# Java Generics

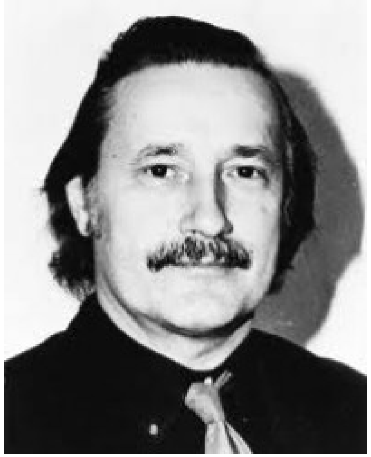
- Mark Perry
- maperry78@yahoo.com.au

# Introduction

- Generics is a type of polymorphism
- Polymorphism definition:
  - Feature that allows values of different data types to be handled using a uniform interface
- Types of polymorphism
  - Ad-hoc (overloading)
  - Parametric (generics, templates)
  - Subtype (inheritance)

# History

- Polymorphism described by Christopher Strachey (67)



- Parametric Polymorphism (ML 76)
- Ada Generics (83)
- Eiffel Generics (86)
- C++ Templates Proposal (88, C++ released 85)
- Java Generics (04, J2SE 5.0)
  - Based on GJ and Pizza (Bracha, Odersky(Scala), Wadler (Haskell) and Stoutamire)

# Terminology

| Term | Example |
|---|---|
| Parameterised Type | List<String> |
| Actual Type Parameter | String |
| Generic Type | List<E> |
| Formal Type Parameter | E |
| Unbound 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 |

# Effective Java

- Advice from Effective Java (second edition, 2008) by Joshua Bloch
- Distinguished Sun Engineer, Google Chief Java Architect (2004)

# Avoid Raw Types

- Raw types are unsafe, parameterised types are safe

```
// Uses raw type (List) - fails at runtime!
public static void main(String[] args) {
   List<String> strings = new ArrayList<String>();
   unsafeAdd(strings, new Integer(42));
   // Runtime cast (type erasure)
   String s = strings.get(0);
}

private static void unsafeAdd(List list, Object o) {
   list.add(o);
}

Test.java:10: warning: unchecked call to add(E) in raw
type List
list.add(o);
        ^
```

# Minor Exceptions

- Must use raw types (includes primitives) in class literals
    - List<String>.class and List<?>.class illegal
    - List.class, String[].class and int.class are legal
- Instanceof
    - Runtime type erasure => can't use on parameterised types other than unbound wildcard

```
// Legitimate use of raw type - instanceof
operator
if (o instanceof Set) { // Raw type
   Set<?> m = (Set<?>) o; // Wildcard type
...
```

# Eliminate Unchecked Warnings

- If you can’t eliminate, prove code is typesafe, then suppress the warning
    - @SuppressWarnings("unchecked") annotation
    - Don't miss real problems amongst noise
    - Use smallest scope
    - Comment raionale why it's safe to suppress

# Prefer Lists to Arrays

- Arrays are covariant
- Generics are invariant

```
// Fails at runtime!
Object[] objectArray = new Long[1];
objectArray[0] = "I don't fit in"; // Throws ArrayStoreException
```

- Arrays are reified, generics have type erasure
- Don't mix well
  - Illegal => new List<E>[], new List<String>[], new E[]
  - Generic array creation error at compile time

# Generic Array Creation

```
// Why generic array creation is illegal - won't compile!
List<String>[] stringLists = new List<String>[1]; // (1)
List<Integer> intList = Arrays.asList(42); // (2)
Object[] objects = stringLists; // (3)
objects[0] = intList; // (4)
String s = stringLists[0].get(0); // (5)
```

- Line 3 – legal as arrays are covariant
- Line 4 – Legal as both runtime types are List
  - Now have List<Integer> in List<String>
- Line 5 – ClassCastException
- Line 1 (generic array creation) is compile error

- Generic types can't return array
- Varargs creates array to hold varargs parameters – produces warning
- Use List<T> instead of T[]
  - Better type safety and interoperability
  - Mixing them gives compile time errors or warnings

# Favour Generic Types

- Bounded type parameter?

# Favour Generic Methods

- Static utility methods

```
// Generic method
public static <E> Set<E> union(Set<E> s1, Set<E> s2) {
    Set<E> result = new HashSet<E>(s1);
    result.addAll(s2);
    return result;
}

…
// note type inference in asList and union
Set<String> guys = new HashSet<String>(
Arrays.asList("Tom", "Dick", "Harry"));
Set<String> stooges = new HashSet<String>(
Arrays.asList("Larry", "Moe", "Curly"));
Set<String> aflCio = union(guys, stooges);
```

# Generic Type Inference

- Generic methods don't need type parameter like generic constructors

```
// Generic static factory method
public static <K,V> HashMap<K,V> newHashMap() {
    return new HashMap<K,V>();
}

// Parameterized type instance creation with static factory
Map<String, List<String>> anagrams = newHashMap();
// generic constructor
Map<String, List<String>> anagrams = new HashMap<String,
ArrayList<String>>();
```

- Collections – emptySet and reverseOrder

# Recursive Type Bound

- Type parameter T bounded by expression involving T
  - Usually Comparable

```
public interface Comparable<T> {
        int compareTo(T o);
}

// Using a recursive type bound to express mutual comparability
public static <T extends Comparable<T>> T max(List<T> list) {...}
```

# Use Bounded Wildcards in APIs

- Invariant typing needs more flexibility

```
// pushAll method without wildcard type - deficient!
public void pushAll(Iterable<E> src) {
        for (E e : src)
                push(e);
}
```

- Can't push Integer using Stack<Number>
- Use bounded wildcard type

# Bounded wildcard use

- Producer

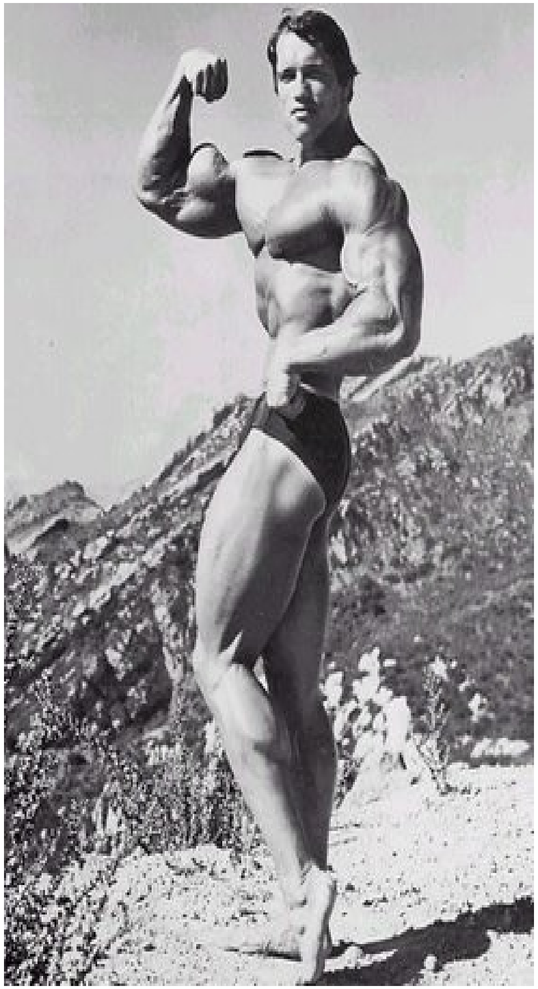
```
// Wildcard type for parameter that serves as an E producer
public void pushAll(Iterable<? extends E> src) {
    for (E e : src)
        push(e);
}
```

- Consumer

```
// Wildcard type for parameter that serves as an E consumer
public void popAll(Collection<? super E> dst) {
    while (!isEmpty()) {
        dst.add(pop());
    }
}
```

# PECS

- Produces Extends
- Consumer Super
- Use on input parameters that represent producers/consumers
- Don't use wildcard return types



# Wildcard Complexity

- Client code should not concern itself with wildcard types
- Type inference is complex (16 pages of language spec)
  - Not always intuitive

```
Set<Integer> integers = ... ;
Set<Double> doubles = ... ;
Set<Number> numbers = union(integers, doubles);

Union.java:14: incompatible types
found : Set<Number & Comparable<? extends Number &
Comparable<?>>>
required: Set<Number>
Set<Number> numbers = union(integers, doubles);
                          ^
```

# Explicit Type Parameter

## Complex Example

```
// explicit type parameter
Set<Number> numbers = Union.<Number>union(integers, doubles);
// original max signature
public static <T extends Comparable<T>> T max(List<T> list);
// revised signature using wildcard types, PECS applied twice
public static <T extends Comparable<? super T>> T max(List<? extends T> list);
```

## Comparable and Comparator are always consumers – use super

```
// consider
List<ScheduledFuture<?>> scheduledFutures = ... ;
// ScheduledFuture subinterfaces Delayed which extends Comparable<Delayed>
```

# Type Parameter/Wildcard Duality

- Consider:

```
// Two possible declarations for the swap method
public static <E> void swap(List<E> list, int i, int j);
public static void swap(List<?> list, int i, int j);
```

- Second is better
- If type parameter appears once in declaration, use wildcard
  - Unbounded type => unbounded wildcard
  - Bounded type => bounded wildcard

# Conjunctive Types

- Little known part of spec
  - Types separated by “&” for generic bounded type
- Use when parameter type must implement multiple interfaces

```
public <T extends MyInterface & Comparable<T> & Serializable>
    void doSomething(T onObject) {...}
```

# Heterogeneous Containers

- Normally we generify the container
- Databases have arbitary columns, generify key instead of container
- As of 1.5, type literal is Class<T>, not Class
  - e.g. String.class has type Class<String>
- Called Type Token

# Heterogenous Container Example

```
// Typesafe heterogeneous container pattern - API
public class Favorites {
    public <T> void putFavorite(Class<T> type, T instance);
    public <T> T getFavorite(Class<T> type);
}
// Typesafe heterogeneous container pattern - client
Favorites f = new Favorites();
f.putFavorite(String.class, "Java");
f.putFavorite(Integer.class, 0xcafebabe);
f.putFavorite(Class.class, Favorites.class);
// get values
String favoriteString = f.getFavorite(String.class);
int favoriteInteger = f.getFavorite(Integer.class);
Class<?> favoriteClass = f.getFavorite(Class.class);
```

# Heterogenous Implementation

- Suprisingly typesafe!
- Keys are of different types - heterogenous

```
// Typesafe heterogeneous container pattern - implementation
public class Favorites {
    private Map<Class<?>, Object> favorites = new HashMap<Class<?>, Object>();
    public <T> void putFavorite(Class<T> type, T instance) {
        if (type == null)
            throw new NullPointerException("Type is null");
        favorites.put(type, instance);
    }
    public <T> T getFavorite(Class<T> type) {
        return type.cast(favorites.get(type));
    }
}
```

# Heterogenous Subtlety

- Uses Map<Class<?>, Object>
- Wildcard type nested, applies not to map, but key
- Means every key can have different type
- Value type (Object) does not guarantee relationship between keys and values
  - Take advantage of Java's weak type system
- Method getFavourite uses dynamic cast

# Malicious Heterogenous Clients

- First limitation
  - Can corrupt using raw Class form, would generate warning
  - Can do same with String into HashSet<Integer>

```
// Achieving runtime type safety with a dynamic cast
public <T> void putFavorite(Class<T> type, T instance) {
        favorites.put(type, type.cast(instance));
}
```

  - See Collection checkedMap, checkedList, etc.

- Second limitation
  - Cannot be used on non-reifiable type (generics)
  - List<String> fails because List<String>.class is an error
  - Technique called Super Type Tokens can address this but has limitations [Gafter 07]
- May need to use bound type parameter or wildcard
  - Used extensively in annotations API
  - Often uses Object.`asSubclass -` casts the `Class` object on which it's called to subclass of argument, on failure throws `ClassCastException`.

# Summary

- Generics has lots of annoyances and gotchas
- Demonstrate broken (weak?) type system
- Java generics improves type safety