Lecture 13
Generics

Varieties of abstraction

Abstraction over **computation**: procedures (methods)

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

Abstraction over **data**: ADTs (classes, interfaces)

```java
Point p1, p2;
```

Abstraction over **types**: polymorphism (generics)

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

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

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

```java
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 elt);
  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);
}

interface List<E> {
  boolean add(E elt);
  E 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

• 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 compatibility)

Type variables are types

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

Declaring and instantiating generics

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>
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

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!
  – 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<N,?>> {
  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 &

Recursively-defined bounds:

// TreeSet works for any type that can be compared
// to itself.
public class TreeSet<T extends Comparable<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
  - Related digression: Java’s array subtyping
- Java realities: type erasure
  - Unchecked casts
  - equals interactions
  - Creating generic arrays

Generic classes are not enough

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

Reminder: static means “no receiver (this parameter)”.

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
  - 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
Generic methods solve the problem

```java
class Utils {
    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;
    }
    static <T2> T2 choose(List<T2> lst) {
        int i = ... // random number < lst.size
        return lst.get(i);
    }
}
```

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

```java
<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.)

```java
<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
    - Related digression: Java’s array subtyping
- Java realities: type erasure
  - Unchecked casts
  - equals interactions
  - Creating generic arrays
Generics and subtyping

<table>
<thead>
<tr>
<th>Number</th>
<th>List&lt;Number&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>List&lt;Integer&gt;</td>
</tr>
</tbody>
</table>

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

```
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

---

How to remember the invariant rule

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

---

Read-only allows covariance

```
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);`

So **covariant** subtyping would be correct:
- `ReadOnlyList<Integer>` is a subtype of `ReadOnlyList<Number>`
- Covariant = type of `ReadOnlyList<T>` changes the **same way** `T` changes

The Java type system conservatively disallows this subtyping
Write-only allows contravariance

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);

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
- Related digression: Java’s array subtyping
- Java realities: type erasure
  - Unchecked casts
  - equals interactions
  - Creating generic arrays