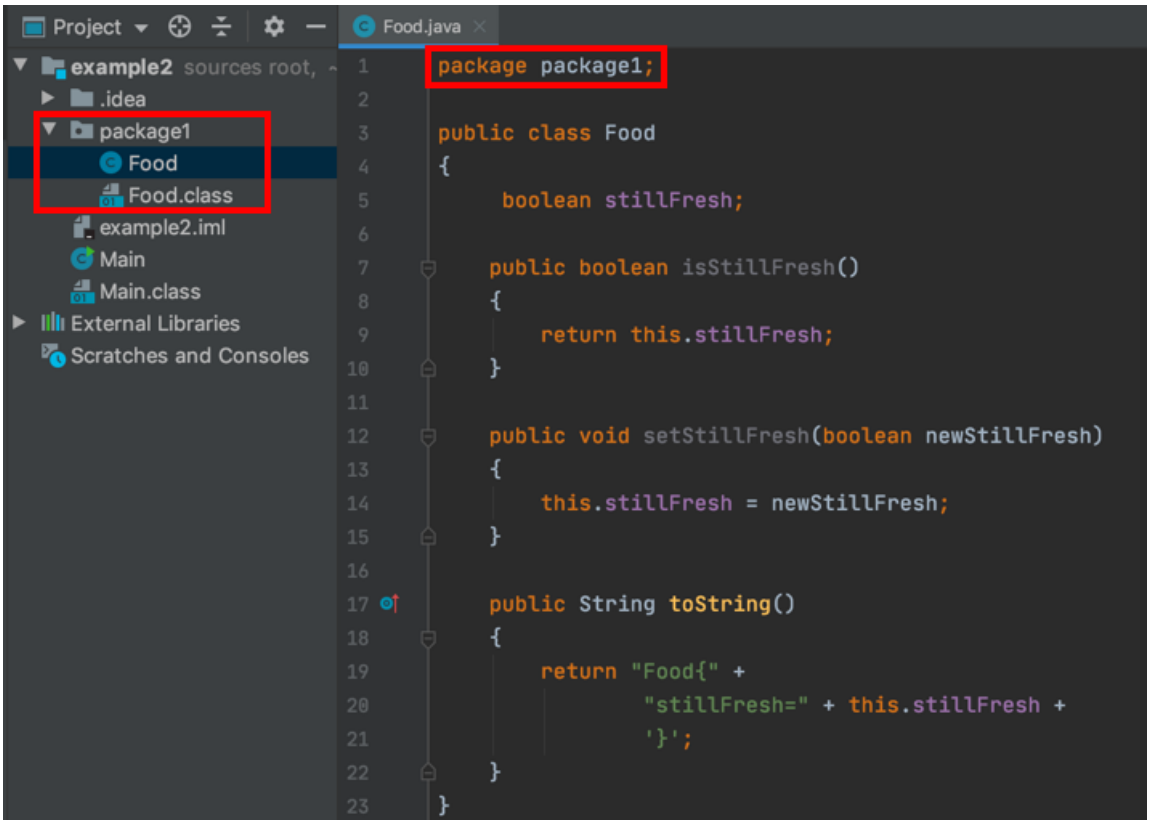


3.11.2 What we'll do

- Create folder for our package: `package1`
  - in folder `package1` create Java class declaration file `Food.java`
    - declare `public` Java class `Food`
    - Declaring code in class `package1` with: `package package1;`
    - Compile file `Food.java`, so in folder `package1` we now have file `Food.class`
- Create-compile-run `Main.java` that uses class Food from `package1`
  - write:

```
import package1.Food;
...
Food food1 = new Food();
```

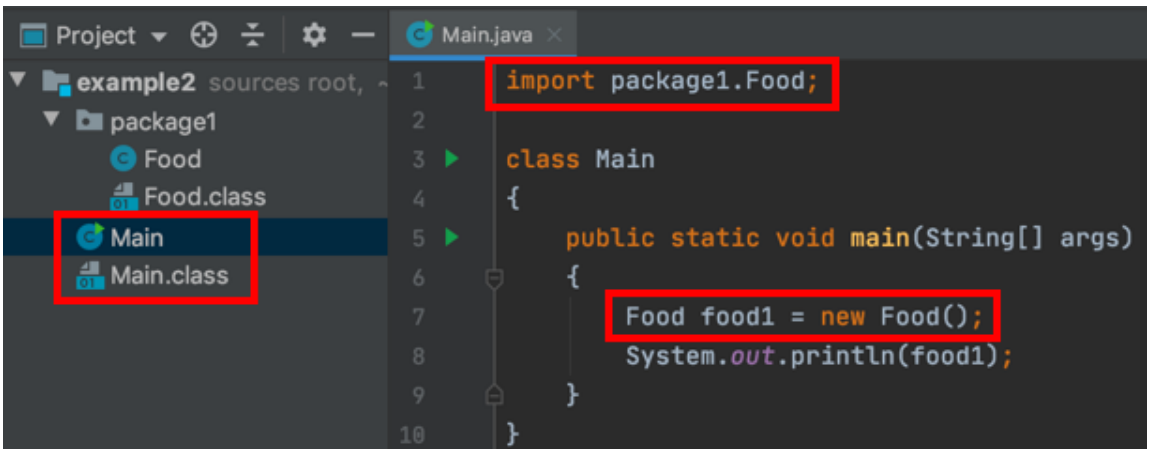
Let's declare Class `Food` in `package1` (file must be created inside directory `package1`):



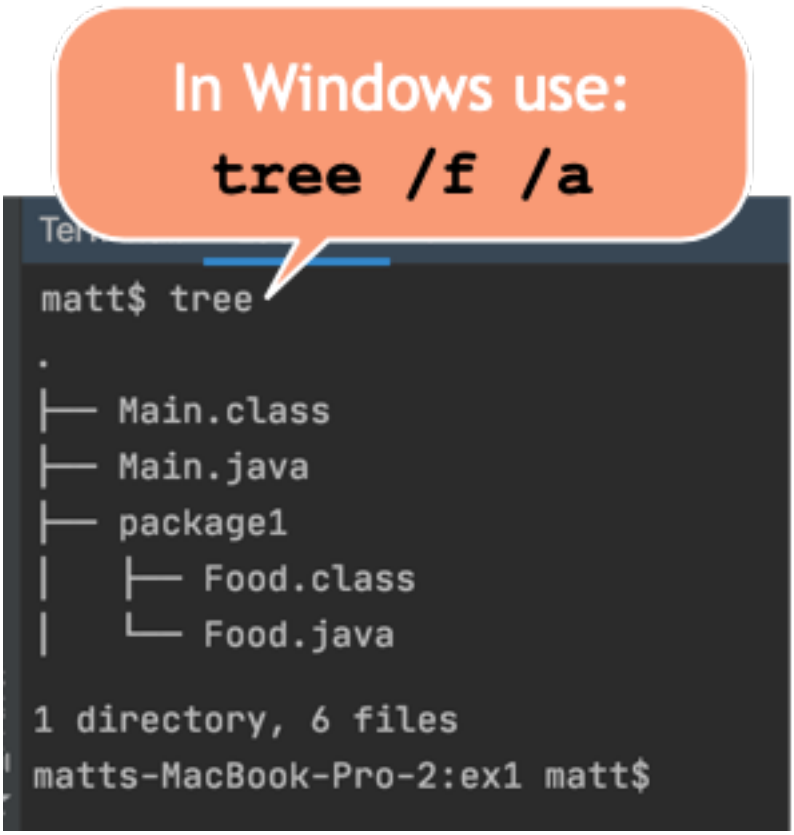
Now in class `Main` (this class is **not** inside directory `package1`):

- we can import `package1` - to have access to any public classes in this package.

So in our `main(...)` method we are now about to create a `Food` object, using the declaration of class `Food` in package `package1`:



3.12 Examine your directory tree



- Directory " . " (the current directory)
  - `Main.java`
  - `Main.class`
- Directory " `package1` "
  - `Food.java`
  - `Food.class`
- (note there may be additional IDEproject configuration files..)

A D.I.Y. (Do-It-Yourself) build tool

3.13 Compile ALL files in one line using `@`

With non-trivial projects, the sequence in which source files are compiles becomes important. To save time and avoid mistakes, we can list the sequence of files to compile in a text file, and use the `@` prefix when using the `javac` Java compiler, to compile the files in the sequence they appear in the text file…

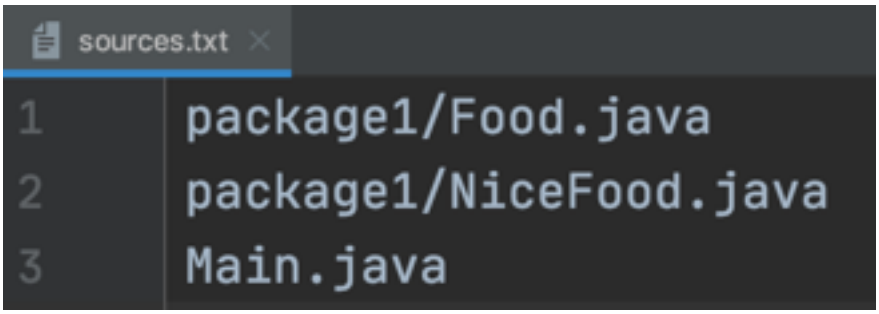
```
$ javac @sources.txt
```

3.13.1 Having to remember to compile `Main.java`, and then all the package folder files can be annoying time-wasting....

- The term "building" refers to compiling and linking together different software components into the final working application
- When using packages, we now have to do some building
  - we should compile all our package classes first, before compiling our `Main` class
- Rather than:
  - `javac package1/*.java`
  - `javac package2/*.java`
  - `javac Main.java`
  - `Etc.`
- A single statement is better!
  - `javac @sources.txt`

3.13.2 Compiling multiple files with `@<file>`

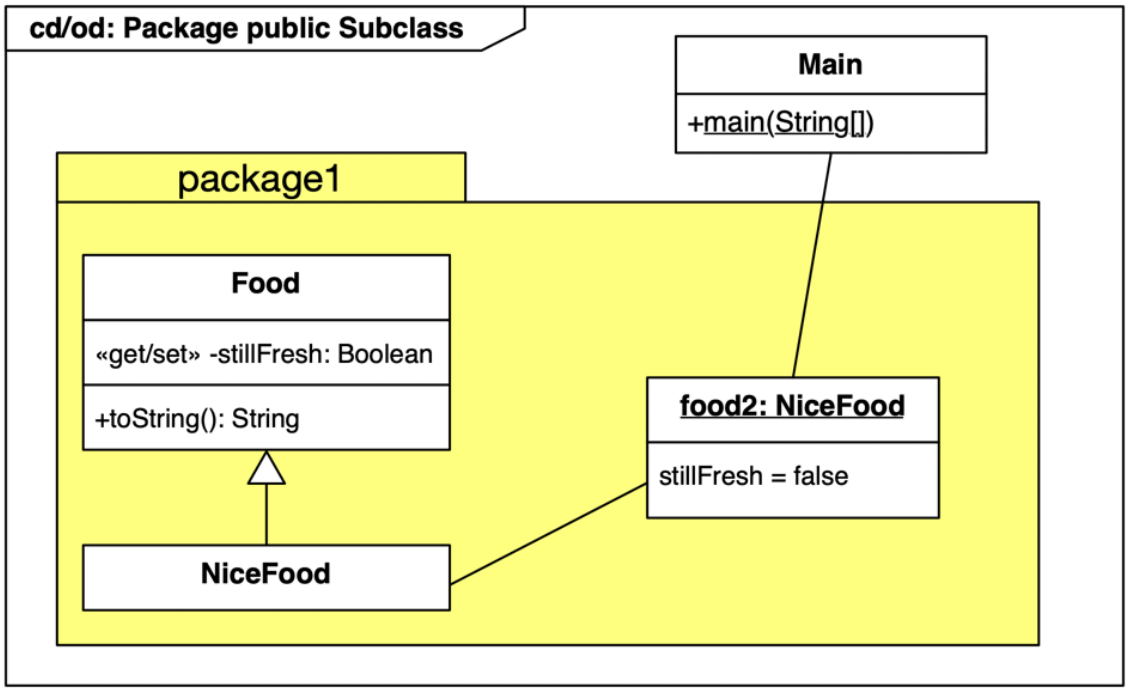
- Create (and keep up-to-date) file `sources.txt` (or whatever you want to call it)
- Java will compile all the files from your list:
  - `javac@sources.txt`




```
matt$ javac @sources.txt
matt$ java Main
(NICE FOOD) still fresh = false
```

3.14 Example 2: "NiceFood" subclass in "package1"

A `Main` class, using a `NiceFood` class in `package1`



3.15 What we'll do

Create new Java class: `package1/NiceFood.java`

- Declaring `NiceFood` as subclass of class `Food`:
  - `NiceFood extends Food;`
- Do **not** write this as a `public` class
  - the implications of this not being a `public` class is what we'll explore here …
- Compile class `NiceFood.java`
  - so in folder `package1` we should now have file `Food.class`
- edit-compile-run `Main.java` that tries to class `NiceFood` from package1
  - write: `NiceFood food2 = new NiceFood();`

3.15.1 `NiceFood.java`

however, we need to explicitly make classes public, so that it is available outside the package …

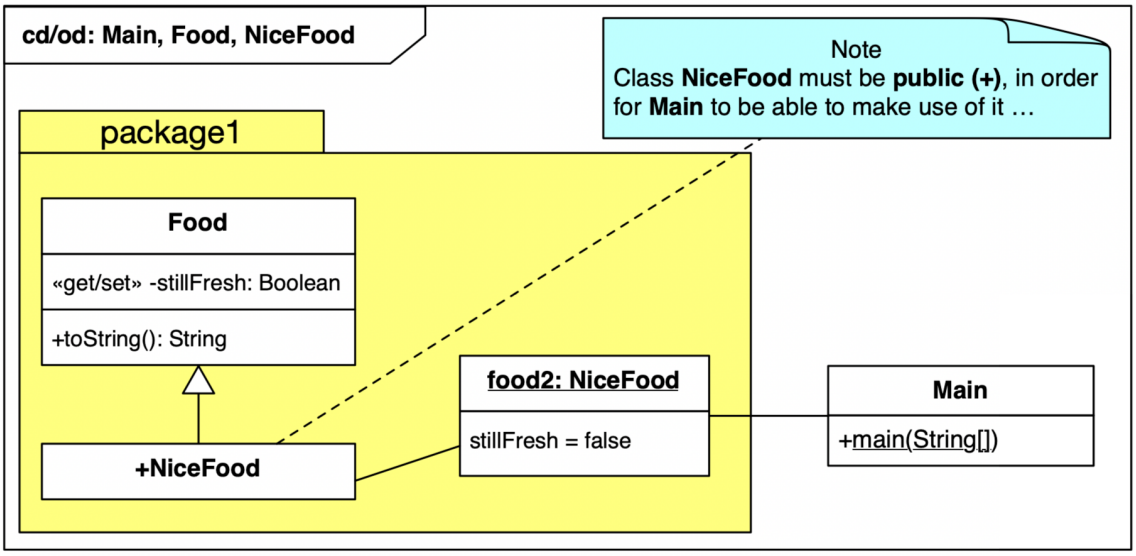
NiceFood.java

```
1   package package1;
2
3   class NiceFood extends Food
4   {
5       public String toString()
6       {
7           return "still fresh = " + this.stillFresh;
8       }
9   }
```

3.15.2 Error – can't access non-public class outside of package ...

Terminal:

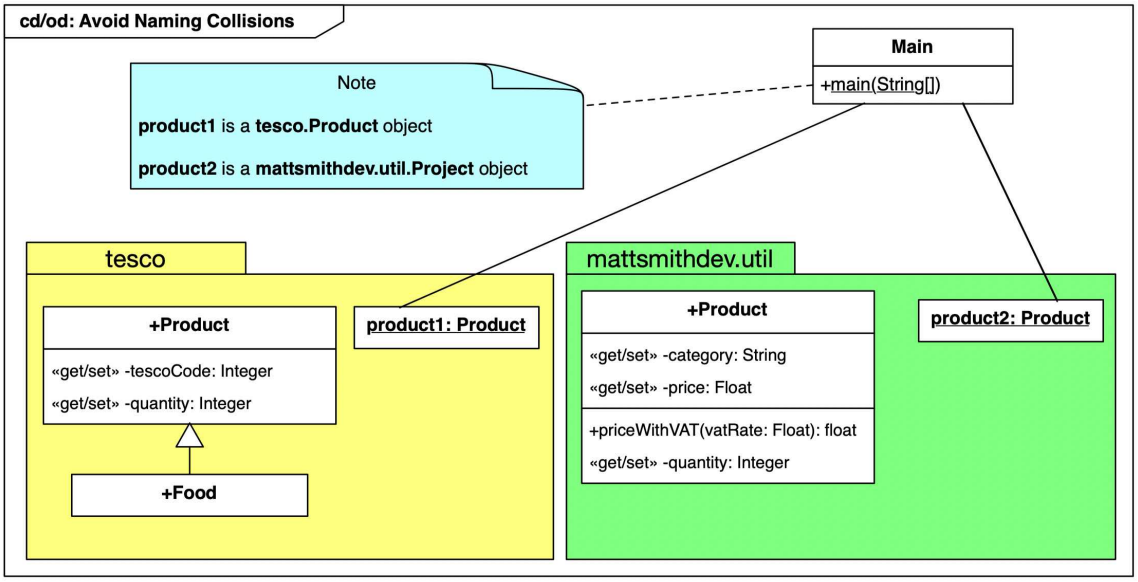
```
matt$ javac Main.java
Main.java:1: error: NiceFood is not public in package1;
                    cannot be accessed from outside package
import package1.NiceFood;
                ^
Main.java:7: error: cannot find symbol
        NiceFood nf1 = new NiceFood();
        ^
  symbol:   class NiceFood
  location: class Main
Main.java:7: error: cannot find symbol
        NiceFood nf1 = new NiceFood();
                           ^
```



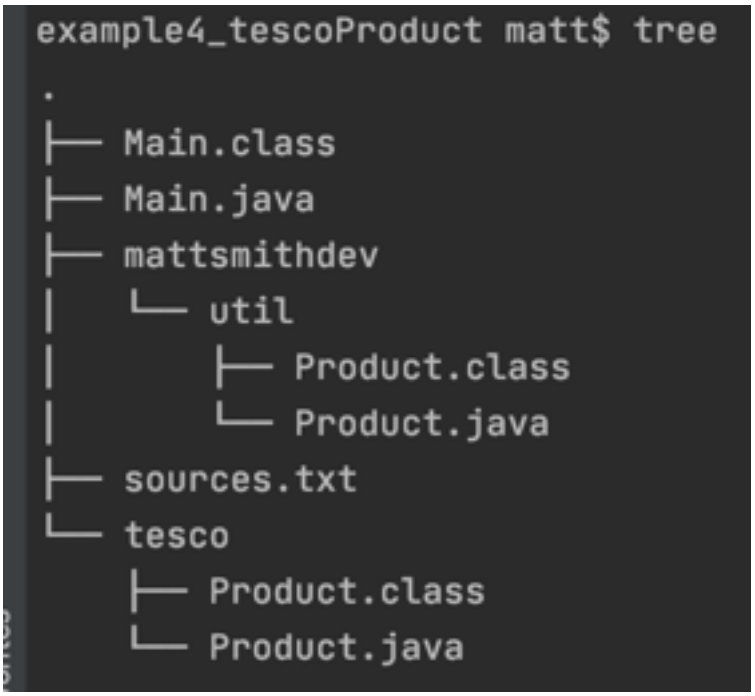
3.16 Avoiding Naming Collisions

The case of the 2 classes named Product …

Situation where 2 classes have same identifier (e.g. Product) but in different packages…



- Create folders for packages
- Write "package" statements for classes
- Compile everything
  - HINT: use `javac sources.txt`



3.17 No "alias" in Java

When working with 2 classes that have the same name, but are declared in different packages, some programming languages (such as PHP) offer a way to declare an 'alias'. However, Java does not offer any such feature. You can read about how cross people are on Stack Overflow …

- https://stackoverflow.com/questions/2447880/change-name-of-import-in-java-or-import-two-classes-with-the-same-name

So when working with 2 classes with the same name in Java, we can import one class, and then have to use the fully qualified class name for the other…

```java
import tesco.*;

class Main
{
    public static void main(String[] args)
    {
        Product product1 = new Product();
        mattsmithdev.util.Product product2 = new mattsmithdev.util.Product();

        System.out.println(product1);
        System.out.println(product2);
    }
}
```

3.18 Summary & Conclusions

Packages

- Java keywords: `package`, `import`
- Grouping together related classes
  - Organised in their own directories: often reverse web domains: `ie.mattsmith`
- Defining a unique `namespace` with keyword `package`
  - Prevents **naming collisions**
  - Can have 2 classes with SAME NAME in different packages
- Get access to a package class or all classes by `import` ing package
- Classes in package need to be `public` for access from outside package…

3.19 Further reading

Learn more about Java packages at:

- Oracle Java docs about packages:
  - https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html
- the `java.io` package declaring the `FilterOutputStream` class used by methods such as `println(...)`
  - https://docs.oracle.com/javase/7/docs/api/java/io/PrintStream.html#println(boolean)

- using built-in packages like `java.lang.system`:
  - https://docs.oracle.com/javase/7/docs/api/java/lang/System.html
- TutorialsPoint Java packages tutorial:
  - https://www.tutorialspoint.com/java/java_packages.htm

3.20 Exercise - declaring a class in your package `tudublin`

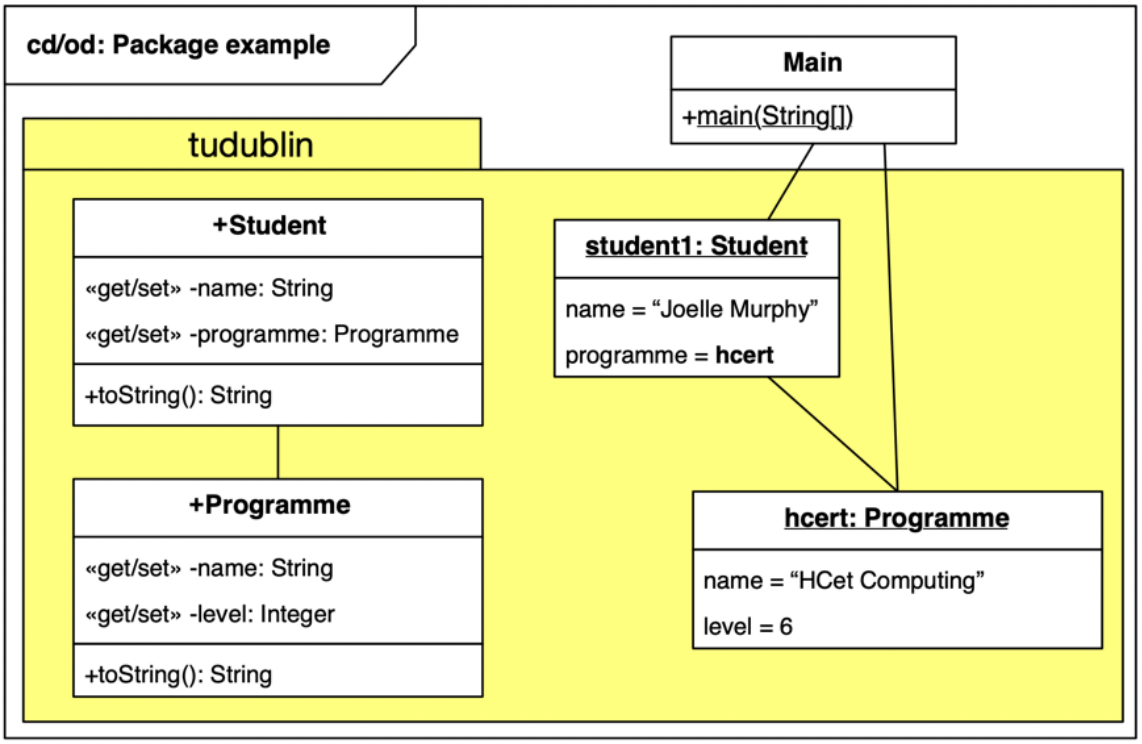


Figure 3.1: Class-Object diagram for package `tudublin`.

AIM:

- practice declaring classes in packages, and working with classes defined in other packages

ACTION:

- create a folder for package `tudublin`
- in your folder, create the following 2 classes for package `tudublin`
  - class `Programme` (File: `tudublin/Programme.java`)
    - public class `Programme` with private properties and public getters/setters:
      - this class represents University **programmes** of study (courses, like the Higher Certification in Computing and the BSc(Hons) degree in computing)
      - `name`: String
      - `level`: Integer
  - class `Student` (File: `tudublin/Student.java`)
    - public class `Student` with private properties and public getters/setters:

        - `name`: String
        - `programme`: Programme (a reference to a Programme object)

- class `Main` (File: `Main.java`)
  - method `main()`:
    - import all classes from package `tudublin`
    - create a `Programme` object `hcert` with name = `HCert Computing` and level = `6`
    - create a `Student` object `student1` with name = `Joelle Murphy` and programme linking to object `hcert`
    - print out each object via the default `toString()` which should confirm the package and class name of the object
- **compile** all classes in your package folder first, e.g. `javac tudubln/*.java`
  - on Windows you'll need to use the backslash `\` character to separate folder names and file names …
- then compile and run your `Main` class

OUTPUT:

```
$ javac tudubln/*.java
$ javac *.java
$ java Main
tudublin.Student@6d06d69c
tudublin.Programme@7852e922
```

ACTION - MORE:

- Now add custom `toString()` methods to each class, so the output describes each object as shown
  - the `toString()` method for `Student` can make use of the `toString()` method for `Programme` objects …
  - try to get you output to exactly match the following:
- again, compile the classes in the package folder first, then compile and run your `Main` class

OUTPUT:

```
$ java Main
(Programme) name = HCert Computing / level = 6
(Student) name = Joelle Murphy /  programme = (Programme) name = HCert Computing (level 6)
```

HINT:

```
hcert = new Programme();
...
student1.setProgramme( hcert );
```

3.21 Exercise - classes with same name in different packages

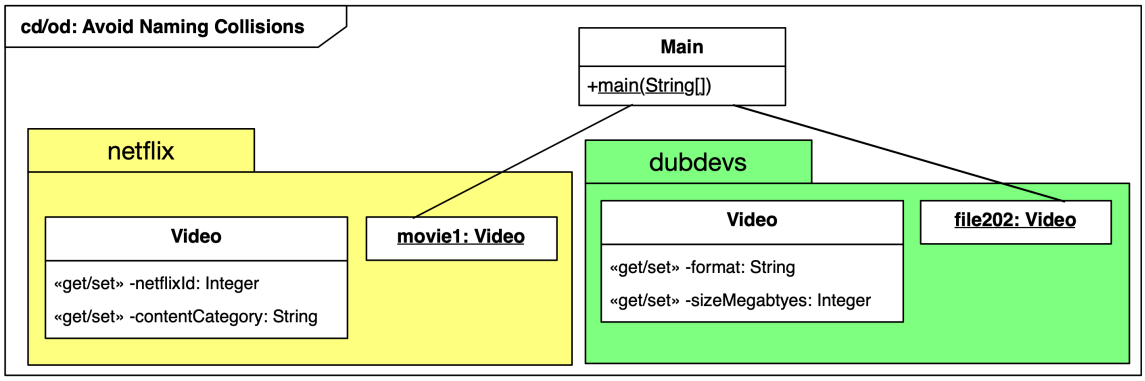


Figure 3.2: Class-Object diagram for packages `netflix` and `dubdevs`.

AIM:

- use packages to avoid naming collisions where 2 classes have the same name

ACTION:

- imagine a Dublin software development company (with package name `dubdevs`) has been asked to work on some software for streaming video company **Netflix**, and both organisations have a class named `Video` …
- create a folder for package `netflix`
- in your folder `netflix`, create the following class for package `netflix`
  - class `Video` (File: `netflix/Video.java`)
    - class `Video` with private properties and public getters/setters:
      - `netflixId`: Integer
      - `contentCategory`: String
- create a folder for package `dubdevs`
- in your folder `dubdevs`, create the following class for package `dubdevs`
  - class `Video` (File: `dubdevs/Video.java`)
    - class `Video` with private properties and public getters/setters:
      - `format`: String
      - `sizeMegabtyes`: Integer
- class `Main` (File: `Main.java`)
  - method `main()`:

    - use a combination of `import` and fully qualified package-class names to create:
        - `Video` object `movie1` of the `netflix` class
        - `Video` object `file2020` of the `dubdevs` class
    - print out each object - using the default `toString()` method inherited from top-level Java class `Object`
- compile the classes in the package folder first, then compile and run your `Main` class

OUTPUT:

```
$ java Main
netflix.Video@6d06d69c
dubdevs.Video@7852e922
```

3.22 Exercise - explore how cannot create object of non-public class in a package

AIM:

- understand that from a class like `Main` that is outside the package, we can only create object-instances of the classes in the package that are `public`

ACTION:

- make a copy of your solution for the previous exercise
- remove the `public` keyword for the declaration of the Netflix `Video` class
- compile class `netflix/Video.java`
  - it should compile fine
- now re-compile your `Main` class

OUTPUT:

You should get a compiler error, since we cannot now create the `movie1` instance-object of the `Video` class in package `netflix`, since that class is no longer `public` …

4

Inheritance - part 2

4.1 Abstract and Final classes and methods

Once we introduce inheritance into an OO programming language there are some special cases we need to provide programming language features for:

- declaring a class from which objects cannot be created
- preventing a method from being overridden by a subclass
- preventing a class from being extend by any subclass
- requiring subclasses to implement a method "signature"

All the above involve the Java keywords: `abstract` and `final`, which we'll explore in this chapter.

4.2 Begin with a single class (project `cat1`)

Let's progressively develop a class hierarchy around cats and dogs and animals. First we'll begin with a class `Cat` from which we'll create an object.

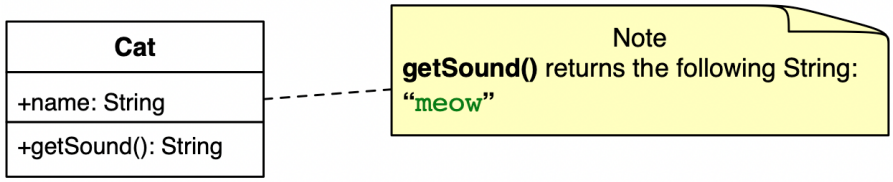


Figure 4.1: Class Diagram for class Cat.

Our `Cat` class has a public String `name` property, and a public method `getSound()` that returns String `meow`. See Figure 4.1:

```
// file: Cat.java
class Cat
{
    public String name;

    public String getSound()
    {
        return "meow";
    }
}
```

Create a `Main` class containing a `main()` method to create a `Cat` object and print out the sound it makes:

```
// file: Main.java
class Main
{
    public static void main(String[] args)
    {
        Cat cat1 = new Cat();
        System.out.println("cat1 makes sound: " + cat1.getSound());
    }
}
```

Running this at the command line we should see:

```
$ java Main
cat1 makes sound: meow
```

4.3 Create superclass `Animal` (project `cat2`)

Animals like cats and dogs have a number of legs, and a name, and make a sound. Let's abstract common properties for all animals to a superclass `Animal`, and make `Cat` extend this general superclass. See Figure 4.2.

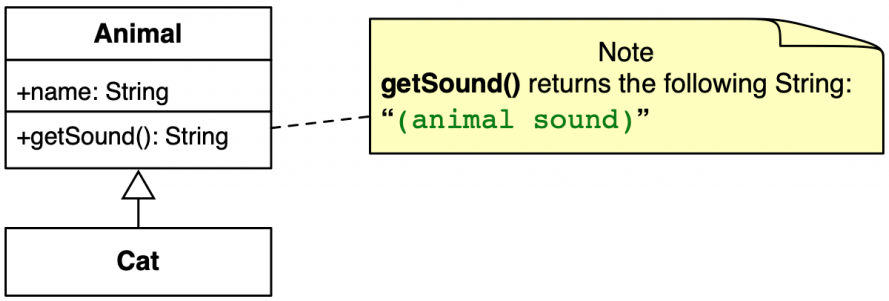


Figure 4.2: class `Cat` extending general class `Animal`.

Here is our general `Animal` class - its `getSound()` method returns a generic `(animal sound)`:

```
// file: Animal.java
class Animal
{
    public String name;

    public String getSound()
    {
        return "(animal sound)";
    }
}
```

Our `Cat` class is very simple now, it simply extends class `Animal`, and inherits the public property `name` and public method `getSound()`:

```
// file: Cat.java
class Cat extends Animal
{
}
```

Create a `Main` class containing a `main()` method are unchanged - we create a `Cat` object and print out its `getSound()` String. Running this at the command line we should see that the `Cat` object `cat1` inherits and returns the generic String from class `Animal`:

```
$ java Main
cat1 makes sound: (animal sound)
```

4.4 make class `Cat` `meow`, by overriding inherited method`getSound()` (project `cat3`)

Our `cat1` object is no longer `meowing` ! Let's fix this by declaring a `getSound()` method for the `Cat` class. This will replace (**override**) the `getSound()` method that it would have inherited from superclass `Animal`. See Figure 4.3.

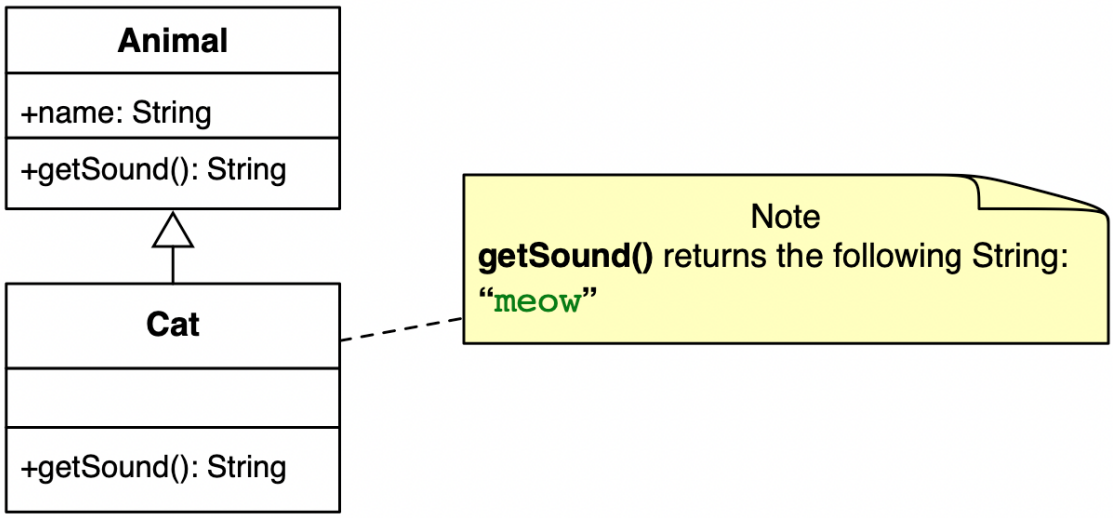


Figure 4.3: class `Cat` extending general class `Animal`.

So we need to refactor (change!) our `Cat` class to declare a `getSound()` method returning `meow`:

```
// file: Cat.java
class Cat extends Animal
{
    public String getSound()
    {
        return "meow";
    }
}
```

Running this at the command line we should see our cat object `meowing` again:

```
$ java Main
cat1 makes sound: meow
```

4.5 Java annotation: `@Override`

- Java offers a special annotation to explicitly state that we are overriding an inherited method `@Override`. So we could write this immediately before our `getSound()` method if we wished:

```
// file: Cat.java
class Cat extends Animal
{
    @Override
    public String getSound()
    {
        return "meow";
    }
}
```

4.5.1 do not confuse "overriding" with "overloading"

Two terms are very similar, but have very different meanings in object-oriented programming:

- **overriding**
  - this refers to when a subclass declares its own specialised declaration, to prevent inheriting that member from a superclass
  - it involves both the **same method name** and the **same sequence and number of arguments**
  - overriding is a power feature of inheritance in most OO languages
- **overloading**
  - this refers to when a subclass declares its own specialised declaration, to prevent inheriting that member from a superclass
  - it involves the **same method name** but a **different** sequence and/or number of arguments
  - overloading is a feature of some OO languages, but not related to inheritance

Both concepts relate to the OO principle of **Polymorphism**:

- the same method name / message name can lead to **different** behaviours

4.6 create an `Animal` object too (project `cat4`)

We have 2 classes, let's create an object of both classes. See Figure 4.4.

Figure 4.4: An `Animal` object and a `Cat` object.

Let's refactor our `main()` method in class `Main` to create an `Animal` object, as well as a `Cat` object; and print out a message about each object's sound:

```
// file: Main.java
class Main
{
    public static void main(String[] args)
    {
        Animal animal1 = new Animal();
        Cat cat1 = new Cat();

        animal1.name = "amy";
        cat1.name = "princiess fifi";

        System.out.println("animal1 makes sound: " + animal1.getSound());
        System.out.println("cat1 makes sound: " + cat1.getSound());
    }
}
```

Running this at the command line we should see our cat object `meow`ing again:

```
$ java Main
animal1 makes sound: (animal sound)
cat1 makes sound: meow
```

4.7 cats are animals (project `cat5`)

A `Cat` object is also an `Animal` object. In OO programming an important concept is that all objects of subclasses can be used wherever an object of a superclass is specified. So we can store a reference to a `Cat` object in a variable that is declared as a reference to an `Animal` class.

See Figure 4.5.

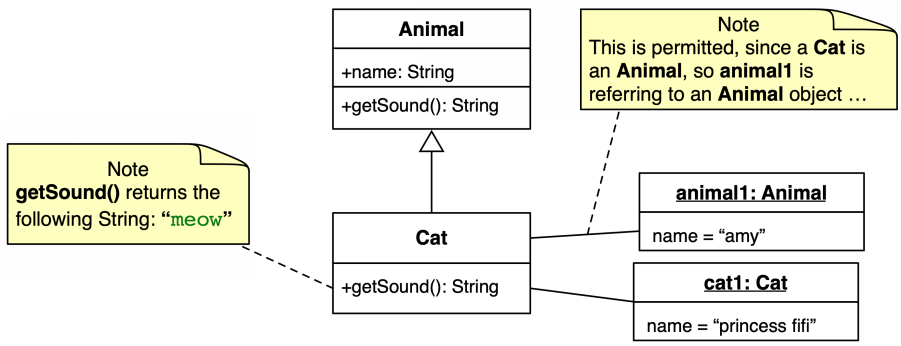


Figure 4.5: An `Animal` reference varaible, referencing a`Cat` object.

Let's refactor our `main()` method in class `Main` to create 2 `Cat` objects, but one will be referred to by a variable specifically for `Cat` objects, and the second will be referred to by variable `animal1` which can refer to any object of class `Animal`, or any of its subclasses.

```
// file: Main.java
class Main
{
    public static void main(String[] args)
    {
        // a Cat object can be referenced by an Animal reference varaible
        Animal animal1 = new Cat();
        Cat cat1 = new Cat();

        animal1.name = "amy";
        cat1.name = "princiess fifi";

        System.out.println("animal1 makes sound: " + animal1.getSound());
        System.out.println("cat1 makes sound: " + cat1.getSound());
    }
}
```

Since both objects are `Cat`, we'll see that both return `meow` when their `getSound()` method is invoked:

```
$ java Main
```

```
animal1 makes sound: meow
cat1 makes sound: meow
```

4.8 Override `toString()` to see animal names (project `cat6`)

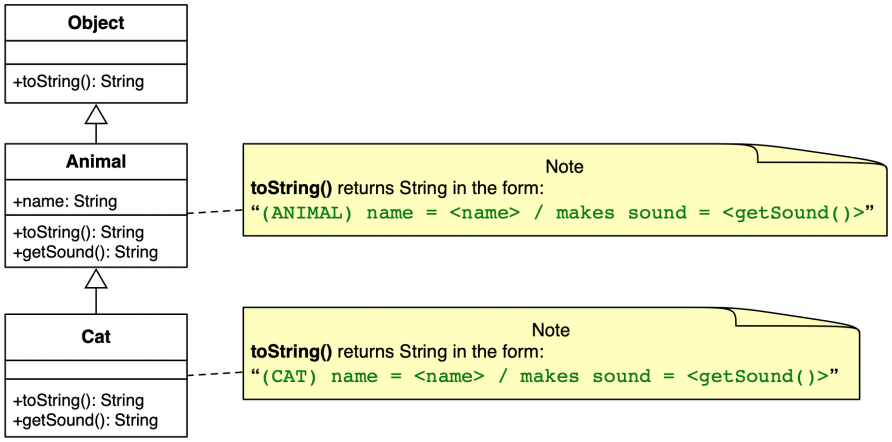


Figure 4.6: Overriding `toString()` for each class.

Rather than doing all the work in `println(...)` statements in our `main()` method, let's override the default `toString()` method inherited from the top-level `Object` class, and make a more meaningful String be returned by both `Animal` and `Cat` objects. See Figure 4.6.

NOTE: While we don't usually show the top-level `Object` class and its toString() method, this is shown in Figure 4.6, to illustrate how class `Animal` is overriding the default `toString()` method it inherits from `Object`; and the class `Cat` is overriding the default `toString()` method it inherits from `Animal`.

Here is our `Animal` class - with its `toString()` method:

```
// file: Animal.java
class Animal
{
    public String name;

    @Override
    public String toString()
    {
        return "(ANIMAL) name = " + this.name + " / makes sound = " + this.getSound();
    }

    public String getSound()
    {
        return "(animal sound)";
    }
}
```

Here is our `Cat` class - with its `toString()` method:

```
// file: Cat.java
class Cat extends Animal
{
    @Override
    public String toString()
    {
        return "(CAT) name = " + this.name + " / makes sound = " + this.getSound();
    }

    @Override
    public String getSound()
    {
        return "meow";
    }
}
```

Let's refactor our `main()` method in class `Main`. We'll make `animal1` an `Animal` object (rather than a `Cat` object), and `cat1` a `Cat` object, so we can see the different behaviour of both classes' `toString()` methods. We can now simplify our `println(...)` statements to print out the object reference variable identifier (variable name) and then invoke the object's `toString()` method:

```
// file: Main.java
class Main
{
    public static void main(String[] args)
    {
        // a Cat object can be referenced by an Animal reference varaible
        Animal animal1 = new Animal();
        Cat cat1 = new Cat();

        animal1.name = "amy";
        cat1.name = "princiess fifi";

        System.out.println("animal1: " + animal1);
        System.out.println("cat1: " + cat1);
    }
}
```

Since both objects are `Cat`, we'll see that both return `meow` when their `getSound()` method is invoked:

```
$ java Main
```

```
animal1: (ANIMAL) name = amy makes sound = (animal sound)
cat1: (CAT) name = princiess fifi makes sound = meow
```

4.9 Abstract classes : Java keyword: `abstract`

Put simply, an `abstract` class is one from which no objects can be instantiated. So this class exists only for subclassing, or declaring static members.

We use the Java keyword `abstract` to declare that a class is abstract (not permitting any object-instances to be created from it).

4.9.1 May make no sense to have objects of some class

Sometimes it would make no sense to have objects of a top-level class

Consider a building company,that might need to store details about 4 types of people:

- Customers
- Managers
- Administrators
- Builders
- It makes sense to generalise name/address/phone variables/methods into a general class `Person`
  - But we would never want a `Person` object in our application– what type of person would that be?
- Java keyword abstract *prevents object-instances being created* from a class

```java
abstract class Person
{
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

```java
class Main
{
    public static void main(String[] args)
    {
        Person p = new Person();
        System.out.println(p1);
    }
}
```

```
matt$ javac *.java
Main.java:5: error: Person is abstract; cannot be instantiated
        Person p = new Person();
                   ^
Main.java:6: error: cannot find symbol
        System.out.println(p1);
                           ^
```

4.10 Make class `Animal` abstract (project `cat7`)

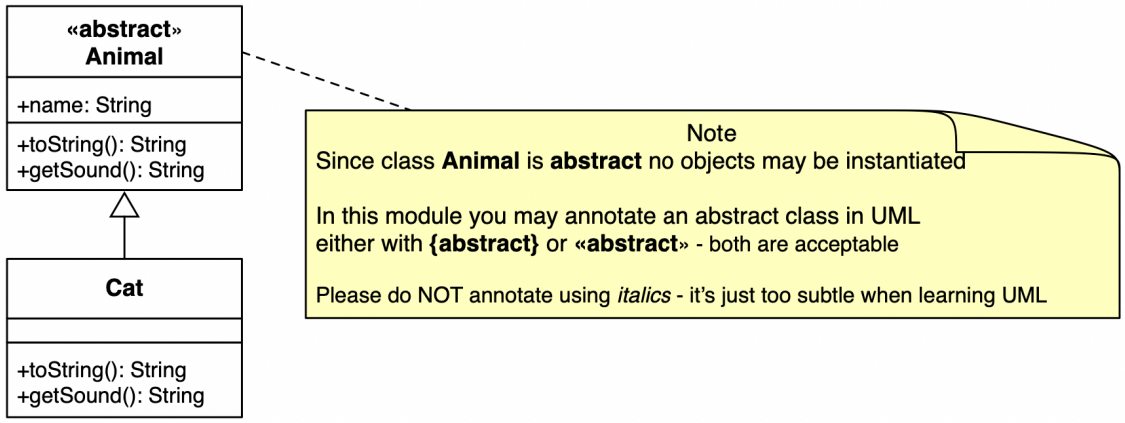


Figure 4.7: `abstract` class `Animal`.

Let's make class Animal abstract. See Figure 4.7.

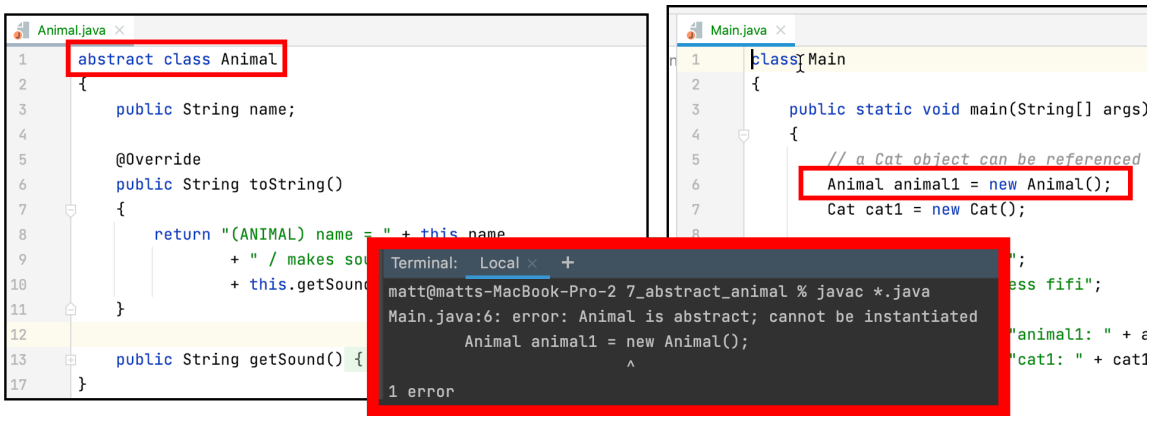


Figure 4.8: Error trying to create instance-object from `abstract` class `Animal`.

Let's make class Animal abstract. When we try to compile our project, we should get an error at the line in our `main(...)` method where we are trying to create an instance-object of `abstract` class `Animal`. See Figure 4.8.

Here is our `Animal` class - declared as `abstract`:

```
// file: Animal.java
abstract class Animal
{
    .... body as before
}
```

When we try to compile with `javac *.java` we'll see an error message:

```
$ javac *.java
Main.java:6: error: Animal is abstract; cannot be instantiated
Animal animal1 = new Animal();
^
1 error
```

4.11 Java `final` classes (UML `leaf` classes) : Java keyword: `final`

Put simply, a `final` class is one for which **no subclasses may be declared**.

We use the Java keyword `final` to declare that a class is final (not permitting subclasses).

4.12 Make class `Cat` final (project `cat8`)

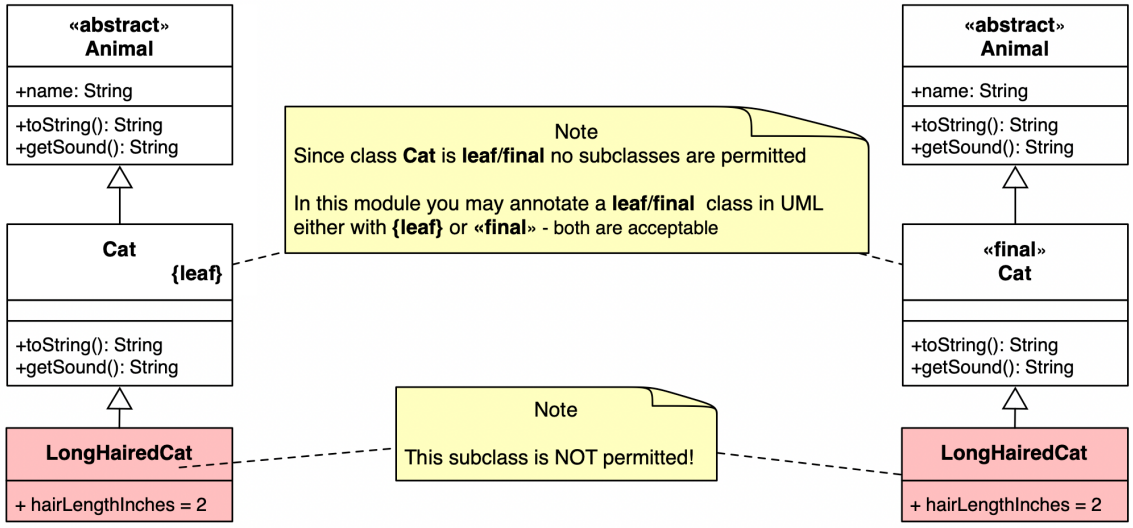


Figure 4.9: `final` class `Cat`.

Let's make class `Cat` a `final` class. See Figure 4.9.

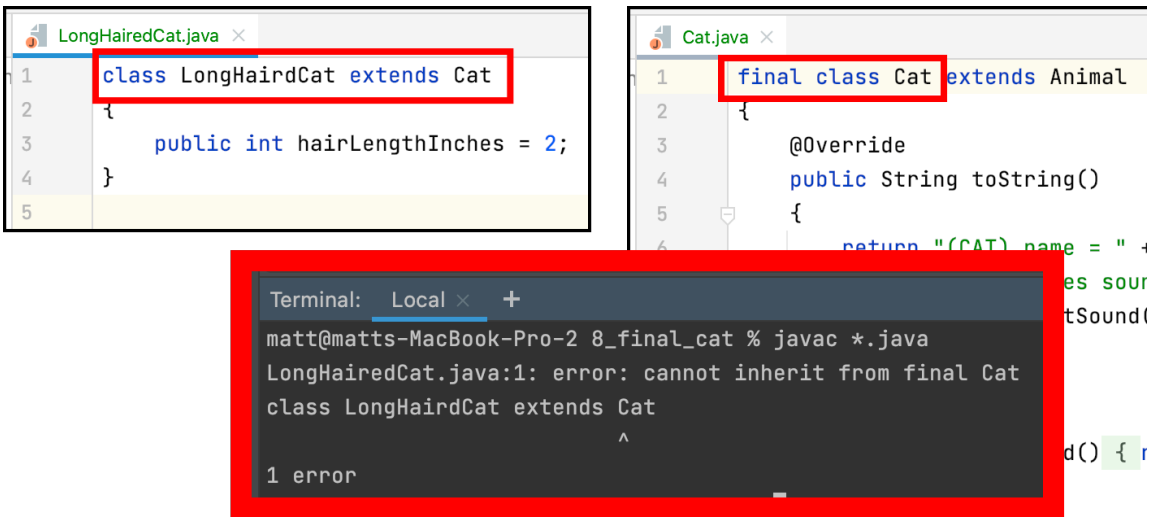


Figure 4.10: Error trying to create subclass of `final` class `Cat`.

Let's now try to declare a subclass of `Cat` named `LongHairedCat`. When we try to compile our project, we should get an error were class `AnLongHairedCatimal` attempts to extend class `Cat`. See Figure 4.10.

Here is our `Cat` class - declared as `final`:

```
// file: Cat.java
final class Cat
{
    .... body as before
}
```

Here is our `LongHairedCat` class - attempting to be a subclass of class `Cat`:

```
// file: LongHairedCat.java
class LongHairdCat extends Cat
{
    public int hairLengthInches = 2;
}
```

When we try to compile with `javac *.java` we'll see an error message:

```
$ javac *.java
LongHairedCat.java:1: error: cannot inherit from final Cat
class LongHairdCat extends Cat
                           ^
1 error
```

4.13 Java `final` method : Java keyword: `final`

A `final` method is one which may **not** be overridden in any subclass.

We use the Java keyword `final` to declare that a method is final (may **not** be overridden).

4.14 Make method `getSound()` final in class `Animal` final (project `cat9`)

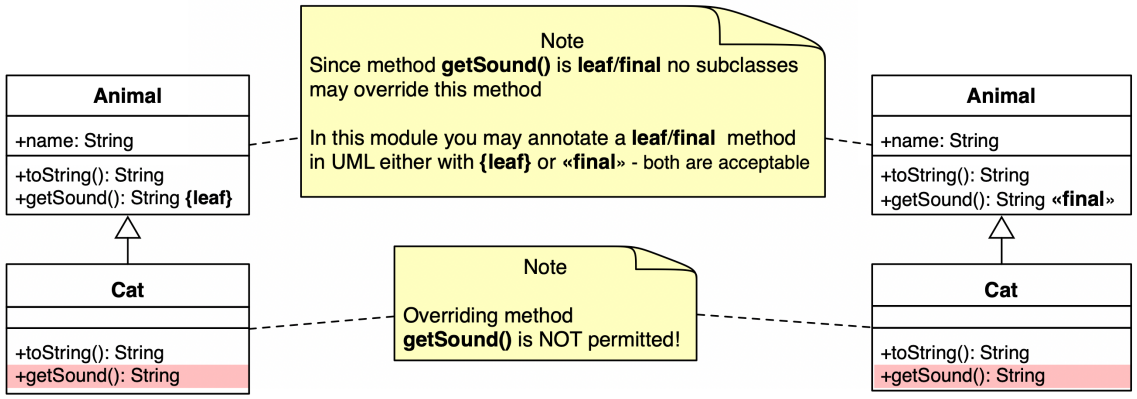


Figure 4.11: `final` method `getSound()` in class `Animal`.

Let's make method `getSound()` final in class `Animal`. See Figure 4.11.

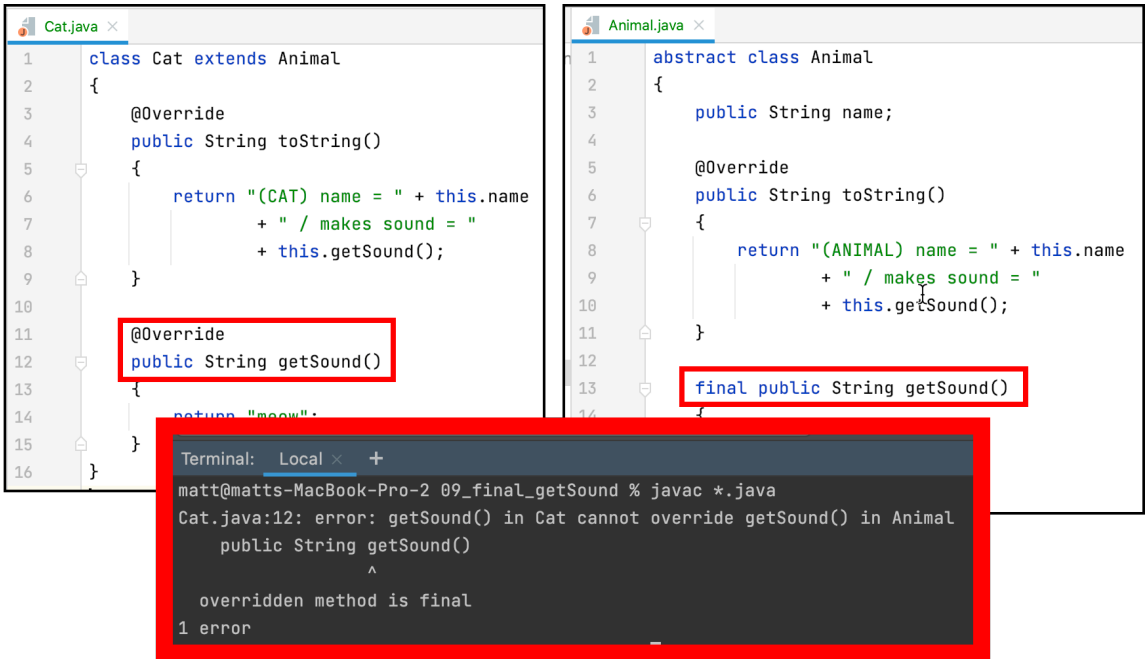


Figure 4.12: Error trying to override inherited `final` method `getSound()` in class `Cat`.

Since in class `Cat` we are trying to override the inherited method `getSound()`, we'll get an error since method `getSound()` final in class `Animal`. See Figure 4.12.

Here is our `Animal` class - declaring method `getSound()` as `final`:

```
// file: Animal.java
abstract class Animal
{
    public String name;

    @Override
    public String toString()
    {
        return "(ANIMAL) name = " + this.name
                + " / makes sound = "
                + this.getSound();
    }

    // method declared as FINAL
    final public String getSound()
    {
        return "(animal sound)";
    }
}
```

Here is our `Cat` class - attempting to override inherited `final` method `getSound()`:

```
// file: Cat.java
class Cat extends Animal
{
    @Override
    public String toString()
    {
        return "(CAT) name = " + this.name
                + " / makes sound = "
                + this.getSound();
    }

    // attempting to overrride inherited final method - not permitted
    @Override
    public String getSound()
    {
        return "meow";
```

```
    }
}
```

When we try to compile with `javac *.java` we'll see an error message:

```
$ javac *.java
Cat.java:12: error: getSound() in Cat cannot override getSound() in Animal
    public String getSound()
                  ^
  overridden method is final
1 error
```

4.15 Java `abstract` method : Java keyword: `abstract`

An `abstract` method is one which **must** be implemented in any concrete (non-abstract) subclass.

We use the Java keyword `abstract` to declare that a method must be implemented by a concrete (non-abstract) subclass.

NOTE

- if a class has **any** (i.e. 1 or more) `abstract` methods, then it should be made an `abstract` class
  - since we cannot create objects of a class that has unimplemented `abstract` methods

4.16 Make method `getSound()` abstract in class `Animal` final (project `cat10`)

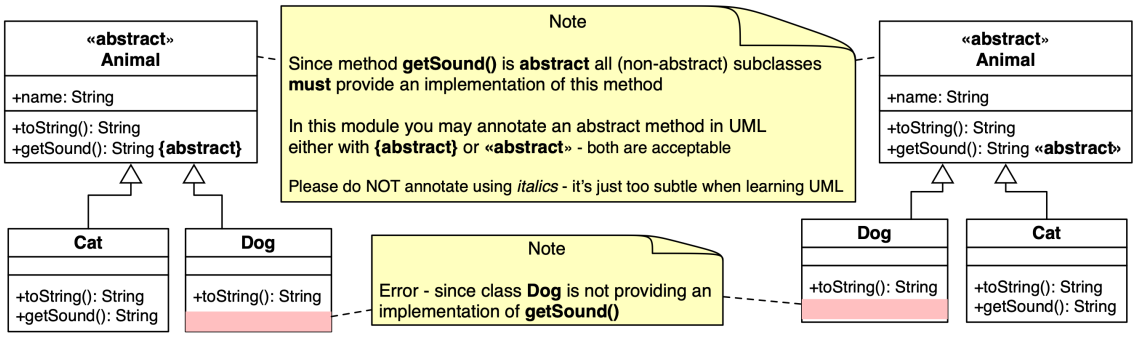


Figure 4.13: `final` method `getSound()` in class `Animal`.

Let's make method `getSound()` abstract in class `Animal`. See Figure 4.13.

Since in class `Dog` we are missing the required implementation of inherited `abstract` method `getSound()`, we'll get an error since method `getSound()` final in class `Animal`. See Figure 4.14.

Here is our `Animal` class - declaring method `getSound()` as `abstaract`:

- NOTE we don't write any body for this method, we just write a semi-colon are the parentheses.

```
// file: Animal.java
abstract class Animal
{
    public String name;

    @Override
    public String toString()
    {
```

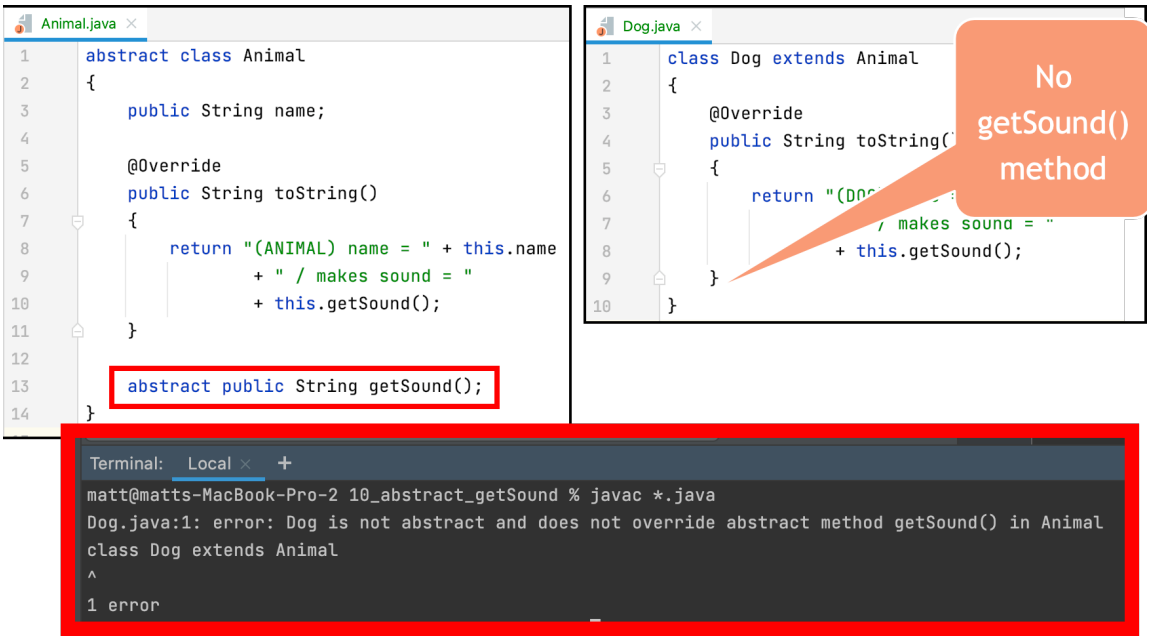


Figure 4.14: Error missing requuired method `getSound()` in class Dog.

```
            return "(ANIMAL) name = " + this.name
                    + " / makes sound = "
                    + this.getSound();
        }

        // we declare this method abstract
        // no method body - just a semi-colon
        abstract public String getSound();
    }
```

Here is our `Cat` class - correctly providing an implementation to override inherited `abstract` method `getSound()`:

```
    // file: Cat.java
    class Cat extends Animal
    {
        @Override
        public String toString()
        {
            return "(CAT) name = " + this.name
                    + " / makes sound = "
                    + this.getSound();
        }
```

```
    // correctly overrriding inherited abstract method
    @Override
    public String getSound()
    {
        return "meow";
    }
}
```

Here is a `Dog` class - incorrectly **not** providing an implementation for method `getSound()`:

```
// file: Dog.java
class Dog extends Animal
{
    @Override
    public String toString()
    {
        return "(DOG) name = " + this.name
                + " / makes sound = "
                + this.getSound();
    }

    // missing method getSound() here ...
}
```

When we try to compile with `javac *.java` we'll see an error message:

```
$ javac *.java
Dog.java:1: error: Dog is not abstract and does not override abstract method getSound() in A
class Dog extends Animal
^
1 error
```

4.17 Working cats and dogs as animals project (project `cat11`)

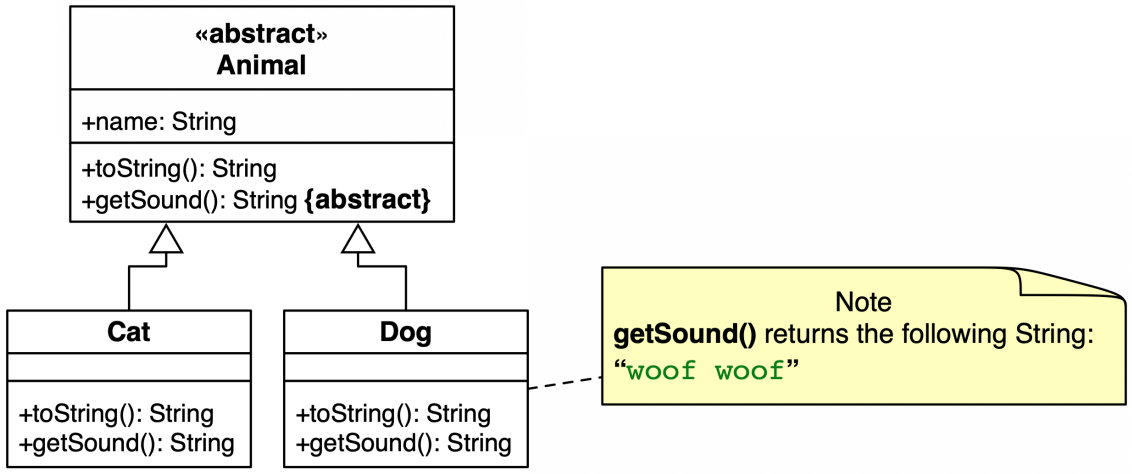


Figure 4.15: `Cat` and `Dog` subclasses of `abstarct` class `Animal`.

Let's add an implementation of method `getSound()` in class `Dog` to get a working project. See Figure 4.15.

Here is a `Dog` class - correctly providing an implementation to override inherited `abstract` method `getSound()`:

```
// file: Dog.java
class Dog extends Animal
{
    @Override
    public String toString()
    {
        return "(DOG) name = " + this.name
                + " / makes sound = "
                + this.getSound();
    }

    // correctly overrriding inherited abstract method
    @Override
    public String getSound()
    {
        return "woof woof";
    }
}
```

Let's refactor our `main()` method in class `Main` - to create a `Cat` and a `Dog` object, and print out their names and sounds via their `toString()` methods:

```
// file: Main.java
class Main
{
    public static void main(String[] args)
    {
        Cat cat1 = new Cat();
        Dog dog1 = new Dog();

        cat1.name = "princiess fifi";
        dog1.name = "spot";

        System.out.println("cat1: " + cat1);
        System.out.println("dog1: " + dog1);
    }
}
```

It should all compile fine, and output the details of our 2 `Animal` objects:

```
$ javac *.java
cat1: (CAT) name = princiess fifi / makes sound = meow
dog1: (DOG) name = spot / makes sound = woof woof
```

4.18 Summary & Conclusions: Inheritance - part 2

Java inheritance

- Java `abstract` before keyword `class`
  - Means that the class is just there to be subclassed
  - We cannot create objects of an abstract class
  - annotated in UML class diagram as `{abstract}` - but I'll accept `«abstract»` as well
- Java `final` before keyword `class`
  - Means that the class may **not** be subclassed
  - We cannot create any subclasses of a final class
  - annotated in UML class diagram as `{leaf}` - but I'll accept `«final»` as well
- Java `final` modifier of method declaration
  - Means that the method may **not** be overridden in any subclass
  - We cannot declare a method with same name and number/types of arguments in any subclass
  - method annotated in UML class diagram as `{leaf}` - but I'll accept `«final»` as well
- Java `abstract` modifier of method declaration
  - Means that the method is just there to be overriden
  - We must make the class containing 1 or more abstract methods an abstract class
  - any concrete (non-abstract) subclass **must** implement (override) the inherited abstract method
  - annotated in UML class diagram as `{abstract}` - but I'll accept `«abstract»` as well

4.19 Exercise - explore abstract classes

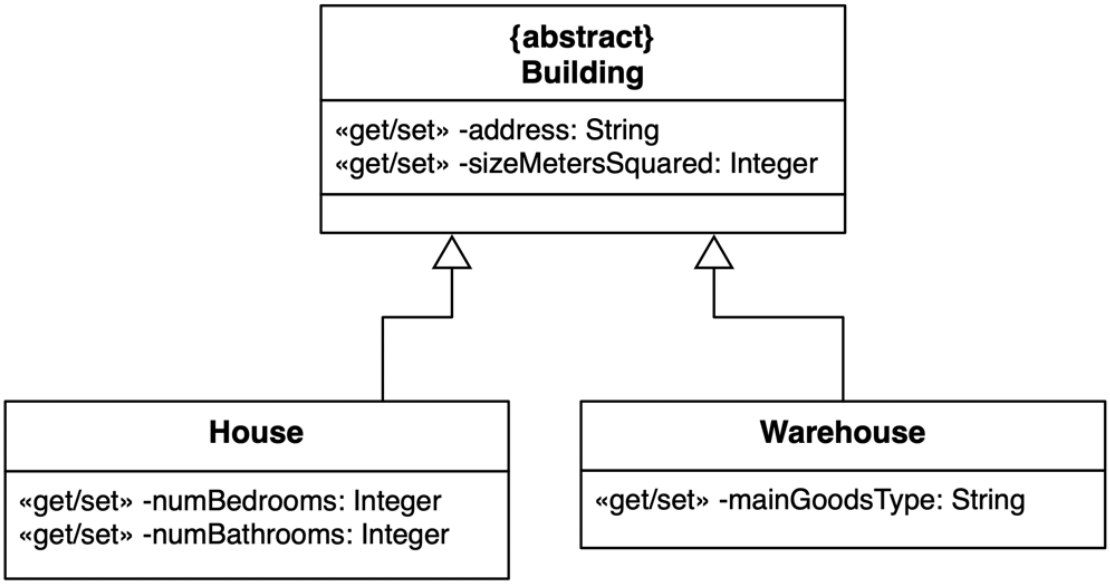


Figure 4.16: Class-Object diagram for `abstract` class `Building`.

AIM:

- practice working with `abstract` classes

ACTION:

- class `Building` (File: `Building.java`)
  - class `Building` as an abstract class, with private properties and public getters/setters:
    - `address`: String
    - `sizeMetersSquared`: Integer
- class `House` (File: `House.java`)
  - class `House`, a subclass of `Building`, with private properties and public getters/setters:
    - `numBedrooms`: Integer
    - `numBathrooms`: Integer
- class `Warehouse` (File: `Warehouse.java`)
  - class `Warehouse`, a subclass of `Building`, with private properties and public getters/setters:
    - `mainGoodsType`: String
- class `Main` (File: `Main.java`)

- `main()` method to:
  - attempt to create an object of class `Building`
    - you should get COMPILER ERROR since `Building` is `abstract`
  - create instance-objects of classes `House` and `Warehouse`

4.20 Exercise - final class

AIM:

- understand the result of having a `final` class

ACTION:

- create a `Person` superclass and a `Student` subclass
  - they do not need any methods or properties for this exercise
- declare the `Person` class as final
- compile your files

OUTPUT:

- you should get an error, stating that `Student` cannot extend Person because `Person` is final

4.21 Exercise - final method

AIM:

- practice working with `final` methods

ACTION:

- create a `Employee` superclass and a `Caretaker` subclass
  - in class `Employee` declare a final method `calculateSalary()` that returns double 20.0
  - in class `Caretaker` declare a method `calculateSalary()` that returns double 55.0
- compile your files

OUTPUT:

- you should get an error, stating that `Caretaker` cannot override inherited final method `calculateSalary()`

4.22 Exercise - abstract method

AIM:

- practice working with `abstract` methods

ACTION:

- create a `Vehicle` superclass and a `Boat` subclass
  - in class `Vehicle` declare an abstract method `getTopSpeed()` the returns a double
    - there should be no method body, just a semi-colon after the parentheses
  - declare class `Boat`
    - but do **not** write any method `getTopSpeed()` for this class
- compile your files

OUTPUT:

- you should get an error, stating that `Boat` cannot extend class `Vehicle` because it does not implement abstract method `getTopSpeed()`

REFINEMENT STEP:

- now declare class `Boat` as abstract
  - you should now be able to compile your classes - since an abstract subclass does **not** need to implement all inherited abstract methods

5

Inheritance - part 3

5.1 Inheritance and `super()`

While for class members, inheritance is straightforward and as you might expect, there are some special issues relating to the inheritance of **constructor** methods. Part most OO programming languages approach to dealing with constructor inheritance is the `super` keyword.

5.2 Constructors are not members, and therefore not inherited

The members of a class include its constants, variables and methods. However, constructor methods are **not** members of a class, and therefore are **NOT** inherited by subclasses.

5.3 Simplest case - top-level

5.4 notes from where?

You don't have to provide any constructors for your class, but you must be careful when doing this. The compiler automatically provides a no-argument, default constructor for any class without constructors. This default constructor will call the no-argument constructor of the superclass. In this situation, the compiler will complain if the superclass doesn't have a no-argument constructor so you must verify that it does. If your class has no explicit superclass, then it has an implicit

superclass of Object, which does have a no-argument constructor.

If a subclass does not declare a constructor, then the no-argument defaulkt will be called - this will try to call a no-argument constructor of the superclass - error if superclass doesn't have a no-argument constructor

constructor chaining

// ———- TODO ————-

5.5 Oracle Java docs about invoking superclass constructors

NOTE:

- This is a **very** important...

When writing a hierarchy of classes, some of which declare constructors, it's very important to understand the following statement in the Java documentation - see Figure 5.2:

> If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the super class does not have a no-argument constructor, you will get a compile-time error.

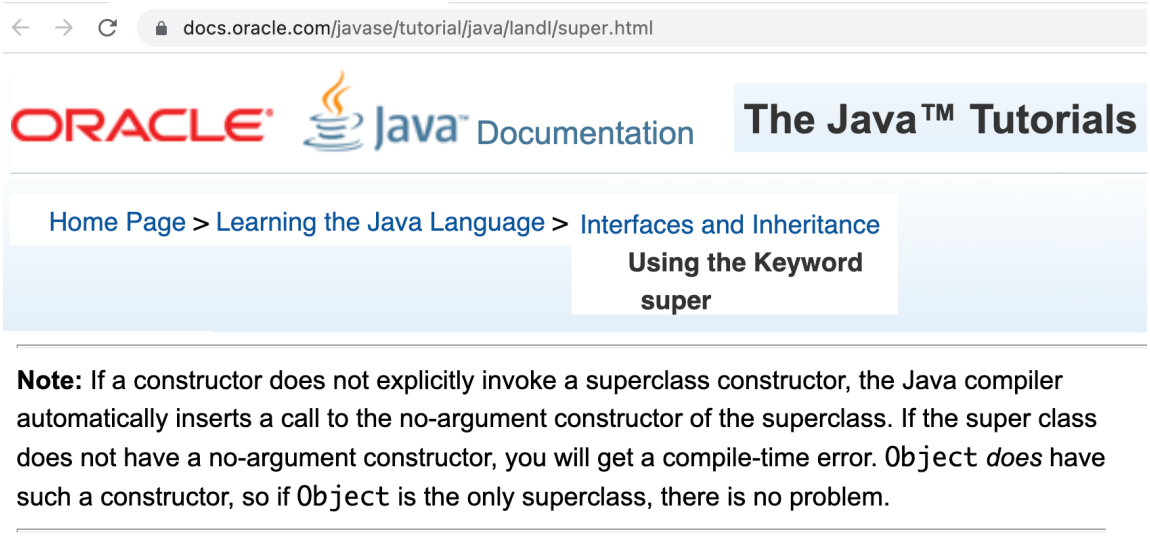


Figure 5.1: Java documentation note about constructor inheritance.

You can find the page here:

- https://docs.oracle.com/javase/tutorial/java/IandI/super.html

5.6 Further reading

You can find some discussion about this topic on Stack Overflow here:

- https://stackoverflow.com/questions/23395513/implicit-super-constructor-person-is-undefined-must-explicitly-invoke-another

And something a bit more advanced here, about why some libraries **require** us to ensure every class has a no-argument constructor declared, if there is any other constructor with arguments:

- https://stackoverflow.com/questions/3078389/why-do-we-need-a-default-no-argument-constructor-in-java

Learn more about the top-leve `Object` class in the docs:

- https://docs.oracle.com/javase/tutorial/java/IandI/objectclass.html

Oracle docs about constructors:

- https://docs.oracle.com/javase/tutorial/java/javaOO/constructors.html

5.7 Exercise - subclass-superclass automatic constructor invocation

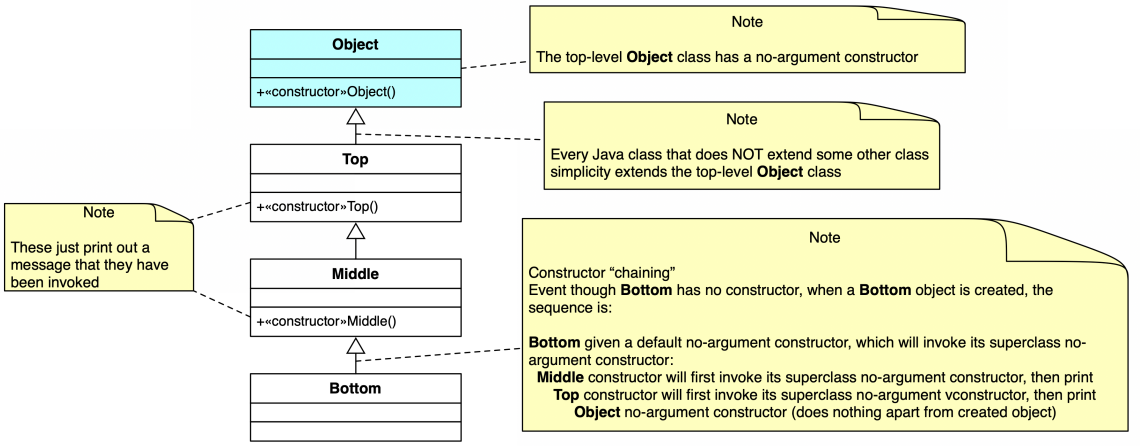


Figure 5.2: Class diagram for constructor "chaining".

AIM:

- see automatic superclass constructor invocation in action

ACTION:

- class `Top` (File: `Top.java`)
  - class `Top` has a single method:
    - constructor that prints out message `Top() constructor invoked`
- class `Middle` (File: `MIddle.java`)
  - class `Middle` has a single method:
    - constructor that prints out message `Middle() constructor invoked`
    -
- class `Bottom` (File: `Bottom.java`)
  - this is an empty class declaration (so will get a no-argument constructor automatically added by the compiler)
- class `Main` (File: `Main.java`)
  - `main()` method to:
    - create a `Bottom` object named `bottom1Object`
    - prints out message `Bottom object created`

OUTPUT:

```
$ java Main

Top() constructor invoked!
Middle() constructor invoked!
Bottom object created
```

6

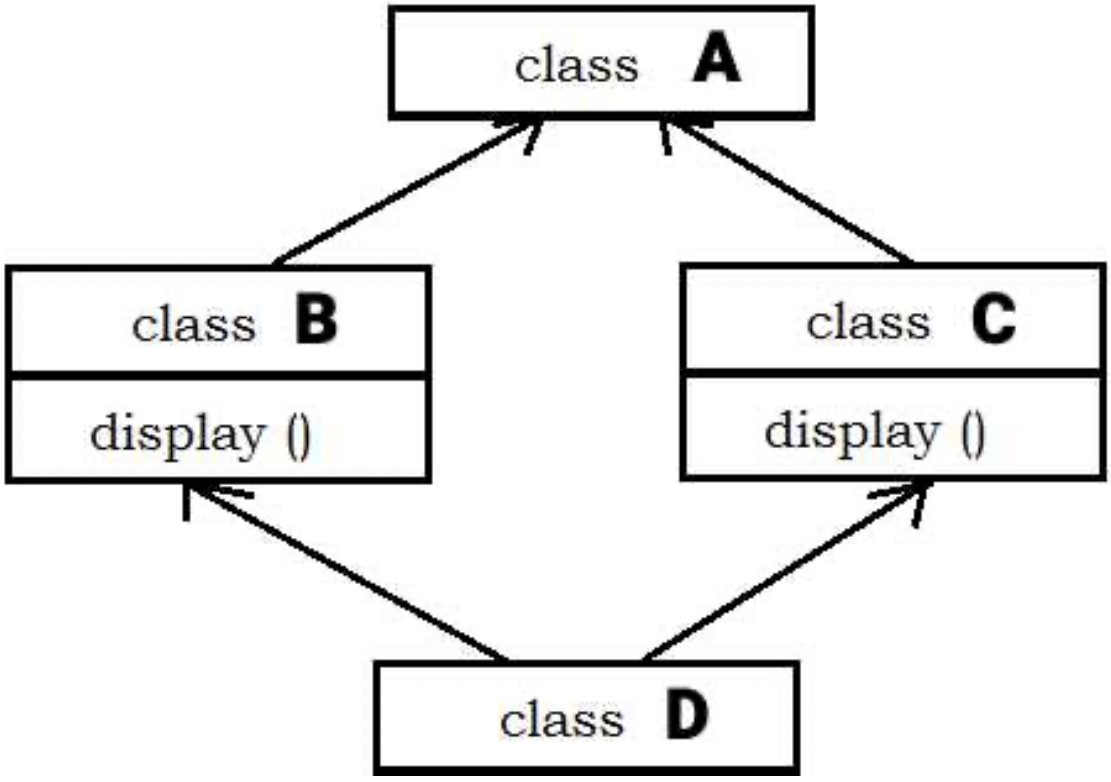
Interfaces

Four core principles - these 3 relate to interfaces:

- Abstraction
- Inheritance
- Polymorphism

Interface

- Declares a set of (automatically public) methods that an `implementing` class must implement
- Offers solution to single-inheritance languages like Java
- Like `abstract` classes , no objects can be created from `Interfaces`



6.1 Java is SINGLE inheritance - to avoid the "Deadly Diamond of Death"

If we allow a class (e .g . Class D) to have more than one superclass

- AND 2 or more of the superclasses have a variable/method the same name
- How do we / Java know WHICH one to inherit ! ! !
- E .g . Does D inherit display ( ) from class B or class C ???
- The problem is avoided if a class can only inherit from one superclass

See:

- http://basicsupdate .blogspot .com/2014/01/deadly -diamond -of -death -problem .html
- https://subhashprogrammingclasses .in/interfaces -in -java/

6.2 Problem with single inheritance - duplication of behaviours in different inheritance hierarchies

- We want to AVOID duplication

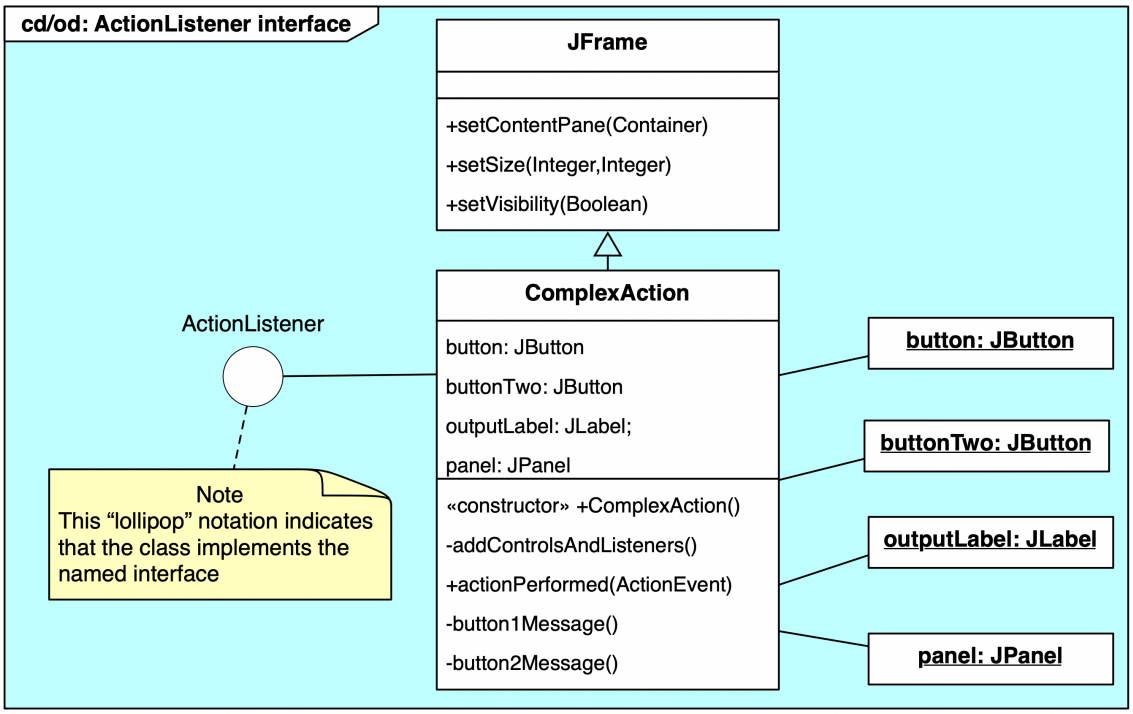
  - Many types of object make a sound:
    - Cat / Dog / Car / Piano
  - But these objects are in different inheritance hierarchies
    - Cat extends Animal
    - Car extends Vehicle
    - Piano extends Instrument
  - But we want a way to say that ALL the above classes have to offer method:
    - `String getSound( )`
- `Interfaces` allow us to declare that classes must implement some behaviours
  - But the actual implementation is left to the class implementing the interface
  - So no "diamond of death| problem , since a interfaces don't offer competing method bodies (no method body in interfaces…)
- In Java a class can only `EXTEND` one superclass , but may implement many interfaces…

6.3 Java interfaces Java keywords:

`interface` and `implements`

6.4 interface ActionListener {…}

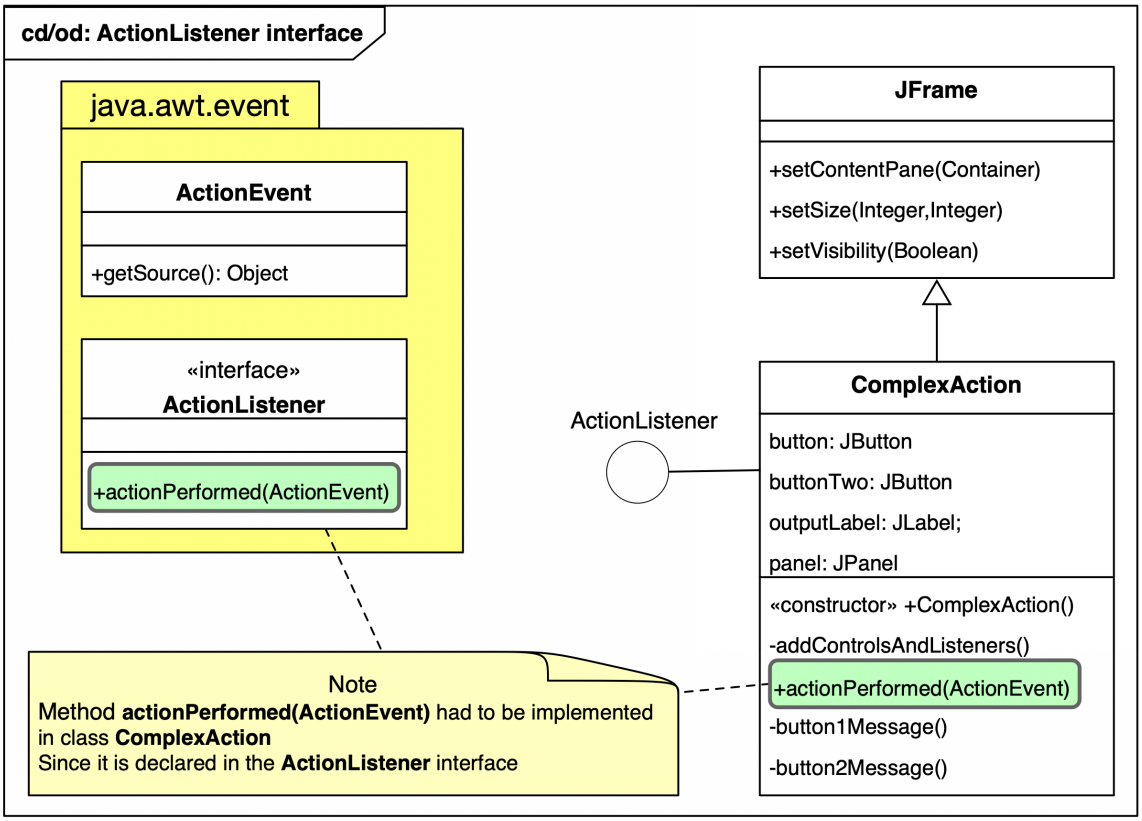
class MyGUI implements ActionListener



NOTE:

- In YOUR class diagrams please use this "lollipop" interface implementation notation

You've met an interface in GUI Programming



NOTE:

In YOUR class diagrams

Please only use the "lollipop" notation to show interface implementation

So the only arrows in your class diagrams are for inheritance …

6.5 An Interface is not a class

- No constructor methods permitted in an interface
  - since cannot create objects from an interface
- Other classes cannot subclass (`extend`) an interface
  - they can only `implement` an interface
  - However , an interface can extend an interface …
- An interface cannot declare any `instance variables`
  - Since they will never be any objects of the interface

6.6 An Interface can

- Declare `static` variables and constants
  - Since these do no related to objects
- Declare `static` methods (with implementations)
  - Since these do no related to objects
- Extend another interface (inheritance)
  - One interface can inherit members from another interface
- One interface can `inherit` members from another interface

```
without interface inheritance:
    class ChevyVolt implements Vehicle, Drivable
    class FordEscort implements Vehicle, Drivable
    class ToyotaPrius implements Vehicle, Drivable
```

```
making interface Vehicle inherit from interface Drivable:
    interface Vehicle extends Drivable

simplies our car classes to:
    class ChevyVolt implements Vehicle
    class FordEscort implements Vehicle
    class ToyotaPrius implements Vehicle
```

6.7 Example interface project - cats/dogs `SoundMaker` interface (project `cat20`)

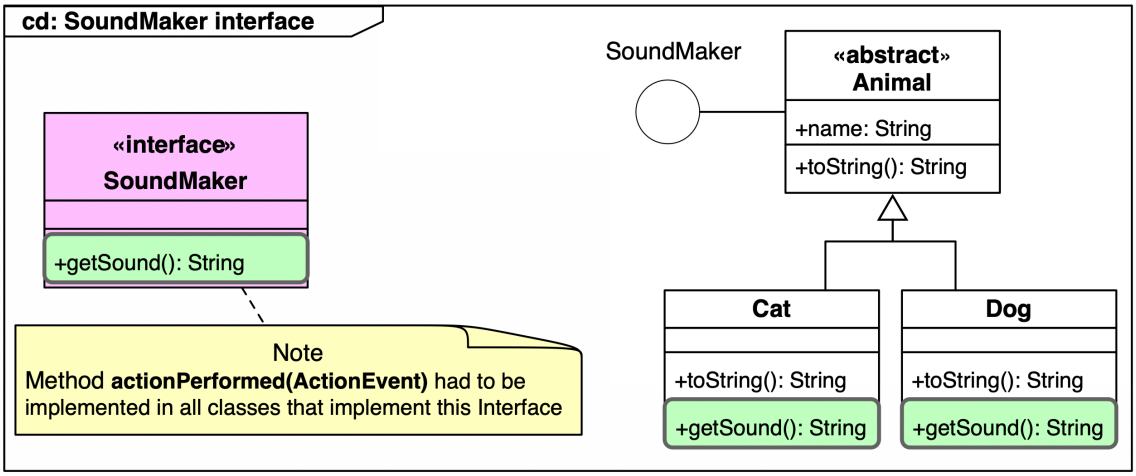


Figure 6.1: Many classes implementing SoundMaker.

Let's create an interface `SoundMaker` that requires implementing classes to declare a method `getSound()`. See Figure 4.15.

First let's declare our `SoundMaker` interface file. All it needs to do is declare that it is an interface, and then give the return type+name+arguments for the method, ending the line with a semi-colon:

```
// file: SoundMaker.java
interface SoundMaker
{
    String getSound();
}
```

NOTE:

- all methods are public for an interface, so it's optional whether to write `public` or not…

We can now create classes to implement the interface. Let's declare abstract class `Animal` to implement our interface - so all concrete (non-abstract) subclasses will have to declare a `getSound()` method. However, since class `Animal` is abstract, this class itself doesn't need to declare a `getSound()` method:

```
// file: Animal.java
abstract class Animal implements SoundMaker
{
    public String name;

    @Override
```

```
    public String toString()
    {
        return "(ANIMAL) name = " + this.name
                + " / makes sound = "
                + this.getSound();
    }
}
```

We can now declare class `Cat` as a subclass of `Animal` declaring a `getSound()` method:

```
// file: Cat.java
class Cat extends Animal
{
    @Override
    public String toString()
    {
        return "(CAT) name = " + this.name
                + " / makes sound = "
                + this.getSound();
    }

    @Override
    public String getSound()
    {
        return "meow";
    }
}
```

We can now also declare class `Dog` as a subclass of `Animal` declaring a `getSound()` method:

```
// file: Dog.java
class Dog extends Animal
{
    @Override
    public String toString()
    {
        return "(DOG) name = " + this.name
                + " / makes sound = "
                + this.getSound();
    }

    @Override
    public String getSound()
```

```
    {
        return "woof woof";
    }
}
```

We can write a `main(...)` method in class `Main` - to create a `Cat` and a `Dog` object, and print out their names and sounds via their `toString()` methods:

```
// file: Main.java
class Main
{
    public static void main(String[] args)
    {
        Cat cat1 = new Cat();
        Dog dog1 = new Dog();

        cat1.name = "princess fifi";
        dog1.name = "spot";

        System.out.println("cat1: " + cat1);
        System.out.println("dog1: " + dog1);
    }
}
```

It should all compile fine, and output the details of our 2 `Animal` objects:

```
$ javac *.java
cat1: (CAT) name = princess fifi / makes sound = meow
dog1: (DOG) name = spot / makes sound = woof woof
```

6.8 Example interface project - cats/dogs/cars/instruments

`SoundMaker` interface (project `cat21`)

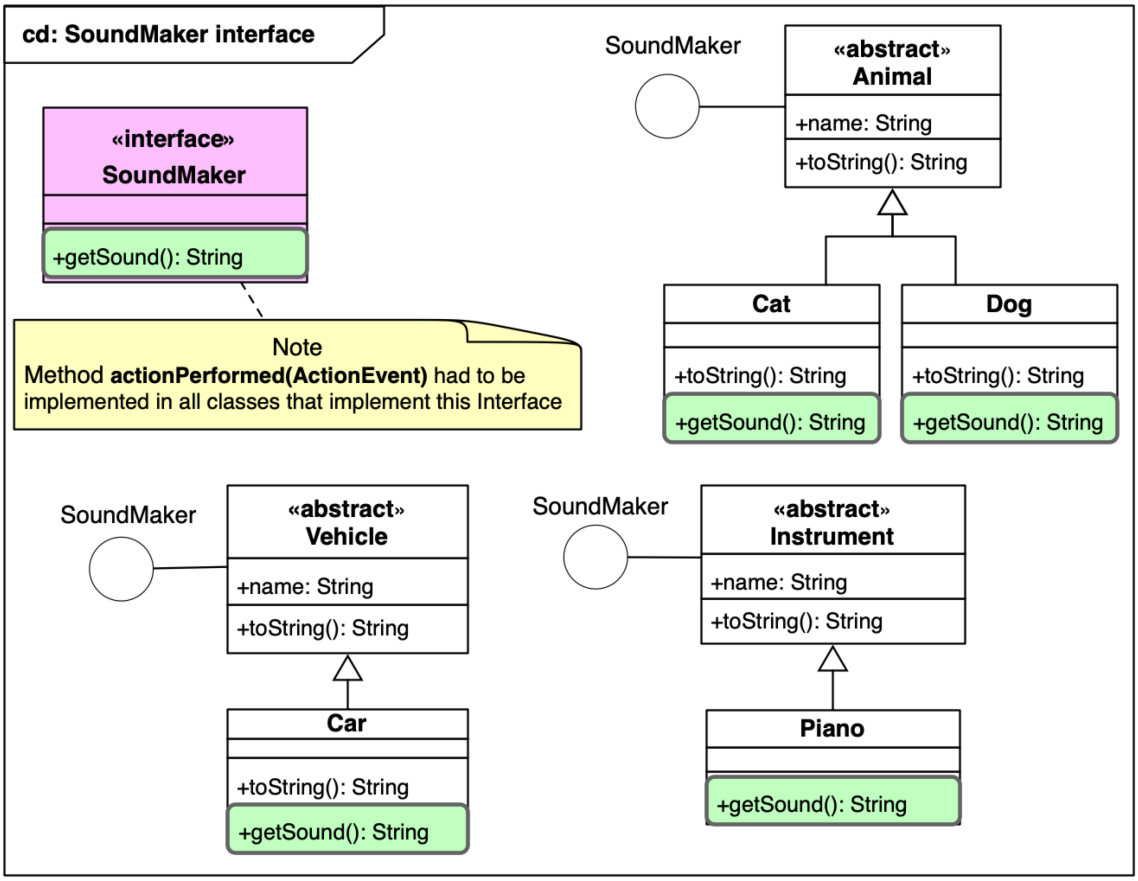


Figure 6.2: Many classes implementing SoundMaker.

So far, what we have done above doesn't seem to add any value to just declaring `toString()` as an abstract method in our `Animal` class. However, the power of interfaces can begin to be seen when we wish to ensure objects of classes that are **not** Animals can also make sounds.

Let's declare some more classes that implement the `SoundMaker` interface. We'll declare abstract classes `Instrument` and `Vehicle` - each with their own concrete subclass. See Figure 6.2.

First let's declare abstract class `Vehicle` to implement our interface:

```
// file: Vehicle.java
abstract class Vehicle implements SoundMaker
{
    public String registration;

    @Override
```

```
    public String toString()
    {
        return "(VEHICLE) registration = " + this.registration
                + " / makes sound = "
                + this.getSound();
    }
}
```

Now we can declare class `Car`, a concrete subclass of `Vehicle`, that implements the required method `getSound()`:

```
// file: Car.java
class Car extends Vehicle
{
    @Override
    public String toString()
    {
        return "(CAR) registration = " + this.registration
                + " / makes sound = "
                + this.getSound();
    }

    @Override
    public String getSound()
    {
        return "vroom vrooom";
    }
}
```

We now declare abstract class `Instrument` to implement our interface:

```
// file: Instrument.java
abstract class Instrument implements SoundMaker
{
    public String category;

    @Override
    public String toString()
    {
        return "(INSTRUMENT) category = " + this.category
                + " / makes sound = "
                + this.getSound();
    }
```

```
}
```

Now we can declare class `Piano`, a concrete subclass of `Instrument`, that implements the required method `getSound()`.

NOTE: Class `Piano` does not declare (override) a `toString()` method, therefore any `Piano` object receiving a `toString()` message will use the method inherited from its `Instrument` superclass:

```
// file: Piano.java
class Piano extends Instrument
{
    @Override
    public String getSound()
    {
        return "plink plonk";
    }
}
```

We can add to our `main(...)` method in class `Main` the instantiation (creation) of a `Car` and a `Paino` object, and print out their names and sounds via their `toString()` methods, in just the same way we did with the `Cat` and `Dog` objects:

```
// file: Main.java
class Main
{
    public static void main(String[] args)
    {
        Cat cat1 = new Cat();
        Dog dog1 = new Dog();

        Car car1 = new Car();
        Piano piano1 = new Piano();

        cat1.name = "princiess fifi";
        dog1.name = "spot";

        car1.registration = "202-D-1999";
        piano1.category = "percussive strings";

        System.out.println("cat1: " + cat1);
        System.out.println("dog1: " + dog1);

        System.out.println("car1: " + car1);
        System.out.println("piano1: " + piano1);
```

```
    }
}
```

It should all compile fine, and output the details of our 2 `Animal` objects, and the car and the piano:

```
$ javac *.java
cat1: (CAT) name = princiess fifi / makes sound = meow
dog1: (DOG) name = spot / makes sound = woof woof
car1: (CAR) registration = 202-D-1999 / makes sound = vroom vrooom
piano1: (INSTRUMENT) category = percussive strings / makes sound = plink plonk
```

6.8.1 Polymorphism: many forms/behaviours...

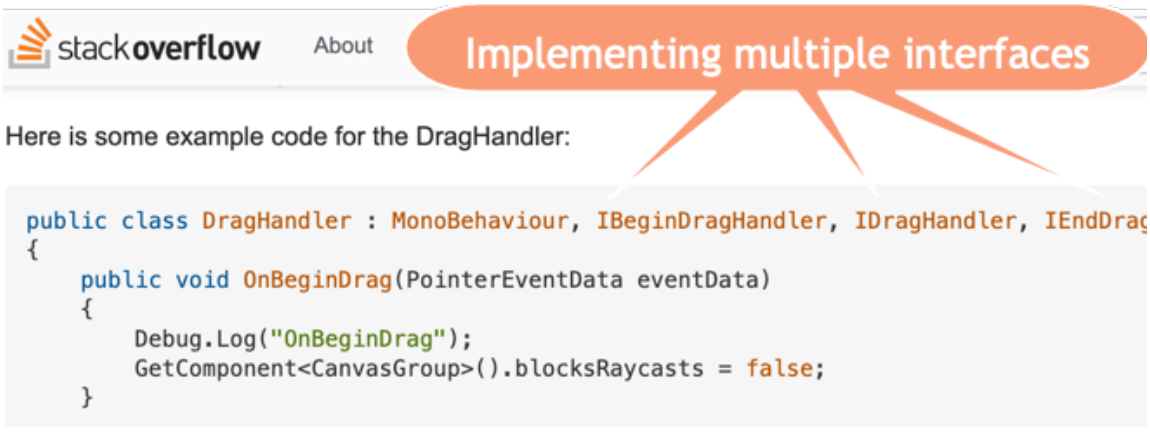
What we see above is a good example of the OOP principle of **Polymorphism**:

- as long as we know an object is a class that implements the **SoundMaker** interface, we know we can send it a `getSound()` message, and we'll get back a String. But the actual declared method being invoked depends on the class of object to which we have sent the message.

Polymorphism:

- Different behaviours from objects , when sent same message
- Different classes can each implement interface methods in ways appropriate to the class
- At **Run-Time** Java examines message received by an object , and that object's class and interface hierarchy, and decides which method implementation to execute...

6.9 Interface vs. Abstract class



- Interfaces
  - Classes can `implement` **multiple** interfaces
  - Interfaces are only for declaring **behaviour**
    - Since cannot have statements that work on the **state** (instance variables) of objects

- Abstract Classes
  - Abstract classes can declare `instance variables` and constructors
    - And so describe how methods work on object **state**
  - Abstract classes can declare **constructors**
  - class can only **inherit** from **one** class (single inheritance)

6.10 Summary & Conclusions

- Coding Abstraction that allows software to be designed for classes to have designed behaviour beyond just one superclass
  - Code can be written to work with any class implementing an interface
  - Good for designing similar behaviour for classes that are otherwise unrelated
- Declaring an Interface:Java keyword: `interface`
- Implementing an Interface:Java keyword: `implements`
- Extending an Interface:Java keyword: `extends`
- Interface on a class diagram `«interface»`
  - Show one class implementing an interface with the "lollipop" graphic

6.11 Reference Sources

- Unity C # interfaces tutorial
  - https://learn .unity .com/tutorial/interfaces
- Stack Exchange debate about whether interfaces should always have "I" prefix
  - https://softwareengineering .stackexchange .com/questions/117348/should -interface -names -begin -with -an -i -prefix
  - My two penny -worth:
    - follow the language conventions – so if C # prefix with an "I" , if Java don't bother
- Article describing `forEach()` default method
  - https://dzone .com/articles/interface -default -methods -java
- Multiple interfaces for Unity C # drag -drog
  - https://stackoverflow .com/questions/55387783/drop -event -not -firing -with -idrophandler -in -unity3d

6.12 default method implementations (Since Java 8)

- Java now permits a default implementation to be declared in an interface
  - Which may, but does not have to be overridden by the implementing class
- Useful to allow addition on new features to all classes implementing an interface
  - E .g . in Java 8 a `forEeach(...)` method was added using a default method to **ALL** classes implementing the `Iterable` interface

I recommend you avoid default method implementations - if you use them in tests or projects you'll have to justify why they were used.

An Interface can define a **default** implementation for methods:

- but then have to avoid diamond of death for interfaces
- https://www .javacodemonk .com/diamond -problem -of -inheritance -in -java -8 -88faf6c9
- In a nutshell
  - If not clear which default method implementation to inherit , then the implementing class MUST override the inherited method with its own implementation…

6.12.1 How Java 8+ avoids the DoD for class implementing multiple interfaces

DoD - Diamond of Death - why is this not a problem when a class implements multiple interfaces?

- https://stackoverflow.com/questions/9860811/do-interfaces-solve-the-deadly-diamond-of-death-issue

for VARIABLES - When a class inherits two variables from parent interfaces, Java insists that any use of the variable name in question be fully qualified.

for METHODS - When a class inherits two Default method implementations for the same method from different Interfaces

- the class must override the method if the priority scheme fails to yield a single winner

6.13 Exercise - new class to implement `SoundMaker` interface

AIM:

- try out implementing the `SoundMaker` interface

ACTION:

- create a new project folder, and copy/create in that the `SoundMaker.java` interface file
- create a new class (unrelated to animals / instruments / vehicles) that makes sound
  - e.g. rain / volcano / doorbell etc.
- your new class should:
  - declare that it implements the `MakesSound` interface
  - declare a `getSound()` method returning a String
- add a `Main` class with `main(...)` method to create an object of your class, and print out the String returned when it is sent the `getSound()` message

6.14 Exercise - understanding / reflecting about GUI programming Java code

AIM:

- making code your own & reflecting on what's happening in a simple GUI program

ACTION:

- you have been provided with a copy of the `ButtonFrame.java` program from:

  COMP H2011 - GUI Programming 21 September - 27 September 2020 Sample Code

- extract the `main()` method into a separate class `Main`
  - and change the identifier (name!) of the object created to `application`
- add the prefix `this.` to any part of a statement that is referring to an instance variable (if any)
- in the order they are created (i.e. starting at the beginning of the `main()` method) prefix each statement that creates an object with it sequence number (object 1, object 2 etc.) and the class it is an instance of:

HINT:

```
public static void main(String args [])
{
    // object 1: ButtonFrame myFrame
    ButtonFrame myFrame = new ButtonFrame();
```

- what is the relationship between the objects? Write a comment each time an object reference variable is being passed as a parameter to a method of another object:

HINT:

```
// object reference `myButton` being passed to method `add()` of object `panel`
panel.add(myButton);
```

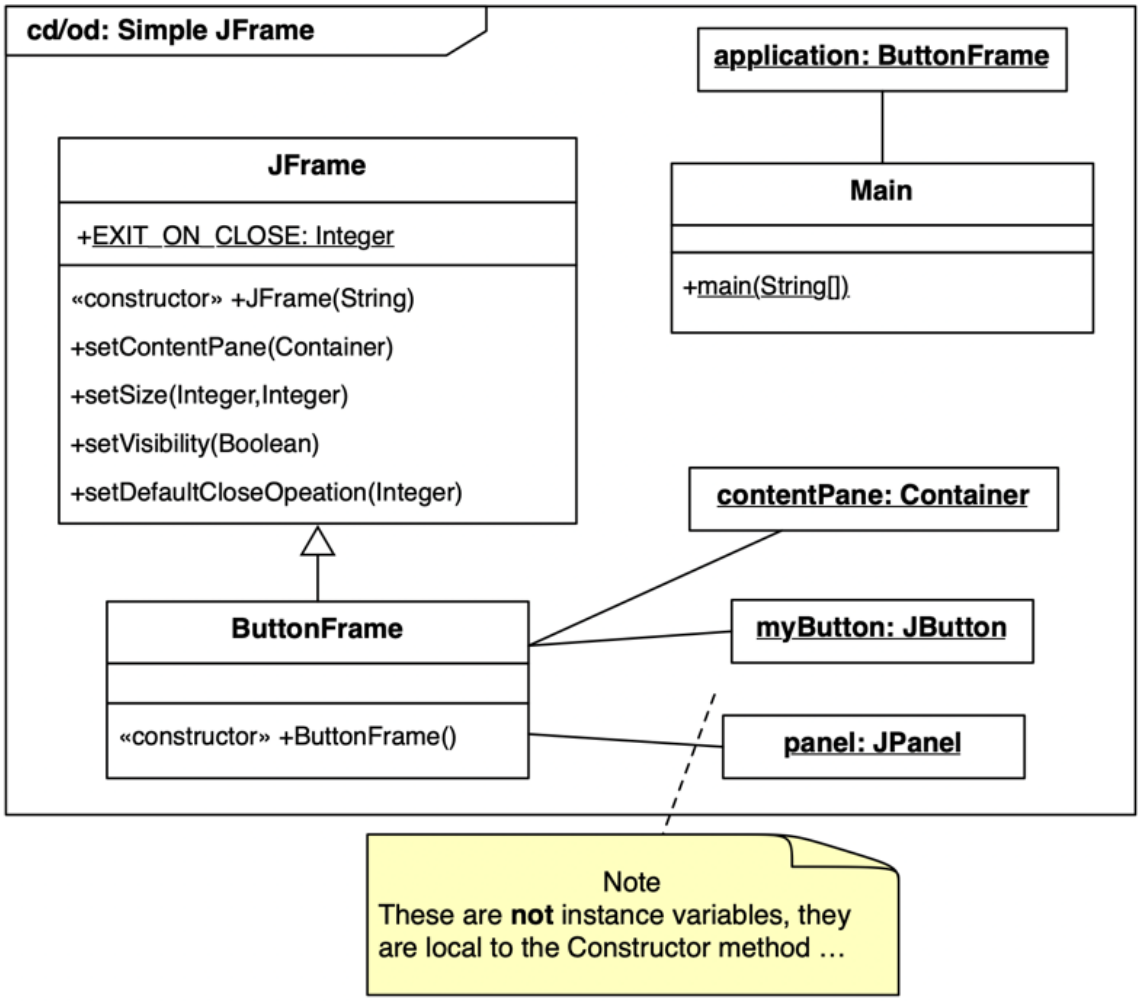


Figure 6.3: Class/Object diagram for ButtonFrame listing.

6.15 Exercise - make code your own - re-write it!

AIM:

- making code your own & separating responsibilities

Do the following:

- you have been provided with a copy of the `ComplexAction.java` program from:

  COMP H2011 - GUI Programming 5 October - 11 October Sample Programme (Lab 4)

- explore this code, and think about the objects being created and the messages being passed
- add the prefix `this.` to any part of a statement that is referring to an instance variable (if any)
  - NOTE: as you get more and more instance variables in the code for a class, "this" all over the place can get annoying - but when **learning** about instance variables (versus class variables versus arguments versus variables local to a method) it is a useful practice - once you've very comfortable with such coding concepts, when languages allow you to miss out the "this", it's okay to do so, unless a coding style requires you to write `this`
- extract the `main()` method into a separate class `Main`
  - and change the identifier (name!) of the object created to `application`
- separate out the declaration of the 2 buttons, so each variable is declared in a statement by itself:

HINT:

```
JButton button = new JButton("First Button");
JButton buttonTwo = new JButton("Second Button");
```

- break up the logic of the `ComplexAction` class, so that adding buttons to panels and registering ActionListeners for user button clicks is in its own method. Do the following:
  - make the `JPanel` an instance variable (so accessible from all methods of the class)
  - create a new method `addControlswAndListeners()` that adds the button listeners to the buttons, and adds the buttons to the panel
  - call method `addControlswAndListeners()` from constructor method `ComplexAction()`

HINT:

```
public ComplexAction()
{
    super("Complex Action Test");
    Container contentPane = getContentPane();
```

```
    addControlswAndListeners();

    ...
}

private void addControlswAndListeners()
{
    this.button.addActionListener(this);
    ...
    this.panel.add(this.button);
    ...
}
```

- let's go further, separating the logic of detecting which object was clicked from its action. Do the following:
  - create 2 new methods:
    - a method `button1Message()` that sets the text of `outputLabel` to `"button 1 clicked"`
    - a method `button2Message()` that sets the text of `outputLabel` to `"button 2 clicked"`
  - change the contents of method `actionPerformed(...)` to use IF-ELSE statements for this logic:

```
IF event source was button 1 THEN invoke `button1Message()`

ELSE IF event source was button 2 THEN invoke `button2Message()`

ELSE set text of `outputLabel` to `"error - unknown source of user event"`
```

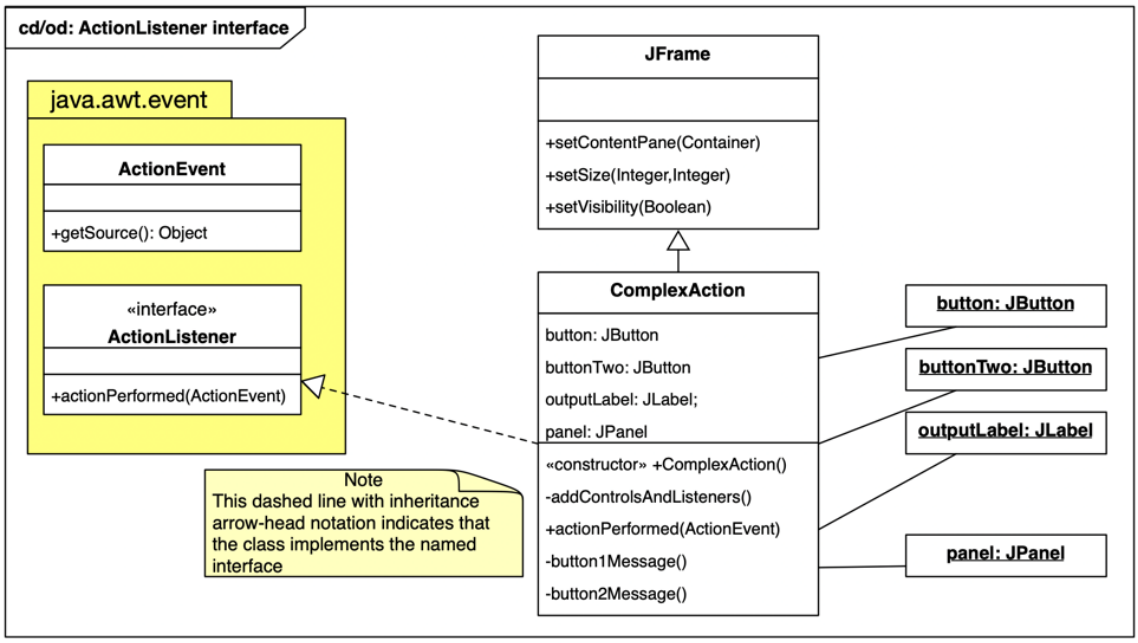


Figure 6.4: Class-Object diagram for refactored`ComplexAction`.

6.16 Exercise - breaking up a program into multiple classes

AIM:

- making code your own & breaking up a program into multiple classes

Do the following:

- make a copy of the previous exercise, and work on that copy for this exercise
- create a new class `ClickManager` that implements the `ActionListener` interface, and has the following members (variables and methods:
  - 2 private variables:
    - `button1` reference to a `JButton` object
    - `button2` reference to a `JButton` object
    - `outputLabel` reference to a `JLable` object
  - a constructor that takes in 2 `JButton` object references and a `JLabel` reference, and stores them in the 3 instance variables
  - move the following methods from `ComplexAction` into this new `ClickManager` class:
    - `button1Message()`
    - `button2Message()`
    - `actionPerformed(...)`

- refactor your class `ComplexAction` as follows:
  - remove the `implements ActionListener` from the class declaration (since out `ClickManager` class is handling user button click events)
  - add a new instance variable `clickManager` that is a reference to a `ClickManager` object, and set its default value to a new object constructed with the other 3 variables:
  - change the `addActionListener(this)` statements for the buttons to add the `clickManager` object to be registered to listen for user button click events

HINT:

```
ClickManager clickManager = new ClickManager(button, buttonTwo, outputLabel);
```

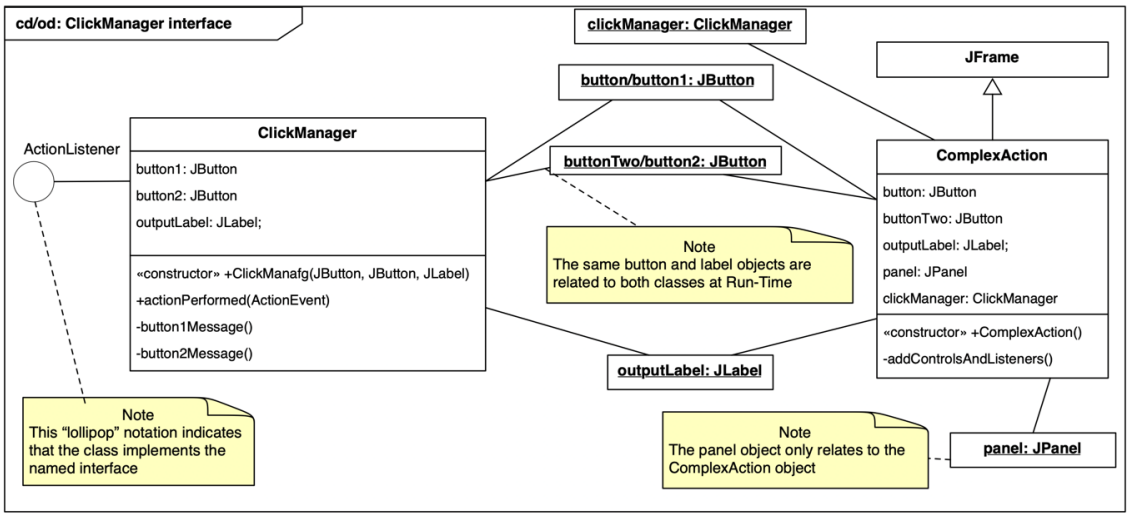


Figure 6.5: Class-Object diagram for new class `ClickManager`.

6.17 Exercise - your own Java interfaces

AIM:

- explore how to declare and implement your own Java interfaces

Do the following:

- Declare an interface named `VatCalculated` requiring classes to implement a method `double priceIncludingVat()`
- Declare a class `Book` which has:
  - private `title` String instance variable - with accessor methods
  - private `price` double instance variable - with accessor methods
  - implements your interface, returning the price plus VAT of 7%

HINT:

```
// VAT at 7%
return this.price * 1.07;
```

- Declare a class `HotelRoom` which has:
  - private `name` String instance variable - with accessor methods
  - private `price` double instance variable - with accessor methods
  - implements your interface, returning the price plus VAT of 3%
- create a `main(...)` method in class `Main` that does the following:
  - create a new `Book` object `book1`: title = `Life of Pi` / price = 10.00
  - create a new `HotelRoom` object `hotelRoom1`: name = `Radison Green` / price = 200.00
  - output the prices of these objects with and without VAT

OUTPUT:

```
 matt$ java Main
book1 price (excl. VAT) = 10.0
book1 price (incl. VAT) = 10.700000000000001
room1 price (excl. VAT) = 200.0
room1 price (incl. VAT) = 206.0
```

NOTE: - you might like to use your `DecimalFormat` skills from an earlier exercise to restrict price outputs to 2d.p. …

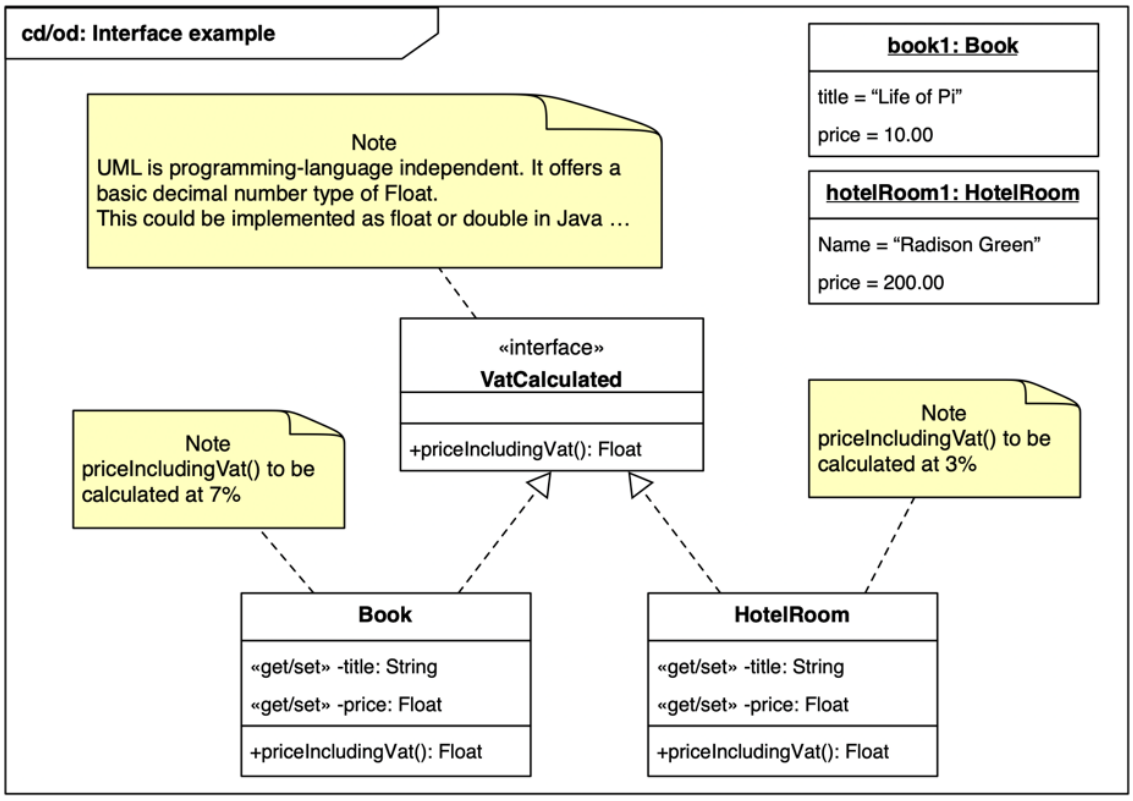


Figure 6.6: Class-Object diagram for interface example.

7

Exceptions

7.1 Exceptions: Java class: `Exception`

Exceptions are a powerful feature of OO programming. They allow us to manage *anticipated* and *recoverable* problems.

Exceptions are different from Errors: - **Errors** are *unexpected* circumstances or events from which it's usually *not possible to recover*, e.g. - Out of memory - No such constant or class

Some examples of things going wrong: - empty string - negative age - invalid menu choice - missing file / bad file name or file extension - file may not upload due to network error - out-of-date code refers to constant no longer defined - data from a user (or JSON web API) wrong data type or malformed - computer run out of memory

7.2 Exercise - write code to generate an exception

- Create a `Main` class containing a `main()` method
  - on separate statements print out `one`, `two`, `three`
  - before the statement to print out `three` write a divide by zero calculation, e.g. `int a = 10 / 0`

OUTPUT:

```
$ java Main
one
two

XX uncaught Exception stack trace XX
```

List of References