Advanced<Java>

Generics

Teach Yourself Generics in 30 seconds

```
public class Box<T> {
  private T contents;
  public T get() { return contents; }
  public void set(T contents) { this.contents = contents; }

  public static void main(String[] args) {
    Box<String> stringBox = new Box<String>();
    stringBox.set("Overstock is hiring!!!");
    String message = stringBox.get();
    System.out.println(message);
  }
}
```

Inheritance

Should **`Box<Integer>`** extend **`Box<Number>`**?

A Box is like an array of length one,
and **`Integer[]`** extends **`Number[]`**...

Array inheritance is broken!!

There is no Santa Claus.

The Easter Bunny was shot by Elmer Fudd.

Array Inheritence

```
Integer[] ints = new Integer[1];
ints[0] = 7; // Integer.valueOf(7);
Number[] numbers = ints;
Number number = numbers[0]; // 7
numbers[0] = new BigDecimal(7);
```

Array access is *covariant*,
but array modification is *contravariant*.

Huh?

Covariance and Contravariance

Suppose `Dog` is a subclass of `Animal`.

Covariance: varies with

Any time you get an element from an array of `Animal`, you could just as well get it from an array of `Dog`.

Contravariant: varies against

Any time you wish to assign to an array of `Dog`, you could just as well assign to an array of `Animal`.

Inheritance is Neither Covariant Nor Contravariant in the Parameter Type.

A **`Box<Dog>`** is not a **`Box<Animal>`**.
A **`Box<Animal>`** is not a **`Box<Dog>`**.

Nor should they be.

A box with an animal might not contain a dog.

You should not try to put an elephant into box meant to hold a dog.(*)

** No elephants were harmed in the making of this slide.*

Thinking Outside the Box, Take 1.

```
public class Box<T> {
  private T contents;
  public T get() { return contents; }
  public void set(T contents) { this.contents = contents; }

  public void takeFrom(Box<T> box) {
    set(box.get());
  }
  public void giveTo(Box<T> box) {
    box.set(get());
  }
}

Box<Dog> crate;
Box<Animal> cage;
cage.takeFrom(crate);  // should work, but won't.
crate.giveTo(cage);  // ditto.
```

Thinking Outside the Box, Take 2.

```
public class Box<T> {
  private T contents;
  public T get() { return contents; }
  public void set(T contents) { this.contents = contents; }

  public void takeFrom(Box<? extends T> box) {
    set(box.get());
  }
  public void giveTo(Box<? super T> box) {
    box.set(get());
  }
}
```

```
Box<Dog> crate;
Box<Animal> cage;
cage.takeFrom(crate); // takeFrom is covariant
crate.giveTo(cage);  // giveTo is contravariant
```

Bounded Types

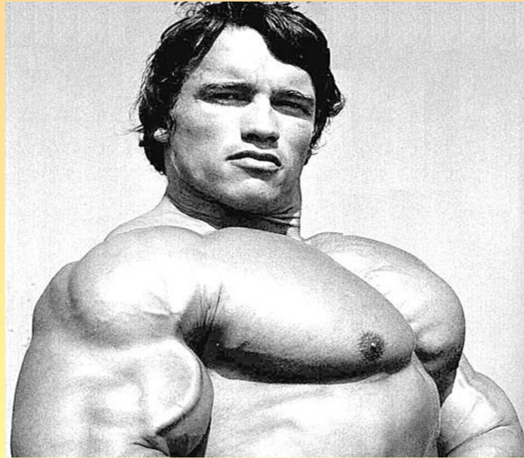
`Box<? extends T>` is a bounded type, meaning
"A **`Box`** of anything which extends **`T`**"

`Box<? super T>` is a bounded type, meaning
"A **`Box`** of anything which **`T`** extends"

As a rule of thumb, clients of generic classes should not need to worry about this sort of stuff,but authors of generic classes may need to.

When should you use which? Josh Bloch sez...

PECS



***P**roducer **E**xtends, **C**onsumer **S**uper*

If you will be using **`Foo<T>`** to *produce* an instance of type `T`, then ask for **`Foo<? extends T>`**.

If you will be using **`Foo<T>`** to *consume* an instance of type `T`, then ask for **`Foo<? super T>`**

```
public void takeFrom(Box<? extends T> box) { ... }
public void giveTo(Box<? super T> box) { ... }
```

PECS - Examples

```
package java.util;
public class Collections {
  public static <T> Set<T> unmodifiableSet(
    Set<? extends T> s) { ... }

  public static <T> void sort(
    List<T> list, // could say extends, but why?
    Comparator<? super T> c) { .. }

  public static <T> T max(
    Collection<? extends T> coll,
    Comparator<? super T> comp) { ... }
}
```

A Simple Example

```
public class SetTest {T
  public static void main(String[] args) {
    Set<Integer> numbers = new HashSet<Integer>();
    numbers.add(0); // autoboxes to Integer.valueOf(0)
    System.out.println(numbers + " contains 0? " +
      numbers.contains(0));
  }
}
```

This outputs

```
[0] contains 0? true
```

A Not So Simple Example

```
public class SetTest {T
  public static void main(String[] args) {
    Set<Long> numbers = new HashSet<Long>();
    numbers.add(0L); // autoboxes to Long.valueOf(0)
    System.out.println(numbers + " contains 0? " +
      numbers.contains(0));
  }
}
```

This outputs

```
[0] contains 0? false
```

????

What Was Sun Thinking?

```
public interface Set<E> extends Collection<E>
{
    boolean add(E o);
    boolean contains(Object o);
    boolean remove(Object o);
    ....
}
```

While there's no harm in calling `dogSet.contains(rat)`, isn't this always a sign of an error?

Why contains() Takes Object

```
Set<Animal> rabidAnimals = ...;
Set<Dog> dogsInPound = ...;

public boolean isPoundSafe() {
  for(Animal rabidAnimal: rabidAnimals) {
    if(dogsInPound.contains(rabidAnimal)) {
      return false;
    }
  }
  return true;
}
```

Bounded Type Parameters

Using bounded types allows code to call methods on variables of generic type that belong to the bound:

```
public class Benchmark<T extends Runnable> {
  public T runnable;
  public loop (int times) {
    for (int i = 0; i < times; i++) {
      runnable.run();
    }
  }
}
```

Multiply Bounded Type Parameters

```
public class RunGroup<T extends Runnable &
Comparable<T>> {
  public List<T> runnables;
  public void runAll() {
    Collections.sort(runnables);
    for (Runnable runnable: runnables) {
      runnable.run();
    }
  }
}
```

Static Type vs Dynamic Type

Recall that the *static type* of a variable or parameter is the type known to the compiler, while the *dynamic type* is the type of the variable at runtime. For example, given:

```
Number x = Integer.valueOf(30);
System.out.print(x.getClass());
```

then the static type of **`x`** is **`Number`**, while the dynamic type is **`Integer`**, so this will print **`java.lang.Integer`**.

Erasure

At runtime, Java *erases* the type arguments to the generic types of objects. Hence, given:

```
List<String> l = new ArrayList<String>();
System.out.println(l.getClass());
```

The result will be only the *raw type*, **`java.util.ArrayList`**; the type argument of **`String`** is lost.

Implications of Erasure

A type parameter cannot be used as a class constant.

```
public class Foo<T> {
  public T bar() {
    new T(); // compiler error
    T.class.newInstance(); // compiler error
  }
}
```

Implications of Erasure

Generic types cannot be used for `instanceOf` expressions:

```
List<?> list = ...;
if (list instanceOf List<String>) { // compiler error
   ...
}
```

Implications of Erasure

Generic exceptions, such as:

```
public class DataException<T> extends Exception {...}
```

are disallowed because the following cannot work:

```
try { ... }
catch(DataException<Integer> dataException) { ... }
```

What Was Sun Thinking?

While erasure wasn't necessary for backwards compatibility, the concern was that without it, libraries written with generics would not be able to interoperate with those written before generics.

This would lead to higher level libraries needing to fork different versions based on which of their lower level dependencies were generified.

Is There Any Hope?

To provided *reified types* now (i.e., to erase erasure) would be backwards incompatable:

```
public List getMessage() { // returns raw type
  List message = new ArrayList();
  message.add("Overstock");
  message.add("is hiring");
  return message;
}

public void foo() {
  // In Java 5 or 6, this gives a warning, but no error
  List<String> message = getMessage();
}
```

Short of first getting rid of all raw types, this is a non-starter

Although....

Java Language Specification, section 4.9:

The use of raw types is allowed only as a concession to compatibility of legacy code. The use of raw types in code written after the introduction of genericity into the Java programming language is strongly discouraged. **It is possible that future versions of the Java programming language will disallow the use of raw types.**

But Thread.stop() Is Still Around...

... and that was deprecated in Java 1.1

Another option, suggested by Neal Gafter, is to add another generic option:

```
class NewCollection<class E> extends Collection<E> { ... }
class NewList<class E> extends NewCollection<E>, List<E>{ ... }
```

This would allow both non-reified and reified generics to live side by side; additionally, the `NewXxx` interfaces could add additional methods.

However, it's not clear that this would not raise the same problems erasure was meant to avoid...

Erasure, Under the Covers

The following contains adult code and explicit byte code, and may not be suitable for younger viewers.
Audience discretion is advised.

Erasure, Under the Covers

Static types are not erased; all the types below can be determined at runtime via reflection:

```
public class IntegerList extends ArrayList<Integer> {
  private List<String> stringList;
  public Set<Integer> reduce(Map<String, Integer> data) {
    ...
  }
}
```

But it's a bit tricky...

```
Type type = IntegerList.class.getGenericSuperclass();
Type[] actualTypeArguments =
  ((ParameterizedType) type).getActualTypeArguments();
System.out.println(actualTypeArguments[0]);
// prints java.lang.Integer
```

Erasure, Under the Covers (cont)

Typically, a type parameter gets converted to **`java.lang.Object`**.

If a type parameter is bounded, it gets converted to its left most bound.

```
public class Amount<T extends Number> {
  public T getValue() { ... }
}
```

The actual signature for getValue will be:

```
  public Number getValue()
```

Multiple Type Bounds, Under the Covers

```
public <T extends List<?> & Comparable<T>>
void foo(T t) {
   t.size();  // method on List
   t.compareTo(t); // method on Comparable
}
```

javap says:

```
public void foo(java.util.List);
 Code:
aload_1
invokeinterface ; //List.size:()I
pop
aload_1
checkcast       ; //class java/lang/Comparable
invokeinterface ; //Comparable.compareTo:(LObject;)I
```

Class Is Now Generic!

```
public final class Class<T> {
  public T newInstance() throws ... { ... }

  public Constructor<T> getConstructor(
    Class<?>... parameterTypes) throws ... { ... }

  public T[] getEnumConstants() { ... }

  public T cast(Object obj) {
    if (obj != null && !isInstance(obj))
        throw new ClassCastException();
    return (T) obj; // unchecked cast, but it's safe
  }
}
```

The cast method not only avoids compiler warnings, but can detect cast errors sooner rather than later.

Generic Methods

```
public class Collections {
   public static final <T> List<T> emptyList() { ... }
}
```

Unlike with constructors, types can be inferred for generic method calls:

```
List<Integer> = Collections.emptyList();
```

... but not always:

```
public void foo(List<Integer> list) {}
public void bar() {
  foo(Collections.emptyList()); //compiler error
  foo(Collections.<Integer>emptyList()); //works
}
```

Why Should You Care?

It couldn't hurt to know more...

... to deliver that next great job

... to deliver that next great job

References

- *Effective Java, Second Edition* – Joshua Bloch
- http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html
- http://gafter.blogspot.com/2006/11/reified-generics-for-java.html
- http://gafter.blogspot.com/2004/09/puzzling-through-erasure-answer.html