Subtype Polymorphism, Subtyping vs. Subclassing, Liskov Substitution Principle, cont.

Announcements
- HW5 due on Tuesday
- Grades
  - Quizzes 1-4 and Exam 1 in Rainbow Grades
  - All HWs except for HW4 in Rainbow Grades, feedback in Submitty
- Exam 2 next Friday, November 3rd
  - Will post review slides and back tests on Tue

A note on RESUBMITs, HW4 and beyond
- Resubmit only if you improved your score
- HW4 Automatic Grading Total is
  if (HW4 Submit >= HW4 Resubmit) then
    return HW4 Submit
  else
    return (HW4 Submit + HW4 Resubmit)/2

A few notes on HW5
- If I have pushed new homework into hw04 repo, you need to pull first, then commit/push
- Read File Paths in hw5.html very carefully
- Don’t include anonymous inner classes in JUnit tests!
- assertTrue/assertFalse, not assert in JUnit tests

Today’s Lecture Outline
- Subtype polymorphism
- Subtyping vs. subclassing
- Liskov Substitution Principle (LSP)
- Function subtyping
- Java subtyping
- Composition: an alternative to inheritance
Today’s Lecture Outline
- New topic: Parametric polymorphism
- Java generics
  - Declaring and instantiating generics
  - Bounded types: restricting instantiations
  - Generics and subtyping. Wildcards
  - Type erasure
  - Java arrays

Liskov Substitution Principle
- Java subtypes (realized with `extends`, `implements`) must be true subtypes
  - Java subtypes that are not true subtypes are dangerous and confusing
- When B is a Java subtype of A, ensure
  1. B, does not remove methods from A
  2. A substituting method B.m has stronger specification than method A.m which it substitutes
- Guarantees substitutability of class B for A

Intuition:
Type Signature is a Specification
- Type signature is a contract too
  E.g., `double f(String s, int i) {...}
  Precondition: arguments are a String and an int
  Postcondition: result is a double
- We need reasoning about behavior and effects, so we added requires, effects, etc.

Function Subtyping
- In programming languages function subtyping deals with substitutability of functions
  - Question: under what conditions on the parameter and return types A,B,C and D, is function A f(B) substitutable for C f(D)
  - Reasons at the level of the type signature
    - Rule: A f(B) is a function subtype of C f(D) iff A is a subtype of C and B is a supertype of D
    - Guarantees substitutability (at the level of signature)

Type Signature of Substituting Method is Stronger
- Method parameters (inputs):
  - Parameter types of A.m may be replaced by supertypes in subclass B.m. “contravariance”
    - E.g., A.m(String p) and B.m(Object p)
  - B.m places no extra requirements on the client!
    - E.g., client: A.a; ... a.m(q). Client knows to provide q a String. Thus, client code will work fine with B.m(Object p), which asks for less: an Object, and clearly, every String is an Object
  - Java does not allow change of parameter types in an overriding method. More on Java overriding shortly

Type Signature of Substituting Method is Stronger
- Method returns (results):
  - Return type of A.m may be replaced by subtype in subclass B.m. “covariance”
    - E.g., Object A.m() and String B.m()
  - B.m does not violate expectations of the client!
    - E.g., client: A a; ... Object o = a.m(). Client expects an Object. Thus, String will work fine
  - No new exceptions. Existing exceptions can be replaced by subtypes
  - Java does allow a subtype return type in an overriding method!
Properties Class from the JDK

Properties stores String key-value pairs. It extends Hashtable so Properties is a Java subtype of Hashtable. What’s the problem?

class Hashtable {
    // modifies: this
    // effects: associates value with key
    public void put(Object key, Object value) {
        // returns: value associated with key
        public Object get(Object key);}

class Properties extends Hashtable {
    // modifies: this
    // effects: associates String value with String key
    public void put(String key, String value) {
        super.put(key, value);}
    // returns: value associated with key
    public String get(String key) {
        return (String) super.get(key);}
    }

Exercise

class Hashtable {
    public void put(Object key, Object value);
    public Object get(Object key);
}

class Properties extends Hashtable {
    public void put(String key, String value);
    public String get(String key);
}

Fall 17 CSCI 2600, A Milanova (based on example by Michael Ernst)

Exercise

Assume Z subtype of Y, Y subtype of X
class A: X m(Y, String)
Which ones are function subtypes of A.m
1. Y m(X, Object)
2. Object m(X, Object)
3. Y m(Z, String)
4. Z m(X, String)

Fall 17 CSCI 2600, A Milanova

Exercise

class Product {
    Product recommend(Product p);
}
Which one is a function subtype of Product.recommend?
class SaleProduct extends Product {
    Product recommend(SaleProduct p);
    SaleProduct recommend(Product p);
    Product recommend(Object p);
    Product recommend(Product p) throws NoSaleException;
}

Fall 17 CSCI 2600, A Milanova (based on example by Michael Ernst)

Reasoning about Specs

Function subtyping reasons with type signatures
Remember, type signature is a specification
  Precondition: requires arguments of given type
  Postcondition: promises result of given type
Compiler can check function subtyping
Specifications (such as PoS specifications) add behavior and effects
  Precondition: stated by requires clause
  Postcondition: stated by modifies, effects, returns
  and throws clauses
Reasoning about specification strength is reasoning about “behavioural subtyping”

Behavioral subtyping generalizes function subtyping
B.m is a “behavioral subtype of/stronger than” A.m
  B.m has weaker precondition than A.m
    This is contravariance
    Just like requirement of function subtyping:
    “B.m’s parameter is a supertype of A.m’s parameter”
  B.m has stronger postcondition than A.m
    This is covariance
    Just like “B.m’s return is a subtype of A.m’s return”
These 2 conditions guarantee B.m’s spec is stronger than A.m’s spec, and B.m is substitutable for A.m
Outline of today’s class

- Function subtyping
- Java subtyping
- Composition: an alternative to inheritance

Java Subtypes

- Java types are defined by classes, interfaces, primitives
- Java subtyping stems from declarations
  - B extends A
  - B implements A
- In a Java subtype, a “substituting” method is an overriding method
  - Has same parameter types
  - Has compatible (same or subtype) return type
  - Has no additional declared exceptions

Overloading vs. Overriding

- If a method has same name, but different parameter types, it overloads not overrides

```java
class HasTable {
    public void put(Object key, Object value);
    public Object get(Object key);
}
class Properties extends HasTable {
    public void put(String key, String value);
    public String get(String key);
}
```

Remember Duration

```java
class Object {
    public boolean equals(Object o);
}
class Duration {
    public boolean equals(Object o);
    public boolean equals(Duration d);
}
Duration d1 = new Duration(10, 5);
Duration d2 = new Duration(10, 5);
System.out.println(d1.equals(d2)); // Compiler chooses family equals(Duration d)
```
Remember Duration

```java
class Object {
   public boolean equals(Object o);
}
class Duration {
   public boolean equals(Object o);
   public boolean equals(Duration d);
}
Object d1 = new Duration(10,5);
Object d2 = new Duration(10,5);
System.out.println(d1.equals(d2));
// Compiler choses equals(Object o)
// At runtime: Duration.equals(Object o)
```
Fragile Base Class Problem

- Previous slide showed an example of the Fragile Base Class Problem
- Fragile Base Class Problem happens when seemingly innocuous changes in the superclass break the subclass

Subclassing is Difficult

- A set that counts the number of attempted additions:

```
class InstrumentedHashSet extends HashSet {
    private int addCount = 0;
    public InstrumentedHashSet(Collection c) {
        super(c);
    }
    public boolean add(Object o) {
        addCount++; return super.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size(); return super.addAll(c);
    }
    public int getAddCount() { return addCount; }
}
```

Subclassing is Difficult

```
InstrumentedHashSet s=new InstrumentedHashSet();
System.out.println(s.getAddCount()); // 0
s.addAll(Arrays.asList("One","Two");
System.out.println(s.getAddCount()); // Prints?
```

InstrumentedHashSet.addAll calls HashSet.addAll calls InstrumentedHashSet.add

Behavior of HashSet.addAll depends on subclass InstrumentedHashSet!

The Yo-yo Problem

- this.add(o) in superclass HashSet calls InstrumentedHashSet.add! Callback.
- Example of the yo-yo problem. Call chain "yo-yos" from subclass to superclass back to subclass
  - InstrumentedHashSet.addAll calls HashSet.addAll calls InstrumentedHashSet.add
  - Behavior of HashSet.addAll depends on subclass InstrumentedHashSet!

Java Subtyping with Interfaces

- Which I call interface inheritance
  - Client codes against type signature of interface methods, not concrete implementations
    - Behavioral specification of an interface method often unconstraint: often just true => false. Any (later) implementation is stronger!
  - Caveat: Java 8’s default methods complicate this!

Java Subtyping with Interfaces

- Why Set and not HashSet?
  - class InstrumentedHashSet implements Set {
    private final Set s = new HashSet();
    private int addCount = 0;
    public InstrumentedHashSet(Collection c) {
        this.addAll(c);
    }
    public boolean add(Object o) {
        addCount++; return s.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size(); return s.addAll(c);
    }
    public int getAddCount() { return addCount; }
    //... Must add all methods specified by Set }
```
Java Subtyping with Interfaces

In JDK and Android SDK
- Implement multiple interfaces, extend single abstract superclass (very common!)
- Abstract classes minimize number of methods new implementations must provide
- Abstract classes facilitate new implementations
- Using abstract classes is optional, so they don’t limit freedom
- Extending a concrete class is rare and often problematic (e.g., Properties, Timestamp, which we saw in Equality lecture)

Why prefer implements A over extends A
- A class has exactly one superclass. In contrast, a class may implement multiple interfaces. An interface may extend multiple interfaces
- Interface inheritance gets all the benefit of subtype polymorphism
- And avoids the pitfalls of subclass inheritance, such as the fragile base class problem, etc.
- Multiple interfaces, single abstract superclass gets most of the benefit

Outline
- Subtype polymorphism
- Subtyping vs. subclassing
- Liskov Substitution Principle (LSP)
- Function subtyping
- Java subtypes
- Composition: an alternative to inheritance

Composition
- Properties extending Hashtable was problematic, thus, we cannot subclass. An alternative solution?
- InstrumentedHashSet extending HashSet was a bad idea too. An alternative?
- Box was not a true subtype of BallContainer. Cannot subclass.
- Composition!

Properties

class Properties { // simplified
    private Hashtable ht = new Hashtable();
    // modifies: this
    // effects: associates value with key
    public void setProperty(String key, String value)
        { ht.put(key, value); }
    // returns: value associated with key
    public void getProperty(String key)
        { return (String) ht.get(key); }
}

InstrumentedHashSet

class InstrumentedHashSet {
    private final Set s = new HashSet();
    private int addCount = 0;
    public InstrumentedHashSet(Collection c) {
        s.addAll(c);
    }
    public boolean add(Object o) {
        addCount++;
        return s.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return s.addAll(c);
    }
    public int getAddCount() { return addCount; }
}
class Box {
    private BallContainer ballContainer;
    private double maxVolume;
    public Box(double maxVolume) {
        this.ballContainer = new BallContainer();
        this.maxVolume = maxVolume;
    }
    public boolean add(Ball b) {
        if (b.getVolume() + ballContainer.getVolume() > maxVolume)
            return false;
        else
            return ballContainer.add(b);
    }
}

Composition

- Implementation reuse without inheritance
  - More common than implementation reuse through inheritance (subclassing)!
  - Easy to reason about
  - Can works around badly-designed classes
  - Disadvantages?

Composition Does not Preserve Subtyping

- InstrumentedHashSet is not a Set anymore
  - So can't substitute it
- It may be a true subtype of Set!
  - But Java doesn't know that
- That nice trick with interfaces to the rescue
  - Declare that the class implements interface Set
  - Requires that such interface exists

Nice Trick with Interfaces

class InstrumentedHashSet implements Set {
    private final Set s = new HashSet();
    private int addCount = 0;
    public InstrumentedHashSet(Collection c) {
        this.addAll(c);
    }
    public boolean add(Object o) {
        addCount++;
        return s.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return s.addAll(c);
    }
    public int getAddCount() { return addCount; }
    // ... Must add all methods specified by Set
}

Today's Lecture Outline

- Parametric polymorphism
  - Java generics
    - Declaring and instantiating generics
    - Bounded types: restricting instantiations
    - Generics and subtyping. Wildcards
    - Type erasure
  - Java arrays

Polymorphism

- Subtype polymorphism
  - What we discussed... Code can use a subclass B where a superclass A is expected
  - E.g., Code A a; a.m() is “polymorphic”, because a can be of many different types at runtime: it can be a A object or an B object. Code works with A and with B (with some caveats!)
  - Standard in object-oriented languages
Polymorphism

- Parametric polymorphism
  - Code takes a type as a parameter
  - Implicit parametric polymorphism
  - Explicit parametric polymorphism
  - Standard in functional programming languages

- Overloading typically referred to as “ad-hoc” polymorphism

Implicit Parametric Polymorphism

- There is no explicit type parameter(s).
  Code is “polymorphic” because it works with many different types. E.g.:
  ```python
  def intersect(sequence1, sequence2):
      result = []
      for x in sequence1:
          if x in sequence2:
              result.append(x)
      return result
  ```

  - As long as sequence1 and sequence2 are of some iterable type, `intersect` works!

Implicit Parametric Polymorphism

- In Python, Lisp, Scheme, others languages
  - There is no explicit type parameter(s); the code works with many different types
  - Usually, there is a single copy of the code, and all type checking is delayed until runtime
    - If the arguments are of type as expected by the code, code works
    - If not, code issues a type error at runtime

Explicit Parametric Polymorphism

- In Ada, Clu, C++, Java
  - There is an explicit type parameter(s)
  - Explicit parametric polymorphism is also known as genericity
  - E.g. in C++ we have templates:
    ```cpp
    template<class V>
    class list_node {
        list_node<V>* prev;
        ...
    }
    template<class V>
    class list {
        list_node<V> header;
        ...
    }
    ```

- Java generics work differently from C++ templates: more type checking on generic code
  - OO languages usually have both: subtype polymorphism (through inheritance: A extends B or A implements B), and explicit parametric polymorphism, referred to as generics or templates
  - Java didn’t have generics until Java 5 (2004)!
Using Java Generics

List<AType> list = new ArrayList<AType>();

AType is the type argument. We instantiated generic (templated) class ArrayList with concrete type argument AType

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

names.add("Ana");

names.add("Katarina");

String s = names.get(0); // what happens here?

Point p = names.get(0); // what happens here?

Point p = (Point) names.get(0); // what happens?

Defining a Generic Class

// generic (templated, parameterized) class
public class Name<TypeVar> {

Convention: TypeVar is 1-letter name such as T for Type, E for Element, N for Number, K for Key, V for Value

Class code refers to the type parameter
E.g. E

To instantiate a generic class, client supplies type arguments
E.g, String as in List<String> names;
Think of it as invoking a "constructor" for the generic class

Generics Clarify Your Code

Without generics
This is known as "pseudo-generic containers"

interface Map {
    Object put(Object key, Object value);
    Object get(Object key);
}

Client code:
Map nodes2neighbors = new HashMap();
String key = ...

nodes2neighbors.put(key, value);

HashSet neighbors = (HashSet) nodes2neighbors.get(key);

With generics

interface Map<K,V> {
    V put(K key, V value);
    V get(K key);
}

Client code:
Map<String,HashSet<String>> nodes2neighbors = new HashMap<String,HashSet<String>>(){};

String key = ...

nodes2neighbors.put(key, value);

HashSet<String> neighbors = nodes2neighbors.get(key);
Today’s Lecture Outline

- Parametric polymorphism
- Java generics
  - Declaring and instantiating generics

  Next time:
  - Bounded types: restricting instantiations
  - Generics and subtyping. Wildcards
  - Type erasure
- Java arrays

Fall 17 CSCI 2600, A Milanova