

CSE 331

Software Design and Implementation

Lecture 14

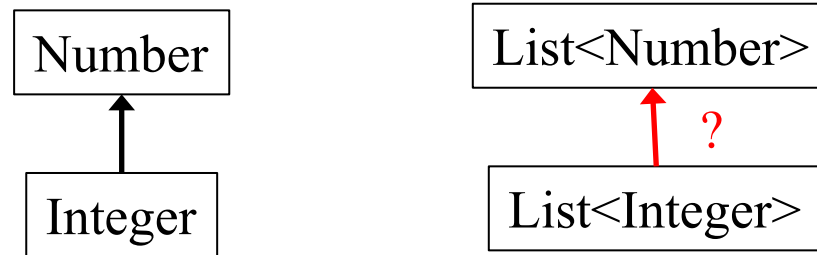
Generics 2

Zach Tatlock / Winter 2017

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*

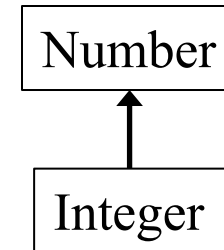
Generics and subtyping



- **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);`

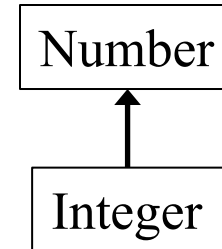
So type `List<Integer>` has:
`Integer get(int index);`

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:

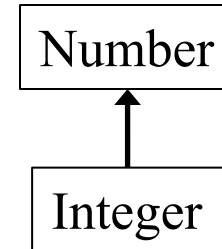
```
boolean add(Integer elt);
```

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

```
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

```
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

```
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

```
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:

```
<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:

```
<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 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:

```
<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

```
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`

Legal operations on wildcard types

```
Object o;
```

```
Number n;
```

```
Integer i;
```

```
PositiveInteger p;
```

```
List<? extends Integer> lei;
```

First, which of these is legal?

```
lei = new ArrayList<Object>();
```

```
lei = new ArrayList<Number>();
```

```
lei = new ArrayList<Integer>();
```

```
lei = new ArrayList<PositiveInteger>();
```

```
lei = 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);
```

Legal operations on 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);`~~

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

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:

```
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

```
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

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

Erasure: Type arguments do not exist at runtime

Equals for a parameterized class

```
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());  
    }  
    ...  
}
```

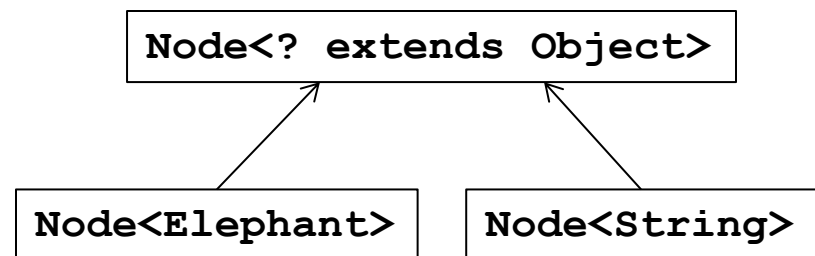
More erasure: At run time, do not know what **E** is and will not be checked, so don't indicate otherwise

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());  
    }  
    ...  
}
```

Works if the type of obj is Node<Elephant> or Node<String> or ...

Leave it to here to “do the right thing” if this and n differ on element type



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

```
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

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

plus casts in client code
→ possibility of run-time errors

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

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

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