Class 13: Inheritance

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.

Inheritance

- For decades considered the bedrock of object oriented software practice, but lately less championed.
- Where can I read about inheritance?
  - 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. We say hashcode is inherited from 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 fields 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
class Person {
    private int id;
    private String name;
    public Person(String n, int i) {
        name = n; id = i; }
    // getters and setters for name and id
    public toString() { return name +":" + id; }
}  
```

- This is the base class.

A Student ISA Person

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

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
What can a subclass do?

- Student can add new fields (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 fields.
- Student cannot remove methods.

Finally, the new class must specify its own constructors; this is likely to involve “super” calls.

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.

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
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() );
  ```
  ```java
  p of type Person, may reference any object that IS-A Person
  p.method(...) works for any method of Person (but not its subclass)
  Any method that we invoke through the p reference 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( ... );
  ```

Type Compatibility

• The single isOlder routine can be used for all of the following calls:
  ```java
  isOlder(p,p), isOlder(s,s), isOlder(e,e), isOlder(p,e),
  isOlder(p,s), isOlder(s,e), isOlder(e,p),
  isOlder(e,s)
  ```

• All in all, we now have leveraged one non-class routine to work for nine different cases

Static and Dynamic types

```java
Student s = new Student(100, "Joe", 4.0 );
Employee e = new Employee(20, "Boss", 100000.0);
Person p = null;
if( getTodaysDay().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

```java
List<String> array = new ArrayList<String>();
```  
Caused a compiler error when trying to call this non-List method:

```java
TestArrayList.java:8:error: cannot find symbol
array.trimToSize(); // ArrayList-only method
```

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, 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 A.

Static and Dynamic types: interface case, cont.

```java
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, 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 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.

• 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
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)

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 it’s 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.

UML for the JDK Map family

From https://tangal36.wordpress.com/2014/09/29/the-efficient-data-structure-for-unique-key-value-entries/
We'll work with the actual Java 8 HashMap code, with the TreeNode support removed to simplify it.

TreeNode support: To cover the case of using a bad hash function, a bucket that has too many entries has its collision list converted to a tree structure. This code has been deleted.

Also, many methods that AbstractMap can handle have been removed to further simplify the code, to HashMap1, provided in the setup project.

This greatly simplified HashMap1 should be readable.

First you'll "seal up" HashMap1. Lots of its data structures are not private. Of course you can't make everything private: you need to be able to call its public API, covered in the JDK Javadoc. You'll end up with HashMap2.

Then we would like to know if we can refactor HashMap to use a sealed-up container for the collision list instead of the open-coded linked list structure it has now, a list of Nodes.

You will replace the list of Nodes with S&W's SequentialSearchST, one for each bucket. Then the list will be hidden inside this "little map" for the bucket.

Look at SeparateChainingHashST on pg. 465, a class like HashMap. It has very neat code because it is using a SequentialSearchST for each bucket's collision list. Here is get from there:

```java
get(Key key) {
    return (Value) st[hash(key)].get(key); }
```

This says: hash the key, find the bucket, and it will have a collision list st, which is a little symbol table (i.e. Map) for the bucket. Just use the key to get the value from this st, and return it from the outer get.

Wow, is that neat! Can we tame the mess in HashMap?

But there's a worry when you make code beautiful, esp. if it was written by real experts.

Is the code ugly because it will run faster that way?

We can draw the collision list structures in both cases and see there's an additional reference at the start of list involved in using SequentialSearchSTs instead of the open-coded list of Nodes.

Let's find out if and how much the performance changes. Be suspicious if your code runs faster than the original HashMap!