CSE 331
Software Design and Implementation

Lecture 13
Generics 1

Zach Tatlock / Winter 2016

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*: ADTs (classes, interfaces)

```
Point p1, p2;
```

Abstraction over *types*: polymorphism (generics)

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

Today!

Why we <3 *love* <3 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"

Any true wizard knows, once you know the name of a thing you can control it.
-- Jerry Sussman

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

Lets us use types

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

// *abstracts* over element type

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

An analogous parameter

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

- Declares a new ***variable***, called a ***(formal) parameter***
- ***Instantiate*** with any ***expression*** of the right type
  - *E.g.*, **lst.add(7)**
- *Type* of **add** is *Integer → boolean*

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

- Declares a new ***type variable***, called a ***type parameter***
- ***Instantiate*** with any (reference) type
  - *E.g.*, **List<String>**
- ***"Type"*** of **List** is *Type → Type*
  - Never just use **List** (in Java for backward-compatiblity)

Type variables are types

Declaration

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

Use

Declaring and instantiating generics

```
class Name<TypeVar1, …, TypeVarN> {…}
interface Name<TypeVar1, …, TypeVarN> {…}
```

– Convention: 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:

```
Name<Type1, …, TypeN>
```

Restricting instantiations by clients

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

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

List1<Date> // OK, Date is a subtype of Object

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

Upper bounds

Revised definition

```
class Name<TypeVar1 extends Type1,
           ...,
           TypeVarN extends TypeN> {…}
```

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

To instantiate a generic class/interface, client supplies type arguments:
`Name<Type1, …, TypeN>`

- 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!
– An enforced precondition on type instantiations

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

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

More examples

```
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();
  …
}
```

Do ***NOT*** copy/paste this stuff into your project unless it is what you want
– *And* you understand it!

More bounds

`<TypeVar extends SuperType>`

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

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

- *Multiple* upper bounds (superclass/interfaces) with **`&`**

Example:

```
// tree set works for any comparable type
public class TreeSet<T extends Comparable<T>> {
    …
}
```

Where are we?

- Done:
  - Basics of generic types for classes and interfaces
  - Basics of *bounding* generics
- Now:
  - Generic *methods* [not just using type parameters of class]
  - Generics and *subtyping*
  - Using *bounds* for more flexible subtyping
  - Using *wildcards* for more convenient bounds
  - Related digression: Java's *array subtyping*
  - Java realities: type erasure
    - Unchecked casts
    - **`equals`** interactions
    - Creating generic arrays

Not all generics are for collections

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

Weaknesses

- 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`**
  - No need to restrict to subclasses of **`Number`**
  - Want to tell clients more about return type than **`Object`**
- Class **`Utils`** is not generic, but the *methods* should be generic

Much better

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

Have to declare type parameter(s)

Have to declare type parameter(s)

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 just 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
}
```

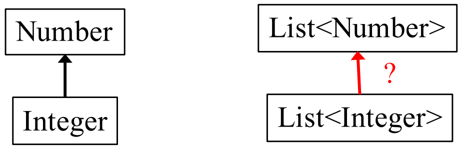
(This one "works" but will make it even more useful later by adding more bounds)

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

Where are we?

- Done:
  - Basics of generic types for classes and interfaces
  - Basics of *bounding* generics
- Now:
  - Generic *methods* [not just using type parameters of class]
  - Generics and *subtyping*
  - Using *bounds* for more flexible subtyping
  - Using *wildcards* for more convenient bounds
  - Related digression: Java's *array subtyping*
  - Java realities: type erasure
    - Unchecked casts
    - `equals` interactions
    - Creating generic arrays

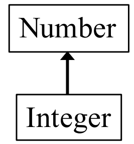
Generics and subtyping



- **`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

Hard to remember?

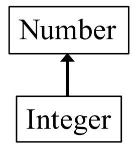
If **`Type2`** and **`Type3`** are different,
then **`Type1<Type2>`** is *not* a subtype of **`Type1<Type3>`**

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:

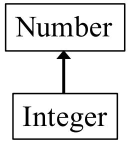
```
  Integer get(int index);
```

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

About the parameters

- So we have seen `List<Integer>` and `List<Number>` are not subtype-related
- But there is subtyping "as expected" on the generic types themselves
- 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>`
  - …

Where are we?

- Done:
  - Basics of generic types for classes and interfaces
  - Basics of *bounding* generics
- Now:
  - Generic *methods* [not just using type parameters of class]
  - Generics and *subtyping*
  - Using *bounds* for more flexible subtyping
  - Using *wildcards* for more convenient bounds
  - Related digression: Java's *array subtyping*
  - Java realities: type erasure
    - Unchecked casts
    - `equals` interactions
    - Creating generic arrays