Project 1 Notes

- JavaTokenizer.java: The interface between clients and implementation of the API, itself having two methods, getNextID and getLineNumber
- The Xref app: this is a client of the JavaTokenizer API/interface. It calls nextId and getLineNumber to do its work.
- Annotator: this is another client of the JavaTokenizer
- Tokenizer: This in the implementation (concrete class) implementing JavaTokenizer.

The interface: could be different

- Here in JavaTokenizer, we have two methods to communicate ids and line numbers, looks like this:
  - String getNextID()
  - int getLineNumber() (to be called after getNextID)
- Recall a possible objection to this API: "The actions should be tied together in one method to ensure that the ID and line number are in fact related"
- But Java only allows a single value to be returned from a method
- So to return id and line number from one method, we would need a little additional object to hold the id and line number together.

FYI: Java vs. Javascript

- Some languages have "literal objects" cooked up as needed to hold data for cases like this. In Javascript, we could put (where the function returns a little object)
  ```javascript```
  ```javascript```
- But Java and similar compiled languages don’t have this feature: all Java objects have proper Java classes.
- So we would create a little class with name and count instance variables to do this, say NameCount, with getters, and create one in the Tokenizer, and return it.
- Then:
  ```java```

Calling the JavaTokenizer API

- We see that Xref has a "tok" instance variable of type JavaTokenizer, as is typical for a client of the API.
- That means its object code can call tok getNextID() to get the next Java identifier and then tok.getLineNumber() to get its line number.
- It can also call tok.equals(o) for any object o and tok.hashCode(), even though these don’t show up in the interface, because a non-null tok is expected to be referencing an Object, and these are methods of Object.
- Note that the Object methods are available on any interface-type object. (Of course they may be the default ones, that just use the reference values.)

Remember, an Interface is a contract

- By agreeing to this API, codified in the Java interface, the two teams have a contract between them:
  - The Tokenizer team implements the API
  - The Xref team implements the code that calls the API
- Note that the interface doesn’t capture everything about the interactions: for example, the constructor syntax is missing, and the treatment of end-of-data.
- But it is an excellent start on the full contract.
The morals of the pa1 story
• Interfaces are crucial to software design, especially for larger programs with multiple source files and multiple teams.
• Interfaces let us treat multiple implementations of a certain API as the same Java type, allowing client code to run using either one.
  • Example: LinkedList and ArrayList are both type List, so code using List can switch off List implementations at will: just change the List creation step.

What's next in our Java type coverage?
• Answer: inheritance in general (Using an interface is a special case of inheritance.)
• One class can be a subclass of another, not just "implements another".
  • If B is a subclass of A, then B ISA A (A is a class)
  • If B implements A, then B ISA A (A is an interface)
  • And any class A is a subclass of Object, so A ISA Object
• Either way, we can write client code that treats a family of types as a single type.

Class Inheritance
• For decades considered the bedrock of object oriented software practice, but lately less championed.
• Where can I read about inheritance and subclasses?
  • Not in S&W: not even in index!  But "inherited methods" are, mostly methods of Object. For example, we can use Object.hashCode() for any object, since its class is a subclass of Object.
  • Weiss, Chap. 4 (also has interfaces)
  • Java tutorial at Oracle, other tutorials on Web

Example: A Student ISA Person
• Recall our Student object, with instance variables id, name, and gpa.
• GPA is specific to students, but id and name could work for any person, so let's set up a Person class
  
```java
public class Person {
    private int id;
    private String name;
    public Person(String n, int i) {
        name = n; id = i; } // set up superclass fields
    public String toString() { return name +": " + id; }
}
```

A Student ISA Person
```java
public class Student extends Person {
    private double gpa;  // additional instance var
    public Student(String n, int i, double gpa) {
        super(n, i);    // set up superclass fields
        this.gpa = gpa;
    }
    public double getGpa() { return gpa; }
    public String toString() { return getName() + "" + getId() + "" + gpa; }
}
```

Using this setup
```java
public class TestPerson {
    public static void main(String args[]) {
        Person p = new Person("Alice", 101);
        Student student = new Student("Joe", 100, 3.5);
        System.out.println(p.getId());
        System.out.println(student.toString());
        System.out.println(student.getGpa());
        System.out.println(student.getName());
    }
}
```

Output:
```
101
(Joe 100 3.5)
3.5
Joe
```
Memory Layout

- Light shading indicates fields that are private, and accessible only by methods of the class. (ignore age, address, and phone here)
- Dark shading in the Student class indicates fields that are not accessible in the Student class but are nonetheless present

Memory Layout, v2, showing String objects

- Strings are separate objects from the Person object itself, so claim other areas of memory.
- Strings are optimized, so "Joe" may be really only in one place.
- Age and gpa are primitive types, housed in the Person object

What can a subclass have?

- Student can add new instance variables (e.g., gpa).
- Student can add new methods (e.g., getGPA).
- Student can override existing methods (e.g., toString).

Two changes are specifically not allowed because they would violate the notion of an IS-A relationship:
- Student cannot remove instance variables.
- Student cannot remove methods.

Finally, the new class normally specifies its own constructor(s); this is likely to involve "super" calls. More on this later.

Overriding a method

- Here toString() is defined in Person and in its subclass Student, and in its superclass, Object.
- We say toString() in Person overrides the toString() in Object and the toString() in Student overrides the toString() in Person.
- Suppose have a Person object, person, and a Student, student.
- Then person.toString() uses Person's toString, and student.toString() uses Student's.
- If neither Person nor Student had toString, then student.toString() would use Object's toString.
- The runtime system searches up the class hierarchy from the class of the variable, looking for a method matching the call.

More formally...

- Any base class methods that are not specified in the derived class are inherited unchanged, with the exception of the constructor.
- Any base class non-constructor method that is declared in the derived class's public section is overridden. The new definition will be applied to objects of the derived class.
- Public base class methods may not be overridden in the private section of the derived class, because that would be tantamount to removing methods and would violate the IS-A relationship.

The constructor and super

- Each derived class should define its constructors.
- If no constructor is written, then a single zero-parameter default constructor is generated. This constructor will call the base class zero-parameter constructor for the inherited portion and then apply the default initialization for any additional data fields (meaning 0 for primitive types, and null for reference types).
Example Constructor

```java
public class Student extends Person {
    public Student(int id, String n, double g) {
        super(id, n); // init. Person
        gpa = g;      // init. our field
        // getGpa, toString omitted
        private double gpa;
    }
    // Note that super must be the first thing in the constructor.
    // If not there, gets automatically added.
}
```

Type Compatibility

- A “Student IS A Person” is describing type compatibility, same as we saw with interfaces. Example:
  ```java
  Student s = new Student(100, "Joe", 4.0);
  Person p = s;
  System.out.println("Name is "+p.getName());
  ```
- p of type Person, may reference any object that IS A Person
- p.method(...) works for any method of Person (or Object)
- Any method that we can invoke through the p reference (i.e. methods of Person) is guaranteed to make sense, since once a method is defined for Person, it cannot be removed by a derived class.

Type Compatibility and Parameter passing

- Suppose age is added to Person and we write
  ```java
  public static boolean isOlder(Person p1, Person p2) {
      return p1.getAge() > p2.getAge();
  }
  ```
- Consider the following declarations, in which constructor arguments are missing to save space:
  ```java
  Person   p = new Person( ... );
  Student  s = new Student( ... );
  Employee e = new Employee( ... ); // another subclass
  ```
- All in all, we now have leveraged one non-class routine to work for nine different cases
- Imagine the huge code reuse if a method takes an array of Person references. Could sort by age across all subtypes.

Recall our example of Interface Point2Dim

- Consider `Point2Dim p = new Point2D(3,4);`
  ```java
  // For Point2Dim definition, see class3, for Point2D, page 77
  ```
- Here the reference variable p is of interface type Point2Dim, and it points to an object of type Point2D, a subtype of Point2Dim.
- We have seen that a reference variable p can have one type, while the object it points to can have another, though it must be a subtype of p’s type. - True in general, since a subclass is a subtype
- The type of the reference variable is called p’s static type, while the type of the object pointed to is called p’s dynamic type.

Static and Dynamic types, subclass case

```java
Student s = new Student(100, "Joe", 4.0);
Employee e = new Employee(20, "Boss", 100000.0);
Person p = null;
if(getTodayDay().equals("Tuesday")
    p = s; else p = e;
System.out.println("Person is "+p.toString());
```

Here p has static type Person because its variable’s type is Person.
It has dynamic type at the println of Student or Employee, depending on the day, and the Student or Employee toString is called.
So p.method(...) can be called for any Person method, and then the actual code that executes is determined by the dynamic type.
Static and Dynamic types: interface case, example 2

Recall from TestArrayList in hw2:
List<String> array = new ArrayList<String>();
array.trimToSize(); // ArrayList-only method

Caused a compiler error when trying to call this non-List method:
TestArrayList.java:8: error: cannot find symbol
array.trimToSize();
Here array has static type List<String> because its variable's type is List<String>.
It has dynamic type ArrayList<String> because it was constructed that way, although in fact type erasure brings it down to the raw type ArrayList during execution.

Static and Dynamic types: interface case, cont.

List<String> array = new ArrayList<String>();

- Here array has static type List<String> because its variable's type is List<String>.
- It has dynamic type ArrayList<String> because it was constructed that way. At runtime, the type is actually just ArrayList, the raw type, by the type erasure of generic types.
- We can call any List method using array.method(...). What code actually executes then is determined by ArrayList, the dynamic type.
- You see it's the same principle whether B is a subclass of A or a B implements A, any case of B is a subtype of A.

Polymorphism

- This type flexibility is an important object-oriented principle known as polymorphism.
- A reference variable that is polymorphic can reference objects of several different types, over time.
- When operations are applied to the reference*, the operation that is appropriate to the actual referenced object is automatically selected

*ones allowed by its static type (or downcast...)

Can we just use an interface? Person? Why use a subclass?

- Yes, we can...
- Then PlainPerson implements Person, Student implements Person, Employee implements Person,
- And we have type compatibility same as subclasses
- What's the difference then?
- The implementation code setup is different

Subclass vs. Interface: where is the code for getName or getId?

Class/subclass approach
- One place, in the base class, called for all subtypes.

Interface approach
- In each implementing class, to satisfy the interface getName() and getId() methods

- This is a big feature: "code reuse", since one body of code handles multiple subtypes, and code is not duplicated across classes.

Subclass vs. Interface: where is the code for getGpa?

Class/subclass approach
- One place, in the subclass
- getGpa is not available via a Person variable, only a Student variable

Interface approach
- In one place, in the implementing class for Student
- getGpa is not in the interface, so not available via a Person variable, only a Student variable.
Interface vs. Subclass: summary of major differences

**Interface approach**
- Methods in common have implementation code in multiple places (disadvantage)
- A class can implement multiple interfaces

**Class/subclass approach**
- Methods in common have implementation code in one place.
- A class can be a subclass of a only a single class (disadvantage)

FYI: UML class diagram, subclass case

- Person-Student-Professor
- Here we see Person with two subclasses, Student and Professor, with appropriate variables
- Note the hollow arrowhead, symbol for inheritance
- Also Address, not involved in the inheritance hierarchy:
  - This ordinary arrow to Address means that Person has an instance variable of object type Address.

FYI: UML class diagram, interface case

- Person-Student-Professor
- Here we see the Person interface implemented by Student and Professor
- Note the hollow arrowhead, symbol for inheritance, and now the dashed arrow shaft, for interface inheritance
- Person here has two properties, firstName and lastName, i.e., has methods getFirstName() and getLastName() or equivalent (it can’t have actual instance variables!)
- Student needs 3 instance variables for firstName and lastName and major

FYI: Downcasts: interface case

```java
Point2D p1 = (Point2D)p;
```
- This is a downcast from p’s Point2D to its subtype Point2D, which will throw a ClassCastException if p’s object is not in fact a Point2D as expected here.
- We can check p’s object’s type with instanceof:
  ```java
  if (p instanceof Point2D) {
   Point2D p1 = (Point2D)p;
     ... do something with p1, like p1.draw()
  }
  ```
- This way, we can code mostly with the more abstract type, and occasionally dip down to the more specific type just when needed. It’s clear what is needed to make PointXY’s work too.

FYI: Downcasts, subclass case

```java
Person s = new Student(100, "Joe", 4.0 ); // upcast
Student p = (Student)s; // downcast
```
- Here s has **static type** Person because its variable’s type is Person. It has **dynamic type** Student, so the downcast will work, ending up with p having static and dynamic type Student.
- But if s’s object wasn’t a Student, this downcast would throw with a ClassCastException.
- We can check s’s dynamic type with instanceof, just like the interface case.

Inheritance Example from JDK

- HashMap extends AbstractMap, and so do six other JDK classes, so AbstractMap is the base class, the superclass.
- AbstractMap has lots of code that all these classes use
- It offloads code from seven JDK classes, making them smaller and easier to support, and reduces code duplication.
- Note that the code in AbstractMap not efficient for get and put: to do get(x), it looks through the entrySet for x.
- So many (all?) of these seven classes override get and put.
FYI: UML for the JDK Map family

From https://thenafi36.wordpress.com/2014/09/29/the-efficient-data-structure-for-unique-key-value-entries/