6.092 - Introduction to Software Engineering in Java

*Lecture 7:*

# Exploring the Java API, Packages & Collections

*Tuesday, January 29*
*IAP 2008*

# Administrivia

HKN Course Evaluations: Now with IAP!

- Help us improve, help students choose
- Survey website active at midnight tonight
- Only active for a few days, so do it soon
- Link to be posted on course website:
  - http://mit.edu/iapjava/

# Course Refresher

What you've learned so far:

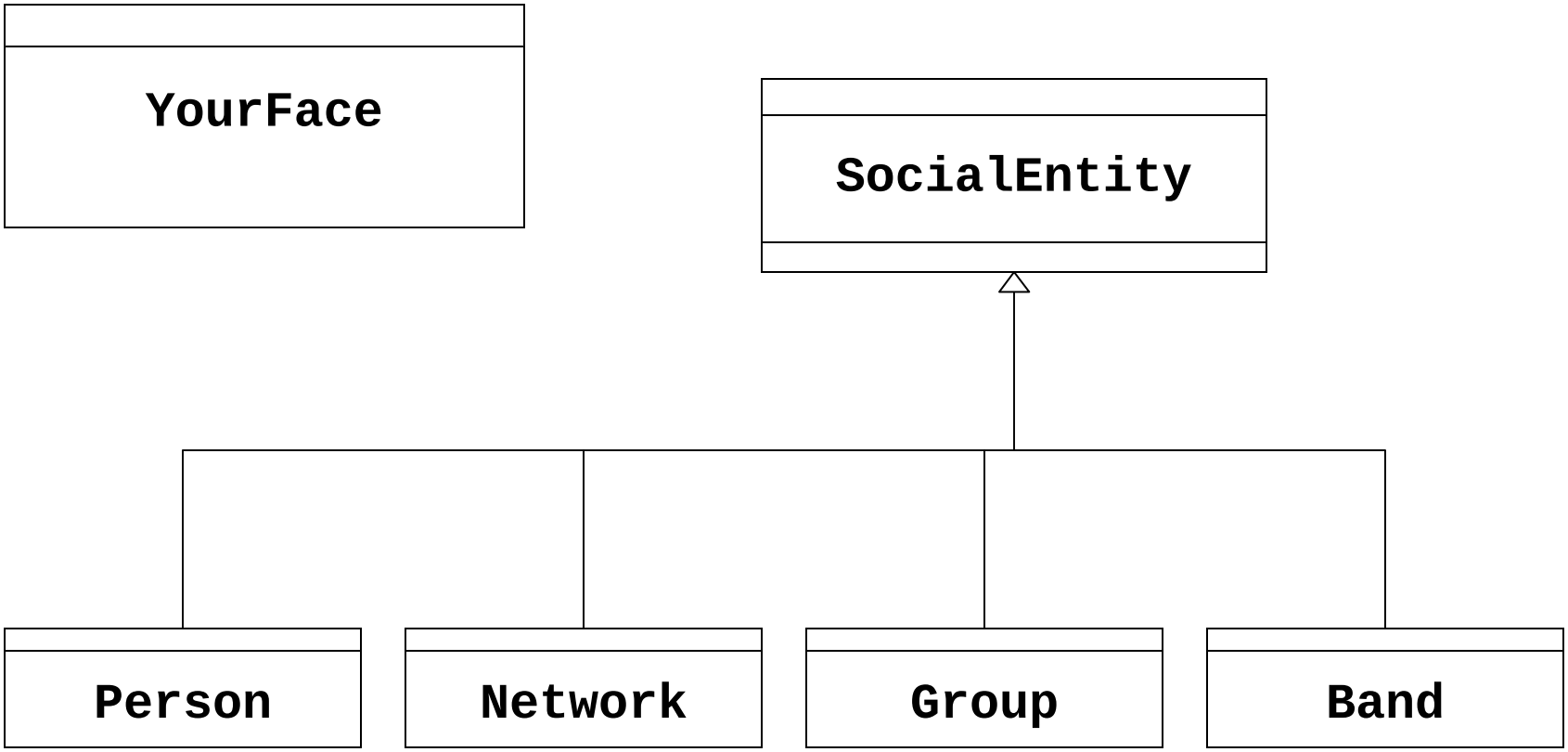
- Intro/Overview
  - compilation, execution
- Java Basics:
  - Structure & Syntax, Variables, Types, & Operators
- Control Flow:
  - Methods & Conditionals, Loops & Arrays
- Object-oriented Programming (OOP):
  - Objects & Classes
  - Inheritance & Abstraction:
    - Classes, Abstract Classes & Interfaces
  - Encapsulation
- Brief Intro to Software Design

# Review: Assignment 6

Modeling YourFace: A Simple Social Network

- Everything is a Social Entity
  - Must implement the `SocialEntity` interface
- Basic properties given for each subtype
- Loose requirements can lead to subtle implementation issues

# Assignment 6: Diagram



## Ideal Solution: `SocialEntity`

```
public interface SocialEntity {

   public String getName();
   public long getId();

}
```

# Ideal Solution: Person

```
public class Person implements SocialEntity {

  protected String name;
  protected long id;
  protected Person[] acquaintances;
  protected Network[] networks;
  protected String location;

  public Person(String name, long id, String location) {
    this.name = name;
    this.id = id;
    this.location = location;
    this.acquaintances = new Person[]{};
    this.networks = new Network[]{};
  }

  // ... getters & setters ...

}
```

# Ideal Solution: Person

```
public class Person implements SocialEntity {
  // ... fields & constructor ...
  // ... other getters & setters ...
  public void setAcquaintances(Person[] acquaintances) {
    this.acquaintances = acquaintances;
  }
  public void setNetworks(Network[] networks) {
    this.networks = networks;
  }
  public String toString() {
    String acqsToString = "\n Acquaintances: ";
    for (Person p : acquaintances) {
      acqsToString += "\n " + p.getName();
    }
    String netsToString = "\n Networks: ";
    for (Network n : networks) {
      netsToString += "\n " + n.getName();
    }
    return "Person #" + getId() + ": " + getName()+
           "\n Location: " + getLocation()+
           acqsToString + netsToString;
  }
}
```

# Ideal Solution: YourFace

```
public class YourFace {
  public static void main(String[] args) {
    // Example: 3 Persons
    Person usman = new Person("Usman Akeju", 0, "Mount Vernon, NY");
    Person evan = new Person("Evan Jones", 1, "Stata");
    Person olivier = new Person("Olivier Koch", 2, "Earth");
    usman.setAcquaintances(new Person[]{evan, olivier});
    evan.setAcquaintances(new Person[]{usman, olivier});
    olivier.setAcquaintances(new Person[]{evan, usman});

    // Example: 3 Networks
    Network mit = new Network("MIT", 3);
    Network canada = new Network("Canada", 4);
    Network france = new Network("France", 5);
    usman.setNetworks(new Network[]{mit});
    evan.setNetworks(new Network[]{mit, canada});
    olivier.setNetworks(new Network[]{mit, france});

    // ... you can do Groups & Bands here ...
    printArray(new Object[]{usman, evan, olivier});
  }
}
```

# Ideal Solution: YourFace

```
public class YourFace {

  // ... see previous slide ...

  public static void printArray(Object[] array) {
    for (Object o : array) {
      System.out.println(o);
      System.out.println();
    }
  }
}
```

# Assignment 6: Recap

- Reminders:
  - Interface fields are treated as ***`final`***
    - generally useless to subclasses unless also ***`static`***
  - Adding accessors & mutators (getters/setters) can be useful
  - Overriding `toString()` can be very useful!
- Caveats:
  - Be mindful of how & when to initialize or assign fields
  - The constructor does not need to take a value for every field
  - Setter methods can help you later!
- Foreshadowing: Where do abstract classes fit?

# Today's Topics

- Packages
- The Java API
- Collections

# Packages

- A way to organize related classes
- Similar to folders in a file system
- Some of you are already using them
  - `package pset6;`

# Packages

```
package com.yourface; //package declaration

public class YourFace {
   // ...
}
```

# Packages

- Package declaration must come before all other code (excluding comments)
- No declaration means “default” package
- Directory structure must mirror package structure
  - E.g., `FooBar` class in a package called `foo.bar` must be in `foo/bar/` directory
- Root package directory (e.g., `foo`) must be in a directory on the CLASSPATH to execute code
  - E.g., run `FooBar`’s main method from command line:
  - `java foo.bar.FooBar`
  - Assumes the current directory contains the `foo` directory

# Packages

- Remember: Visibility
  - Can limit visibility of classes, constructors, fields, or methods to a single package
  - Must omit visibility keywords (e.g., **`public`** and **`protected`**) for “default”/“package” visibility

# Packages: Importing

- To access the many useful packages & classes included with Java, one must ***import*** them
  - Use the **`import`** keyword
    - After package declaration, before class declaration
  - Can import a single class
    - `import java.io.BufferedReader;`
  - Can import an entire package
    - `import java.io.*;`

# Packages: Importing

```
package com.foocorp.payroll;

import com.foocorp.payroll.workers.Engineer;
import com.foocorp.payroll.workers.Manager;
import com.foocorp.payroll.workers.PaidEmployee;

// could have also done this:
// import com.foocorp.payroll.workers.*;

public class FooCorporation {
  // ...
}
```

# Packages: java.lang

- Contains fundamental Java classes
  - `Object`
  - `String`
  - `System`
  - `Math`
  - ***Many more!***
- *Never needs to be imported*
  - You automatically have access to it all

# The Java API

- “Application Programming Interface”
  - Documentation for every public class that Java provides
  - Packages, classes, fields, interfaces, methods, inheritance, plus descriptions
  - *Always open* when coding complex software; even seasoned Java programmers use/need it

# The Java API: A Tour

http://java.sun.com/javase/6/docs/api/index.html

# The Java API

- Who writes the API?
  - Programmers, when they write their code
  - Documentation functionality built into special comment block: `/** ... */`
  - Uses HTML for formatting
  - Documentation generated by `javadoc` program

# The Java API: `javadoc` comments

```
package java.lang;

/**
 * Class <code>Object</code> is the root of the class hierarchy.
 * Every class has <code>Object</code> as a superclass. All objects,
 * including arrays, implement the methods of this class.
 *
 * @author  unascribed
 * @version 1.73, 03/30/06
 * @see     java.lang.Class
 * @since   JDK1.0
 */
public class Object {
  // ...
}
```

# Collections

- The Problem: arrays are limited
  - Not resizable
  - Not useful for creating mappings between objects (requires at least three arrays)
  - Not useful for keeping track of duplicate objects
  - Not useful for constant-time operations

# Collections

- The Solution: Collections
  - Better way to create dynamic groupings (`Set`), orderings (`List`), and mappings (`Map`) between objects
  - Mirror mathematical constructs
  - Are automatically resized to fit new members
  - Live in `java.util` package

# Collections Framework

- Basic useful Interfaces
  - `Collection`
    - generic container, most of the framework implements this
  - `List`
    - stores multiple items in an explicit order (repeated elements allowed)
    - `ArrayList`, `LinkedList`, etc.
  - `Set`
    - stores unique items in no particular order
    - `HashSet`, `SortedSet`, etc.
  - `Map`
    - stores unordered key-value pairs (like a dictionary; keys are unique)
    - `HashMap`, `Hastable`, `TreeMap`, etc.

- Good programming practice:
  - Don’t expose underlying types unless absolutely necessary!
    - E.g., declare as `Map`, instantiate as `HashMap()`

# Collections

- Basic useful methods:
  - `add`
  - `addAll`
  - `remove`
  - `clear`
  - `isEmpty`
  - `size` (not length!)
  - `toArray`
- *See API for more + usage!*

# Collections vs. Arrays

- Instantiation
  - Array:
    - `Person[] pa = new Person[10];`
  - Collection:
    - `Set s = new HashSet();`
- Adding a member
  - Array:
    - `pa[0] = new Person(...);`
  - Collection:
    - `s.add(new Person(...));`

# Collections vs. Arrays

- Iteration over all members
  - Array:
    - `for (int i = 0; i < pa.length; i++) { pa[i].doSomething(); }`

    ***OR***

    - `for (Person p : pa) { p.doSomething(); }`
  - Collection:
    - `Iterator i = s.iterator();`
    - `while (i.hasnext()) { ((Person)i.next()).doSomething(); }`

    ***OR***

    - `for (Object p : s) { ((Person)p).doSomething(); }`

- Note that, for a `Map`, you must iterate over its *entries*, or *keys*, or *values*; see entrySet(), keySet(), and values() methods in the `Map` API

# Collections: Generics

- Collections can hold objects of different runtime types, though we generally don't and shouldn't

- Generics allow one to specify the type of the elements in a `Collection`
  - Avoids messy casting
  - Enables us to use more than just plain `Object`

# Generified Collections vs. Arrays

- Equivalent functionality:
  - `Person[] p = new Person[10];`
  - `List<Person> al = new ArrayList<Person>();`

- Iteration, revisited:

```
Set<Person> s = new HashSet<Person>();
 // look, ma, no casting!
Iterator<Person> i = s.iterator();
while (i.hasnext()) { p = i.next(); }
```

*OR*

```
for (Person p : s) { p.doSomething(); }
```

- We specify the collection type at declaration *and* instantiation, using angle brackets (<>)

# Assignment 6, Revisited: `Person`

```
package yourface.entities;
import java.util.ArrayList;

public class Person implements SocialEntity {

  protected String name;
  protected long id;
  protected List<Person> acquaintances;
  protected List<Network> networks;
  protected String location;

  public Person(String name, long id, String location) {
    this.name = name;
    this.id = id;
    this.location = location;
    this.acquaintances = new ArrayList<Person>;
    this.networks = new ArrayList<Network>;
  }

  // ... getters & setters ...

}
```

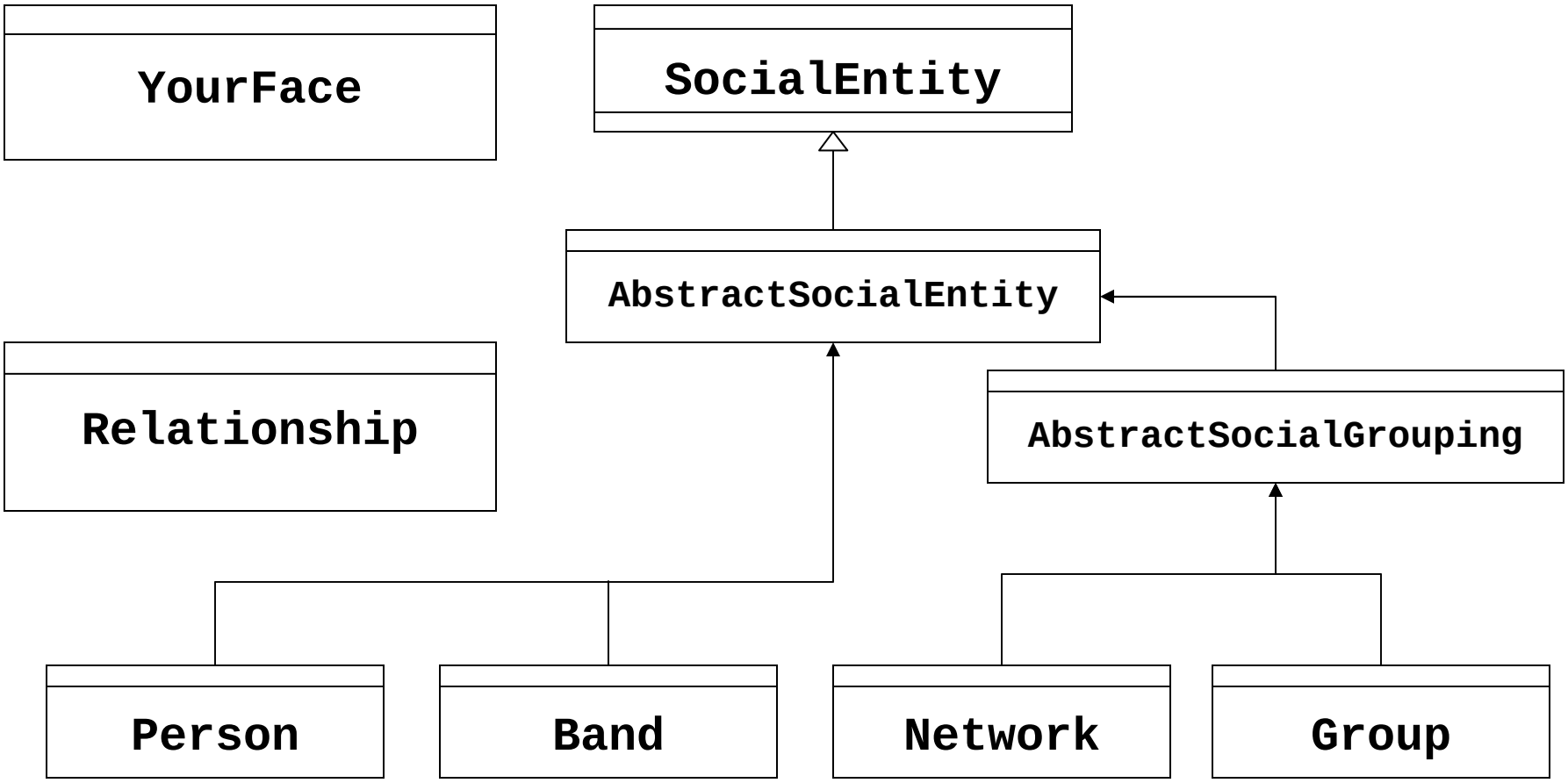
# Assignment 7:
# Refining YourFace

- Use your current knowledge about Packages, Collections, and Abstract Classes to refine your code from Assignment 6

# Assignment 7: Refining YourFace

- Reinforce the Abstraction Barrier!
  - Use Abstract Classes
- Improve functionality!
  - use Collections
- Add organizational structure!
  - use packages

# Assignment 7: Example Diagram



# Assignment 7: Refining YourFace

- ***Tips:***
  - Start ***now***
  - Stay for the lab hour
  - Ask questions often (in person or via email)
  - Reuse your old code as much as you can
  - Use the Java API!
    - Along with other references listed on the course homepage
  - Take advantage of office hours