Java Generics

Jonathan Windle

University of East Anglia

J.Windle@uea.ac.uk

May 26, 2017

Overview I

Overview II

- Wildcard use case
- Random Generic things

4 Summary

- What we should know...?

Typing Data Structures/Raw Types

```
ArrayList arr = new ArrayList();
```

- By default ArrayList is designed to store Object references.
- This means anything can be added except primitive types.
- ArrayList is **completely generic** in that it can store anything. It is said to be a Raw Type.

```
arr.add("Boop");
arr.add(8004);
String str = (String) arr.get(0);
```

- arr is raw type, when getting something from the arr it is necessary to cast as get returns Object

Issues with Raw Types

- Completely dependant on the user to correctly use structure.
- If the user misuses the casting and cast the wrong type, an Exception will be thrown. This is detected at run time.

Generics

- Enforces a data structure to only contains `Objects` of a certain type using Generic syntax.
- Removes the need for casting, ensuring type safety.
- Any potential casting errors are detected at Compile time.
- <> Diamond operator means type is ensured from the left hand side of the assignment, not the same as assigning a raw type.
- Interfaces are commonly Generic, such as Comparable and Comparator. This means there is no need to cast and interfaces can be used freely.

Basics Summary

- Means of enforcing type safety on data structures without defining multiple classes for each type.
- Allow for early error detection at compile time.
- Removes need for casting.

Compiling Generic Code

- Two possible strategies:
    - Create a new class for every different type used (code specialisation) - C++ not Java.
    - Use one general class and determine types at runtime (code sharing).

Code Specialisation - C++

- Compiler generates a new representation for every instantiation of a generic type or method
- At compile time:
  1. Form a list of all types of the data structure defined in the code.
  2. Create a new class of that data structure and compile seperately.
- Benefits:
  - No impact on runtime performance.
  - Easy to optimize compilation.
- Problems:
  - You need to know at compile time all possible types.

Code Sharing: Type Erasure - Java

- Compiler generates code for only one representation of a generic type, by erasing the Generic type and replacing with `Object`.
- At compile time:
  1. All types are stripped from a generic and compiled as a raw type.
  2. Type checks and casts are automatically added. These are performed at runtime.
- Benefit:
  - No need to create extra files which may not be needed.
- Problem:
  - Extra type checking takes time, slower execution.

Simple Generic Data Structures

- `<E>` `E` represents the enforced type chosen.
- Can still instantiate a raw type of any generic.
- `<K,V,E,S>` You can have as many types as you want and use any valid identifier.

Generics in nested classes

- Nested classes can have the same generic type as outer class, due to always being associated with an instance of the outer class.

```
public class Pair<K,V> {
        public class Inner {
        K in1; // K is same type as outer class
        V in2; // V is same type as outer class
    }
}
```

- Static nested classes **cannot** refer to generic type of enclosing class.

```
public class Pair<K,V> {
        public static class Inner {
    // Cannot reference either K or V due to static instance
    }
        public static class Inner<K,V> {
    // Type can be set independently to the outer object
        K in1; // K is same label, could be different Type
        V in2; // V is same label, could be different Type
    }
}
```

Enforcing Generic Restrictions

- The type of a generic can be typed using the extends keyword.
- `<T extends Number>` Means you can only type the generic to `Number` or a subclass of `Number`.
- Making generics comparable:
- NOT ALLOWED - without enforcing restriction

```
public class Pair<K,V> implements Comparable<K,V>> {
        private K key;
    private V value;
    public int compareTo(Pair<K,V> other) {
        return key.compareTo(other.key); // NOT ALLOWED Key not necessarily comparable
    }
}
```

- ALLOWED: - with enforcing using `extends Comparable`

```
public class Pair<K,V> implements Comparable<K extends Comparable,V>> {
        private K key;
    private V value;
    public int compareTo(Pair<K,V> other) {
        return key.compareTo(other.key); // K Has to be Comparable, this is allowed
    }
}
```

Enforcing Generic Restrictions - Cont.

- Type Erasure replaces the generic type with the least specific restriction.
- If requirement is several interfaces, they can be enforced by & e.g. `<T extends Comparable<T> & Cloneable>`

Generic Methods

- Work in a very similar way to classes. Type scope is limited to that method only though.
- Don't have to explicitly pass the type arguments, it's inferred from arguments passed.
- `insertionSort()` is a generic method:

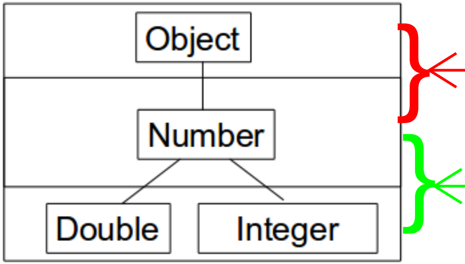
```
String[] sa = new String[10];
insertionSort(sa); // Infers String type

Integer[] in = new Integer[50];
insertionSort(in); // Infers integer type
```

Wildcards

- Three types of wildcard:
  - ?: denotes set of all types
  - ? extends Foo Denotes a family of subtypes of type Foo
  - ? super Bar Denotes a family of supertypes of type Bar
- The main use of wildcards is to overcome the problem with inheritance and generics.

Wildcards - Cont.

```java
Wrapper<?> raw; // Unbounded, any type.
Wrapper<? super Number> up; // Number or any superclass
Wrapper<? extends Number> down; // Number or any subclass
up = new Wrapper<Object>(); // This is OK
down = new Wrapper<Double>(); // This is also OK.
```



- In this case, up can be in the top half or Number.
- down can be anything in the middle or lower half.

Why we need wildcards

- There is no inheritance between generics of different types.
- You can store subclasses in in a typed structure e.g.:

```java
LinkedList<Number> m2 = new LinkedList<>();
m2.add(new Integer(11)); // Is allowed
m2.add(new Double(2.5)); // Is allowed
```

- You cannot store generic subclasses e.g.:

```java
LinkedList<Number> m = new LinkedList<Integer>(); // Not allowed
LinkedList<Number> m = new LinkedList<Double>(); // Not allowed
```

- `LinkedList<Number>` is not a superclass of `LinkedList<Double>`.
- Type erasure does not allow for this. Generic collections are invariant

Wildcard use case

- This will not work due to no inheritance between generics.
- `ArrayList<String>` is not the same as `ArrayList<Object>`

```java
// This can only be used for Object arrays.
static void printList(ArrayList<Object> list) {
        for (Object elem : list) {
                System.out.println(elem);
        }
}
ArrayList<String> strArr = new ArrayList<>();
strArr.add("Boop");
printList(strArr); // This won't compile.
```

- With the use of wildcard ? this will work with any `ArrayList` and keep Type safety

```java
// This can be used with ANY ArrayList
static void printList(ArrayList<?> list) {
        for (Object elem : list) {
                System.out.println(elem);
        }
}
ArrayList<String> strArr = new ArrayList<>();
ArrayList<Card> cardArr = new ArrayList<>();
printList(strArr); // This is OK
printList(cardArr); // This is OK
```

Generic Arrays don't work

- Array's have a dynamic type, i.e. `Object[]` can store `Integer` references, but type erasure does not use it.
- Solution is to cast Generic arrays to `Object[]` or use `ArrayList`.

Summary

- Generics are a way of enforcing a type on a data structure.
- Errors can be found at compile time
- Restrictions at Class level can be put in place using `extends`.
- Restrictions on a type can be removed at the Object level using `wildcards`.

What we should know...?

- Benefits of generics
- Differences between code sharing (Type Erasure) and Code specialisation and which language does what.
- Understanding of restrictions that can be put on.
- Methods can be generic so they can be used with generic classes.

The End