CSE 331
Software Design and Implementation

Lecture 15
Generics<2>

Leah Perlmutter / Summer 2018
Announcements
Announcements

• Quiz 5 is due tomorrow
• Homework 6 due tomorrow
• Section tomorrow!
  – Subtyping – now with worksheet!
  – HW7 (Dijkstra’s algorithm)
Big picture

- Last time: Generics intro
- Subtyping and Generics
- Using *bounds* for more flexible subtyping
- Using *wildcards* for more convenient bounds
- Digression: Java’s *unsoundness*(es)
- Java realities: *type erasure*
Review
interface List<T> {
    boolean add(T elt);
    T get(int index);
}

So type List<Number> has:
    boolean add(Number elt);
    Number get(int index);

So type List<Integer> has:
    boolean add(Integer elt);
    Integer get(int index);

Java subtyping is *invariant* with respect to generics
    – Neither List<Number> nor List<Integer> subtype of other
    – Not covariant and not contravariant

- Subtype needs stronger spec than super
- Stronger method spec has:
  - weaker precondition
  - stronger postcondition
Generic types and subtyping

- `List<Integer>` and `List<Number>` are not subtype-related
  - No subtyping relationships based on the type argument

- Generic types can have subtyping relationships relying on the “base” type

- Example: If `HeftyBag` extends `Bag`, then
  - `HeftyBag<Integer>` is a subtype of `Bag<Integer>`
  - `HeftyBag<Number>` is a subtype of `Bag<Number>`
  - `HeftyBag<String>` is a subtype of `Bag<String>`
  - ...
Overview

• Last time: Generics intro
• Subtyping and Generics
• Using *bounds* for more flexible subtyping
• Using *wildcards* for more convenient bounds
• Digression: Java’s *unsoundness*(es)
• Java realities: *type erasure*
Overview: Bounds and Wildcards

Now: *Type bounds* e.g. `<T extends Number>`
- How to use *type bounds* to write reusable code despite invariant subtyping
- Elegant technique using generic methods
- General guidelines for making code as reusable as possible

Next: *Java wildcards* e.g. `<? extends Number>`
- Essentially provide the same expressiveness
- *Less verbose*: No need to declare type parameters that would be used only once
- *Better style* because Java programmers recognize how wildcards are used for common idioms
  - Easier to read (?) once you get used to it
Bounds
Best type for `addAll`

```java
interface Set<E> {
    // Adds all elements in c to this set
    // (that are not already present)
    void addAll(_______ c);
}
```

What is the best type for `addAll`'s parameter?
- Allow as many clients as possible…
- … while allowing correct implementations
Best type for `addAll`

```java
interface Set<E> {
    // Adds all elements in c to this set
    // (that are not already present)
    void addAll(_______ c);
}

void addAll(Set<E> c);
```

Too restrictive:
- Does not let clients pass other collections, like `List<E>`
- Better: use a supertype interface with just what `addAll` needs
- This is not related to invariant subtyping [yet]
Best type for `addAll`

```java
interface Set<E> {
    // Adds all elements in c to this set
    // (that are not already present)
    void addAll(_______ c);
}

void addAll(Collection<E> c);
```

Too restrictive:

- Client cannot pass a `List<Integer>` to `addAll` for a `Set<Number>`
- Should be okay because `addAll` implementations only need to read from `c`, not put elements in it
- This is the invariant-subtyping limitation
Best type for **addAll**

```java
interface Set<E> {
    // Adds all elements in c to this set
    // (that are not already present)
    void addAll(_______ c);
}

<T extends E> void addAll(Collection<T> c);
```

The fix: A bounded generic type parameter

- Now client *can* pass a `List<Integer>` to `addAll` for a `Set<Number>`
- `addAll` implementations won’t know what element type `T` is, but will know it is a subtype of `E`
  - So it cannot add anything to collection `c` refers to
  - But this is enough to implement `addAll`
Revisit copy method

Earlier we saw this:

```java
<T> void copyTo(List<T> dst, List<T> src) {
    for (T t : src)
        dst.add(t);
}
```

Now we can do this, which is more useful to clients:

```java
<T1, T2 extends T1> void copyTo(List<T1> dst, List<T2> src) {
    for (T2 t : src)
        dst.add(t);
}
```
Big picture

• Last time: Generics intro
• Subtyping and Generics
• Using *bounds* for more flexible subtyping
• Using *wildcards* for more convenient bounds
• Digression: Java’s *unsoundness(es)*
• Java realities: *type erasure*
Wildcards
Wildcards

Syntax: For a type-parameter instantiation (inside the `<…>`), can write:

- `? extends Type`, some unspecified subtype of `Type`
- `?`, is shorthand for `? extends Object`
- `? super Type`, some unspecified supertype of `Type`

A wildcard is essentially an *anonymous type variable*

- Each `?` stands for some possibly-different unknown type
- Use a wildcard when you would use a type variable exactly once, so no need to give it a name
- Avoids declaring generic type variables
- Communicates to readers of your code that the type’s “identity” is not needed anywhere else
Examples

[Compare to earlier versions using explicit generic types]

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

  – More flexible than void addAll(Collection<E> c);
  – More idiomatic than (but semantically identical to)
    <T extends E> void addAll(Collection<T> c);
More examples

<T extends Comparable<T>> T max(Collection<T> c);
   – No change because T used more than once

<T> void copyTo(List<? super T> dst,
   List<? extends T> src);

Why this “works”?
   – Lower bound of T for where callee puts values
   – Upper bound of T for where callee gets values
   – Callers get the subtyping they want
     • Example: copy(numberList, integerList)
     • Example: copy(stringList, stringList)
PECS: **Producer Extends, Consumer Super**

Where should you insert wildcards?

Should you use `extends` or `super` or neither?

- Use `? extends T` when you *get* values (from a *producer*)
  - No problem if it’s a subtype
- Use `? super T` when you *put* values (into a *consumer*)
  - No problem if it’s a supertype
- Use neither (just `T`, not `?`) if you both *get* and *put*

```java
<T> void copyTo(List<? super T> dst,
    List<? extends T> src);
```
More on lower bounds

• As we’ve seen, lower-bound ? super T is useful for “consumers”

• For upper-bound ? extends T, we could always rewrite it not to use wildcards, but wildcards preferred style where they suffice

• But lower-bound is only available for wildcards in Java
  – This does not parse:
    
    ```java
    <T super Foo> void m(Bar<T> x);
    ```
  – No good reason for Java not to support such lower bounds except designers decided it wasn’t useful enough to bother
? versus Object

? indicates a particular but unknown type

```java
void printAll(List<?<> lst) {...}
```

Difference between `List<?>` and `List<Object>`:
- Can instantiate `?` with any type: `Object`, `String`, ...
- `List<Object>` is restrictive; wouldn't take a `List<String>`

Difference between `List<Foo>` and `List<?> extends Foo`:
- In latter, element type is **one** unknown subtype of `Foo`
  
  Example: `List<?> extends Animal` might store only Giraffes but not Zebras
- Former allows anything that is a subtype of `Foo` in the same list
  
  Example: `List<Animal>` could store Giraffes and Zebras
Reasoning about wildcard types

Consider all possible instantiations of the wildcard type!
Reasoning about wildcard types

Object \( o \);
Number \( n \);
Integer \( i \);
PositiveInteger \( p \);

List<? extends Integer> \( lei \);

First, which of these is legal?
\( lei = \text{new ArrayList<Object>}(()) \);
\( lei = \text{new ArrayList<Number>}(()) \);
\( lei = \text{new ArrayList<Integer>}(()) \);
\( lei = \text{new ArrayList<PositiveInteger>}(()) \);
\( lei = \text{new ArrayList<NegativeInteger>}(()) \);

Which of these is legal?
\( lei.add(o); \)
\( lei.add(n); \)
\( lei.add(i); \)
\( lei.add(p); \)
\( lei.add(null); \)
\( o = lei.get(0); \)
\( n = lei.get(0); \)
\( i = lei.get(0); \)
\( p = lei.get(0); \)
Reasoning about wildcard types

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

List<? super Integer> lsi;

First, which of these is legal?
lsi = new ArrayList<Object>;
lsi = new ArrayList<Number>;
lsi = new ArrayList<Integer>;
lsi = new ArrayList<PositiveInteger>;
lsi = new ArrayList<NegativeInteger>;

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);
A wildcard is essentially an *anonymous type variable*

- Each ? stands for some possibly-different unknown type
- Use a wildcard when you would use a type variable exactly once, so no need to give it a name

Reasoning about Wildcards
- Consider all possible instantiations of the wildcard type!
Big picture

• Last time: Generics intro
• Subtyping and Generics
• Using bounds for more flexible subtyping
• Using wildcards for more convenient bounds
• Digression: Java’s unsoundness(es)
• Java realities: type erasure
Type Unsoundness
Type systems

• Prove absence of certain run-time errors
• In Java:
  – methods/fields guaranteed to exist
    • compare to, eg, python
  – programs without casts don’t throw ClassCastException
• Type system *unsound* if it fails to provide its stated guarantees
Java arrays

We know how to use arrays:
- Declare an array holding Type elements: Type[]
- Get an element: x[i]
- Set an element x[i] = e;

Java included the syntax above because it’s common and concise.

But can reason about how it should work the same as this:

```java
class Array<T> {
    public T get(int i) { ... "magic" ... }
    public T set(T newVal, int i) { ... "magic" ... }
}
```

So: If Type1 is a subtype of Type2, how should Type1[] and Type2[] be related??
Array subtyping

• Given everything we have learned, if \texttt{Type1} is a subtype of \texttt{Type2}, then \texttt{Type1[]} and \texttt{Type2[]} should be unrelated
  – Invariant subtyping for generics
  – Because arrays are mutable

• But in Java, if \texttt{Type1} is a subtype of \texttt{Type2}, then \texttt{Type1[]} is a subtype of \texttt{Type2[]}
  – Not true subtyping: the subtype does not support setting an array index to hold a \texttt{Type2}
  – Java (and C#) made this decision in pre-generics days
    • Else cannot write reusable sorting routines, etc.
  – Backwards compatibility means it’s here to stay
Big picture

- Last time: Generics intro
- Subtyping and Generics
- Using *bounds* for more flexible subtyping
- Using *wildcards* for more convenient bounds
- Digression: Java’s *unsoundness*(es)
- Java realities: *type erasure*
Type Erasure
Type erasure

All generic types become type `Object` once compiled
- Big reason: backward compatibility with ancient byte code
- So, at run-time, all generic instantiations have the same type

```java
List<String> lst1 = new ArrayList<String>();
List<Integer> lst2 = new ArrayList<Integer>();
lst1.getClass() == lst2.getClass()  // true
```

Cannot use `instanceof` to discover a type parameter

```java
Collection<String> cs = new ArrayList<String>();
    if (cs instanceof Collection<String>) {  // illegal
        ...
    }
```
You cannot create objects or arrays of a parameterized type
(Actual type info not available at runtime)
Generics and casting

Casting to generic type results in an important warning

```java
List<Cat> cats = new ArrayList<Cat>(); // ok
List<?> mystery = cats;
List<String> ls = (List<String>) mystery; // warn
ls.add("not a cat"); // undetected error
...
Cat c = cats.remove(0); // ClassCastException
```

- Compiler gives an unchecked warning, since this is something the runtime system will not check for you
- Usually, if you think you need to do this, you're wrong

Object can also be cast to any generic type 😞

```java
public static <T> T badCast(T t, Object o) {
    return (T) o; // unchecked warning
}
```
The bottom-line

- Java guarantees a `List<String>` variable always holds a (subtype of) the *raw type* `List`
- Java does not guarantee a `List<String>` variable always has only `String` elements at run-time
  - Will be true unless unchecked casts involving generics are used
  - Compiler inserts casts to/from `Object` for generics
    - If these casts fail, hard-to-debug errors result: Often far from where conceptual mistake occurred
- Don’t ignore warnings!
  - You’re violating good style/design/subtyping/generics
  - You’re risking difficult debugging
class Node {
...

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

class Node<E> {
    ...
    @Override
    public boolean equals(Object obj) {
        if (!((obj instanceof Node<?>))) {
            return false;
        }
        Node<?> n = (Node<?>) obj;
        return this.data().equals(n.data());
    }
    ...
    Leave it to here to “do the right thing” if this and n differ on element type
    }
    }
    }
}
Summary: Type Erasure

• At runtime, Java does not know the exact types of generics
• Sort of awkward but required for backward compatibility
Wrapup
Generics clarify your code

```java
interface Map {
    Object put(Object key, Object value);
    ...
}

interface Map<Key, Value> {
    Value put(Key key, Value value);
    ...
}
```

plus casts in client code
→ possibility of run-time errors
Tips when writing a generic class

• Start by writing a concrete instantiation
  – Get it correct (testing, reasoning, etc.)
  – Consider writing a second concrete version

• Generalize it by adding type parameters
  – Think about which types are the same or different
  – The compiler will help you find errors

• As you gain experience, it will be easier to write generic code from the start
Summary

**Type bounds** e.g. `<T extends Number>`
- Make code more flexible!

**Java wildcards**
- Anonymous type variables (used only once)
  
  - `? extends Type`, some unspecified subtype of `Type`
  
  - `? super Type`, some unspecified supertype of `Type`

**Type Erasure**
- Java doesn’t know generic types at runtime
  
  - necessary for backward compatibility
Announcements
Announcements

• Quiz 5 is due tomorrow
• Homework 6 due tomorrow
• Section tomorrow!
  – Subtyping – now with worksheet!
  – HW7 (Dijkstra’s algorithm)