Polymorphism (generics)

CSE 331
University of Washington
Varieties of abstraction

• Abstraction over computation: procedures
  
  ```java
  int x1, y1, x2, y2;
  Math.sqrt(x1*x1 + y1*y1);
  Math.sqrt(x2*x2 + y2*y2);
  ```

• Abstraction over data: ADTs (classes, interfaces)
  
  ```java
  Point p1, p2;
  ```

• Abstraction over types: polymorphism (generics)
  
  ```java
  Point<Integer>, Point<Double>
  ```
  
  — Applies to both computation and data
Why we ♥ abstraction

• Hide details
  – Avoid distraction
  – Permit the details to change later

• Give a meaningful name to a concept

• Permit reuse in new contexts
  – Avoid duplication: error-prone, confusing
  – Programmers hate to repeat themselves
A collection of related abstractions

interface ListOfNumbers {
    boolean add(Number elt);
    Number get(int index);
}
interface ListOfIntegers {
    boolean add(Integer elt);
    Integer get(int index);
}
... and many, many more

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

Declares a new variable, called a formal parameter
Declares a new type variable, called a type parameter
Instantiate by passing an Integer:
l.add(7);
myList.add(myInt);
The type of add is Integer → boolean
The type of List is Type → Type

Instantiate by passing a type:
List<Float>
List<List<String>>
List<T>
Restricting instantiation by clients

boolean add1(Object elt);
boolean add2(Number elt);
add1(new Date()); // OK
add2(new Date()); // compile-time error

interface MyList1<E extends Object> {...}
interface MyList2<E extends Number> {...}
MyList1<Date>      // OK
MyList2<Date>      // compile-time error
Using type variables

Code can perform any operation permitted by the bound

```java
interface MyList1<E extends Object> {  
    void m(E arg) {  
        arg.asInt();  // compiler error  
    }  
}

interface MyList2<E extends Number> {  
    void m(E arg) {  
        arg.asInt();  // OK  
    }  
}
```
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();
}
```
Type variables are types

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

class MySet<T> implements Set<T> {
    // rep invariant:
    //   non-null, contains no duplicates
    List<T> theRep;
}
Generics and subtyping

Integer is a subtype of Number

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

Use our subtyping rules to find out
List<Number> and List<Integer>

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

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

Java subtyping is covariant with respect to generics
Immutables lists

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

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

Why would we want this?
Write-only lists

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

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

WriteOnlyList<Eagle> hotelCalifornia;

Why would we want this?
Covariant subtyping is restrictive
Solution: wildcards

interface Set<E> { 
    // Adds all of the elements in c to this set 
    // if they're not already present (optional operation). 
    void addAll(Set<E> c); 
}

interface Set<E> { 
    void addAll(Collection<E> c); 
}

interface Set<E> { 
    void addAll(Collection<? extends E> c); 
}

Problem 1:
Set<Number> s;
List<Number> l;
s.addAll(l);

Problem 2:
Set<Number> s;
List<Integer> l;
s.addAll(l);
Using wildcards

class HashSet<E> implements Set<E> {
    void addAll(Collection<? extends E> c) {
        // What can this code assume about c?
        // What operations can this code invoke on c?
        ...
    }
}

Wildcards are written at declarations, not uses
A missing extends clause means extends Object
There is also “? super E”
Arrays and subtyping

Integer is a subtype of Number

Is Integer[] a subtype of Number[]?

Use our subtyping rules to find out
(Same question as with Lists)

Same answer with respect to true subtyping

Different answer in Java!

Integer[] is a Java subtype of Number[]

Java subtyping disagrees with true subtyping
**Integer[] is a Java subtype of Number[]**

```java
Number n;
Number[] na;
Integer i;
Integer[] ia;

na[0] = n;
na[1] = i;
n = na[0];
i = na[1];

ia[0] = n;
ia[1] = i;
n = ia[0];
i = ia[1];

ia = na;
Double d = 3.14;

na = ia;
na[2] = d;
i = ia[2];
```

Why did the Java designers do this?
Not all generics are for collections

class MyUtils {
    static Number sumList(List<Number> l) {
        Number result = 0;
        for (Number n : l) {
            result += n;
        }
        return result;
    }
}
Not all generics are for collections

class MyUtils {
    static T sumList(Collection<T> l) {
        // ... black magic within ... 
    }
}

Where is this type variable declared?
Not all generics are for collections

class MyUtils {
    static
    <T extends Number> T sumList(Collection<T> l) {
        // ... black magic within ...
    }
}

How to declare a type parameter to a method
public static
<T extends Comparable<T>>
void sort(List<T> list) {
    // ... use list.get() and T.compareTo(T)
}

Actually:
<T extends Comparable<? super T>>
Tips when writing a generic class

1. Start by writing a concrete instantiation
2. Get it correct (testing, reasoning, etc.)
3. Consider writing a second concrete version
4. Generalize it by adding type parameters
   – Think about which types are the same & different
   – Not all ints are the same, nor are all Strings
   – The compiler will help you find errors
Eventually, it will be easier to write the code generically from the start
   – but maybe not yet
Generics clarify your code

interface Map {  
    Object put(Object key, Object value);  
    equals(Object other);  
}

interface Map<Key, Value> {  
    Value put(Key key, Value value);  
    equals(Object other);  
}

Generics usually clarify the implementation  
sometimes ugly: wildcards, arrays, instantiation  
Generics always make the client code prettier and safer