Subtype Polymorphism, cont. Parametric Polymorphism and Java Generics

Announcements

- HW5 due on Friday
- Grades
  - All quizzes and Exam 1 in the LMS
  - All HWs except for HW4 in the Homework Server
- Exam 2 next Tuesday, November 3rd
  - Review slides and back tests on Announcements page tonight

Announcements

- A note on Resubmits, HW4 and beyond
  - Resubmit only if you improve your score

  HW4 Automatic Grading Total is
  if (HW4 Submit >= HW4 Resubmit) then
  return HW4 Submit
  else
  return (HW4 Submit + HW4 Resubmit)/2

Announcements

- A Few Notes on HW5
  - Read note on File Paths in hw5.html: hw5/data/filename.csv

  - You may run into scalability issues when fed large dataset. Common issue: graph construction
  - Don’t call checkRep
  - Change data structures
  - DO NOT TEST ON THE HOMEWORK SERVER! Resolve scalability issues before you submit.

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

- 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

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)
  - Reason at the level of the type signature
    - Rule: If A is a subtype of C and B is a supertype of D then A f(B) is a function subtype of C f(D)
    - Guarantees substitutability of A f(B) for C f(D)

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)

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 “behavioral subtyping”

Reason about Specs

- 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 generalizes the requirement of function subtyping: “B.m’s parameter is a supertype of A.m’s parameter”
  - Contravariance
- B.m has stronger postcondition than A.m
  - Generalizes “B.m’s return is a subtype of A.m’s return”
  - Covariance
- These 2 conditions guarantee B.m’s spec is stronger than A.m’s spec, and B.m is substitutable for A.m

Overloading vs. Overriding in Java

- A method family contains multiple implementations of same name + parameter types sub-signature (no return type)
- Which method family is determined at compile time based on compile-time types
  - E.g., family put(Object key, Object value) or family put(String key, String value)
- Which implementation from the method family runs, is determined at runtime based on the runtime type of the receiver
Java Subtyping Guarantees

- A variable’s runtime type (i.e., the class of its runtime object) is a Java subtype of the variable’s declared class (Not true in C++)!
- Object o = new Date(); // OK
- Date d = new Object(); // Compile-time error
- Thus, objects always have implementations of the method specified at the call site
- Client: B b; ... b.m() // Runtime object has m()
- If all subtypes are true subtypes, spec of runtime target m() is stronger than spec of B.m()

Subclassing is Difficult

- A set that counts the number of attempted additions:
  ```java
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; }
}
```

- 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

Fragile Base Class Problem

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

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?

Subclassing is Difficult

- InstrumentedHashSet is a true subtype of HashSet. But... Something goes quite wrong here

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
  - `HashSet.addAll` calls `InstrumentedHashSet.add`
  - `InstrumentedHashSet.addAll` calls `HashSet.addAll`
  - Behavior of `HashSet.addAll` depends on subclass `InstrumentedHashSet`!

Why Set and not HashSet?

- 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!
  - Facilitates composition and wrapper classes as in the `InstrumentedHashSet` example

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!

- Facilitates composition and wrapper classes as in the `InstrumentedHashSet` example

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

Java Subtyping with Interfaces

In JDK and the 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 the Equality lecture)

Outline

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

- Properties is not a true subtype of Hashtable. Thus, cannot subclass. An alternative solution?
- Subclassing is a bad idea for the InstrumentedHashSet too. An alternative?
- Box is not a true subtype of BallContainer. Cannot subclass.
- Composition!

Properties

```java
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

```java
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; }
}
```

Box

```java
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
  - Adds level of indirection
  - Tedious to write
  - Does not preserve subtyping

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

```java
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 an `A` object or an `B` object. Code works with `A` and with `B` (with some caveats!)
  - Standard in object-oriented languages

- 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:

```
template<class V>
class list_node {
    list_node<V>* prev;
    list list<V> header;
    ... // constructors, methods
}
```

```
template<class V>
class list {
    list_node<V> header;
    ... // constructors, methods
}
```

- Usually templates are implemented by creating multiple copies of the generic code, one for each concrete type argument, then compiling
- Problem: if you instantiate with the “wrong” type argument, C++ compiler gives us long, cryptic error messages referring to the generic (templated) code in the STL.

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

```
class MySet<T> { 
    // rep invariant: non-null,
    // contains no duplicates
    List<T> theRep;
    T lastLookedUp;
}
```

// generic (templated, parameterized) class public class Name<TypeVar, ... 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.,
- To instantiate a generic class, client supplies type arguments
  - E.g., String m in List<String> name;
  - Think of it as invoking a “constructor” for the generic class
Example: a Generic Interface

```java
// Represents a list of values
public interface List<E> {
    public void add(E value);
    public void add(int index, E value);
    public E get(int index);
    public int indexOf(E value);
    public boolean isEmpty();
    public void remove(int index);
    public void set(int index, E value);
    public int size();
}
```

```java
public class ArrayList<E> implements List<E> {
    // Implementation...
}
```

```java
public class LinkedList<E> implements List<E> {
    // Implementation...
}
```

Generics Clarify Your Code

Without generics

- This is known as “pseudo-generic containers”

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

Client code:

```java
Map nodes2neighbors = new HashMap();
String key = …
nodes2neighbors.put(key, value);
HashSet neighbors = (HashSet) nodes2neighbors.get(key);
```

Casts in client code. Clumsy. If client mistakenly puts non-HashSet value in map, ClassCastException at this point.

With generics

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

Client code:

```java
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
  - Bounded types: restricting instantiations
  - Generics and subtyping. Wildcards
  - Type erasure
- Java arrays

Bounded Types Restrict Instantiation by Client

```java
interface MyList1<E extends Object> { … }
```

`MyList1` can be instantiated with any type. Same as

```java
interface MyList1<E> { … }
```

```java
interface MyList2<E extends Number> { … }
```

`MyList2` can be instantiated only with type arguments that are `Number` or subtype of `Number`

```java
MyList1<Date> // OK
MyList2<Date> // what happens here?
```

Why Bounded Types?

- Generic code can perform operations permitted by the bound

```java
class MyList1<E extends Object>
    void m(E arg) {
        arg.intValue(); // compile-time error; Object does not have intValue()
    }
```

```java
class MyList2<E extends Number>
    void m(E arg) {
        arg.intValue(); // OK. Number has intValue()
    }
```

Fall 15 CSCI 2600, A Milanova (modified from example by Michael Ernst)
Another Example

```java
public class Graph<N> implements Iterable<N>
    private final Map<N, Set<N>> node2neighbors;
public Graph(Set<N> nodes, 
    Set<Tuple<N, N>> edges) {
    ...
}

public interface Path<N, P extends Path<N, P>> 
    extends Iterable<N>, Comparable<Path<?, ?>> {
public Iterator<N> iterator(); ...
}
```

Fall 15 CSCI 2600, A Milanova (examples by Michael Ernst)

Bounded Type Parameters

- `<Type extends SuperType>`
  - An upper bound, type argument can be `SuperType` or any of its `subtypes`

- `<Type super SubType>`
  - A lower bound, type argument can be `SubType` or any of its `supertypes`

Exercise

- Given this hierarchy with X, Y and Z:
  - What are valid instantiations of generics
    - class A<T extends X> { ... }
    - class A<T extends Z> { ... }
    - class A<T super Z> { ... }
    - class A<T super X> { ... }

Declarations of a Generic Method

```java
class MyUtils {
    <T extends Number>
    T sumList(Collection<T> l) {
        ...
    }
}
```

Fall 15 CSCI 2600, A Milanova (modified from example by Michael Ernst)

Generic Method Example: Sorting

```java
public static <T extends Comparable<T>>
    void sort(List<T> list) {
        // use of get & T.compareTo<T>
        // T e1 = l.get(...);
        // T e2 = l.get(...);
        // e1.compareTo(e2);
        ...
    }
```

Fall 15 CSCI 2600, A Milanova (modified from example by Michael Ernst)

Another Generic Method Example

```java
public class Collections {
    ...
    public static <T> void copy(List<T> dst, List<T> src) {
        for (T t : src) {
            dst.add(t);
        }
    }
```

Fall 15 CSCI 2600, A Milanova (modified from example by Michael Ernst)

We can use T.compareTo<T> because T is bounded by Comparable<T>!

When you want to make a single (often static) method generic
da class, precede its return type by type parameter(s).
More Bounded Type Examples

\[
\langle T \text{ extends } \text{Comparable} \langle T \rangle \rangle
\]

\[
T \text{ max}(\text{Collection} \langle T \rangle \ c);
\]

\[
\langle T \rangle \text{ void copy(List} \langle T2 \text{ super } T \rangle \text{ dst, List} \langle T3 \text{ extends } T \rangle \text{ src);}
\]

(Actually, must use wildcard \langle ? \rangle --- more on this later:

\[
\langle T \rangle \text{ void copy(List} \langle ? \text{ super } T \rangle \text{ dst, List} \langle ? \text{ extends } T \rangle \text{ src); }
\]

\[
\langle T \text{ extends } \text{Comparable} \langle T2 \text{ super } T \rangle \rangle
\text{ void sort(List} \langle T \rangle \text{ list)}
\]

(same, must use \langle ? \text{ super } T \rangle \rangle

Next time

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