CS 2103: Class 4

Jacob Whitehill

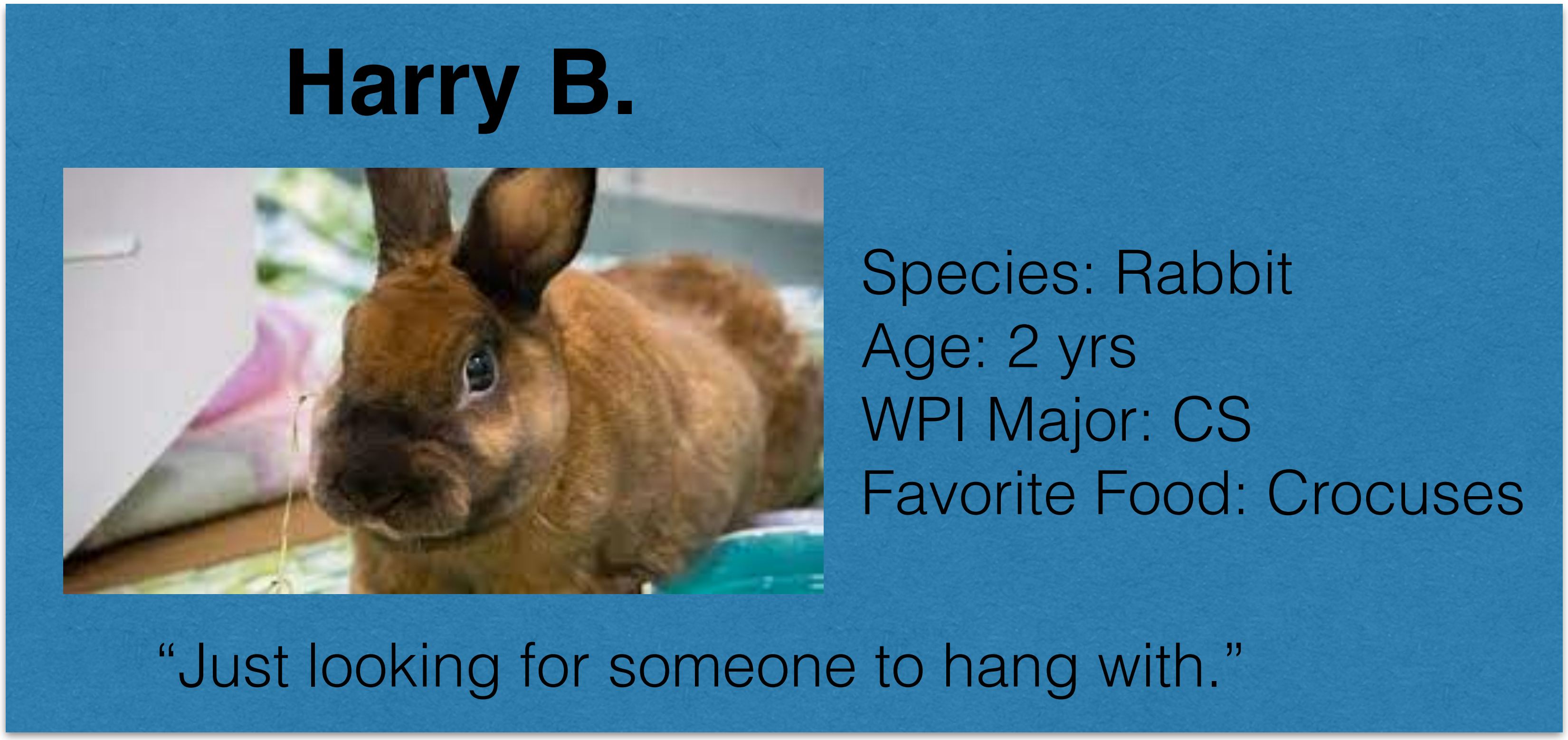
Inheritance vs. ownership

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 `Profile` in PetDate.net to have a profile picture (`Image`).



Harry B.

Species: Rabbit
Age: 2 yrs
WPI Major: CS
Favorite Food: Crocuses

"Just looking for someone to hang with."

Inheritance versus ownership

- Suppose we want every `Profile` 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`
  - `void convertToGrayscale () { ... }`

Inheritance versus ownership

- Rather than duplicate these methods, `Profile` can “borrow” this functionality from `Image` via:
  - Inheritance
  - Ownership

Inheritance

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

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

- How handy!

Inheritance

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

- Advantage:

  1. Simple — just two words in the declaration.

Inheritance

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

- Disadvantages:

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

Inheritance

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

- Disadvantages:

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

Inheritance

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

- Disadvantages:

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

Inheritance

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

- Disadvantages:

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

     ```
     ...
     profile.convertToGrayscale();  // yuck!
     ```

Ownership

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

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

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

Ownership

-
  ```
  class Profile {
    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* (`Profile`) to another class *B* (`Image`).

Ownership

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

    ...
  }
  ```

- Advantages:

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

Ownership

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

    ...
  }
  ```

- Advantages:

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

Ownership

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

    ...
  }
  ```

- Advantages:

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

Ownership

-
  ```
  class Profile {
    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.

Inheritance vs. Ownership

- Use inheritance sparingly — each class can have at most one parent class.

- Inheritance usually conveys an “is a” relationship (e.g., `Fish` **is a**n `Animal`).

- Ownership often conveys a “has a” relationship (e.g., a `Profile` **has a**n `Image`).

Design choice

- When making architectural decisions in OOP, there are usually **trade-offs**.
- Overall, for this example I would recommend ownership rather than inheritance.

Notation

Notation

- Using consistent naming conventions while programming in any language is:
  - Not important for the compiler/interpreter.
  - Very important for other humans.

Notation in CS 2103

- The name of a **Java class** should be in **mixed-case** like this:
  - `class ImageAnalyzer`
- It should **not** be any of the following:
  - `class imageAnalyzer  // camel-case`
  - `class IMAGEANALYZER  // all-caps`
  - `class Image_Analyzer  // underscored`
  - `class imageanalyzer  // all lower-case`
  - `class IMageAnalyzer  // just sloppy`

Notation in CS 2103

- Instance variables, local variables, method parameters, and instance methods should all be written in camel-case, e.g.:

    -
      ```
      class Person {
        private int _minAge, _maxAge;

        public void sendMessage (Person personToWrite) {
          Message theMessage = new Message();
          // ...
        }
      }
      ```

Notation in CS 2103

- Some programmers like to denote each instance variable with an underscore _ or an "m" that precedes the rest of the name, e.g.:

  -
    ```
    class Person {
      private int _minAge, _maxAge;

      public void sendMessage (Person personToWrite) {
        Message theMessage = new Message();
        // ...
      }
    }
    ```

- Either (or none) is ok — just be consistent.

Notation in CS 2103

- Some programmers like to denote each instance variable with an underscore _ or an “m” that precedes the rest of the name, e.g.:

  -
    ```
    class Person {
      private int mMinAge, mMaxAge;

      public void sendMessage (Person personToWrite) {
        Message theMessage = new Message();
        // ...
      }
    }
    ```

- Either (or none) is ok — just be consistent.

Notation in CS 2103

- Constants (values that never change) should be declared as `static final` and be named with **all-caps with underscores**, e.g.:

  -
    ```
    class Person {
      protected static final int MAX_AGE = 130;
      // ...
    }
    ```

Code Structure in CS 2103

- Keep methods <= 50 lines for readability.
    - If your method is much longer, that's likely a sign that your method is trying to do too much and should be decomposed into multiple methods.

Access modifiers

Access modifiers

- One way to avoid bugs in a programming project is to allow the programmer to **access only what they need** (“need to know basis”).

- **Rationale**: If a variable/method in class `A` cannot be accessed from class B, then class `B` cannot possibly mess it up.

Access modifiers

- To facilitate this “need to know” behavior, Java offers four access modifiers:
    - Most restrictive `private`
    - (default) — “package-private”
    - `protected`
    - Least restrictive `public`

private

- Only methods within the same class can access the variable/method/class.

  -
    ```
    public class A {
      private int _number;
      public void f () {
        _number = 5;  // ok
      }
    }
    public class B {
      public void g () {
        final A a = new A();
        a._number = 5;  // error
      }
    }
    ```

private

- Not even subclasses can access private members of a parent/ancestor class:

  -
    ```
    public class A {
      private int _number;
      public void f () {
        _number = 5;  // ok
      }
    }
    public class S extends A {
      public void g () {
        _number = 5;  // error
      }
    }
    ```

(default) package-private

- Java classes can belong to “packages”, e.g.: `java.util.ArrayList` is in the `java.util` package.

- Classes in the `java.util` package must be in the `java/util` directory and must declare “`package java.util;`” at the top of the file.

(default) package-private

- Package-private variables/methods/classes can be accessed by every class within the same package:

  -
    ```
    package somePackage;
    public class A {
      String _name;  // no modifier; hence, package-private
    }
    ```

  -
    ```
    package somePackage;
    public class B {
      public void f () {
        final A a = new A();
        a._name = "Zeus";  // ok
      }
    }
    ```

protected

- `protected` class members can be accessed from classes within the same package **and by subclasses**:

  -
    ```
    public class A {
      protected int _number;
      public void f () {
        _number = 5;  // ok
      }
    }
    public class S extends A {
      public void g () {
        _number = 5;  // ok
      }
    }
    ```

`public`

- `public` class members can be accessed from **any class**.

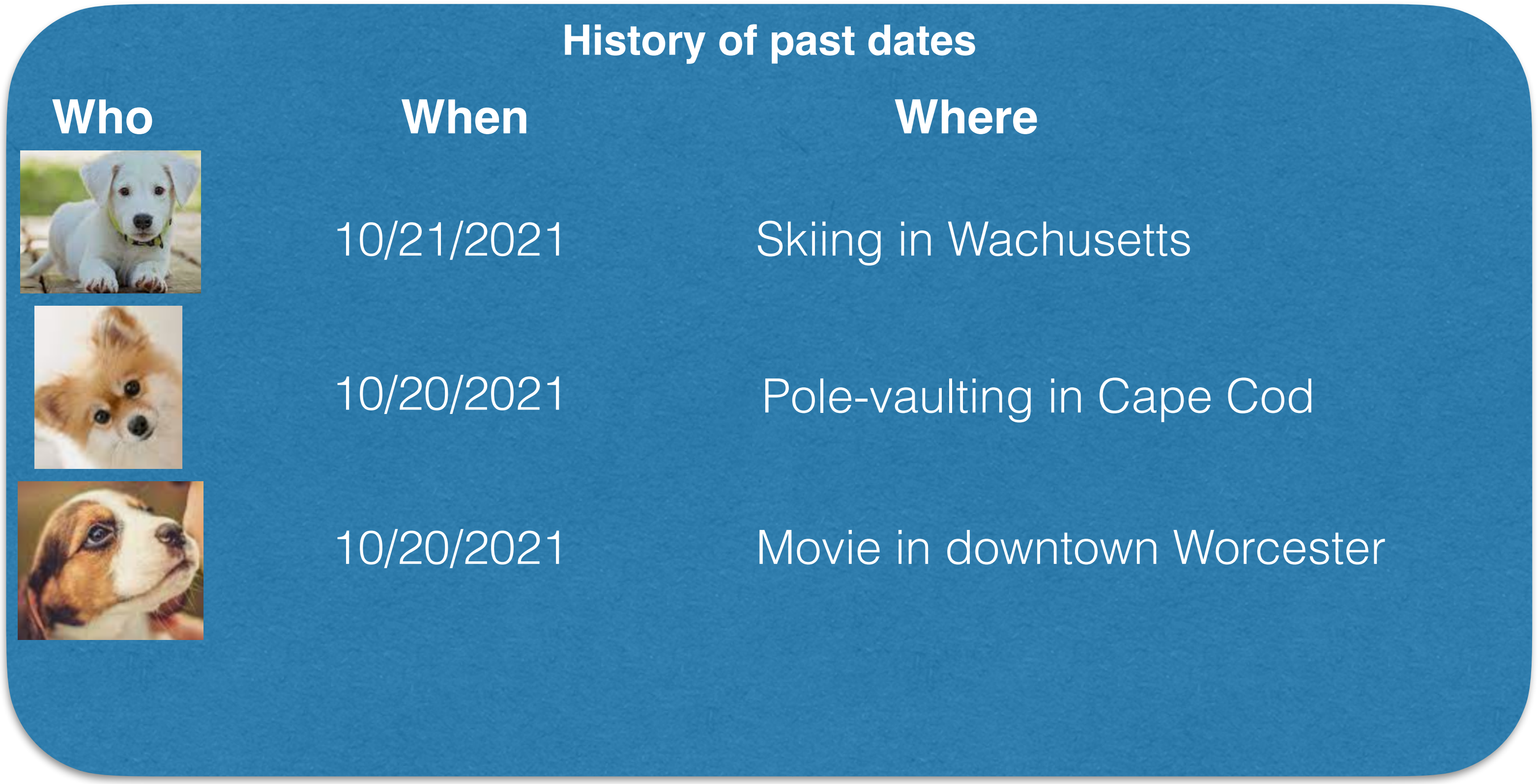
Guidelines on using privacy modifiers

- In “real-world” projects involving large teams of programmers:
    - If you make something public, someone will eventually use it.
    - If you later decide it’s too dangerous to keep public, it will be difficult to restrict access (since code will break).
    - Hence, start with the most restrictive access you can get away with.
    - When needed, provide the least access needed to do the job.
        - E.g., if only subclasses need access, then make it `protected`.

Interfaces

History of past dates

- PetDate.net allows users to look at their history of past dates, e.g.:



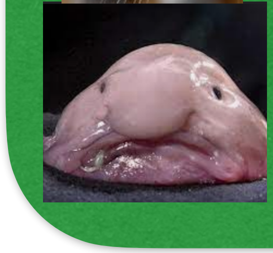
History of past dates

Who	When	Where
	10/21/2021	Skiing in Wachusetts
	10/20/2021	Pole-vaulting in Cape Cod
	10/20/2021	Movie in downtown Worcester

List of new members

- It also allows users to search through new PetDate.net members who recently joined:

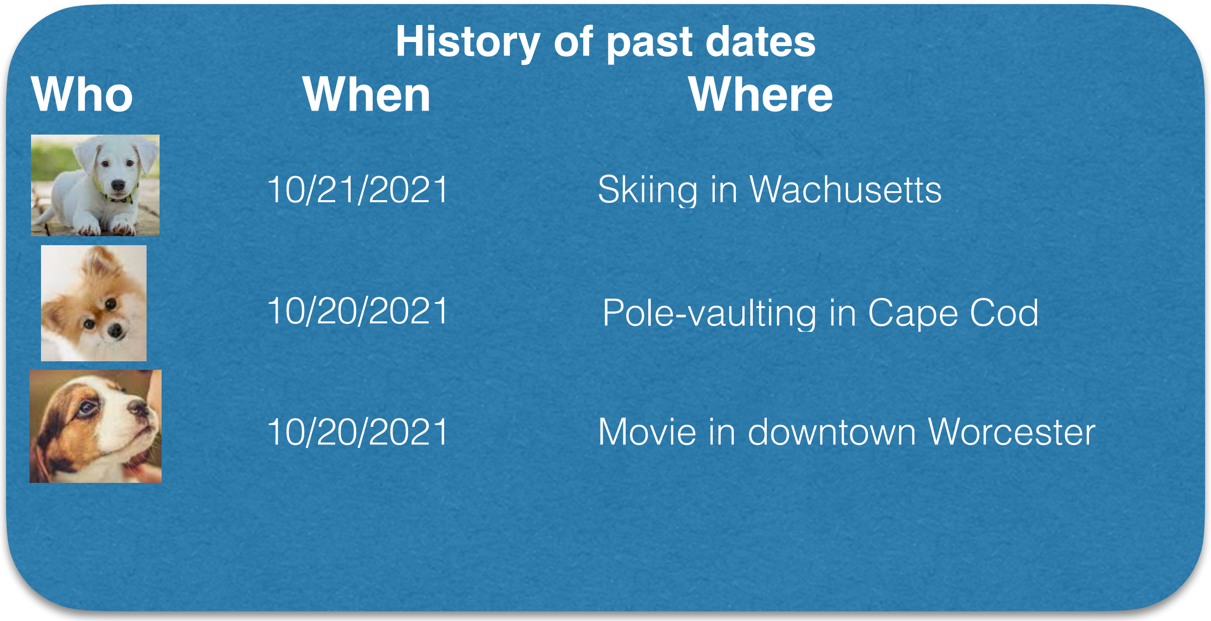
Newly joined members

Pic	Age	Name
	21	Leonardo
	29	Matt
	81	Humphrey

Modeling problem

- We want to be able to call:

  -
    ```
    final ListBox dateListBox = new ListBox();
    dateListBox.addItem(dateWithMike);
    dateListBox.addItem(dateWithFrank);
    dateListBox.addItem(dateWithHairy);
    ```



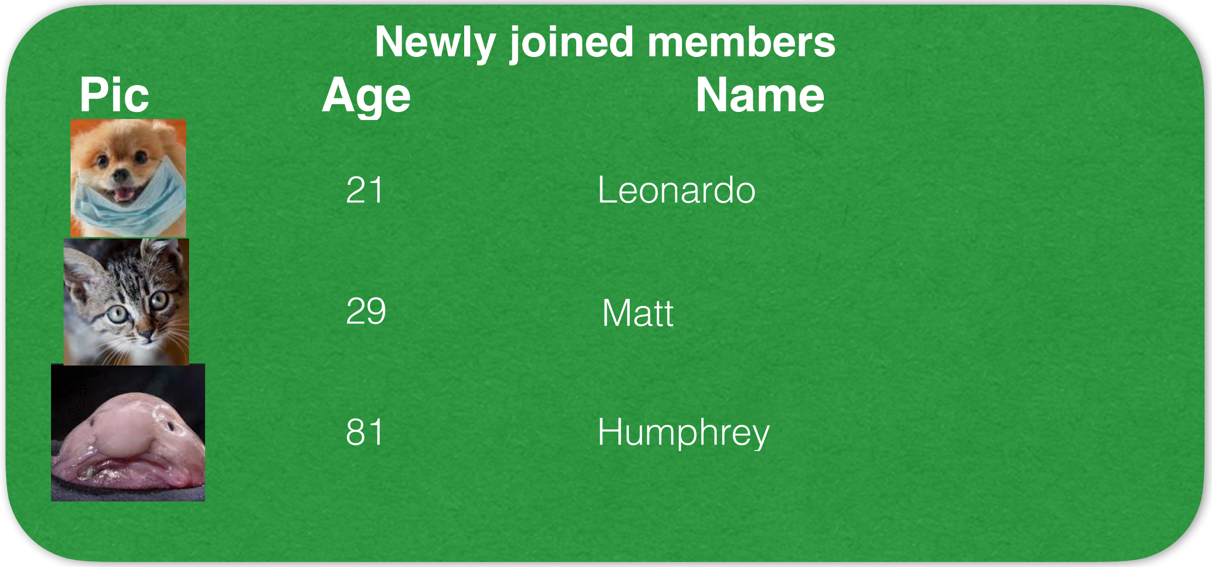
History of past dates

Who	When	Where
	10/21/2021	Skiing in Wachusetts
	10/20/2021	Pole-vaulting in Cape Cod
	10/20/2021	Movie in downtown Worcester

Modeling problem

- But we also want to be able to call:

  -
    ```
    final ListBox petsListBox = new ListBox();
    petsListBox.addItem(leonardo);
    petsListBox.addItem(matt);
    petsListBox.addItem(humphrey);
    ```



Newly joined members

Pic	Age	Name
	21	Leonardo
	29	Matt
	81	Humphrey

Modeling problem

- Suppose we want to create a general GUI component `ListBox` that can display a list of “things” that contain a **picture** and a **description**.

-
  ```
  class ListBox {
    public void addItem (         item) {
      ...
    }
  }
  ```

- What **type** should go in the blank so that we can add both **pets** and **dates**?

Modeling problem

- Strategy 1 — create a common ancestor class:

    -
      ```
      class ListableObject {
        private Image _image;
        private String _description;
      }
      ```

    -
      ```
      class Pet extends ListableObject {
          …
      }
      ```

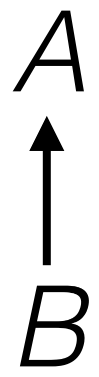
    -
      ```
      class Date extends ListableObject {
        …
      }
      ```

Modeling problem

- We could then define `addItem` to take an `item` of type `ListableObject`.

-
  ```
  class ListBox {
    public void addItem (ListableObject item) {
      ...
    }
  }
  ```

- Problem: cannot add an item from a class *B* that already has a parent class *A*.



Modeling problem

- Using the class hierarchy is the wrong tool for this job.
- All we want is enforce that every object we add to the `ListBox` must have a **picture** and a **description**.
- Other than that, **we don't care** what kind of object it is.

Java interfaces

- **Strategy 2**: use Java **interfaces**.
    - An **interface** is a collection of methods signatures & descriptions of what they do. (**Signature**: a method's name, parameters, and return type.)
    - Interfaces are a **more flexible kind of type** than classes.
    - Interfaces allow you to specify a **set of methods** that an object must support.

Java interface: definition

- We can create a Java **interface** as follows:

  -
    ```
    /**
     * Interface for any object that wants to be shown inside
     * a ListBox. It must have an image and a description.
     */
    interface Listable {
      /**
       * Returns the image associated with this item
       */
      public Image getImage ();

      /**
       * Returns the description associated with this item
       */
      public String getDescription ();
    }
    ```

- Interfaces contain method **names**, **parameters**, and **return types**, but **no bodies**.

Java interface: definition

- We can create a Java **interface** as follows:

    -
      ```
      /**
       * Interface for any object that wants to be shown inside
       * a ListBox. It must have an image and a description.
       */
      interface Listable {
        /**
         * Returns the image associated with this item
         */
        public Image getImage ();

        /**
         * Returns the description associated with this item
         */
        public String getDescription ();
      }
      ```

- Methods with no bodies are called **abstract**.

Java interface: implementation

- Before we can use an interface, we must implement it.

- We can implement the interface in `Pet`:

  -
    ```
    class Pet extends Person implements Listable {
      private Image _profilePic;
      ...
      public Image getImage () {
        return _profilePic;
      }
      public String getDescription () {
        return getName();  // from superclass (Being)
      }
    }
    ```

- **Implementing** an interface: create a body for every method in the interface.

Java interface: implementation

- Before we can use an interface, we must implement it.

- We also implement the interface in `Date`:

  -
    ```
    class Date implements Listable {
      private Image _snapshot;
      private String _whatHappened;

      public Image getImage () {
        return _snapshot;
      }
      public String getDescription () {
        return _whatHappened;
      }
    }
    ```

- **Implementing** an interface: create a body for every method in the interface.

Implementing an interface

- If any method body is missing, then it won't compile:

  -
    ```
    class Date implements Listable {
      private Image _snapshot;
      private String _whatHappened;


      // No implementation of getImage()

      public String getDescription () {
        return _whatHappened;
      }
    }
    ```

  -
    ```
    Date.java:1: error: Date is not abstract and
    does not override abstract method getImage() in
    Listable
    ```

Modeling problem

- Using the `Listable` interface, we can enforce that every `item` supports `getImage()` and `getDescription()` methods, without requiring a specific parent class.

-
  ```
  class ListBox {
    public void addItem (Listable item) {
      ...
    }
  }
  ```

Interfaces as types

Types in Java

- In Java, every declared variable has a **type**, e.g.:

```
String str;  // str is a String
Image image;  // image is an Image.
Object obj;  // obj is an Object.
int someNum;  // someNum is an int
```

- The type of the object specifies which methods can be called on it, e.g.:

```
str.length();  // ok
obj.length();  // won't compile;
```

Interfaces as types

- Once you have defined an interface and implemented it in one or more classes, you can:
  - Declare a variable of the interface type

    -
      ```
      final Listable item1 = new ClassicDate();
      final Listable item2 = new Pet();

      ...

      item1.getImage();  // ok
      item2.getDescription();  // ok
      ```

Interfaces as types

- Once you have defined an interface and implemented it in one or more classes, you can:
  - Declare a variable of the interface type
  - Declare a parameter of the interface type

    -
      ```
      void addItem (Listable item) {
        drawImage(item.getImage());
        writeDescription(item.getDescription());
      }
      ```

Interfaces as types

- Once you have defined an interface and implemented it in one or more classes, you can:
  - Declare a variable of the interface type
  - Declare a parameter of the interface type
  - Return a variable of the interface type

    -
      ```
      Listable getListItem () {
        final Listable date = new ClassicDate();
        return date;
      }
      ```

Interfaces as types

- Once you have defined an interface and implemented it in one or more classes, you can:
  - Declare a variable of the interface type
  - Declare a parameter of the interface type
  - Return a variable of the interface type
  - Declare an array of variables of the interface type

    -
      ```
      final Listable[] dateHistory =
        new Listable[getNumDates()];
      ```

Interfaces as types

- A class can implement any number of interfaces, e.g.:

  ```
  class Pet implements Listable, Serializable {
    ...
  }
  ```

- In contrast, a class can have at most one parent class.

- In this sense, interfaces are more flexible.

Interfaces as types

- Interfaces **cannot** be instantiated:
  - `final Listable item = new Listable();  // wrong`

Interfaces as types

- Interfaces **cannot** be instantiated:
    - `final Listable item = new Listable();  // wrong`
- Why not?
    - What code should be executed in the following?
        - `item.getImage();`

Interfaces as types

- Interfaces **cannot** be instantiated:
  - `final Listable item = new Listable();   // wrong`
- Why not?
  - What code should be executed in the following?
    - `item.getImage();`

The `getImage()` method is **abstract** in the interface — no implementation!

Exercise

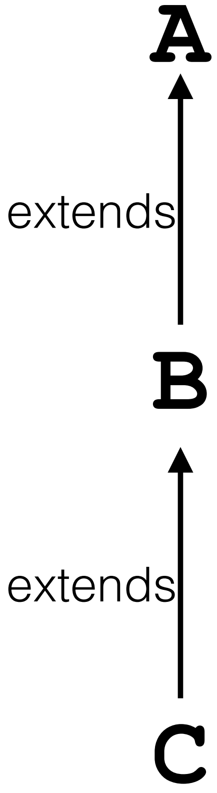
- Suppose we have an interface `Identifiable`:

  -
    ```
    interface Identifiable {
      String getName();
      String getAddress();
      long getSSN();
    }
    ```
- Create a class `Person` that implements `Identifiable`.

Subinterfaces

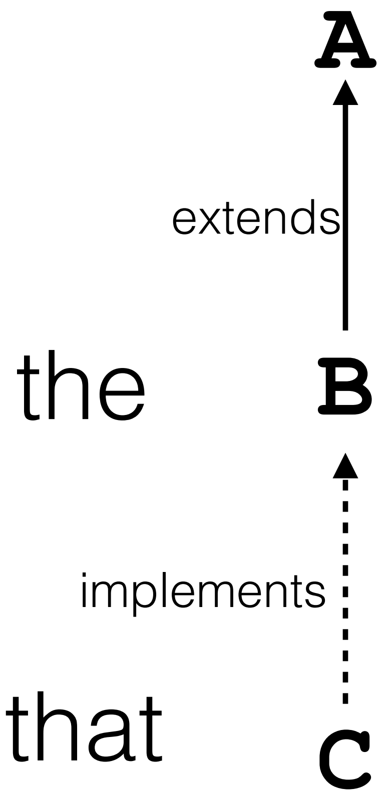
Subclasses (review)

- Suppose there is a class **A**, and another class **B** that inherits from **A**.
- Now, suppose a third class, **C**, inherits from **B**.
- Then **C** inherits the union of the methods in both **A** and **B**.



Subinterfaces

- Similarly to how classes can have subclasses, interfaces can have subinterfaces.
- The subinterface inherits all the methods from the parent interface.
- A class **`C`** that implements the subinterface **`B`** (that extends interface **`A`**) must implement the *union* of the methods in **`A`** and **`B`**.



Example

-
  ```
  interface A {
    void method1 (int num);
    String method2 ();
  }

  interface B extends A {
    void method3 (String word);
  }

  class C implements B {
    public void method1 (int num) {
    }
    public String method2 () {
      return ...
    }
    public void method3 (String word) {
    }
  }
  ```

Exercise

```
class Location {
        double _longitude, _latitude;
        public Location (double longitude, double latitude) {
                _latitude = latitude;
                _longitude = longitude;
        }
}

interface Locatable {
        Location getCurrentLocation ();
}

interface HealthMonitor {
        int getBloodPressure (int bodyZone);
        int getColdestBodyZone ();
}

class AppleWatch implements Locatable, HealthMonitor {
        // TODO: implement methods so this class compiles
        // Requirement: no method may return null
}
```

Interfaces as contracts

Interfaces as contracts

- So far, we have discussed how a Java interface defines a **type** of object.

- One of the main purposes of interfaces is to **guarantee that certain methods exist**, e.g.:

  -
    ```
    interface Listable {
      Image getImage ();
      String getDescription();
    }
    ```

  -
    ```
    interface SmileDetector {
      public boolean isImageSmiling (Image face);
    }
    ```

Interfaces as contracts

- Interfaces also facilitate **division of labor** between members of a team.

- Interfaces separate **what** a class does:

```
interface SmileDetector {
  /**
   * Returns whether or not the specified face is smiling.
   * @param face the face (48x48 pixels) to analyze.
   * @return whether the face is smiling.
   */
  public boolean isImageSmiling (Image face);
}
```

Just the signature

Interfaces as contracts

- Interfaces also facilitate **division of labor** between members of a team.

- …from **how** it does it:

```
class NeuralNetworkSmileDetector implements SmileDetector {
  private float[][] _weights;

  /**
   * Returns whether or not the specified face is smiling.
   * @param face the face (48x48 pixels) to analyze.
   * @return whether the face is smiling.
   */
  public boolean isImageSmiling (Image face) {
    ...
  }
}
```

The actual implementation

Interfaces as contracts

- This leads to a natural **division of labor**:
  - The **user** of an interface does not have to care how it is implemented.
  - The **implementer** does not have to care how it is used.

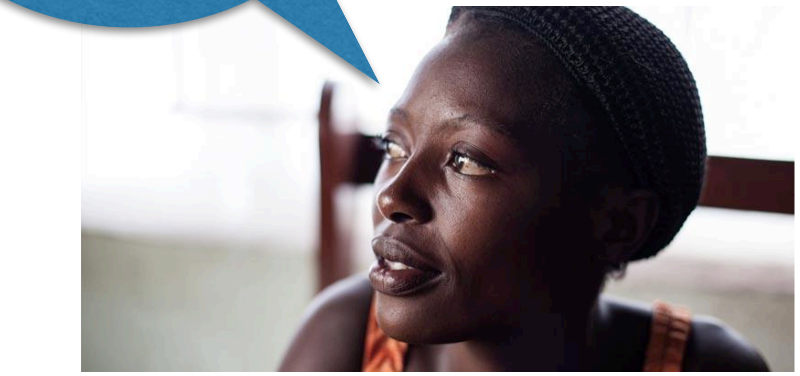
Interfaces as contracts

```
public class MyGame {
  void someMethod () {
    SmileDetector detector = ...
    ...
    if (detector.isImageSmiling(im)){
      ...
    }
  }
}
```

Ok I'll **use** the detector in my game.

```
public class SomeSmileDetector
  implements SmileDetector {
  ...
  boolean isImageSmiling (Image face) {
    float minDistance = ...
  }
}
```

I'll **implement** the smile detector.



Interface

Interfaces as contracts

- The interface serves as a software contract between user and implementer.
- It acts as a “wall”:
    - Whatever changes behind the “wall” **doesn’t affect the other programmer**.

Interfaces as contracts

- The interface specifies mutual requirements between implementor and user.

```
interface SmileDetector {
  /**
   * Returns whether or not the specified face is smiling.
   * @param face the face (48x48 pixels) to analyze.
   * @return whether the face is smiling.
   */
  public boolean isImageSmiling (Image face);
}
```

- In this case, the user is required to pass in a face of size 48x48. The implementor is required to produce an estimate (true/false) of whether the face is smiling.

Key-points: classes, interfaces, & OO design

1. Classes bundle together a **coherent** set of **actions** (methods) and **attributes** (instance variables).

2. Common actions and attributes can be **factored out** of multiple classes using **inheritance** — this can yield a class **hierarchy** of both **abstract** (non-instantiable) and **concrete** classes.

Key-points: classes, interfaces, & OO design

3. Interfaces allow the programmer to specify a **set of methods** that every implementing class is required to offer.

4. Interfaces also serve as a **software contract** that naturally supports a **division of labor** among programmers.

5. In Java, a class can inherit from **at most one parent class**, but can implement **any number** of interfaces. Hence, before using inheritance, ask yourself whether an interface would do the job just as well.