# COMP6700/2140 Generic Types

**Alexei B Khorev and Josh Milthorpe**

Research School of Computer Science, ANU

24 March 2017

## Generic Programming

Often we want to write code that works with objects of different types, without specifying the particular types involved. How can we do this without sacrificing the safety of compile-time type checking?

Use a *generic type* – a type that is parameterized over other types e.g. `ArrayList<T>`

Prior to Java 5, the only generic type was *Array*, "[ ]", where the type of its elements could regarded as the parameter-type: `T[] "=" Array<T>`.

Other data structures (`Vector`, `List`, `Hashtable`, …) allowed objects of any type, but there was no way to declare (or ensure) that a data structure contained type-*homogeneous* elements. Even if all your elements in a Vector were of a single type e.g. String, when you extracted them, the explicit cast was required, since the return type of the extracted element was ‘Object‘:

```
Vector v = // ...
String str = (String) v.elementAt(i);
```

As well as cluttering the code, the cast operation may result in a run-time error if a non-String element was previously inserted.

## Parameterized Types

Java 5 introduced *generic types*. A generic type is a class or interface which is defined with other classes or interfaces as *parameters*. The actual type values are provided during instantiation (similar to method's invocation with actual parameters).

```
class A<T> { // more than one type parameter can be used, class A<E,K>
    private T field;
    public A (T param) {
        this.field = param;
    }
    public T getField() {
        return this.field;
    }
}
```

Instances of a generic class are defined with the type parameter given a value — an existing type:

```
A<String> aS = new A<String>("I get it!");
A<Integer> aI = new A<Integer>(100); // autoboxing: 100 -> (Integer)100
```

Type parameters are not types themselves! There is no `class T` defined anywhere in the API. The type parameter `T` is **not** a part of the class *A* name (compiler removes generic information from the class definition — so called *type erasure* — and `A.java` → `A.class`). Examples: Cell.java, ClassTest.java.

## Generics: Naming Conventions

Naming guidelines: type parameter names are single, uppercase letters, which is quite different from the variable naming convention — the difference between type variable and an ordinary class or interface name should be very clear. The most commonly used type parameter names:

- `E` — Element (in the collection type DS)
- `K` — Key (in *Hashtable* and the like)
- `V` — Value (in *Hashtable* and the like, `Hashtable<K,V>`)
- `N` — Number (of *Number* type)
- `T,S,U,V` — Type (anything, really)

**Syntax enhancement in JDK 7**

Instantiating a generic container is now easier. Instead of

```
Map<String, List<Trade>> trades = new TreeMap<String, List<Trade>> ();
```

use the *diamond operator* <>:

```
Map<String, List<Trade>> trades = new TreeMap<>();
```

and compiler will select the right type values from the left side.

## Generic constructors and methods

Type parameters can also be used within method and constructor signatures to create *generic methods* and *generic constructors*. Similar to declaring a generic type, but the type parameter's scope is limited to the method or constructor in which it's declared.

```java
public class BoxG<T> { //"borrowed" from Java Tutorial
    private T t;
    public void add(T t) { this.t = t; }
    public T get() { return t; }

    public <U> void inspect(U u){ // a method with a parameterized parameter type
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }
    public static void main(String[] args) {
        BoxG<Integer> integerBox = new BoxG<Integer>();
        integerBox.add(new Integer(10));
        integerBox.inspect("some text");
    }
}
```

The output (BoxG.java):

```
T: java.lang.Integer
U: java.lang.String
```

## Generic Static Methods

*Static* methods may also be parameterized:

```
public static <U> void fillBoxes(U u, ArrayList<Box<U>> boxes) {
    for (Box<U> box : boxes) {
        box.add(u);
    } // assuming that this is a part of a non-generic class Box definition
}
```

The type parameters are included in the method call:

```
Fruit pineapple = new Fruit("pineapple", 3.0); // creating a 3kg heavy pineapple
ArrayList<Box<Fruit>> fruitBoxes = new ArrayList<Box<Fruit>>();
Box.<Fruit>fillBoxes(pineapple, fruitBoxes);
```

Compiler can *infer* the type value of the parameter itself, so the explicit declaration isn't often necessary:

```
Box.fillBoxes(pineapple, fruitBoxes); // compiler infers that U is Fruit.
```

## Bounded Type Parameters, Sub-typing and Wildcards

A generic class definition may involve manipulations which do not make sense for every type, e.g. addition of two instances of the parameter type. In such case, one would like to constrain the type parameter values used in instantiation.

Such a constraint is declared via *bounded type parameters*. (Example: BoxBT.java):

```
public <U extends Number> void inspect(U u){ ... } // U is (sub-)type of Number
class CMP<T extends Number & Comparable<T>> { ... } // T is also comparable
```

With such a bound, the invocation `integerBox.inspect("some text")` is illegal because the value of the type parameter `U` is `String` which does not extend `Number`.

Generic classes may also have bounded type parameters.

Sub-typing of parameterized types: `Integer` is a subtype of `Number`, but `Box<Integer>` is **not** a subtype of `Box<Number>` (see diagram on the a few slides below). The method `doSomething(Box<Number> n)` will not accept a parameter of type `Box<Integer>`. The right way to declare such methods is to use *wildcards* (demo in BoxWC.java):

```
public void doSomething(Box<? extends Number> box) {... // upper bound wildcard
public void doSomething(Box<? super Integer> box) {... // lower bound wildcard
```

## Wildcard Rules

The theory behind sub- and super-typing can be quite confusing. Some attempts to explain it in an accessible manner may result in dissatisfaction and feeling of being intellectually insulted — as, for example, the *old* version of the Java Tutorial. Generics section with lions, birds and cages "explanations" (no longer offered online).

At the first encounter, it is better to postpone attempts to develop the full understanding of how and why this works, and to accept the following principles and apply them without much deliberation:

The **Get and Put Principle**

- use an `extends` wildcard when you only get values out of a structure,
- use a `super` wildcard when you only put values into a structure,
- and don't use a wildcard when you both get and put.

Another way to put it: **PECS**: *producer* `extends`, *consumer* `super`.

- Producer: `<? extends T>`
- Consumer: `<? super T>`

Example: GenMethods.java.

## Arrays and Generic Lists

The temptation to use both arrays and generic lists in the same code should be avoided. Try to use only one data structure: when performance is the ultimate goal, use arrays (but be aware of the cost of copying if there will be many add-remove operations); if you primarily need flexibility, use generic lists and choose their implementation carefully (`ArrayList` or `LinkedList`).

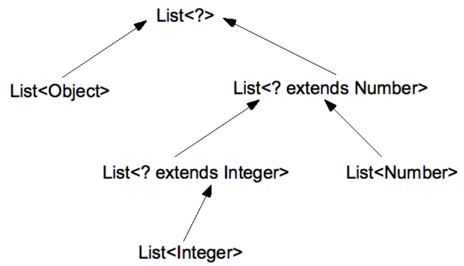
The reason arrays and generic lists mix badly together is their very different behaviour during compilation and run-time: arrays provide run-time type safety but not compile-time type safety and vice versa for generics.

- Arrays are *reified* and *covariant* — they *retain* and enforce their type information, and their subtyping behaviour is the same as that of their elements: if `Sub` is a subtype of `Super`, then `Sub[]` is a *subtype* of `Super[]`; this allows to treat arrays like “ordinary” types.
- Generic Lists are *erased* and *invariant* — compilation *removes any information about the type parameter*. Even if `T1 extends T2`, `List<T1>` is *not a subclass* of `List<T2>`.

For these reasons, one cannot create an array of a generic type, a parameterized type, or a type parameter. All these array creations are invalid: `new List<E>[]`, `new List<String>[]` and `new E[]`. All will result in generic array creation errors at compile time (because it is not type safe).

## Taxonomy of Generic Types

`java.lang.Number` is the parent to all wrapper classes of primitives except `bool` and `char`. Although `Integer` is a subclass of `Number`, `List<Integer>` is not a subclass of `List<Number>`:



Taxonomy of the generic *List* types

## Use of Wild Cards

A generic method to sum the elements of a generic *List*

```
static double sum(List<? extends Number> list ) {
    double sum = 0.0;
    for (Number n : list)
        sum += n.doubleValue();
    return sum;
}
```

The wildcard “?” indicates that the method `sum` requires a *List* of any subtype of *Number* (including itself) — the *bounded wildcard* with an *upper bound*.

A bounded wildcard with a *lower bound* `List<? super Integer>` matches any super-type of `List<Integer>` (including itself):

1. `List<Integer>`
2. `List<Number>`
3. `List<Serializable>`
4. `List<Comparable<Integer>>`
5. `List<Object>`

## Generics Glossary of Terms

(from Joshua Bloch's *Effective Java*, 2ed)

| **Term** | **Example** |
|---|---|
| Parametrised Type | `List<String>` |
| Actual Type Parameter | `String` |
| Generic Type | `List<E>` |
| Formal Type Parameter | `E` |
| Unbounded Wildcard Type | `List<?>` |
| Raw Type | `List` |
| Bounded Type Parameter | `<E extends Number>` |
| Recursive type bound | `<T extends Comparable<T>>` |
| Bounded wildcard type | `List<? extends Number>` |
| Generic Method | `static <E> List<E> asList(E[] a)` |
| Type Token | `String.class` |

## The Dream of Java... Denied?

**Back in 1995... "The Feel of Java"**

> *Java is a blue collar language. It's not PhD thesis material but a language for a job. Java feels very familiar to many different programmers because we preferred tried-and-tested things (James Gosling "Feel of Java", Talk at* OOPSLA 1996*).*

**After Generics became part of the language**

1. "I am completely and totally humbled. Laid low. I realise now that I am simply not smart at all. I made the mistake of thinking that I could understand generics. I simply cannot. I just can't. This is really depressing. It is the first time that I've ever not been able to understand something related to computers, in any domain, anywhere, period."
2. "I'm the lead architect here, have a PhD in physics, and have been working daily in Java for 10 years and know it pretty well. The other guy is a very senior enterprise developer (wrote an email system that sends 600 million emails/year with almost no maintenance). If we can't get [generics], it's highly unlikely that the 'average' developer will ever in our lifetimes be able to figure this stuff out." (both citations by Josh Bloch at *JavaPolis 2007*.)

If you want to become a Java generics expert, read Angelika Langer's 427-page (!) *Java Generics FAQ* (also there is a book "Java Generics and Collections" by Maurice Naftalin and Philip Wadler, O'Reilly 2006). But first ask yourself: "Can I go to `C++` or Scala instead?"

## Generics Intimidation

**Cay Hortsmann** (interview *Java Champion*, Feb 2008)

When students first see the API with thousands of classes, they despair. I used to be able to tell them, “That’s OK, at least the language itself is very simple.” But that was before this:

```
static <T extends Object & Comparable<? super T>> T
                      Collections.max(Collection<? extends T> coll);
```

As a student, you need to stay within a safe subset of the Java language and the API so that you can use it as a tool to learn some good computer science.

**Joshua Bloch** (talk at *JavaPolis2007*): “We simply cannot afford another *wildcards*”.

“Typical” declarations and compiler errors and warnings involving generics:

```
Enum<E extends Enum<E>> { ... };
<T extends Object & Comparable<? super T>>
              T Collections.max(Collection<? extends T> col) ;
public <V extends Wrapper<? extends Comparable<T>>>
              Comparator<V> comparator() { ... };
error: equalTo(Box<capture of ?>) in Box<capture of ?> cannot
              be applied to (Box<capture of ?>)
    equal = unknownBox.equalTo(unknownBox)
Arrays.asList(String.class, Integer.class) // Warning!
```

## Further Reading

- Horstmann *Core Java for the Impatient*, Ch. 6.1, 6.2, 6.3$^{(*)}$, 6.4$^{(*)}$, 6.5$^{(*)}$, 6.6
- Oracle *The Java Tutorials*: Generics
- The follow up “extra-curricular” Lecture on Generic Methods