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

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

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

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> { ... }
```

Upper bound on type argument

```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?
```

Fall 15 CSCI 2600, A Milanova
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()
  }
}
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)

Bounded Type Parameters

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

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

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

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

Fall 15 CSCI 2600, A Milanova (modified from slide 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);
  ...
}
```

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)

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);
    }
  }
}
```

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)

Java Wildcards

```java
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)

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
Generics and Subtyping

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

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!

Fall 15 CSCI 2600, A Milanova

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

- Java subtyping is invariant with respect to generics: if A ≠ B, then C<A> has no subtyping relationship with C<B>
- Thus, List<Number> and List<Integer> are unrelated through subtyping!

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)

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>?
  NO!

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!
- Is WriteList<Integer> subtype of WriteList<Number>?
  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)
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);
    <T extends E> void addAll(Collection<T> c);
    // Not good either. Can't have
    // <T extends Collection<Integer>>
    // 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>!
}
```

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.

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

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

Legal Operations on 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

How to Use Wildcards

Object o;
Number n;
Integer i;
PositiveInteger p;
List<? extends Integer> lei;
First, which of these is legal?

- `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>();`

Legal Operations on Wildcards

Object o;
Number n;
Integer i;
PositiveInteger p;
List<? super Integer> lsi;
First, which of these is legal?

- `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>();`

Fall 15 CSCI 2600, A Milanova (based on slide due to Michael Ernst)
Wildcards allow Subtyping for Generics

Object
  Number
    Integer
  List/Object
  List/Number*
    List/<T extends Number>*
  List,Integer
  ArrayList<Integer>*
  List<Integer>
  List<Double>*
  List<? extends Number>
  List<?>
  List<Integer>*
  List<Double>*
  ArrayList<Integer>*
  List<?>

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

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

Equals for a Generic Class

```java
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());
        }
    }
}
```

Equals for a Generic Class

```java
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.

- 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.

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
Arrays and Subtyping

- Integer is subtype of Number
- Integer[] is subtype of Number[]?
- Use our subtyping rules to find out (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”.

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() != y.hashCode().

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

CFG and Def-use Pairs

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

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

Exceptions

```java
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.