Java Collections

Early versions of Java lacked generics...

```java
interface Collection {
    /* Return true if the collection contains o */
    boolean contains(Object o);
    
    /* Add o to the collection; return true if
    * the collection is changed. */
    boolean add(Object o);
    
    /* Remove o from the collection; return true if
    * the collection is changed. */
    boolean remove(Object o);
    ...
}
```

Java Collections

The lack of generics was painful when using collections, because programmers had to insert manual casts into their code...

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

... and people often made mistakes!

Using Java Collections

This limitation was especially awkward because built-in arrays do not have the same problem!

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

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

Arrays → Generics

One can think of the array “brackets” as a kind of parameterized type; a type-level function that takes one type as input and yields another type as output.

```java
Object[] a = ...
String[] a = ...
Integer[] a = ...
Button[] a = ...
```

We should be able to do the same thing with object types generated by classes!
Proposals for adding Generics to Java

With generics, the Collection interface becomes...

```java
interface Collection<T> {
    /* Return true if the collection contains x */
    boolean contains(T x);
    /* Add x to the collection; return true if the collection is changed. */
    boolean add(T x);
    /* Remove x from the collection; return true if the collection is changed. */
    boolean remove(T x);
    ...
}
```

Using Java Collections

Using generics, no casts are needed...

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

… and mistakes (usually) get caught!

With generics, no casts are needed...

```java
Collection<String> c = ...
c.add("Hello")
c.add(1979);
/* Illegal: static error! */
```

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

Static Type checking

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

```java
Collection<String> c = ...
c.add("Hello") /* Okay */
c.add(1979); /* Illegal: static error! */
```

Subtyping

Subtyping extends naturally to generic types.

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

class ArrayList<T> implements List<T> { ... }

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

String is a subtype of object so...

```java
LinkedList<String> ls = new LinkedList<String>();
Linkedlist<String> lo = new LinkedList<Object>();

ls = lo;
lo.add(2110);  // Suppose this is legal
ls.class(); /* throws exception */
```

But what would happen at run-time if we were able to actually execute this code?
Array Subtyping

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

```java
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 = as[0];  //Type-checks: as is a String array
```

What happens when this code is run?

It throws an `ArrayStoreException`!

Printing Collections

Suppose we want to write a helper 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 = ...;
p(c);  /* Illegal: Collection<Integer> is not a subtype of Collection<Object>! */
```

To get around this problem, Java’s designers added wildcards to the language:

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

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

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

Wildcards

To get around this problem, Java’s designers added wildcards to the language:

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

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

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

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

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

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

but `c` could be a Collection of anything, not just supertypes of Integer

Bounded Wildcards

Sometimes it is useful to know some information about a wildcard. Can do this by adding bounds:

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

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

```java
Collection<Circle> c = ...;
dolt(c);  /* Legal! */
```

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

“?” super is useful for when you are only giving values to the object, such as putting values into a Collection

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

```java
Collection<Circle> c = ...;
dolt(c);  /* Legal! */
```

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

“?” extends is useful for when you are only receiving values from the object, such as getting values out of a Collection
Wildcards can be nested. The following receives Collections from an Iterable and then gives floats to those Collections.

```java
void doIt(Iterable<? extends Collection<? super Float>> cs) {
    for (Collection<? super Float> c : cs)
        c.add(0.0f);
}
```

... 

```java
List<Set<Float>> l = ... 
doIt(l); /* Legal! */
```

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

```java
Iterable<Iterable<Float>> i = ... 
doIt(i); /* Illegal! */
```

```java
ArrayList<? extends Set<? super Number>> a = ... 
doIt(a); /* Legal! */
```

Returning to the printing example, another option would be to use a method-level type parameter...

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

```java
... 
collections(c); /* More explicitly: this.<Integer>print(c) */
```

But wildcards are preferred when just as expressive.

Suppose we want to concatenate a whole list of lists into one list. We want the return type to depend on what the input type is.

```java
<T> List<T> flatten(List<? extends List<T>> ls) {
    List<T> flat = new ArrayList<T>();
    for (List<T> l : ls)
        flat.addAll(l);
    return flat;
}
```

```java
... 
is = flatten(is);
ss = flatten(ss);
```

```java
Note that we are both receiving values from ts and giving values to ts, so we can't use a wildcard.
```

The Comparable<T> interface 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);
}
```

```java
Integer, Double, Character, and String are all Comparable with themselves
```

Suppose we need two parameters to have similar types.

```java
<T> void replaceAll(List<T> ts, T x, T y) {
    for (int i = 0; i < ts.size(); i++)
        if (Objects.equals(ts.get(i), x))
            ts.set(i, y);
}
```

```java
... 
```

```java
Suppose we want to look up a value in a sorted list

```java
<T extends Comparable<? super T>>//bounded type parameter
int indexOf(List<T> sorted, T x) { // no null values
    int min = 0;
    int max = l.size();
    while (min < max) {
        int guess = (min + max) / 2;
        int comparison = x.compareTo(l.get(guess));
        if (comparison < 0)
            max = guess;
        else if (comparison == 0)
            return guess;
        else
            min = guess + 1;
    }
    return -1;
}