CSE 331
Software Design and Implementation

Lecture 14
Generics⟨1⟩

Leah Perlmutter / Summer 2018

Announcements

Announcements

- Quiz 5 is due Thursday
- Homework 6 due Thursday
- Midterm grades and feedback will be out this evening

Generics

Outline (lec14 and lec15)

- Basics of generic types for classes and interfaces
- Basics of *bounding* generics
- Generic *methods* [not just using type parameters of class]
- Generics and *subtyping*
- Using *bounds* for more flexible subtyping
- Using *wildcards* for more convenient bounds
- Digression: Java's *unsoundness*(es)
- Java realities: *type erasure*

Varieties of abstraction

Abstraction over *computation*: procedures (methods)

```
int x1, y1, x2, y2;
Math.sqrt(x1*x1 + y1*y1);
Math.sqrt(x2*x2 + y2*y2);
```

Abstraction over *data*: Data structures

```
Point p1, p2;
```

Abstraction over *implementations*: Specifications

```
* @requires x >= 0
* @return square root of x
```

Abstraction over *types*: polymorphism (generics)

```
Point<Integer>, Point<Double>
```

Today!

Why we ♥ abstraction

Hide details
- Avoid distraction
- Permit details to change later

Give a *meaningful name* to a concept

Permit *reuse* in new contexts
- Avoid duplication: error-prone, confusing
- Save reimplementation effort
- Helps to "Don't Repeat Yourself"

Related abstractions

```
interface ListOfStrings {
  boolean add(String elt);
  String get(int index);
}
interface ListOfNumbers {
  boolean add(Number elt);
  Number get(int index);
}
```

Related abstractions

```
interface ListOfStrings {
  boolean add(String elt);
  String get(int index);
}
interface ListOfNumbers {
  boolean add(Number elt);
  Number get(int index);
}
```

… and many, many more

```
// Type abstraction
// abstracts over element type E
interface List<E> {
  boolean add(E n);
  E get(int index);
}
```

Type abstraction lets us use these types:

```
List<String>
List<Number>
List<Integer>
List<List<String>>
…
```

Formal parameter vs. type parameter

```
interface ListOfIntegers {
  boolean add(Integer elt);
  Integer get(int index);
}
```

- Declares a new ***variable*** **`elt`**, called a ***(formal) parameter***
- ***Instantiate*** by passing in an ***argument*** interpretable as **`Integer`**
  - *E.g.*, **`lst.add(7)`**
- Scope of **`elt`** (declared in method header) is the entire method body

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

- Declares a new ***type variable*** **`E`**, called a ***type parameter***
- ***Instantiate*** by passing in an ***argument*** interpretable as any reference type
  - *E.g.*, **`List<String>`**
- Scope of **`E`** (declared in class header) is the entire class

Scope

Declaration

Use

```
class NewSet<E> implements Set<E> {
  // rep invariant:
  //   non-null, contains no duplicates
  // …
  List<E> theRep;
  E lastItemInserted;
  …
}
```

Use

Use

Declaring and instantiating generics

Parameter Parameter

```
class MyClass<TypeVar1, …, TypeVarN> {…}
interface MyInterface<TypeVar1, …, TypeVarN> {…}
```

- Convention: Type variable has one-letter name such as: **`T`** for **`Type`**, **`E`** for **`Element`**, **`K`** for **`Key`**, **`V`** for **`Value`**, …

To instantiate a generic class/interface, client supplies *type arguments*:

```
MyClass<String, …, Date> = new MyClass<>();
```

Argument Argument

Restricting instantiations by clients

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

method parameter's type restricts which arguments can be passed in

```
interface List1<E extends Object> {…}
interface List2<E extends Number> {…}
```

type parameter's upper bound restricts which type arguments can be passed in

```
List1<Date> // OK, Date is a subtype of Object
List2<Date> // compile-time error, Date is not a
            // subtype of Number
```

Declaring and instantiating generics: syntax with bounds

```
class MyClass<TypeVar1 extends TypeBound1,
              ...,
              TypeVarN extends TypeBoundN> {…}
```

- (same for interface definitions)
- (default upper bound is `Object`)

To instantiate a generic class/interface, client supplies type arguments:

`MyClass<String, …, Date>`

- Compile-time error if type is not a subtype of the upper bound

Using type variables

Code can perform any operation permitted by the bound

- Because we know all instantiations will be subtypes!

```
class Foo1<E extends Object> {
  void m(E arg) {
    arg.asInt();  // compiler error, E might not
    ...           // support asInt()
  }
}

class Foo2<N extends Number> {
  void m(N arg) {
    arg.asInt();  // OK, since Number and its
    ...           // subtypes support asInt()
  }
}
```

More bounds

`<TypeVar extends SuperType>`

- One *upper bound*; accepts given supertype or any of its subtypes

`<TypeVar extends ClassA & InterfaceB & InterfaceC & …>`

- *Multiple* upper bounds (superclass/interfaces) with `&`
  - accepts an argument that matches **all** the bounds

`public class TreeSet<T extends Comparable<T>> {...}`

- Recursively-defined bounds
  - TreeSet accepts any type that can be compared to itself

Outline

- Basics of generic types for classes and interfaces
- Basics of *bounding* generics
- Generic *methods* [not just using type parameters of class]
- Generics and *subtyping*
- Using *bounds* for more flexible subtyping
- Using *wildcards* for more convenient bounds
- Digression: Java's *unsoundness*(es)
- Java realities: *type erasure*

Generic Methods

Generic classes are not enough

```
class Utils {
  public static double sumList(List<Number> lst) {
      double result = 0.0;
      for (Number n : lst) {
        result += n.doubleValue();
      }
      return result;
  }
  public static Object choose(List<Object> lst) {
      int i = … // random number < lst.size
      return lst.get(i);
  }
}
```

Cannot pass `List<Double>`

Cannot pass `List<Kitten>`

Reminder: `static` means "no receiver (`this` parameter)".

`List<Double>` is not a subtype of `List<Number>` !
We will see why soon.

Weaknesses of generic classes

- Would like to use **sumList** for any subtype of **Number**
  - For example, **Double** or **Integer**
  - But as we will see, **List<Double>** is not a subtype of **List<Number>**
- Would like to use **choose** for any element type
  - i.e., any subclass of **Object**
  - Want to tell clients more about return type than **Object**
- Class **Utils** is not generic, but the *methods* should be generic

Generic methods solve the problem

```
class Utils {
  public static double sumList(List<T1> lst){
      double result = 0.0;
      for (Number n : lst) { // T1 also works
        result += n.doubleValue();
      }
      return result;
  }
  public static T2 choose(List<T2> lst) {
      int i = … // random number < lst.size
      return lst.get(i);
  }
}
```

error: cannot find symbol: class T1

Need to ensure T1 subtype of Number

error: cannot find symbol: class T2

Generic methods solve the problem

```
class Utils {
  public static <T1 extends Number> double sumList(List<T1> lst){
      double result = 0.0;
      for (Number n : lst) { // T1 also works
        result += n.doubleValue();
      }
      return result;
  }
  public static <T2> T2 choose(List<T2> lst) {
      int i = … // random number < lst.size
      return lst.get(i);
  }
}
```

Generic methods solve the problem

```
class Utils {
  public static <T1 extends Number> double sumList(List<T1> lst){
      double result = 0.0;
      for (Number n : lst) { // T1 also works
        result += n.doubleValue();
      }
      return result;
  }
  public static <T2> T2 choose(List<T2> lst) {
      int i = … // random number < lst.size
      return lst.get(i);
  }
}
```

What if T1 and T2 had the same name?

Insert a type parameter declaration in the method header!

Generic methods solve the problem

```
class Utils {
  public static <T1 extends Number> double sumList(List<T1> lst){
      double result = 0.0;
      for (Number n : lst) { // T1 also works
        result += n.doubleValue();
      }
      return result;
  }
  public static <T2> T2 choose(List<T2> lst) {
      int i = … // random number < lst.size
      return lst.get(i);
  }
}
```

Scope of T1 is the body of `sumList`

Scope of T2 is the body of `choose`

Using generics in methods

- Instance methods can use type parameters of the class
- Instance methods and static methods can have their own type parameters
  - Generic methods
- Callers to generic methods need not explicitly instantiate the methods' type parameters
  - Compiler usually figures it out for you
  - *Type inference*

More examples

```
<T extends Comparable<T>> T max(Collection<T> c) {
  …
}

<T extends Comparable<T>>
void sort(List<T> list) {
   // … use list.get() and T's compareTo
}
```

(This one works, but we will make it even more useful later by adding more bounds.)

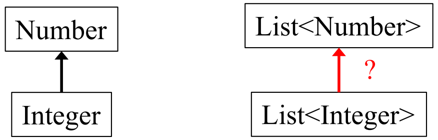
```
<T> void copyTo(List<T> dst, List<T> src) {
   for (T t : src)
      dst.add(t);
}
```

Outline

- Basics of generic types for classes and interfaces
- Basics of *bounding* generics
- Generic *methods* [not just using type parameters of class]
- Generics and *subtyping*
- Using *bounds* for more flexible subtyping
- Using *wildcards* for more convenient bounds
- Digression: Java's *unsoundness*(es)
- Java realities: *type erasure*

Generics and Subtyping

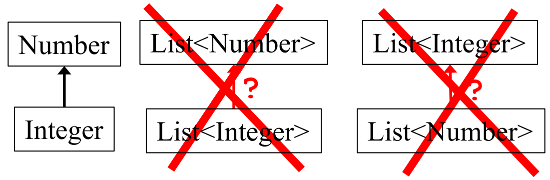
Generics and subtyping



- **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);
```

- Subtype needs stronger spec than super

- Stronger method spec has:
  - weaker precondition
  - stronger postcondition

Java subtyping is *invariant* with respect to generics

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

Generics and subtyping

If **T2** and **T3** are different types,
then for all **Foo**, **Foo<T2>** is *not* a subtype of **Foo<T3>**

Previous example shows why:

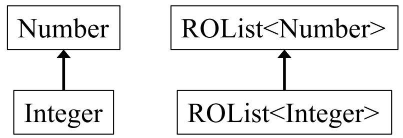
– Observer method prevents one direction
– Mutator/producer method prevents the other direction

If our types have only observers or only mutators, then one direction of subtyping would be sound

– Java's type system is not expressive enough to allow this

Read-only allows covariance (in theory)

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



Type **ReadOnlyList<Number>** has method:

```
  Number get(int index);
```

Type **ReadOnlyList<Integer>** has method:

```
  Integer get(int index);
```

- Subtype method needs:
  - weaker pre
  - stronger post

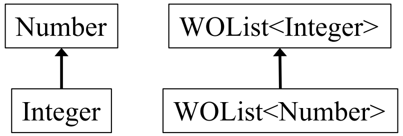
So *covariant* subtyping would be correct:

– **ROList<Integer>** is a subtype of **ROList<Number>**
– Covariant = type of **ROList<T>** changes the same way **T** changes

The Java type system conservatively disallows this subtyping

Write-only allows contravariance (in theory)

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



Type **WriteOnlyList<Number>** has method:

```
  boolean add(Number elt);
```

Type **WriteOnlyList<Integer>** has method:

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

- Subtype method needs:
  - weaker pre
  - stronger post

So *contravariant* subtyping would be correct:
- **WOList<Number>** is a subtype of **WOList<Integer>**
- Contravariant = type of **ROList<T>** changes opposite to **T**

The Java type system conservatively disallows this subtyping

Generic types and subtyping

- **List<Integer>** and **List<Number>** are not subtype-related
- Generic types can have subtyping relationships
- 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>**
  - …

Outline

- Basics of generic types for classes and interfaces
- Basics of *bounding* generics
- Generic *methods* [not just using type parameters of class]
- Generics and *subtyping*
- Using *bounds* for more flexible subtyping
- Using *wildcards* for more convenient bounds
- Digression: Java's *unsoundness*(es)
- Java realities: *type erasure*

Stay tuned for lec15!

Announcements

Announcements

- Quiz 5 is due Thursday
- Homework 6 due Thursday
- Midterm grades and feedback will be out this evening

- Thank you for coming to class today! ☺