

## Parametric Polymorphism and Java Generics

## Announcements

- One day extension on HW5
  - Because of an error in my HW5 config
- HW6 out, due November 10
- Grades
  - All quiz and Exam grades in the LMS
  - All HW grades in the Homework Server
- Quiz 7 today

2

## Exam 2

- Topics included in Exam 2
  - Reasoning about ADTs
  - Exceptions
  - Testing
  - Equality
  - Subtype polymorphism, LSP, Java subtyping, etc.
  - Parametric polymorphism and Java generics
- Review slides and practice tests on Announcements page

3

## Announcements

- There is a penalty for excessive submissions on the Homework Server
  - 20 submissions: no penalty
  - Over 20: about 1pt for 10 submissions
- We will waive the penalty on HW5, however, it will apply on HW6 and beyond

Fall 15 CSCI 2600, A Milanova

4

## Today's Lecture Outline

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

Fall 15 CSCI 2600, A Milanova

5

## 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)

6

## 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)

7

## 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)

8

## 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)

9

## 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)

10

## Java Wildcards

```
public static
<T> void copy(List<? super T> dst,
              List<? extends T> src)

<T extends Comparable<? super T>>
void sort(List<T> list)
```

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

11

## Today's Lecture Outline

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

Fall 15 CSCI 2600, A Milanova

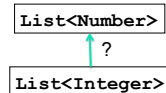
12

## Generics and Subtyping

- Integer is a subtype of Number



- Is List<Integer> a subtype of List<Number>?



Fall 15 CSCI 2600, A. Milanova

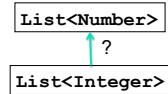
13

## Use Function Subtyping Rules to Find Out!

```

interface List<Number> {
    boolean add(Number elt);
    Number get(int index);
}

interface List<Integer> {
    boolean add(Integer elt);
    Integer get(int index);
}
  
```



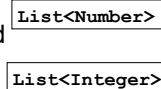
- Function subtyping: subtype must have **supertype parameters** and **subtype return!**

14

## What is the Subtyping Relationship Between List<Number> and List<Integer>

- Java subtyping is **invariant** with respect to generics: if  $A \neq B$ , then  $C<A>$  has no subtyping relationship with  $C<B>$

- Thus, List<Number> and List<Integer> are unrelated through subtyping!



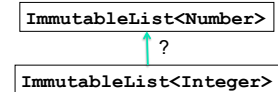
15

## Immutable Lists

```

interface ImmutableList<Number> {
    Number get(int index);
}

interface ImmutableList<Integer> {
    Integer get(int index);
}
  
```



Fall 15 CSCI 2600, A. Milanova (example due to Michael Ernst)

16

## Write-only Lists

```

interface WriteOnlyList<Number> {
    boolean add(Number elt);
}

interface WriteOnlyList<Integer> {
    boolean add(Integer elt);
}
  
```

- Is WriteOnlyList<Integer> subtype of WriteOnlyList<Number>?
- NO!
- Is WriteOnlyList<Number> subtype of WriteOnlyList<Integer>?
- YES!

Fall 15 CSCI 2600, A. Milanova (example due to Michael Ernst)

17

## Getting Stuff Out of WriteList

```

interface WriteList<Number> {
    boolean add(Number elt);
    Number get(int index);
}

interface WriteList<Integer> {
    boolean add(Integer elt);
    Object get(int index);
}
  
```

- Is WriteList<Number> subtype of WriteList<Integer>?
- YES!

**Contravariant subtyping:** because the subtyping relationship between the composites (WriteList<Number> is subtype of WriteList<Integer>) is the **opposite** of the subtyping relationship between their type arguments (Integer is subtype of Number)

18

## Invariance is Restrictive (Because it Disallows Subtyping)

### Java solution: **wildcards**

```
interface Set<E> {
    // Adds all elements in c to this set
    // if they are not already present.
    void addAll(Set<E> c);
    void addAll(Collection<E> c);
    void addAll(Collection<? extends E> c);
    <T extends E> void addAll(Collection<T> c);
}
```

Not good. Can't have  
Set<Number> s; List<Number> l;  
s.addAll(l); // List & Set unrelated

Not good either. Can't have  
Set<Number> s; List<Integer> l; s.addAll(l);  
This is because of invariance: List<Integer> is a  
subtype of Collection<Integer> but Collection<Integer>  
is not a subtype of Collection<Number>!

Solution: wildcards.  
? is the wildcard.

19

## Java Wildcards

- A wildcard is essentially an anonymous type variable
  - Use ? if you'd use a type variable exactly once
- ? appears at **use** sites of the generic type, not at declaration sites
- Purpose of the wildcard is to make a library more flexible and easier to use by allowing limited subtyping

20

## Using Wildcards

```
class HashSet<E> implements Set<E> {
    void addAll(Collection<? extends E> c) {
        // What does this give us about c?
        // i.e., what can code assume about c?
        // What operations can code invoke on c?
    }
}
```

This is **use** of the  
parameter type E

- There is also <? **super** E>
- Intuitively, why <? **extends** E> makes sense here?

Fall 15 CSCI 2600, A Milanova (based on slide due to Michael Ernst)

21

## Legal Operations on Wildcards

```
Object o;
Number n;
Integer i;
PositiveInteger p;

List<? extends Integer> lei;
First, which of these is legal?
lei.add(o);
lei.add(n);
lei.add(i);
lei.add(p);
o = lei.get(0);
n = lei.get(0);
i = lei.get(0);
p = lei.get(0);
lei = new ArrayList<Object>();
lei = new ArrayList<Number>();
lei = new ArrayList<Integer>();
lei = new ArrayList<PositiveInteger>();
lei = new ArrayList<NegativeInteger>();
```

Which of these is legal?

Fall 15 CSCI 2600, A Milanova (slide due to Michael Ernst)

22

## Legal Operations on Wildcards

```
Object o;
Number n;
Integer i;
PositiveInteger p;

List<? super Integer> lsi;
First, which of these is legal?
lsi.add(o);
lsi.add(n);
lsi.add(i);
lsi.add(p);
lsi.add(null);
o = lsi.get(0);
n = lsi.get(0);
i = lsi.get(0);
p = lsi.get(0);
lsi = new ArrayList<Object>();
lsi = new ArrayList<Number>();
lsi = new ArrayList<Integer>();
lsi = new ArrayList<PositiveInteger>();
```

Which of these is legal?

Fall 15 CSCI 2600, A Milanova (slide due to Michael Ernst)

23

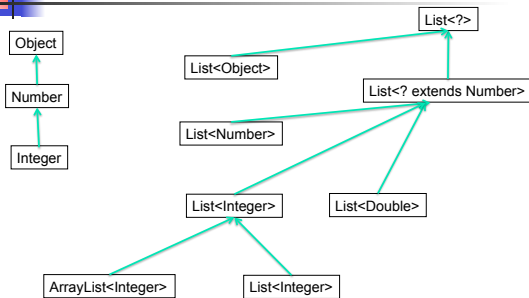
## How to Use Wildcards

- PECS: Producer Extends, Consumer Super**
- Use <? **extends** T> when you **get** (read) values from a **producer**
- Use <? **super** T> when you **add** (write) values into a **consumer**
- E.g.:  
<T> void copy(List<? super T> dst,  
                    List<? extends T> src)
- Use neither, just <T>, if both **add** and **get**

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

24

## Wildcards allow Subtyping for Generics



Fall 15 CSCI 2600, A. Milanova

25

## Today's Lecture Outline

- Parametric polymorphism
- Java generics
  - Declaring and instantiating generics
  - Bounded types: restricting instantiations
  - Generics and subtyping. Wildcards (briefly)
  - **Type erasure**
- Arrays
- Review problems

Fall 15 CSCI 2600, A. Milanova

26

## Type Erasure

- All **type arguments** become Object when compiled
    - Reason: backward compatibility with old bytecode
    - At runtime all generic instantiations have same type
- ```

List<String> lst1 = new ArrayList<String>();
List<Integer> lst2 = new ArrayList<Integer>();
lst1.getClass() == lst2.getClass() // true
  
```
- Cannot use instanceof to find type argument
- ```

Collection<?> cs = new ArrayList<String>();
if (cs instanceof Collection<String>) {
    // compile-time error
}
  
```
- Must use equals() on elements of generic type

27

## Equals for a Generic Class

```

class Node<E> {
    ...
    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Node<E>))
            return false;
        Node<E> n = (Node<E>) obj;
        return this.data().equals(n.data());
    }
}
  
```

At runtime, JVM has no knowledge of type argument. Node<String> is same as Node<Elephant>. instanceof is a compile-time error.

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

28

## Equals for a Generic Class

```

class Node<E> {
    ...
    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Node<E>)) {
            return false;
        }
        Node<E> n = (Node<E>) obj;
        return this.data().equals(n.data());
    }
}
  
```

Same here. JVM has no knowledge of type argument. Node<String> will cast to Node<Elephant>. Casting results in a compile-time warning, but not error.

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

29

## Today's Lecture Outline

- Parametric polymorphism
- Java generics
  - Declaring and instantiating generics
  - Bounded types: restricting instantiations
  - Generics and subtyping. Wildcards (briefly)
  - Type erasure
- **Arrays**
- Review questions

Fall 15 CSCI 2600, A. Milanova

30

## Arrays and Subtyping

```

graph TD
    Integer --> Number
    IntegerArray[Integer[]] --> NumberArray[Number[]]
  
```

- Integer is subtype of Number
- Is Integer[] a subtype of Number[]?
- Use our subtyping rules to find out Integer[]

(Just like with List<Integer> and List<Number>)

- Again, the answer is NO!
- Different answer in Java: in Java Integer[] is a Java subtype Number[]!
  - The Java subtype is not a true subtype!
  - Known as "problem with Java's covariant arrays"

## Integer[] is a Java subtype of Number[]

```

Number n;
Number[] na;
Integer i;
Integer[] ia;
na[0] = n;
na[1] = i;
n = na[0];
i = na[1]; //what happens?
ia[0] = n; //what happens?
ia[1] = i;
n = ia[0];
i = ia[1];

ia = na; //what
// happens here?
Double d = 3.14;
na = ia; //what?
na[2] = d; //?
i = ia[2];
  
```

## Today's Lecture Outline

- Parametric polymorphism
- Java generics
  - Declaring and instantiating generics
  - Bounded types: restricting instantiations
  - Generics and subtyping. Wildcards (briefly)
  - Type erasure
- Arrays
- Review questions

## Prints What?

```

Object d1 = new Duration(10,5);
Object d2 = null;
System.out.println(d1.equals(d2));
  
```

## True or False?

Question 2. If there are non-null reference values  $x$ ,  $y$  and  $z$  such that  $x.equals(y)$  returns false,  $y.equals(z)$  returns true and  $x.equals(z)$  returns true, then equals is not transitive.

Question 3. The consistency property requires that for every non-null  $x$  and  $y$ , such that  $x.equals(y)$  is false,  $x.hashCode() \neq y.hashCode()$ .

Question 4. Integer f(String) is a function subtype of Number f(Object).

## CFG and Def-use Pairs

```

graph TD
    Start(( )) --> S1[s=0; x=0;]
    S1 --> Cond1{x < y}
    Cond1 -- True --> B1[x=x+3; y=y+2;]
    Cond1 -- False --> B2[x+y<10]
    B1 --> B2
    B2 -- True --> N5[5. s=s+x+y]
    B2 -- False --> N6[6. s=s+x-y]
    N5 --> Merge(( ))
    N6 --> Merge
    Merge --> End[return s;]
  
```

```

int f(int y) {
1.  int s = 0;
   int x = 0;
2.  while (x < y) {
3.    x = x+3;
   y = y+2;
4.    if (x+y < 10)
5.      s = s+x+y;
   else
6.      s = s+x-y;
7.  } // end-if
8.  return s;
}
  
```

Is it possible to cover def-use pair (6,5)?

"Merge" node



## True or False

- Specification tests is just another name for black-box tests.



## Exceptions

```
void m() {  
    ...  
    try {  
        String s = new String("car");  
        String sub = s.substring(4); // IOOBE  
    }  
    catch (RuntimeException e) {  
        e.printStackTrace();  
    }  
    // the rest of m  
}
```

- a) catch block catches exception then m proceeds
- b) exception terminates m and propagates to the caller of m.