Material on generics

JavaHyperText entry
generics

Look at lecture notes page of course website, row for this lecture, and download demo code.

Java Collections

Early versions of Java lacked generics...

interface Collection {
    /** Return true iff the collection contains ob */
    boolean contains(Object ob);

    /** Add ob to the collection; return true iff
     * the collection is changed. */
    boolean add(Object ob);

    /** Remove ob from the collection; return true iff
     * the collection is changed. */
    boolean remove(Object ob);
    ...
}

Using Java Collections

Limitation seemed especially awkward because built-in arrays do not have the same problem!

String[] a = ...
   a[0] = "Hello"
   a[1] = "World"
   ...
   for (String s : a) {
      System.out.println(s);
   }

In late 1990s, Sun Microsystems initiated a design process to add
generics to the language ...

Announcements

Late ones until Sunday night.

We tell you soon whether your A6 can be used in A7.

We make A7 available and demo it.
Arrays → Generics

Array of Strings, ArrayList of strings --- same concept with a different syntax

We should be able to do the same thing with object types generated by classes!

| Object[] oA = ... // array of objects |
| String[] sa = ... // array of Strings |
| ArrayList<Object> oA = ... // ArrayList of Objects |
| ArrayList<String> oA = ... // ArrayList of Strings |

Arrays of Strings, ArrayList of strings --- same concept with a different syntax

We should be able to do the same thing with object types generated by classes!

Proposals for adding Generics to Java

Andrew Meyers

Turing Award winner Barbara Liskov

Nate Foster

PolyJ

Pizza/GJ

LOOJ

... all based on parametric polymorphism.

Winners:

Gilad Bracha

David Stoutamire

Phil Wadler

Martin Odersky

Winner: Gilad Bracha

Reason: Proposal did not require changes to the Java Virtual Machine.

Why not? Their proposal uses type erasure. All notions of type are erased from the program. (It could look like a Python program, which doesn’t have types.) Of course, there are checks for improper casting and such.

Generic Collections

Can be a lot more complicated

```
interface Iterable<T> {
    default void forEach(Consumer<? super T> action)
}

class Arrays {
    /** Sort b according to the natural ordering. */
    static <T extends Comparable<? super T>> void parallelSort(T[] b)
    
    WOW! Who can understand THAT!!
}
```

Using Java Collections

With generics, no casts are needed...

```
Collection<String> c = ...
c.add("Hello")
c.add("World");
...
for (String s : c) {
    System.out.println(s + " : " + s.length());
}
```

… and mistakes (usually) get caught!
Type checking (at compile time)

The compiler can automatically detect uses of collections with incorrect types...

```java
// This is Demo0
Collection<String> c = ...;
c.add("Hello");  // okay */
c.add(1979);  /* Illegal: syntax error! */
```

Generally speaking, Collection<String> behaves like the parameterized type Collection<T> where all occurrences of T have been replaced by String.

Subtyping

Subtyping extends naturally to generic types.

```java
interface Collection<T> { ... }
collection<T> extends Collection<T> { ... }
collection<T> implements List<T> { ... }
```

/* The following statements are all legal. */
```java
List<String> l = new LinkedList<String>();
ArrayList<String> a = new ArrayList<String>();
collection<String> c = a;
l = a;
```

Subtyping interface
```java
Collection<T> { ... }
collection<T> extends Collection<T> { ... }
collection<T> implements List<T> { ... }
```

Subtyping class
```java
LinkedList<T> implements List<T> { ... }
ArrayList<T> implements List<T> { ... }
```

/* The following statements are all legal. */
```java
List<String> l = new LinkedList<String>();
ArrayList<String> a = new ArrayList<String>();
collection<String> c = a;
l = a;
```

Array Subtyping

Java’s type system allows the analogous rule for arrays:

```java
// This is Demo1
String[] as = new String[10];
Object[] ao = new Object[10];
ao = as;  //Type-checks: considered outdated design
ao[0]= 2110;  //Type-checks: Integer subtype Object
String s = ao[0];  //Type-checks: as is a String array
```

What happens when this code is run? TRY IT OUT!
It throws an ArrayStoreException! Because arrays are built into Java right from beginning, it could be defined to detect such errors.

Array Subtyping

Java’s type system allows the analogous rule for arrays:

```java
// This is Demo1
String[] as = new String[10];
Object[] ao = new Object[10];
ao = as;  //Type-checks: considered outdated design
ao[0]= 2110;  //Type-checks: Integer subtype Object
String s = ao[0];  //Type-checks: as is a String array
```

Is this legal? TRY IT OUT!

Subtyping

String[] is a subtype of Object[]

...is ArrayList<String> a subtype of ArrayList<Object>?

```java
// This is Demo1
ArrayList<String> ls = new ArrayList<String>();
ArrayList<Object> lo = new ArrayList<Object>();
lo = ls;  //Suppose this is legal
lo.add(2110);  //Type-checks: Integer subtype Object
String s = ls.get(0);  //Type-checks: ls is a List<String>
```

TRY IT OUT! The answer is NO. ArrayList<String> is NOT a subtype of ArrayList<Object>

A type parameter for a method

```java
Note 2
/** Replace all values x in list by y. */
public void replaceAll(List<Double> ts, Double x, Double y) {
    for (int i = 0; i < ts.size(); i++)
        if (Objects.equals(ts.get(i), x))
            ts.set(i, y);
}
```

We would like to rewrite the parameter declarations so this method can be used for ANY list, no matter the type of its elements.
Try replacing `Double` by some “Type parameter” `T`, and Java will still complain that type `T` is unknown.

```java
/** Replace all values x in list ts by y. */
public void replaceAll(List<Double> ts, Double x, Double y) {
    for (int i = 0; i < ts.size(); i++)
        if (Objects.equals(ts.get(i), x))
            ts.set(i, y);
}
```

Somehow, Java must be told that `T` is a type parameter and not a real type. Next slide says how to do this.

---

### Printing Collections

Suppose we want to write a method to print every value in a `Collection<T>`.

```java
void print(Collection<Object> c) {
    for (Object x : c) {
        System.out.println(x);
    }
}
```

```java
... Collection<Integer> c = ...;
print(c); /* Illegal: Collection<Integer> is not a * subtype of Collection<Object>! */
```

---

### Wildcards

To get around this problem, `wildcards` were added.

```java
void print(Collection<?> c) {
    for (Object x : c) {
        System.out.println(x);
    }
}
```

```java
... Collection<Integer> c = ...
c.add(42);
print(c); /* Legal! */
```

One can think of `Collection<?>` as a “Collection of some unknown type of values”.

---

### Bounded Wildcards

We can’t add values to collections whose types are wildcards ...

```java
void doIt(Collection<?> c) {
    c.add(42); /* Illegal! */
}
```

```java
... Collection<String> c = ...
doit(c); /* Illegal */
```

42 can be added to:
- `Collection<Integer>`
- `Collection<Number>`
- `Collection<Object>`

but `c` could be `Collection` of anything, not just super types of `Integer`

---

How to say that? can be a supertype of `Integer`?

42 can be added to:
- `Collection<Number>`
- `Collection<Float>`

```java
void doIt(Collection<? super Integer> c) {
    c.add(42); /* Legal! */
}
```

```java
... Collection<Object> c = ...
doIt(c); /* Legal! */
doIt(c); /* Illegal! */
doIt(c); /* Illegal! */
```

42 can be added to `Collection<Integer>` or some supertype of `Integer`, and `42` can be added to any such `Collection`.

```
/* super` is useful when you are only giving values to the object, such as putting values into a Collection.
```

Now `c` can only be a `Collection` of `Integer` or some supertype of `Integer`, and `42` can be added to any such `Collection`. 
**Bounded Wildcards**

"? extends" is useful when you are only receiving values from the object, such as getting values out of a Collection.

```java
void doIt(Collection<? extends Shape> c) {
    for (Shape s : c)
        s.draw();
}
```

---

**Generics Methods**

Here’s the printing example again. Written with a method type-parameter.

```java
<T> void print(Collection<T> c) {
    for (T x : c)
        System.out.println(x);
}
```

But wildcards are preferred when just as expressive.

---

**Catenating Lists**

The return type depends on what the input type is.

```java
/** Return the flattened version of lists. */
<T> List<T> flatten(List<? extends List<T>> lists) {
    List<T> flat = new ArrayList<T>();
    for (List<T> l : lists)
        flat.addAll(l);
    return flat;
}
```

---

**Interface Comparable**

Interface Comparable<T> declares a method for comparing one object to another.

```java
interface Comparable<T> {
    /* Return a negative number, 0, or positive number
     * depending on whether this is less than,
     * equal to, or greater than that */
    int compareTo(T that);
}
```
Our binary search

Type parameter: anything T that implements Comparable<T>

```java
/** Return h such that c[0..h] <= x < c[h+1..]. */
public static <T extends Comparable<T>> int indexOf1(List<T> c, T x) {
    int h = -1;
    int t = c.size();
    // inv: h < t && c[0..h] <= x < c[t..]
    while (h + 1 < t) {
        int e = (h + t) / 2;
        if (c.get(e).compareTo(x) <= 0)
            h = e;
        else t = e;
    }
    return h;
}
```

Those who fully grok generics write:

Type parameter: anything T that implements Comparable<T>

```java
/** Return h such that c[0..h] <= x < c[h+1..]. */
public static <T extends Comparable<? super T>> int indexOf1(List<T> c, T x) {
    int h = -1;
    int t = c.size();
    // inv: h < t && c[0..h] <= x < c[t..]
    while (h + 1 < t) {
        int e = (h + t) / 2;
        if (c.get(e).compareTo(x) <= 0)
            h = e;
        else t = e;
    }
    return h;
}
```

Sir Tony Hoare

Inside every large program is a small program struggling to get out.  
The unavoidable price of reliability is simplicity.

> There are two ways of constructing a software design.  One way is to make it so simple that there are obvious no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.  
> — Tony Hoare

Edsger W. Dijkstra

Beauty is our business.  
How do we convince people that in programming simplicity and clarity — in short, what mathematicians call elegance — are not a dispensable luxury but a crucial matter that decides between success and failure?  
Simplicity and elegance are unpopular because they require hard work and discipline to achieve and education to be appreciated.

Donald Knuth

Programs are meant to be read by humans and only incidentally for computers to execute.  
Everyday life is like programming, I guess.  If you love something you can put beauty into it.  
The best practice is inspired by theory.

> Computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty.  A programmer who sub unconsciously views himself as an artist will enjoy what he does and will do it better.  
> — Donald Knuth