Interfaces and Abstract Classes in Java

What we covered last class

- Introduction to **classes**
  - Represent specific **objects**
  - Methods describe **interactions**
  - Inheritance allows for **subclassing**
- Discussion of access modifiers
  - public - open to **all**
  - private - open to **object**
  - protected - open to **subclasses**

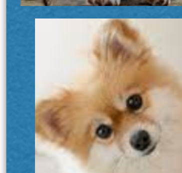
Classes from Last Class

- ```
  class Being {
    private String _name;
    private String _address;
  }
  ```
- ```
  class Pet extends Being {
    private Being _owner;
    private Image _profilePicture;
    private Date[] _history;
  }
  ```
- ```
  class Date {
    private Time _time;
    private Image _snapshot;
    private String _description;
  }
  ```
  ```

Running Examples from Last Class

- PetDate.net a website for tracking time you spent with your, or a friend's pet

History of past dates

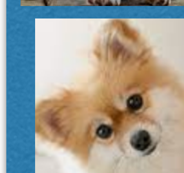
Who	When	Where
	10/21/2021	Skiing @ A-Basin
	10/20/2021	Climbing @ Golden Cliffs
	10/20/2021	Movie in Denver

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 @ A-Basin
	10/20/2021	Climbing @ Golden Cliffs
	10/20/2021	Movie in Denver

List of new members

- It also allows users to search through new PetDate.net pets who recently been added:

Newly joined members

Pic	Age	Name
	21	Leonardo
	29	Matt
	27	Humphrey

Modeling problem

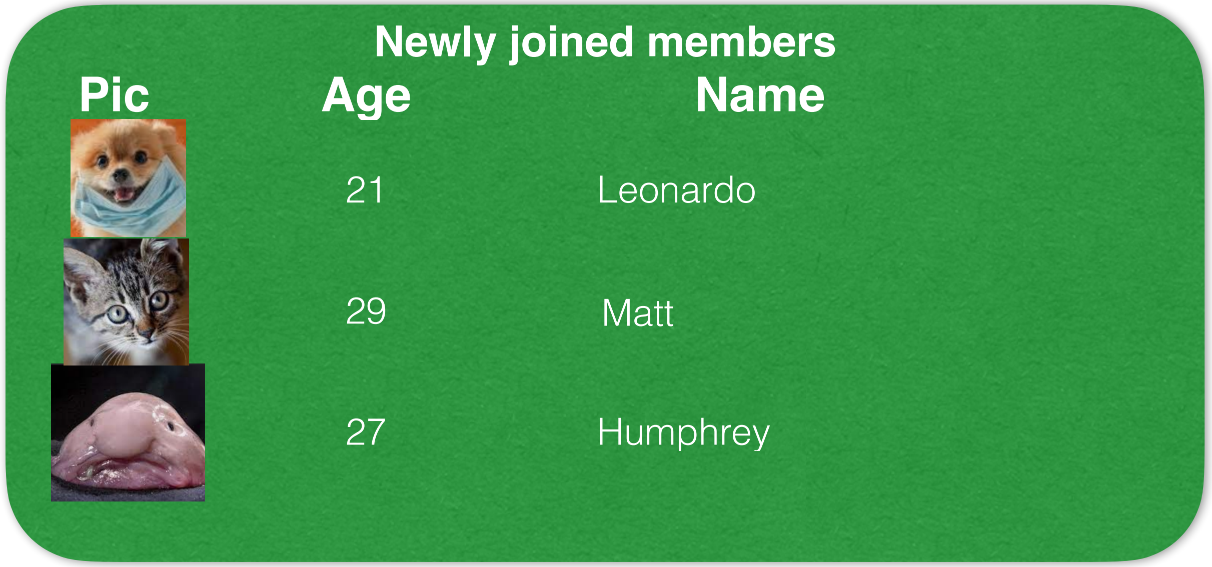
- We want to be able to call:
  - ```
    final ListBox dateListBox = new ListBox();
    dateListBox.addItem(dateWithMike);
    dateListBox.addItem(dateWithFrank);
    dateListBox.addItem(dateWithHairy);
    ```


    ```

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);
    ```


    ```

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) {
      drawImage(item.getImage());
      drawText(item.getDescription());
    }
  }
  ```

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

Modeling problem

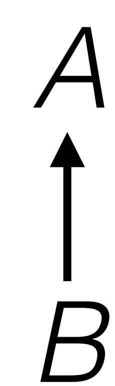
- Strategy 1 — create a common ancestor class:
  - ```
    class ListableObject {
      public Image getImage () { ... }
      public String getDescription () { ... }
    }
    ```
  - ```
    class Pet extends ListableObject {
       …
    }
    ```
  - ```
    class Date extends ListableObject {
      …
    }
    ```
    ```

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 (Pet)* that already has a parent class *A (Being)*.


  ```

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 Being 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 Date();
      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`.
    ```

Solution

```
class Person implements Identifiable {

  public String getName() {
    return "Bo";
  }

  public String getAddress() {
    return "1 Main St";
  }

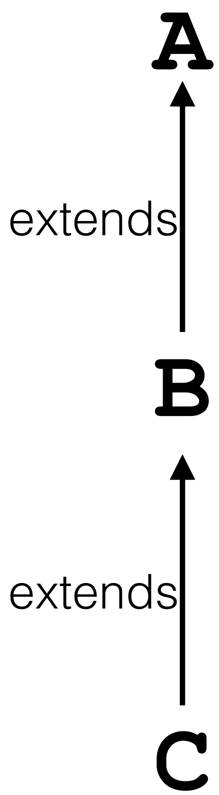
  public long getSSN() {
    return 1234;
  }

}
```

Subinterfaces

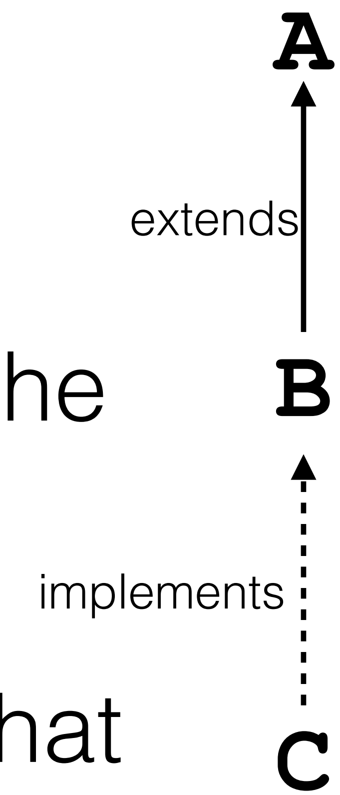
Subclasses (review)

- 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

- Suppose we have an interface `Listable`:
  - ```
    interface Listable {
      public Image getImage ();
      public String getDescription ();
    }
    ```
- Create a subinterface `NamedListable`.
  - Returns a name for the object
    ```

Solution

- ```
  interface NamedListable extends Listable
  {
    public String getName();
  }
  ```

- ```
  class NamedPet extends Pet implements
  NamedListable {
     public String getName() {
        return “Spot”;
     }
  }
  ```

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.

Abstract classes

Abstract classes

- In addition to classes and interfaces, Java also offers a programming construct with some features of both: abstract classes.

- This is useful for refactoring classes that have some variables and/or methods in common.

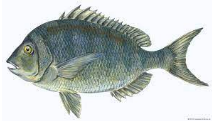
- Abstract classes are allowed to contain:
  - Variables
  - Concrete methods (signature & implementation)
  - Abstract methods (signature only)

Abstract classes

- Abstract methods must be declared with `abstract`.
- Abstract classes must be declared with `abstract`.
- Abstract classes cannot be instantiated.
- To be useful, an abstract class must be extended by a concrete subclass.
- The subclass must implement all of the abstract methods.

Refactoring with abstract classes: example

```
class Fish {
  private ArrayList<String> _messages;
  void sendMessage (String message) {
    _message.add(message);
  }
  public void draw (Graphics g) {
    // Draw:
```



```
  }
}
```

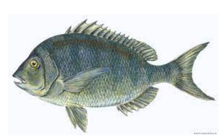
```
class Pterodactyl {
  private ArrayList<String> _messages;
  void sendMessage (String message) {
    _message.add(message);
  }
  public void draw (Graphics g) {
    // Draw:
```

```
  }
}
```

Refactoring with abstract classes: example

```
abstract class Animal {
  private ArrayList<String> _messages;
  void sendMessage (String message) {
    _message.add(message);
  }
  public abstract void draw (Graphics g);
}
```

```
class Fish extends Animal {
  @Override // asks the compiler to check that parent method exists
  public void draw (Graphics g) {
    // Draw:
```



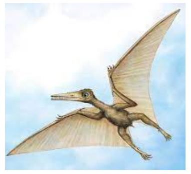
```
  }
}
```

```
class Pterodactyl extends Animal {
  @Override // asks the compiler to check that parent method exists
  public void draw (Graphics g) {
    // Draw:
```

```
  }
}
```

Refactoring with abstract classes: example

- ```
  abstract class Animal {
    private ArrayList<String> _messages;
    void sendMessage (String message) {
      _message.add(message);
    }
    public abstract void draw (Graphics g);
  }
  ```
- `class Fish extends Animal { ... }`
- `class Pterodactyl extends Animal { ... }`
- ```
  Animal animal = new Animal();  // error: cannot
  instantiate abstract class
  Animal animal = new Pterodactyl();  // ok
  animal.sendMessage("Yo!");
  animal.draw(g);
  ```



Refactoring with abstract classes: example

- Why not just define an empty `draw()` method in `Animal` and make the class non-`abstract`?

- If `Animal` were a regular class, it could be instantiated, but nothing would be drawn.

- With `abstract`, we can let the compiler prevent a(nother) programmer from erroneously creating an `Animal` object.

Exercise

```
abstract class A {
    abstract void q ();
}

interface X {
    void m (Object o);
    X n ();
}

interface Y {
    int p (int j);
    A me ();
}

class C implements X, Y {
    // Your code here!
    // No returning null!
}
```

Solution

```
abstract class A {
    abstract void q ();
}

interface X {
    void m (Object o);
    X n ();
}

interface Y {
    int p (int j);
    A me ();
}

class C implements X, Y {
    // Your code here!
    // No returning null!
}
```

```
class C implements X, Y {

  void m (Object o) { return; }

  X n () { return this;}

  int p (int j) { return j+1; }

  A me () {
    return new A_concrete();
  }

  class A_concrete extends A {
    void q() { return; }
  }

}
```

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.

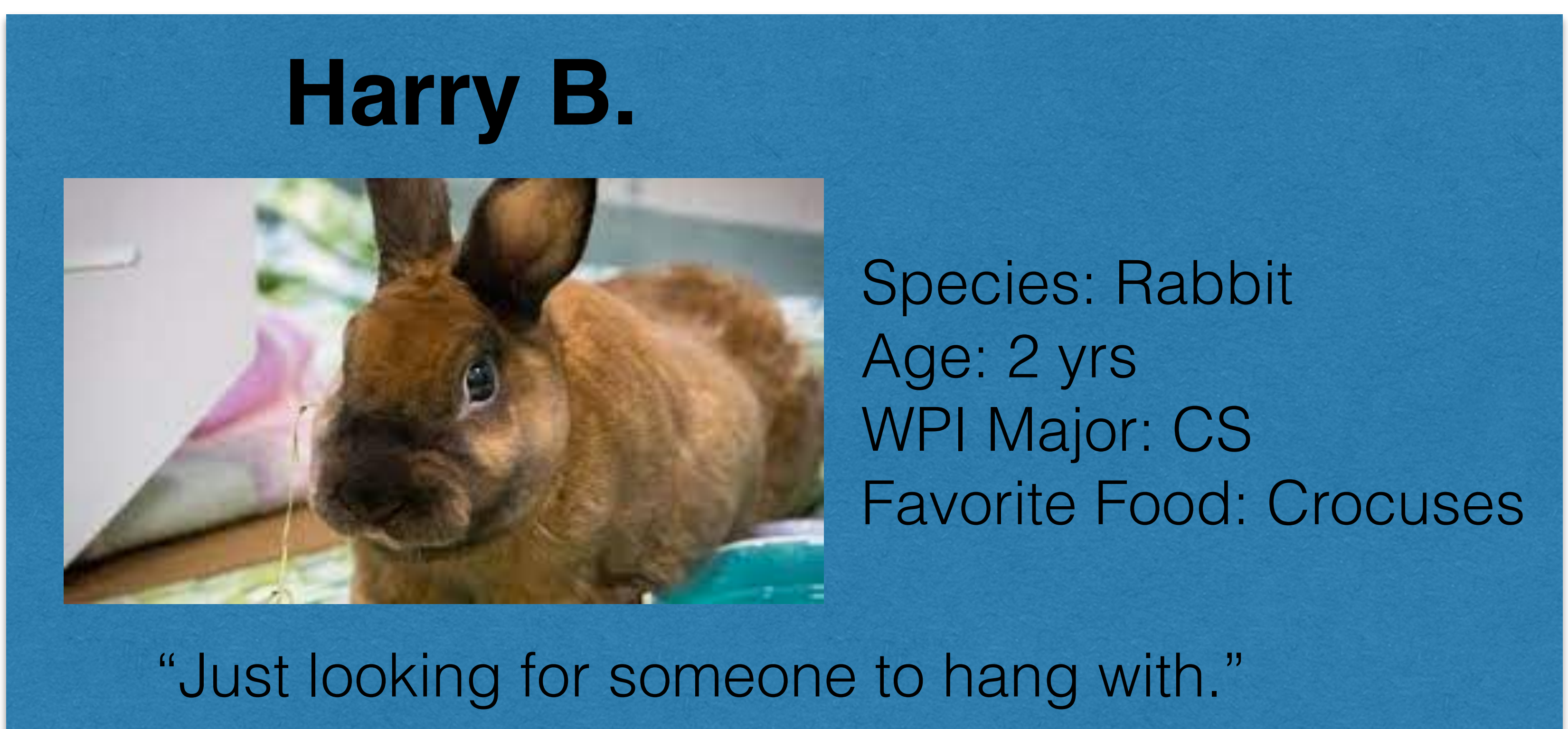
5. In Java, a class can inherit from **at most one parent class**, but can implement **any number** of interfaces. Hence, before using inheritance, ask yourself whether an interface would do the job just as well.

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`).



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.

Type-safety and casting

- ```
  interface Date {
    public int getNumParticipants ();
    public String getActivity ();
    public String getTime ();
    ...
  }
  ```
  ```

Type-safety and casting

- ```
  abstract class AbstractDateImpl implements Date {
    private String _activity, _time;
    public String getActivity () {
      return _activity;
    }
    public String getTime () {
      return _time;
    }
    public abstract int getNumParticipants ();
  }
  ```
  ```

Type-safety and casting

- ```
  class SimpleDate extends AbstractDateImpl {
    private Couple _couple;
    public int getNumParticipants () {
      return 2;
    }
  }
  ```

- ```
  class Couple {
    private Pet _pet1, _pet2;
    public Couple (Pet pet1, Pet pet2) {
      _pet1 = pet1;
      _pet2 = pet2;
    }
  }
  ```

Type-safety and casting

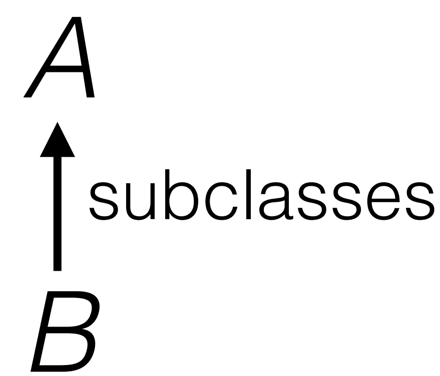
- ```
  class DoubleDate extends AbstractDateImpl {
    private Couple _couple1, _couple2;
    public int getNumParticipants () {
      return 4;
    }
    public Couple[] getCouples () {
      return new Couple[] { _couple1, _couple2 };
    }
  }
  ```
  ```

Casting

- Objects are **cast** into different classes/interfaces when we assign them to variables declared of different types:

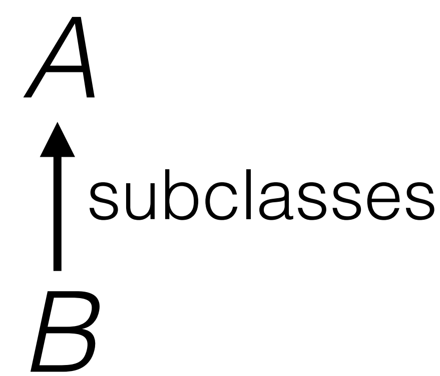
```
class A { ...
}
class B extends A { ...
}

final B b = new B();
```



Casting

- Objects are **cast** into different classes/interfaces when we assign them to variables declared of different types:




```
class A { ...
}
class B extends A { ...
}

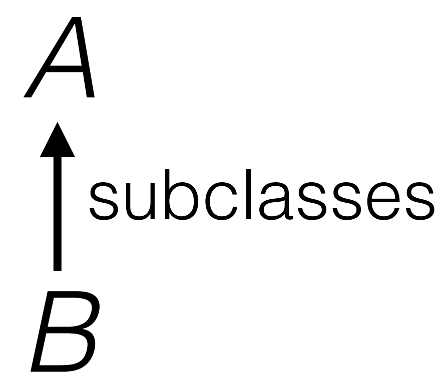
final B b = new B();
final A a = b;       // Upcast from B to A.
```

Casting

- Objects are **cast** into different classes/interfaces when we assign them to variables declared of different types:

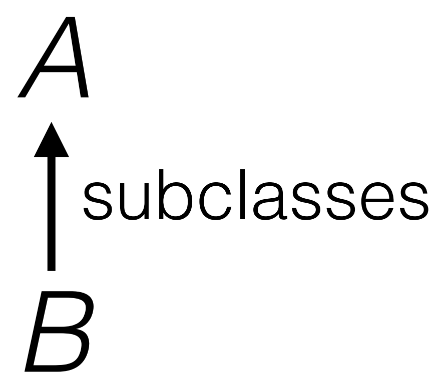
```
class A { ...
}
class B extends A { ...
}

final B b = new B();
final A a = b;          // Upcast from B to A.
final B b2 = (B) a;   // Downcast from A to B.
```



Casting

- Objects are **cast** into different classes/interfaces when we assign them to variables declared of different types:




```
class A { ...
}
class B extends A { ...
}

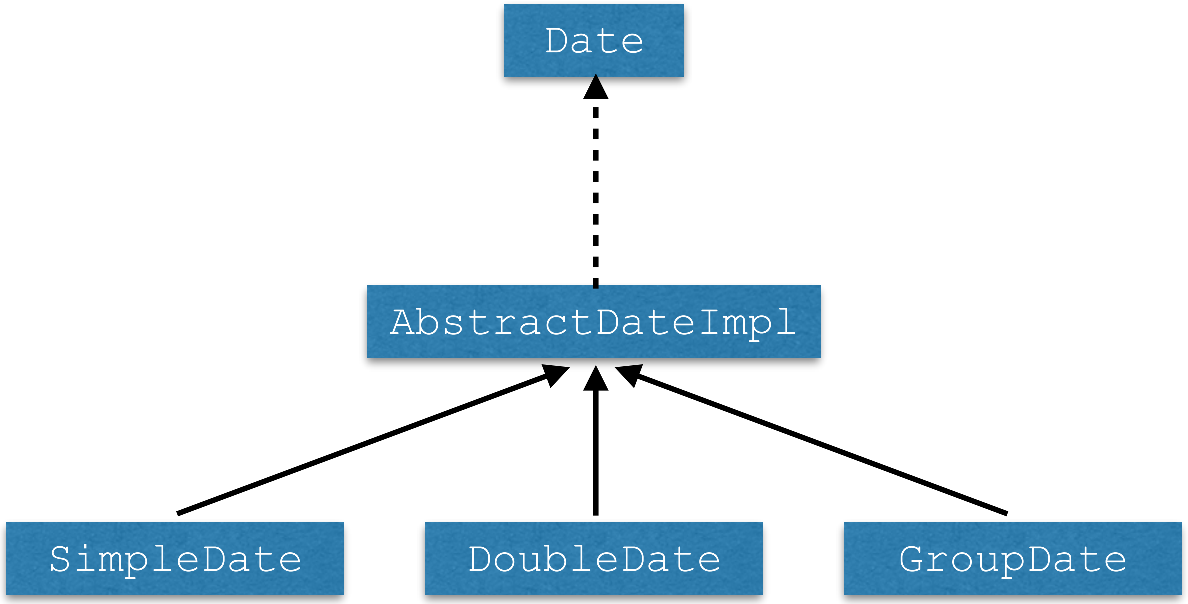
final B b = new B();
final A a = b;         // Upcast from B to A.
final B b2 = (B) a;   // Downcast from A to B.
```

The terms **upcast** and **downcast** have to do with the class hierarchy, in which parent classes are "above" child classes.

Run-time type-checking

- The JVM enforces type-safety at **run-time**:
  - Every object knows what kind of class it is, what its parent class is, and all the interfaces that it implements.
  - If you attempt to downcast an object into a type with which it is not compatible, then your program will throw a `ClassCastException`.
    - Your program will terminate.

Run-time type-checking



- Example:

```
final Date gd = new GroupDate();
final SimpleDate sd = (SimpleDate) gd;   // run-time error
```

- This will result in a `ClassCastException` because a `GroupDate` is never also a `SimpleDate`.

Run-time type-checking

- We can also cast to an interface type, e.g.:

```
final Object o = new SimpleDate();
final Date date = (Date) o;
```

- Since not every object of `Object` class is guaranteed to implement the `Date` interface, we must “downcast” to `Date`.

- At run-time, the JVM will check whether `o` is of some class that implements `Date`, and throw a `ClassCastException` if it is not.

Dynamic dispatch

- Suppose we have the following class hierarchy:
    - ```
      class A {
        public void m () {
          System.out.println("A");
        }
      }
      class B extends A {
        public void m () {
          System.out.println("B");
        }
      }
      final A a = new B();
      a.m();
      ```
- Which version of `m` will be called?
      ```

Dynamic dispatch

- In Java, every object “remembers” the class from which it was instantiated.

- Since the same object can have multiple types, the method that is invoked depends on the class from which the object was *instantiated.*

  - This is determined by the JVM at run-time and is known as **dynamic dispatch**.

Dynamic dispatch: exercise

```java
public class A {
        void m () {
                System.out.println("A");
        }
}
public class B extends A {
        void m () {
                System.out.println("B");
        }
}
```

```java
public class C {
        A someOtherMethod () {
                return new B();
        }
        void someMethod () {
                final A a = new B();
                a.m();

                final A a2 = (A) someOtherMethod();
                a2.m();

                B b = (B) (new A());
                b = new B();
                b.m();

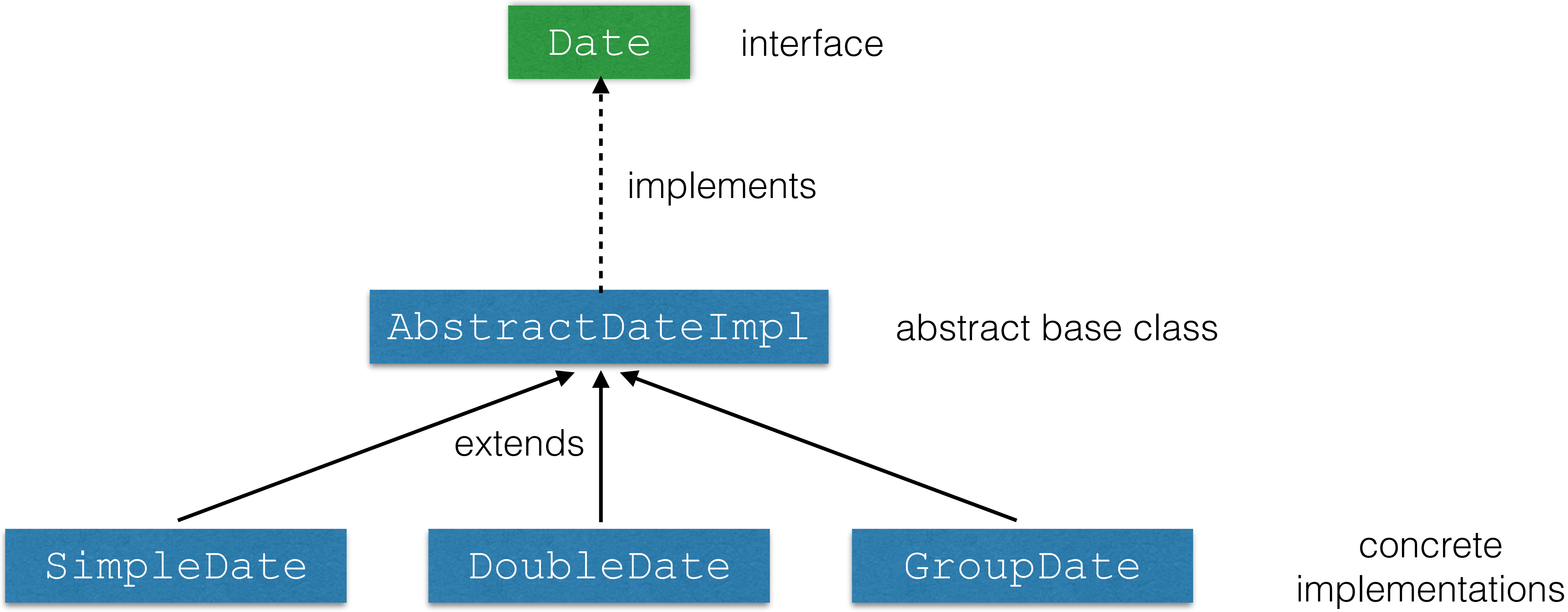
                final B b2 = new B();
                ((A) b2).m();
        }
}
```

Type-safety and casting

- ```
  class GroupDate extends AbstractDateImpl {
    private Couple[] _couples;
    private Pet[] _singles;
    public int getNumParticipants () {
      return 2*_couples.length + _singles.length;
    }
    public Couple[] getCouples () {
      return _couples;
    }
  }
  ```
  ```

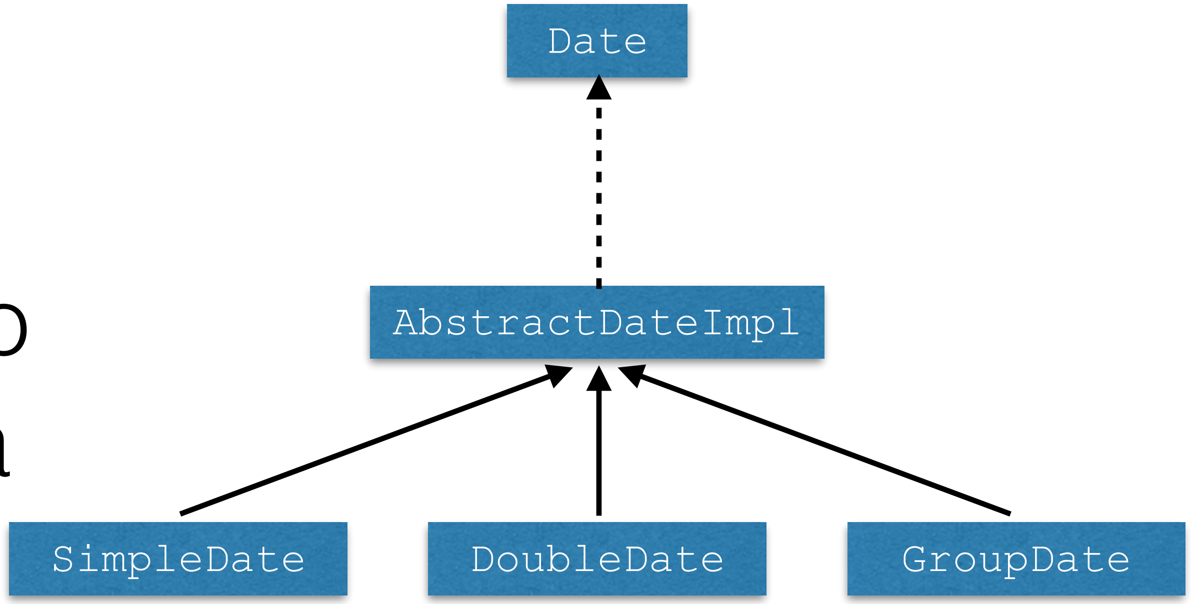
Example

- We can visualize these in one class/interface diagram:



Objects can have multiple types

- When using interfaces and class hierarchies to model data relationships, the **same object** can have **multiple types**.



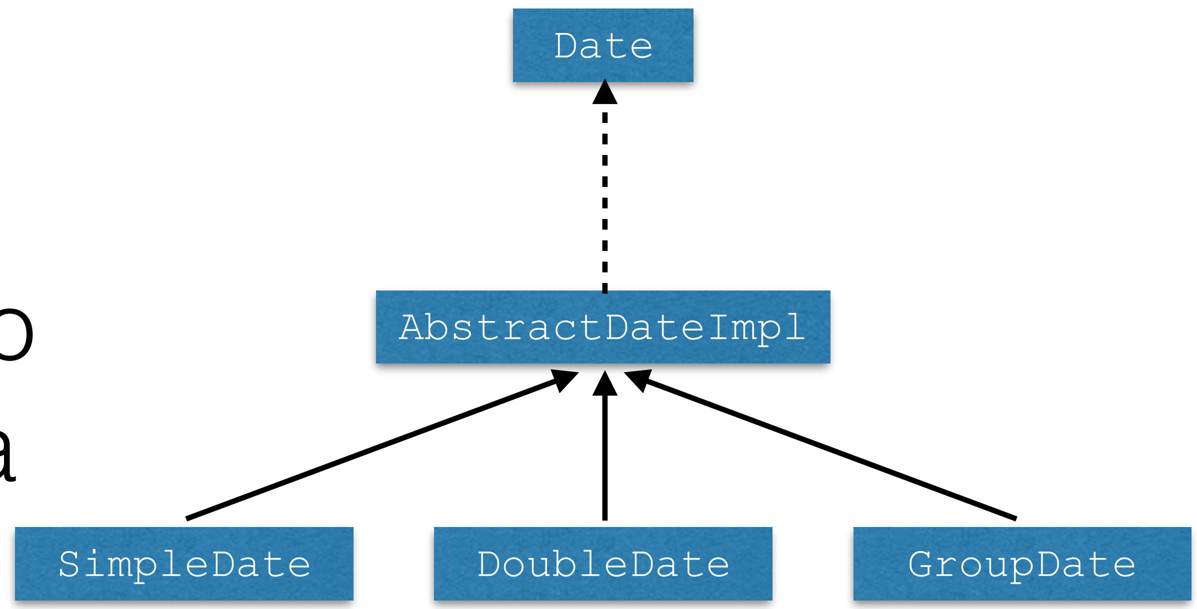
- For example, every `DoubleDate` object can also be referred to as a `Date` or a `AbstractDateImpl`:

```
final DoubleDate ddate =
  new DoubleDate();
final Date d = ddate;
final AbstractDateImpl a = ddate;
```

Objects can have multiple types

- When using interfaces and class hierarchies to model data relationships, the **same object** can have **multiple types**.

- For example, every `DoubleDate` object can also be referred to as a `Date` or a `AbstractDateImpl`:




```
final DoubleDate ddate =
  new DoubleDate();
final Date d = ddate;
final AbstractDateImpl a = ddate;
```

All three of the declared variables — `ddate`, `d`, and `a` — all point to the same object (which is a `DoubleDate`).

Type-safety in Java

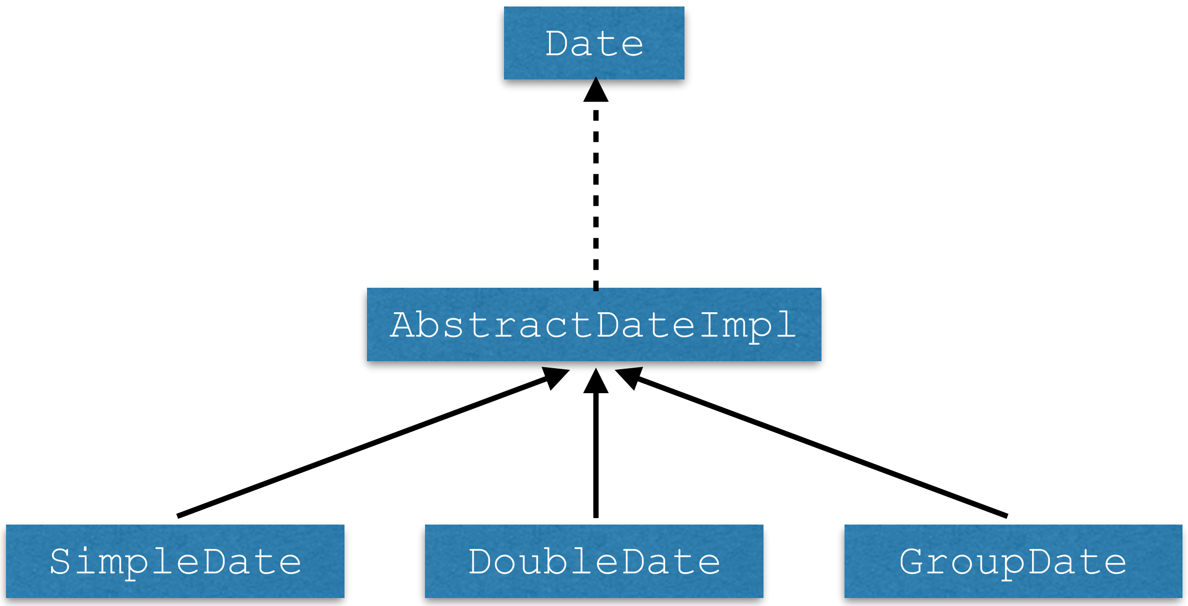
- Since Java is a type-safe language, it enforces that methods can only be called on objects that implement them.
  - E.g., cannot allow `getCouples()` to be called on a `String` object.
- Two kinds of type-checking:
  - Compile-time (Java compiler).
  - Run-time (Java Virtual Machine).

Objects can have multiple types

- The **declared type** of the variable dictates which methods can be called on the object:

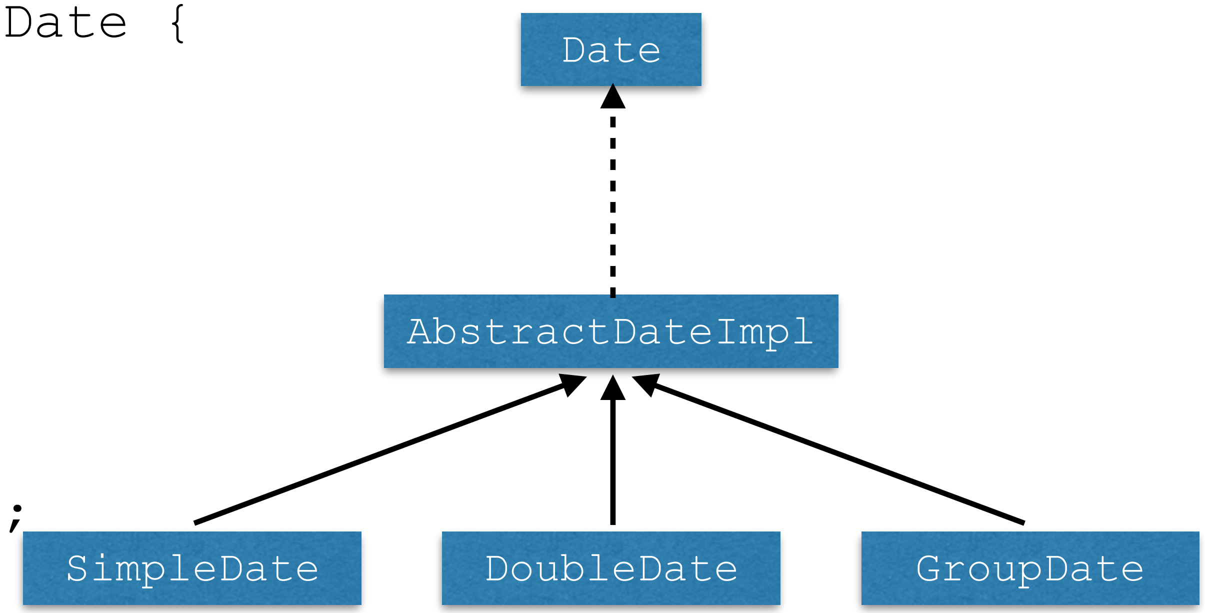
```
final DoubleDate ddate =
  new DoubleDate();
final Date d = ddate;
final AbstractDateImpl a = ddate;
```

Declared types



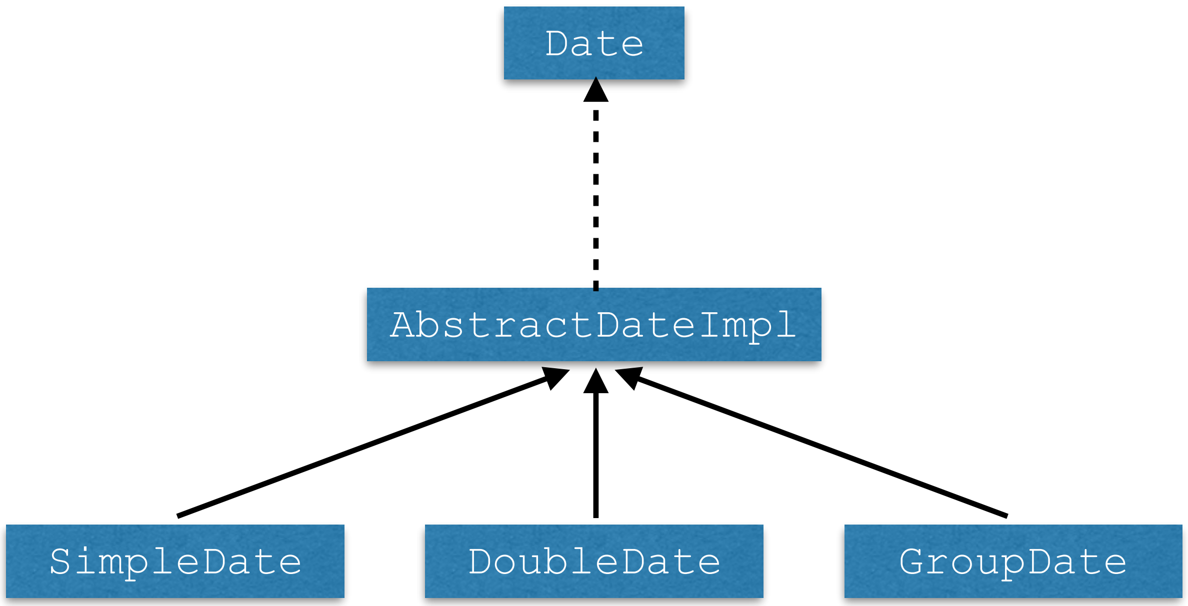
Objects can have multiple types

- ```
  interface Date {
    public int getNumParticipants ();
    public String getActivity ();
    public String getTime ();
    ...
  }
  ```
- ```
  abstract class AbstractDateImpl implements Date {
    private String _activity, _time;
    public String getActivity () {
      return _activity;
    }
    public String getTime () {
      return _time;
    }
    public abstract int getNumParticipants ();
  }
  ```
- ```
  class DoubleDate extends AbstractDateImpl {
    private Couple _couple1, _couple2;
    public int getNumParticipants () {
      return 4;
    }
    public Couple[] getCouples () {
      return new Couple[] { _couple1, _couple2 };
    }
  }
  ```


  ```

Objects can have multiple types

- Method calls will compile only if they are compatible with the object's declared type.



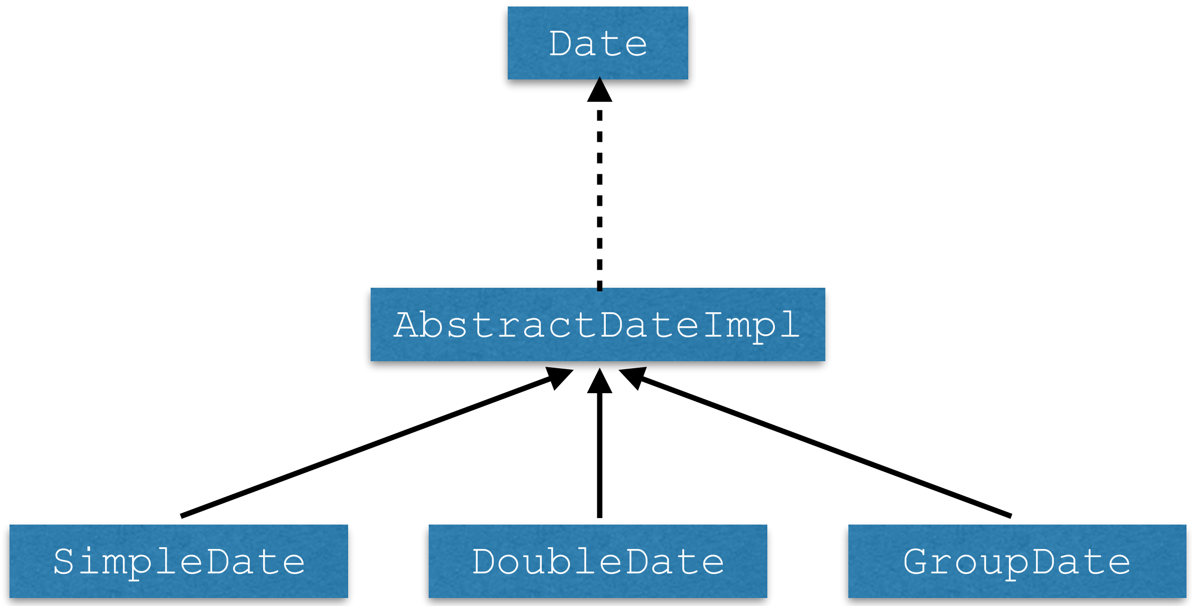

```
final DoubleDate ddate =
    new DoubleDate();
final Date d = ddate;
final AbstractDateImpl a = ddate;
```

```
1    d.getNumParticipants();
2    a.getNumParticipants();
3    d.getCouples();
4    a.getCouples();
5    ddate.getCouples();
```

Which of these lines of code will compile?

Objects can have multiple types

- Method calls will compile only if they are compatible with the object's declared type.




```
final DoubleDate ddate =
  new DoubleDate();
final Date d = ddate;
final AbstractDateImpl a = ddate;

d.getNumParticipants(); // ok
a.getNumParticipants(); // ok
d.getCouples();   // not ok
a.getCouples();   // not ok
ddate.getCouples();  // ok
```

Compile-time type-checking

- Java verifies at **compile-time** that the type of object assigned to a variable is consistent with the variable's declared type:

```
// not ok — an Integer is never a string
final String s = new Integer(1234);
```

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

- 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`.