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*

Generics and subtyping

- Integer is a subtype of Number
- Is List<Integer> a subtype of List<Number>?
- Use subtyping rules (stronger, weaker) to find out…
List<Number> and List<Integer>

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
    – Not covariant and not contravariant
    – Neither List<Number> nor List<Integer> subtype of other

Invariance of Java’s subtyping

If Type2 and Type3 are different, then Type1<Type2> is not a subtype of Type1<Type3>

Previous example shows why:
   – Observer method prevents “one direction”
   – Mutator/producer method prevents “the other direction”

If our types have only observers or only mutators, then one direction of subtyping would be sound
   – But Java’s type system does not “notice this” so such subtyping is never allowed in Java

Read-only allows covariance

interface List<T> {
    T get(int index);
}

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

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

So covariant subtyping would be correct:
    – List<Integer> a subtype of List<Number>

But Java does not analyze interface definitions like this
    – Conservatively disallows this subtyping

Write-only allows contravariance

interface List<T> {
    boolean add(T elt);
}

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

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

So contravariant subtyping would be correct:
    – List<Number> a subtype of List<Integer>

But Java does not analyze interface definitions like this
    – Conservatively disallows this subtyping
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*

More verbose first

Now:
– 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

Then: *Java wildcards*
– 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

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

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

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

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

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

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)
More on lower bounds

• As we’ve seen, lower-bound \( ? \text{ super } T \) is useful for “consumers”

• For upper-bound \( ? \text{ sub } T \), 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:
    \[ <T \text{ super } \text{ Foo}> \text{ void } m(\text{Bar}<T> \text{ x}); \]
  – No good reason for Java not to support such lower bounds except designers decided it wasn’t useful enough to bother

 Legal operations on wildcard types

Object \( o \);
Number \( n \);
Integer \( i \);
PositiveInteger \( p \);
List\(<\text{? extends Integer}>\) \( \text{lei} \);

First, which of these is legal?
\begin{align*}
\text{lei} &= \text{new ArrayList<Object>}; \\
\text{lei} &= \text{new ArrayList<Number>}; \\
\text{lei} &= \text{new ArrayList<Integer>}; \\
\text{lei} &= \text{new ArrayList<PositiveInteger>}; \\
\text{lei} &= \text{new ArrayList<NegativeInteger>};
\end{align*}

Which of these is legal?
\begin{align*}
\text{lei}.\text{add}(o); \\
\text{lei}.\text{add}(n); \\
\text{lei}.\text{add}(i); \\
\text{lei}.\text{add}(p); \\
\text{lei}.\text{add}(null);
\end{align*}

Object \( o \);
Number \( n \);
Integer \( i \);
PositiveInteger \( p \);
List\(<\text{? super Integer}>\) \( \text{lsi} \);

First, which of these is legal?
\begin{align*}
\text{lsi} &= \text{new ArrayList<Object>}; \\
\text{lsi} &= \text{new ArrayList<Number>}; \\
\text{lsi} &= \text{new ArrayList<Integer>}; \\
\text{lsi} &= \text{new ArrayList<PositiveInteger>}; \\
\text{lsi} &= \text{new ArrayList<NegativeInteger>};
\end{align*}

Which of these is legal?
\begin{align*}
\text{lsi}.\text{add}(o); \\
\text{lsi}.\text{add}(n); \\
\text{lsi}.\text{add}(i); \\
\text{lsi}.\text{add}(p); \\
\text{lsi}.\text{add}(null);
\end{align*}
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 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

Two unsoundnesses in Java

- One well-known and intentional
  - array subtyping
- One discovered this week(!!!)
  - a subtle interaction between generic bounds and null

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 Type1 is a subtype of Type2, then Type1[] and Type2[] should be unrelated
  - Invariant subtyping for generics
  - Because arrays are mutable

- But in Java, if Type1 is a subtype of Type2, then Type1[] is a subtype of Type2[]
  - Not true subtyping: the subtype does not support setting an array index to hold a 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

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<?> cs = new ArrayList<String>();
if (cs instanceof Collection<String>) { // illegal
...
}
```
Generics and casting

Casting to generic type results in an important warning

```
List<?> lg = new ArrayList<String>();  // ok
List<String> ls = (List<String>) lg;  // warn
```

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

– Most common real need is creating arrays with generic element types (discussed shortly), when doing things like implementing `ArrayList`.

Object can also be cast to any generic type 😊

```
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

• So, two reasons not to ignore warnings:
  – You’re violating good style/design/subtyping/generics
  – You’re risking difficult debugging

Recall equals

```
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<?})) {
            return false;
        }
        Node<?> n = (Node<?>) obj;
        return this.data().equals(n.data());
    }
    ...
}

**Generics and arrays**

public class Foo<T> {
    private T aField;        // ok
    private T[] anArray;     // ok
    public Foo() {
        aField = new T();      // compile-time error
        anArray = new T[10];   // compile-time error
    }
}

You cannot create objects or arrays of a parameterized type
(Actual type info not available at runtime)
Necessary array cast

```java
public class Foo<T> {
    private T aField;
    private T[] anArray;

    @SuppressWarnings("unchecked")
    public Foo(T param) {
        aField = param;
        anArray = (T[])(new Object[10]);
    }
}
```

You can declare variables of type `T`, accept them as parameters, return them, or create arrays by casting `Object[]`
- Casting to generic types is not type-safe, so it generates a warning
- Rare to need an array of a generic type (e.g., use `ArrayList`)

Some final thoughts…

Generics clarify your code

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

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

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