CS 2103: Class 3

Jacob Whitehill

Avoid redundancy:
Program decomposition & Code refactoring

Redundancy: example

```
void resetAccount (State state) {
        final String name = state.getLoginName();
        if (! state.isLoggedIn()) {
                state.logIn(name);
        }
        final Account account = getCustomerAccountByName(name);
        final float balance = account.getBalance();
        if (account.needCreditCheck()) {
                if (! account.creditIsOk()) {
                        throw new BadCreditException("Credit is bad");
                }
        }
        if (balance < 0 || state.mustPayAll()) {
                payBalance(account, balance);
                state.getWindowManager().sendConfirmationEmail(account.getEmail(), "Confirmation");
        }
}

void redeemGiftCard (State state, float giftCardAmount) {
        final String name = state.getLoginName();
        if (! state.isLoggedIn()) {
                state.logIn(name);
        }
        final Account account = getCustomerAccountByName(name);
        final float balance = account.getBalance();
        account.setBalance(balance + giftCardAmount);

        if (balance < 0 || state.mustPayAll()) {
                if (askUser(name, "Pay balance?")) {
                        payBalance(account, balance);
                        state.getWindowManager().sendConfirmationEmail(account.getEmail(), "Thanks");
                }
        }
}
```

What are different ways of creating helper methods to "factor out" the common code below?

```
void resetAccount (State state) {
        final String name = state.getLoginName();
        if (! state.isLoggedIn()) {
                state.logIn(name);
        }
        final Account account = getCustomerAccountByName(name);
        final float balance = account.getBalance();
        if (account.needCreditCheck()) {
                if (! account.creditIsOk()) {
                        throw new BadCreditException("Credit is bad");
                }
        }
        if (balance < 0 || state.mustPayAll()) {
                payBalance(account, balance);
                state.getWindowManager().sendConfirmationEmail(account.getEmail(), "Confirmation");
        }
}

void redeemGiftCard (State state, float giftCardAmount) {
        final String name = state.getLoginName();
        if (! state.isLoggedIn()) {
                state.logIn(name);
        }
        final Account account = getCustomerAccountByName(name);
        final float balance = account.getBalance();
        account.setBalance(balance + giftCardAmount);

        if (balance < 0 || state.mustPayAll()) {
                if (askUser(name, "Pay balance?")) {
                        payBalance(account, balance);
                        state.getWindowManager().sendConfirmationEmail(account.getEmail(), "Thanks");
                }
        }
}
```

Version 1

```
Account getAccount (State state) {
        final String name = state.getLoginName();
        if (! state.isLoggedIn()) {
                state.logIn(name);
        }
        return account = getCustomerAccountByName(name);
}

void payBalanceAndConfirm (State state, Account account, float balance, String message) {
        payBalance(account, balance);
        state.getWindowManager().sendConfirmationEmail(account.getEmail(), message);
}

void resetAccount (String name) {
        final Account account = getAccount(state);
        final float balance = account.getBalance();
        if (account.needCreditCheck()) {
                if (! account.creditIsOk()) {
                        throw new BadCreditException("Credit is bad");
                }
        }
        if (balance < 0 || state.mustPayAll()) {
                payBalanceAndConfirm(state, account, balance, "Confirmation");
        }
}

void redeemGiftCard (String name, float giftCardAmount) {
        final Account account = getAccount(state);
        final float balance = account.getBalance();
        account.setBalance(balance + giftCardAmount);

        if (balance < 0 || state.mustPayAll()) {
                if (askUser(name, "Pay balance?")) {
                        payBalanceAndConfirm(state, account, balance, "Thanks");
                }
        }
}
```

Version 2 (too heavy)

```
// Overly factored -- lots of if statements, which make it hard to understand
void resetAndRedeemGiftCardHelper (State state, boolean hasGiftCard, float giftCardAmount,
                                   boolean checkCredit, boolean askToPayBalance, String message) {
        final String name = state.getLoginName();
        if (! state.isLoggedIn()) {
                state.logIn(name);
        }
        final Account account = getCustomerAccountByName(name);
        final float balance = account.getBalance();
        if (hasGiftCard) {
                account.setBalance(balance + giftCardAmount);  // might be a slow operation
        }
        if (checkCredit && account.needCreditCheck()) {
                if (! account.creditIsOk()) {
                        throw new BadCreditException("Credit is bad");
                }
        }

        if (balance < 0 || state.mustPayAll()) {
                if (! asToPayBalance || askUser(name, "Pay balance?")) {
                        payBalance(account, balance);
                        state.getWindowManager().sendConfirmationEmail(account.getEmail(), message);
                }
        }
}

void resetAccount (String name) {
        resetAndRedeemGiftCardHelper(name, false, 0, true, false, "Confirmation");
}
void redeemGiftCard (String name, float giftCardAmount) {
        resetAndRedeemGiftCardHelper(name, true, giftCardAmount, false, true, "Thanks");
}
```

Refactoring with helper methods: considerations

- Does each helper method have a cohesive definition, or does it “glue” together random parts?

- Is the refactored code easier or harder to read than before?

- Is the amount of code reduced?

Avoid redundancy: Program decomposition & Code refactoring: Classes

Redundancy in classes

- Consider the following classes:

```
class X {
  int _someVar;

  String hi (String name) {
    return "Hi: " + name + " " + someMethod();
  }

  int someMethod () {
    return _someVar * 6;
  }
}
```

```
class Y {
  int _someVar;

  int someMethod () {
    return _someVar * 6;
  }

  void bye () {
    System.out.println("Bye: " + someMethod());
  }
}
```

Redundancy in classes

- Consider the following classes:

```
class X {
  int _someVar;

  String hi (String name) {
    return "Hi: " + name + " " + someMethod();
  }

  int someMethod () {
    return _someVar * 6;
  }
}
```

```
class Y {
  int _someVar;

  int someMethod () {
    return _someVar * 6;
  }

  void bye () {
    System.out.println("Bye: " + someMethod());
  }
}
```

Redundancy

Redundancy in classes

- Using **inheritance**, we can refactor these as:

```
class X extends Z {
  String hi (String name) {
    return "Hi: " + name + " " + someMethod();
  }
}

class Y extends Z {
  void bye () {
    System.out.println("Bye: " + someMethod());
  }
}
```

```
class Z {
  int _someVar;

  int someMethod () {
    return _someVar * 6;
  }
}
```

Redundancy in classes

- Using **ownership**, we can refactor these as:

```
class X {
  Z _z;

  String hi (String name) {
    return "Hi: " + name + " " + _z.someMethod();
  }
}
```

```
class Y {
  Z _z;

  void bye () {
    System.out.println("Bye: " + _z.someMethod());
  }
}
```

```
class Z {
  int _someVar;

  int someMethod () {
    return _someVar * 6;
  }
}
```

Redundancy in classes

- Two of the chief ways of reducing redundancy in classes are:
  - **Inheritance**: factor out the redundant methods & instance variables into a common parent class *P*, and then inherit/subclass from *P* in multiple subclasses.
  - **Ownership**: factor out the redundant methods & instance variables into a class *S*, and then add an instance variable to *S* in multiple other classes.

Why Java?

Java

- Java is a compiled, “mid-level” language that runs on a virtual machine.
- “High”, “low” and “mid”-level languages refer to the level of abstraction.
- More abstract than C:
  - E.g., can’t manipulate memory directly using pointers.
- Less abstract than Python:
  - E.g., can’t just call “`range`” to create a list of numbers; need to manually construct an array.

Compilation/Translation

- Before a programming language such as C/C++ can be executed by the physical CPU, it must be compiled into something the CPU can understand.

- The native language of a CPU is its **assembly language**.

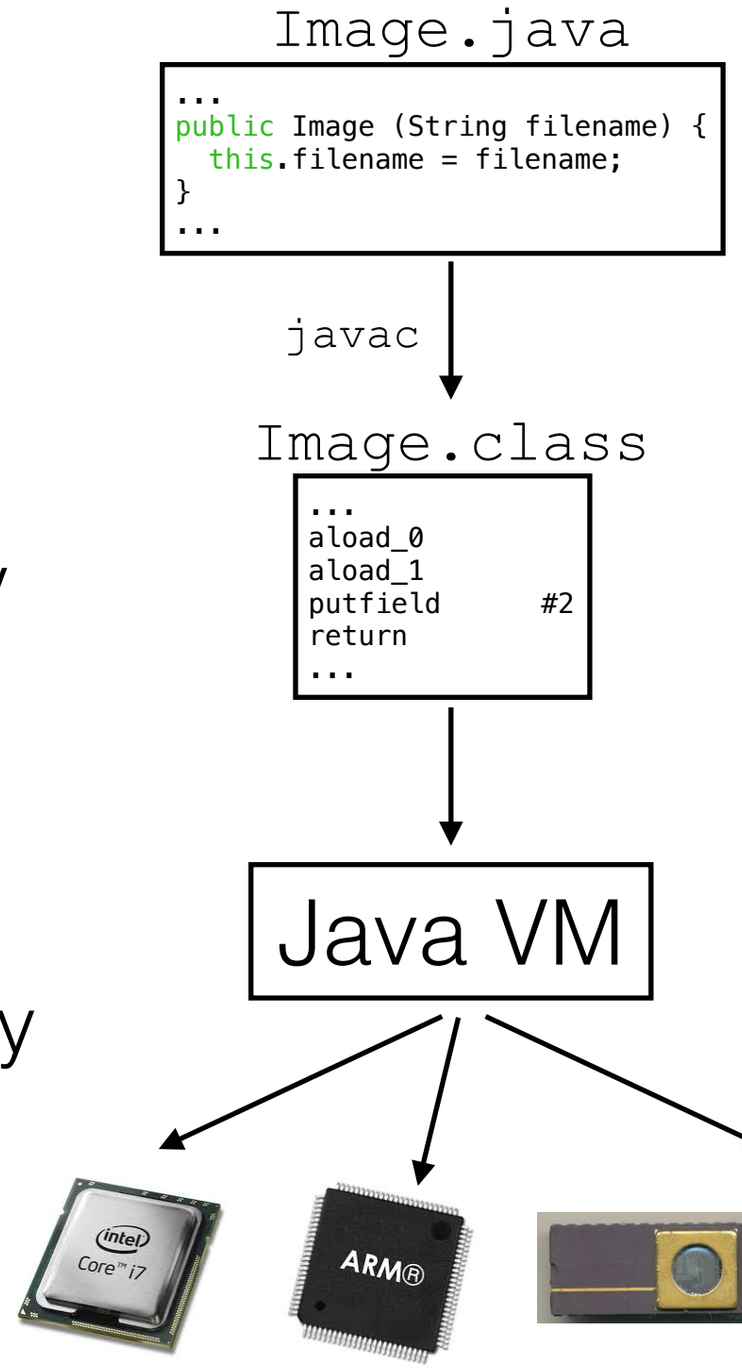
Java VM

- Java code is **not** compiled into assembly language instructions that can be directly executed on the host CPU (e.g., Intel i7, ARM).

- Instead Java runs on a **virtual machine** (VM).

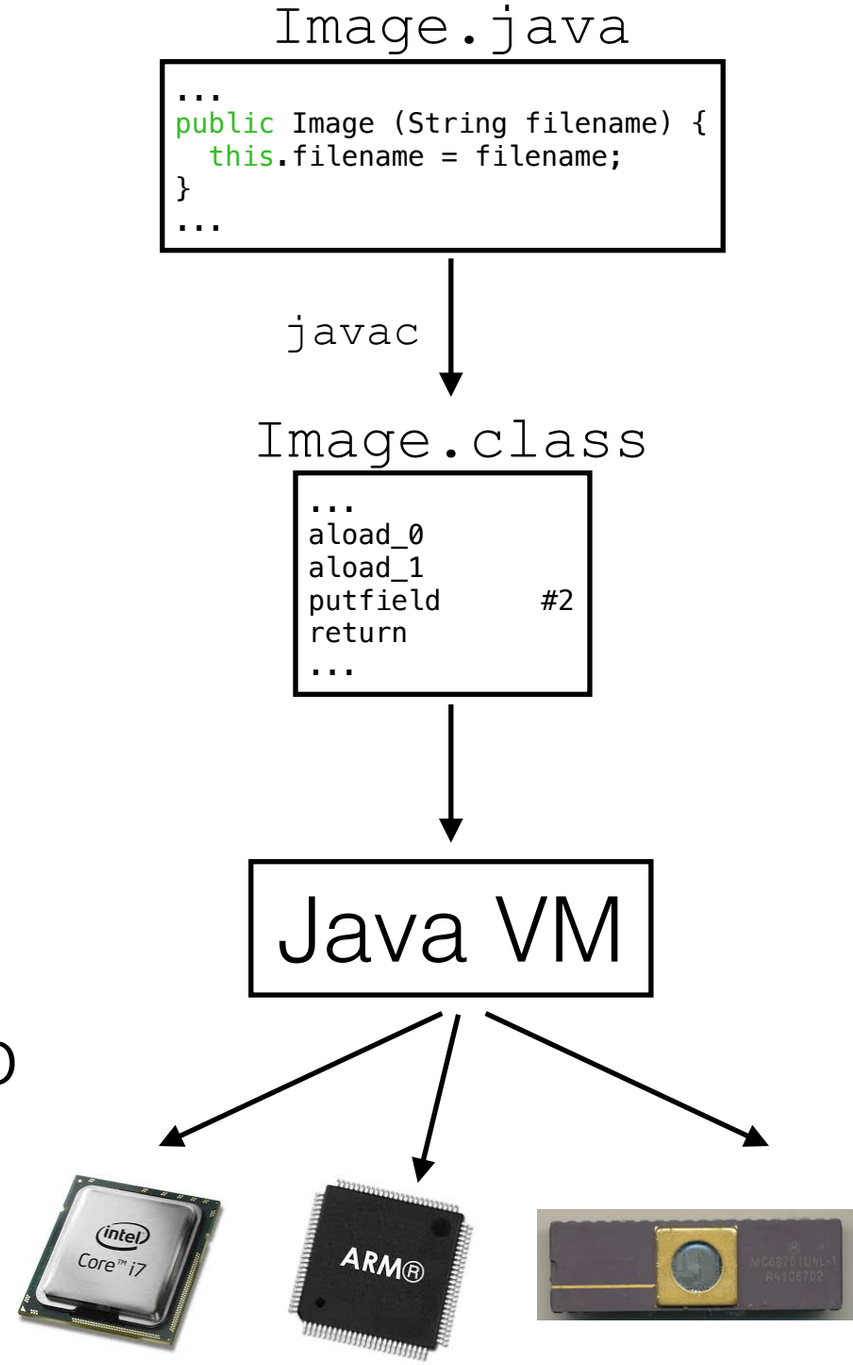
Java VM

- The `javac` compiler compiles Java source (.java) into bytecode (.class).
- These bytecode files are the “native language” of the Java VM.
- Java VM implementations exist for many operating systems and hardware platforms.
- This makes Java very portable because the same .class files can be run on many devices without being recompiled.



Java VM

- The VM has implications for how to manage memory in Java:
  - Once an object is no longer needed, it is automatically deallocated.
- It is impossible to make certain kinds of mistakes that ubiquitous in C/C++.
- The programmer does not have to keep track of which memory blocks to “free”.



Why Java?

- Java is arguably **more secure** than some languages (e.g., C) because of features such as:
  - Type checking
  - Array-bounds checking

Why Java?

- Java is usually **slower** than C:
  - Java runs on a VM;
    C runs directly on underlying CPU.
  - Java implements run-time security features;
    C just assumes everything is fine.

Why Java?

- Java is particularly well-suited for:
  - Enterprise computing.
  - Mobile app development (specifically Android).
  - (Some) scientific simulations.

Enterprise computing

- Enterprise computing applications typically involve **complex business logic**; they include:
  - Large-scale billing systems for healthcare, insurance, etc.
  - Online banking platforms
  - Stocks & options trading systems

Mobile app development

- Java offers (fairly) high performance, security, and **portability**:

- Compiled Java apps can run on many different hardware platforms.

Java in 2020

- The software landscape is changing:
  - **Server-side**: Node.js is being used for more and more large-scale web applications.
  - **Client-side**: Javascript+HTML5 is increasingly powerful, and highly portable. Google now promotes a new language, Kotlin.
  - **Scientific computing**: Python is very popular.

Classes

Classes

- Java supports the creation of **objects** that all belong to a particular **class**.

- An object relates a set of *attributes* with a set of *actions* that can manipulate the attributes.

- Examples of a class:
  - `Person`
  - `Profile`
  - `Message`
  - `Image`

Anatomy of a class

```
public class Person {
   private String _name;
   private int _age;

   public Person () {
   }
   public Person (String name, int age) {
      _name = name;
      _age = age;
   }
   public Message createMessageFor (Person other) {
      Message message = new Message(...);
      // ...
      return message;
   }
}
```

Anatomy of a class

```
public class Person {
   private String _name;
   private int _age;

   public Person () {
   }
   public Person (String name, int age) {
      _name = name;
      _age = age;
   }
   public Message createMessageFor (Person other) {
      Message message = new Message(...);
      // ...
      return message;
   }
}
```

instance (or member) variables

Anatomy of a class

```
public class Person {
   private String _name;
   private int _age;

   public Person () {
   }
   public Person (String name, int age) {
      _name = name;
      _age = age;
   }
   public Message createMessageFor (Person other) {
      Message message = new Message(...);
      // ...
      return message;
   }
}
```

instance methods (or member functions)

Anatomy of a class

```
public class Person {
   private String _name;
   private int _age;

   public Person () {
   }
   public Person (String name, int age) {
      _name = name;
      _age = age;
   }
   public Message createMessageFor (Person other) {
      Message message = new Message(...);
      // ...
      return message;
   }
}
```

Constructors that are invoked when a new object of the class is instantiated

Anatomy of a class

```
public class Person {
   private String _name;
   private int _age;

   public Person () {
   }
   public Person (String name, int age) {
      _name = name;
      _age = age;
   }
   public Message createMessageFor (Person other) {
      Message message = new Message(...);
      // ...
      return message;
   }
}
```

access modifiers
(more on these later)

Subclasses

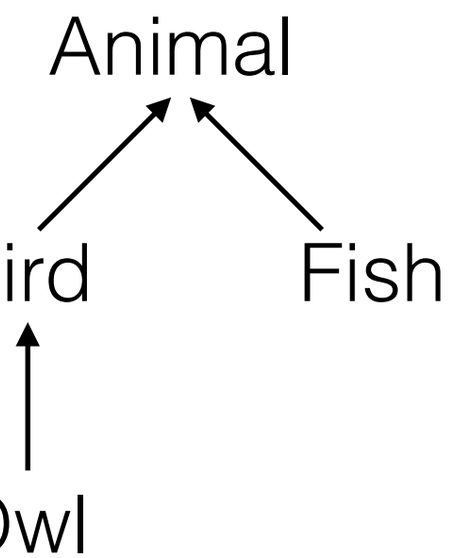
- We can also create a more specialized class of objects by subclassing/inheriting/extending a base class, e.g.:

```
class Animal {
  private String _name;
  private int _age;
  public void eat () { ...
  }
}
```

```
class Fish extends Animal {
  public void swim () { ...
  }
}
```

```
class Bird extends Animal {
  public void flapWings () { ...
  }
}
```

```
class Owl extends Bird {
  public void twistHeadAllTheWayAround () { ...
  }
}
```



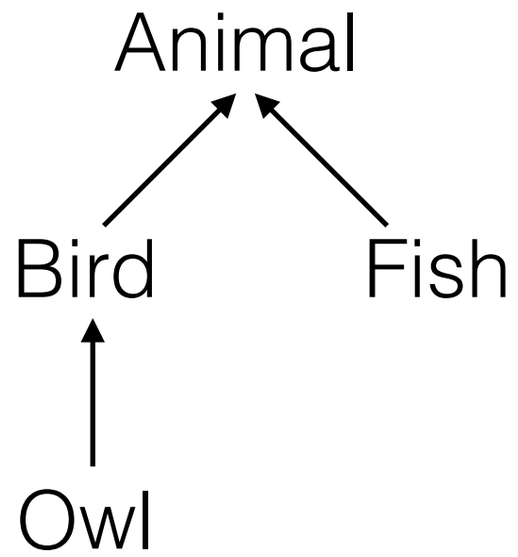
Subclasses

- An **abstract class** is a class that cannot be instantiated; only one of its (non-abstract) subclasses can be instantiated:

```
abstract class Animal {
  private String _name;
  private int _age;
  public void eat () { ...
  }
}

...

Animal animal = new Animal();  // compiler error
Fish fish = new Fish();  // ok
```



Subclasses

- Note that, in Java, each class may have **at most** one parent class.

- Hence, you should think carefully about which class to choose as a parent (if any).

- Do not subclass from a parent class too eagerly — think carefully whether it's worth it.

Subclasses

- In this course I will assume you are already familiar with:
  - subclassing with `extends`
  - overriding methods and calling parents' implementations using `super`
- Our focus will be on **how to use** these tools effectively.

Designing classes

- A class should represent a **coherent** set of attributes and actions that **belong together**.
    - It's not just an arbitrary way of grouping parts of the code.

- The name of a class should be a **singular noun** that can be **pluralized**.

Examples of **well**-conceived classes

- `Person` — name, age, hobbies, preferences, etc.
- `FriendRequest` — a request that two people become friends.
- `Image` — a buffer to store the values of the pixels along with methods to modify them.

Examples of **ill**-conceived classes

- `MovieWithProductionCompany`
  - This is combining information from two different things.

Examples of **ill**-named classes

- `ImageAnalyzing`

  - Does the following really make sense?

```
ImageAnalyzing analyzing =
  new ImageAnalyzing(); // bad
```

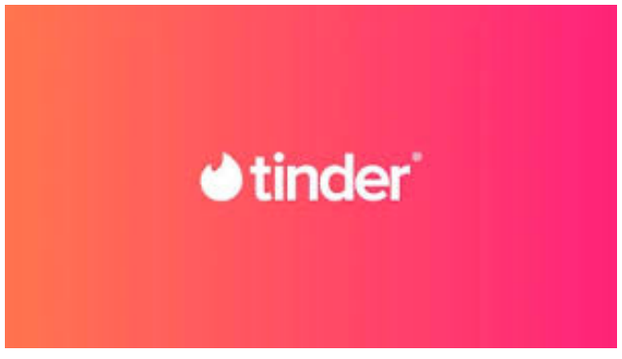
  - What does it mean to be an “object” of “analyzing”??

Examples of **ill**-named classes

- `ImageAnalyzer`
  - This works much better:

```
ImageAnalyzer analyzer =
  new ImageAnalyzer();  // better
```

You may have heard of:



match.com

But have you heard of:

?

PetDate.net

- PetDate.net is a revolutionary new startup that facilitates dating for pets.
- Suppose you are a software engineer building PetDate.net's dating service — what classes would you need to create?

Possible classes

- Pet (and maybe some specific subclasses?)
- Owner
- Dates:
  - Classic (2-pet) dates
  - Double dates
  - Group dates
- Messages
- Relationships
- …

A few possible classes

```
abstract class Being {
  private String _name;
  private String _address;
}

class Person extends Being {
  private String _creditCardNumber;
}

class Pet extends Being {
  private Person _owner;
  private Image _profilePicture;
  private Date[] _history;
}

abstract class Date {
  private Time _time;  // when it happened
  private String _description;  // what went on
}

class ClassicDate extends Date {
  private Pet _p1, _p2;
  private Image _snapshot;
  private String _whatHappened;
}
```

Inheritance vs. ownership

Inheritance versus ownership

- Inheritance is an **often overused** tool in OOP.

- Very often, **ownership** should be used instead.

Inheritance versus ownership

- Suppose we want every `Pet` in PetDate.net to have a profile picture (`Image`).

- `Image` has several methods, including:
  - `double getWidth() { ... }  // gets width of image`
  - `double getHeight() { ... }  // gets height of image`
  - `boolean isGrayscale () { ... }  // checks if it's grayscale`

Inheritance versus ownership

- Rather than duplicate these methods, `Pet` can "borrow" this functionality from `Image` via:

- Inheritance

- Ownership

Inheritance

- ```
  class Pet extends Image {  // Inheritance
     ...
  }
  ```

- Now, `Pet` automatically has `getWidth()`, `getHeight()`, and `isGrayscale()` methods.

- How handy!
  ```

Inheritance

- ```
  class Pet extends Image {  // Inheritance
     ...
  }
  ```

- Advantage:

1. Simple — just two words in the declaration.
  ```

Inheritance

- ```
  class Pet extends Image {  // Inheritance
     ...
  }
  ```

- Disadvantages:

1. Inflexible: With Java, `Pet` can no longer inherit from any other parent class.
  ```

Inheritance

- ```
  class Pet extends Image {  // Inheritance
     ...
  }
  ```

- Disadvantages:

2. Awkward semantics: is `Pet` really a special type of `Image`??
  ```

Inheritance

- ```
  class Pet extends Image {  // Inheritance
     ...
  }
  ```

- Disadvantages:

3. Unsafe: `Image` has many other methods that have nothing to do with a `Pet`, e.g.:
   `boolean isGrayscale() { ... }`
  ```

Inheritance

- ```
  class Pet extends Image {  // Inheritance
     ...
  }
  ```

- Disadvantages:

3. Unsafe: `Image` has many other methods that have nothing to do with a `Pet`. We do not want these methods to be callable on objects of type `Pet` (could be dangerous):

```
Pet person;
pet.isGrayscale();  // yuck!
```
```

Ownership

- ```
  class Pet {
    private Image _image;  // ownership
  }
  ```

- Alternatively, `Pet` can *own* an `Image` object.

- To access `Image`'s `getWidth()` and `getHeight()` methods, `Pet` just needs to **delegate** to `Image`...
  ```

Ownership

```
class Pet {
  private Image _image;  // ownership

  public double getImageWidth () {
    return _image.getWidth();  // delegation
  }

  public double getImageHeight () {
    return _image.getHeight();  // delegation
  }
}
```

- **Delegation**: “forward” a message sent to class *A* (`Pet`) to another class *B* (`Image`).

Ownership

- ```
  class Pet {
    private Image _image;  // ownership

    ...
  }
  ```

- Advantages:

    1. Flexible: still allows `Pet` to inherit from any other class.
  ```

Ownership

- ```
  class Pet {
    private Image _image;  // ownership

    ...
  }
  ```

- Advantages:

  2. Safer: `Pet` only exposes the *necessary* functionality of `Image` that it needs.
  ```

Ownership

- ```
  class Pet {
    private Image _image;  // ownership

    ...
  }
  ```

- Advantages:

  3. Cleaner semantics: `Pet` and `Image` are (appropriately) no longer part of the same class hierarchy.
  ```

Ownership

```
class Pet {
  private Image _image;  // ownership

  public getImageWidth () {
    return _image.getWidth();  // delegation
  }

  public getImageHeight () {
    return _image.getHeight();  // delegation
  }
}
```

- Disadvantage:

  1. More code: we have to write delegating methods.

Design choice

- When making architectural decisions in OOP, there are usually **trade-offs**.

- Overall, for this example I would recommend ownership rather than inheritance.