Announcements

- HW6 will be out tonight as usual
  - If you are taking late days on HW5, pull repo before pushing!
- Check your grades in Rainbow grades
- Quiz 6 today

Exam 2 next time

- Topics included in Exam 2
  - Assertions and exceptions
  - Testing
  - Equality
  - Subtype polymorphism, LSP, Java subtyping
  - Parametric polymorphism and Java generics
- Review slides and back test on Announcements page

Today’s Lecture Outline

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

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

- 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!

 Explicit Parametric Polymorphism

- In Ada, Clu, C++, Java
- There is an explicit type parameter(s)
- Explicit parametric polymorphism is also known as genericty
- 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;
  ...
}
```
- 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 :(

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)!

Using Java Generics

```java
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 17 CSCI 2600, A Milanova (modified from an example by Michael Ernst)
Defining a Generic Class

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

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();
}
```

Generics Clarify Your Code

Without generics

```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);
```

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);
```

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

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

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

MyList1 can be instantiated with any type. Same as

interface MyList1<E> { ... }

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

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

Bounded Type Parameters

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

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

Fall 17 CSCI 2600, A Milanova (modified from slide by Michael Ernst)

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

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

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 17 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
- Java arrays
- Review questions

Generics and Subtyping

- Integer is a subtype of Number
- List<Integer> a subtype of List<Number>? (Diagram)

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!

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!

Immutable Lists

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

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!

Sample due to Michael Ernst
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)

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. 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

Purpose of the wildcard is to make a library more flexible and easier to use by allowing limited subtyping

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);
lei.add(null);

This is use of the parameter type E.

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);
lsi.add(null);

This is use of the parameter type E.

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?
    }
}

There is also List<? super E>

Intuitively, why <? extends E> makes sense here?
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*

---

Arrays and Subtyping

- **Integer** is subtype of **Number**
- Is **Integer[]** a 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”

---

Today’s Lecture Outline

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

---

Prints what?

```java
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 \(true\), 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 \(f(Object)\).

True of False

- Specification tests is just another name for black-box tests.
- An exception always indicates an unrecoverable error.

CFG and Def-use Pairs

```
int f(int y) {
  int s = 0;
  int x = 0;
  while (x < y) {
    x = x + 3;
    y = y + 2;
    if (x + y < 10)
      s = s + x + y;
    else
      s = s + x - y;
  } // end-while
  return s;
}
```

Min # tests to get 100% branch coverage?
Can we cover def-use pair (6,5)?

Exceptions

```
void m() {
  ... try {
    String s = new String("car");
    String sub = s.substring(4); // 100BE
  } 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.
```