

## 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 3<sup>rd</sup>
  - Review slides and back tests on Announcements page tonight

2

## Announcements

- A note on Resubmits, HW4 and beyond
- Resubmit only if you improve your score
- HW4 Automatic Grading Total is  
if (HW4 Submit  $\geq$  HW4 Resubmit) then  
return HW4 Submit  
else  
return (HW4 Submit + HW4 Resubmit)/2

Fall 15 CSCI 2600, A. Milanova

3

## 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.

Fall 15 CSCI 2600, A. Milanova

4

## Today's Lecture Outline

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

Fall 15 CSCI 2600, A. Milanova

5

## Today's Lecture Outline

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

Fall 15 CSCI 2600, A. Milanova

6

## 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**

7

## 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: 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)!**

Fall 15 CSCI 2600, A. Milanova

8

## 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 15 CSCI 2600, A. Milanova

9

## 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"

10

## 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

Fall 15 CSCI 2600, A. Milanova (based on slides by Mike Ernst)

12

## 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()`

13

## Subclassing is Difficult

Before:

```
class A {
    private int c=0;
    void inc1() { c++; }
    void inc2() { c++; }
}
class B extends A {
    @Override
    void inc2() {
        inc1();
    }
}
```

After a tiny change:

```
class A {
    private int c=0;
    void inc1() { inc2(); }
    void inc2() { c++; }
}
class B extends A {
    @Override
    void inc2() {
        inc1();
    }
}
```

Fall 15 CSCI 2600, A Milanova

14

## 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

Fall 15 CSCI 2600, A Milanova

15

## 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; }
}
```

Fall 15 CSCI 2600, A Milanova (based on example by Michael Ernst)

16

## Subclassing is Difficult

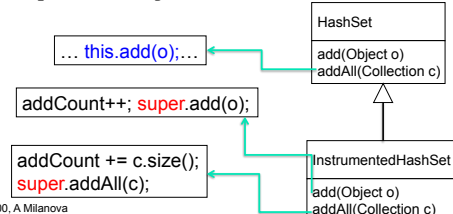
- `InstrumentedHashSet` is a true subtype of `HashSet`. But... Something goes quite wrong here

```
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; }
}
```

17

## 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?
```



Fall 15 CSCI 2600, A Milanova

## 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`!

19

## 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
}
```

Fall 15 CSCI 2600, A Milanova (based on example by Michael Ernst)

20

## 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 unconstrained: often just `true` => `false`. Any (later) implementation is stronger!
  - Facilitates composition and wrapper classes as in the `InstrumentedHashSet` example

Fall 15 CSCI 2600, A Milanova

21

## 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)

22

## 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

23

## Outline

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

Fall 15 CSCI 2600, A Milanova

24

## 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!

Fall 15 CSCI 2600, A. Milanova

25

## Properties

Wrapper class

The delegate

```
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);
    }
}
```

Fall 15 CSCI 2600, A. Milanova

26

## InstrumentedHashSet

The delegate

```
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; }
}
```

Fall 15 CSCI 2600, A. Milanova

27

## Box

The delegate

```
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);
    }
}
```

Fall 15 CSCI 2600, A. Milanova

28

## Composition

- Implementation reuse without inheritance
  - More common than implementation reuse through inheritance (subclassing)!
- Easy to reason about
- Can work around badly-designed classes
- Disadvantages
  - Adds level of indirection
  - Tedious to write
  - Does not preserve subtyping

Fall 15 CSCI 2600, A. Milanova

29

## 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

Fall 15 CSCI 2600, A. Milanova (based on slide by Michael Ernst)

30

## 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
}
```

31

## Today's Lecture Outline

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

Fall 15 CSCI 2600, A Milanova

32

## 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

Fall 15 CSCI 2600, A Milanova

33

## 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

Fall 15 CSCI 2600, A Milanova

34

## Implicit Parametric Polymorphism

- There is no **explicit type parameter(s)**. Code is "polymorphic" because it works with many different types. E.g.:

```
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!

Fall 15 CSCI 2600, A Milanova

35

## 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

Fall 15 CSCI 2600, A Milanova

36

## 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;
    ...
}

template<class V>
class list {
    list_node<V> header;
    ...
}
```

37

## Explicit Parametric Polymorphism

- Instantiating classes from previous slide with **int**:  
**typedef list\_node<int> int\_list\_node;**  
**typedef list<int> int\_list;**
- 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 :)

38

## Explicit Parametric Polymorphism

- 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)!

39

## 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?
```

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

40

## Defining a Generic Class

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

Declaration of type parameter

Use of type parameter

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

41

## Defining a Generic Class

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

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

42

## Example: a Generic Interface

```
// 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();
}

public class ArrayList<E> implements List<E> {
    ...
}

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

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

43

## 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);
```

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

44

## Generics Clarify Your Code

### 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);
```

No casts. Compile-time checks prevent client from putting non-HashSet value. Somewhat clumsy, verbose instantiations.

45

## Today's Lecture Outline

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

Fall 15 CSCI 2600, A Milanova

46

## Bounded Types Restrict Instantiation by Client

```
interface MyList1<E extends Object> { ... }
```

MyList1 can be instantiated with any type. Same as

```
interface MyList1<E> { ... }
```

Upper bound on type argument

```
interface MyList2<E extends Number> { ... }
```

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

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

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

47

## Why Bounded Types?

- Generic code can perform operations permitted by the bound

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

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)

48

## Another Example

```
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)

49

## 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

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

50

## 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> { ... } ?
```

Fall 15 CSCI 2600, A Milanova

51

## Declaring a Generic Method

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

Declaration of type parameter

Uses of type parameter

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

52

## Generic Method Example: Sorting

```
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);
    ...
}
```

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

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

53

## Another Generic Method Example

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

When you want to make a single (often static) method generic in a class, precede its return type by type parameter(s).

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

54

## More Bounded Type Examples

```
<T> extends Comparable<T>>
    T max(Collection<T> c);

<T> void copy(List<T2 super T> dst,
              List<T3 extends T> src);
(actual, must use wildcard ? --- more on this later:
<T> void copy(List<? super T> dst,
              List<? extends T> src); )

<T> extends Comparable<T2 super T>>
    void sort(List<T> list)
(same, must use ? with super: <? super T>)
```

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

55

## Next time

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

Fall 15 CSCI 2600, A Milanova

56