Programmieren II

Generics

Alexander Fraser

fraser@cl.uni-heidelberg.de

(Based on material from Jedi, Jeff Meister and T. Bögel)

June 11, 2014

Outline

Outline

Outline

Traversing collections

A) Traversing collections with `for-each`

```
for (Object o : collection)
    System.out.println(o);
```

B) Using `Iterators`

- Iterators allow traversing trough collections
- Each collection provides an iterator with the `.iterator()` method

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

- `Iterator.remove()`: *modify the collection* ***during iteration***

Iterator example: filtering a list

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext(); )
        if (!cond(it.next()))
            it.remove();
}
```

- Works for **any** Collection

Outline

Sorting Collections

Simple case

- `Collections.sort(l)` (where `l` is a `List`, for instance)
- Natural ordering of elements (works for all standard Java data types out of the box)
- In order to sort a Collection, its elements need to `implement Comparable`
- Overview of classes implementing `Comparable`: `http://docs.oracle.com/javase/tutorial/collections/interfaces/order.html`

Writing Comparable types (classes)

Comparable interface

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

- In order to sort collections with your own classes, you have to implement `Comparable`!

compareTo method

- Compares the object with another object (o)
- **returns negative int**, if o is less than the object for which the method is called
- **returns 0**, if both objects are equal
- **returns positive int**, if o is greater

Comparator

- Default ordering: natural order
- Different behavior: you need a `Comparator`
- Class that compares two elements of the same type

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Example: Person Comparator

- Normally, sorting persons by their name first is ok
- One scenario: we want to sort them by birthyear for a company anniversary

```
import java.util.Comparator;

public class YearFirstPersonComp implements Comparator<Person>
    {

    public int compare(Person arg0, Person arg1) {
        // sort persons by their birthyear
        return (arg0.getBirthYear() - arg1.getBirthYear());
    }

}
```

Sorting a list of Persons

```
Collections.sort(personList, new YearFirstPersonComp());
```

Outline

Sorted collections I

`SortedSet` interface

- `head/tailSet(E e)` returns sub-sets of elements less/greater than `e`
- `subSet(E from, E to)` returns a sub-set with values between `from` and `to`
- `first/last()` retrieves first/last element
- Concrete implementation: `TreeSet`
- All elements in a sorted set need to implement `Comparable`
- Optional comparator can be specified to adjust ordering strategy
- Constructors:
  - `TreeSet()`
  - `TreeSet(Comparator comp)`
  - `...`

Sorted collections II

SortedMap interface

- Keys are ordered
- Concrete implementation: `TreeMap`
- Methods similar to `SortedSet`
  - `firstKey()`
  - `subMap(K from, K to)`
  - `...`

Outline

Summary

Collections

- `Collection` framework contains multiple classes to conveniently store collections of objects
- Ordered (insertion-order) collections with duplicates: `List` (e.g. `ArrayList`, `LinkedList`)
- Sets of elements without duplicates and no ordering: `Set` (e.g. `HashSet`)
- Sets of elements without duplicates and ordering: `SortedSet` (e.g. `TreeSet`)
- Mapping from keys to values: `Map` (e.g. `HashMap`, `TreeMap`)

Outline

Source

Most of the Generics section slides are from the Java Education and Development Initiative (Jedi)
`https://jedi.java.net/`

Outline

Source

The first few slides on Generics, and the Wildcards section (later) are from Jeff Meister

`http://cseweb.ucsd.edu/~jmeister/`

Java 1.4: Life Before Generics

Java code used to look like this:

```
List listOfFruits = new ArrayList();
listOfFruits.add(new Fruit(“Apple”));
Fruit apple = (Fruit) listOfFruits.remove(0);
listOfFruits.add(new Vegetable(“Carrot”)); // Whoops!
Fruit orange = (Fruit) listOfFruits.remove(0); // Run-time error
```

Problem: Compiler doesn’t know listOfFruits should only contain fruits

A Silly Solution

We could make our own fruit-only list class:

```
class FruitList {
    void add(Fruit element) { … }
    Fruit remove(int index) { … }
    …
}
```

But what about when we want a vegetable-only list later? Copy-paste? Lots of bloated, unmaintainable code?

Java 1.5: Now We're Talking

Now, Java code looks like this:

```
List<Fruit> listOfFruits = new ArrayList<Fruit>();
listOfFruits.add(new Fruit("Apple"));
Fruit apple = listOfFruits.remove(0);
listOfFruits.add(new Vegetable("Carrot")); // Compile-time error
```

Hooray! Compiler now knows listOfFruits contains only Fruits

- So remove() must return a Fruit
- And add() cannot take a Vegetable

13 An Introduction to Generics

Topics

- Why Generics?
- Declaring a Generic Class
  - "Primitive" Limitation
- Constrained Generics
- Declaring a Generic Method

Generics

- Included in Java's latest release
- Problem with typecasting:
  - Downcasting is a potential hotspot for *ClassCastException*
  - Makes our codes wordier
  - Less readable
  - Destroys benefits of a strongly typed language
  - Example: *ArrayList* object

```
String myString = (String) myArrayList.get(0);
```

Generics

- Why generics?
  - Solve problem with typecasting
- Benefits:
  - Allow a single class to work with a wide variety of types
  - Natural way of eliminating the need for casting
  - Preserves benefits of type checking
  - Example: *ArrayList* object

```
//myArrayList is a generic object
String myString = myArrayList.get(0);
```

Generics

- Caution:

```
Integer data = myArrayList.get(0);
```

  - Removal of downcasting doesn't mean that you could assign anything to the return value of the *get* method and do away with typecasting altogether
  - Assigning anything else besides a *String* to the output of the get method will cause a compile time type mismatch

```
found: java.lang.String
required: java.lang.Integer
```

Generics

```
1 //Code fragment
2 ArrayList<String> genArrList =
3                                 new ArrayList<String>();
4 genArrList.add("A generic string");
5 String myString = genArrList.get(0);
6 //int myInt = genArrList.get();
7 JoptionPane.showMessageDialog(this, myString);
```

Declaring a Generic Class

- For the previous code fragment to work, we should have defined a generic version of the *ArrayList* class

- Java's newest version already provides users with generic versions of all Java *Collection* classes

Declaring a Generic Class

```
1  class BasicGeneric<A> {
2      private A data;
3      public BasicGeneric(A data) {
4          this.data = data;
5      }
6      public A getData() {
7          return data;
8      }
9  }
10 //continued...
```

Declaring a Generic Class

```
11 public class GenSample {
12    public String method(String input) {
13        String data1 = input;
14        BasicGeneric<String> basicGeneric = new
15                                BasicGeneric<String>(data1);
16        String data2 = basicGeneric.getData();
17        return data2;
18    }
19 //continued...
```

Declaring a Generic Class

```
20    public Integer method(int input) {
21        Integer data1 = new Integer(input);
22        BasicGeneric <Integer> basicGeneric = new
23                              BasicGeneric <Integer>(data1);
24        Integer data2 = basicGeneric.getData();
25        return data2;
26    }
27 //continued...
```

Declaring a Generic Class

```
20      public static void main(String args[]) {
21          GenSample sample = new GenSample();
22          System.out.println(sample.method(
23                                  "Some generic data"));
24          System.out.println(sample.method(1234));
25      }
26  }
```

Declaring a Generic Class

- Declaration of the BasicGeneric class:

```
class BasicGeneric<A>
```

  - Contains type parameter: `<A>`
  - Indicates that the class declared is a generic class
  - Class does not work with any specific reference type

- Declaration of field:

```
private A data;
```

  - The field *data* is of generic type, depending on the data type that the *BasicGeneric* object was designed to work with

Declaring a Generic Class

- Declaring an instance of the class
  - Must specify the reference type to work with
  - Examples:

```
BasicGeneric<String> basicGeneric = new
                    BasicGeneric<String>(data1);
```

    - Class works with variables of type *String*

```
BasicGeneric<Integer> basicGeneric = new
                    BasicGeneric<Integer>(data1);
```

    - Class works with variables of type *Integer*

Declaring a Generic Class

- Declaration of the *getData* method:

```
public A getData() {
    return data;
}
```

  - Returns a value of type A, a generic type
  - The method will have a runtime data type
  - After you declare an object of type *BasicGeneric*, A is bound to a specific data type

Declaring a Generic Class

- Instances of the *BasicGeneric* class

```
BasicGeneric<String> basicGeneric = new
                        BasicGeneric<String>(data1);
String data2 = basicGeneric.getData();
```

  - *basicGeneric* is bound to *String* type
  - No need to typecast

```
BasicGeneric<Integer> basicGeneric = new
                        BasicGeneric<Integer>(data1);
Integer data2 = basicGeneric.getData();
```

  - *basicGeneric* is bound to *Integer* type
  - No need to typecast

Generics: "Primitive" Limitation

- Java generic types are restricted to reference types and won't work with primitive data types
  - Example:

```
BasicGeneric<int> basicGeneric = new
                        BasicGeneric<int>(data1);
```

- Solution:
  - Wrap primitive types first
  - Can use wrapper types as arguments to a generic type

Constrained Generics

- Preceding example:
  - Type parameters of class *BasicGeneric* can be of any reference data type

- May want to restrict the potential type instantiations of a generic class
  - Can limit the set of possible type arguments to subtypes of a given type bound

Constrained Generics

- Limiting type instantiations of a class
  - Use the *extends* keyword in type parameter

```
class ClassName <ParameterName extends ParentClass>
```

  - Example: generic ScrollPane class
    - Template for an ordinary *Container* decorated with scrolling functionality
    - Runtime type of an instance of this class will often be a subclass of *Container*
    - The static or general type is *Container*

Constrained Generics

```
1  class ScrollPane<MyPane extends Container> {
2      ...
3  }
4  class TestScrollPane {
5      public static void main(String args[]) {
6          ScrollPane<Panel> scrollPane1 =
7                                  new ScrollPane<Panel>();
8          // The next statement is illegal
9          ScrollPane<Button> scrollPane2 =
10                                 new ScrollPane<Button>();
11     }
12 }
```

Constrained Generics

- Gives added static type checking
  - Guarantee that every instantiation of the generic type adheres to assigned bounds
  - Can safely call any methods found in the object's static type
- No explicit bound on the parameter
  - Default bound is *Object*
  - An instance can't invoke methods that don't appear in the *Object* class

Declaring a Generic Method

- Java also allows us to declare a generic method
- Generic Method
  - Polymorphic methods
  - Methods parameterized by type
- Why generic method?
  - Type dependencies between the arguments and return value are naturally generic
  - But the generic nature change from method call to method call rather than class-level type information

Declaring a Generic Method

```
1 class Utilities {
2     /* T implicitly extends Object */
3     public static <T> ArrayList<T> make(T first) {
4         return new ArrayList<T>(first);
5     }
6 }
```

Declaring a Generic Method

- Java also uses a type-inference mechanism
  - Automatically infers the types of polymorphic methods based on the types of arguments
  - Lessens wordiness and complexity of a method invocation

- To construct a new instance of *ArrayList<Integer>*, we would simply have the following statement:

```
Utilities.make(Integer(0));
```

Summary

- Why Generics?
- Declaring a Generic Class

```
class ClassName<TypeParameter> {
    ...
}
```

  - "Primitive" Limitation

Summary

- Constrained Generics

```
class ClassName<ParameterName extends ParentClass>
```

- Declaring a Generic Method
  - Example:

```
public static <T> ArrayList<T> make(T first) {
    return new ArrayList<T>(first);
}
```

Subtyping

Since Apple is a subtype of Object, is List<Apple> a subtype of List<Object>?

```
List<Apple> apples = new ArrayList<Apple>();
List<Object> objs = apples; // Does this compile?
```

Seems harmless, but no! If that worked, we could put Oranges in our List<Apple> like so:

```
objs.add(new Orange()); // OK because objs is a List<Object>
Apple a = apples.remove(0); // Would assign Orange to Apple!
```

An Aside: Subtyping and Java Arrays

- Java arrays actually have the subtyping problem just described (they are covariant)
- The following obviously wrong code compiles, only to fail at run-time:

```
Apple[] apples = new Apple[3];
Object[] objs = apples; // The compiler permits this!
objs[0] = new Orange(); // ArrayStoreException
```

- Avoid mixing arrays and generics (trust me)

Wildcard Types

- So, what *is* List<Apple> a subtype of?
- The supertype of all kinds of lists is List<?>, the List of unknown
- The ? is a wildcard that matches anything
- We can't add things (except null) to a List<?>, since we don't know what the List is really of
- But we can retrieve things and treat them as Objects, since we know they are at least that

Bounded Wildcards

- Wildcard types can have upper and lower bounds
- A List<? extends Fruit> is a List of items that have unknown type but are all at *least* Fruits
  - So it can contain Fruits and Apples but not Peas
- A List<? super Fruit> is a List of items that have unknown type but are all at *most* Fruits
  - So it can contain Fruits and Objects but not Apples

Bounded Wildcards Example

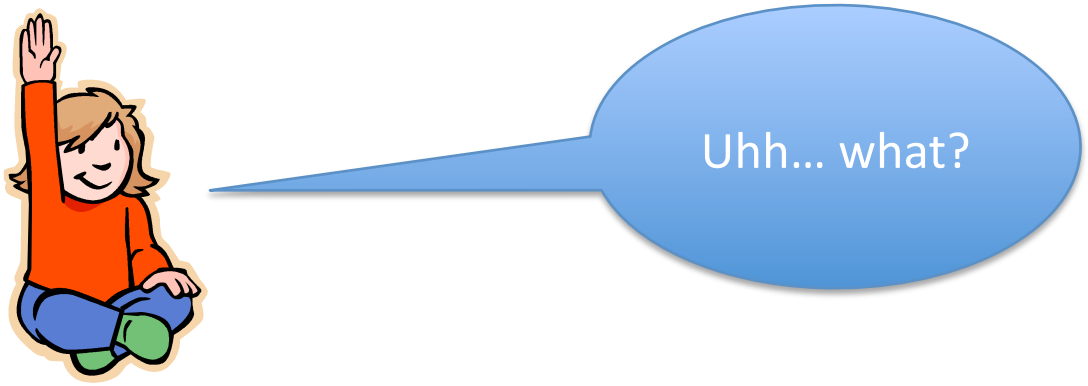
```
class WholesaleVendor<T> {
    void buy(int howMany, List<? super T> fillMeUp) { … }
    void sell(List<? extends T> emptyMe) { … }
    …
}

WholesaleVendor<Fruit> vendor = new WholesaleVendor<Fruit>();
List<Food> stock = …;
List<Apple> overstockApples = …;

// I can buy Food from the Fruit vendor:
vendor.buy(100, stock);
// I can sell my Apples to the Fruit vendor:
vendor.sell(overstockApples);
```

Josh Bloch's Bounded Wildcards Rule

- Use <? extends T> when parameterized instance is a T producer (for reading/input)
- Use <? super T> when parameterized instance is a T consumer (for writing/output)



How Generics are Implemented

- Rather than change every JVM between Java 1.4 and 1.5, they chose to use *erasure*
- After the compiler does its type checking, it *discards* the generics; the JVM never sees them!
- It works something like this:
  - Type information between angle brackets is thrown out, e.g., List<String> → List
  - Uses of type variables are replaced by their upper bound (usually Object)
  - Casts are inserted to preserve type-correctness

Pros and Cons of Erasure

- Good: Backward compatibility is maintained, so you can still use legacy non-generic libraries
- Bad: You can't find out what type a generic class is using at run-time:

```
class Example<T> {
    void method(Object item) {
        if (item instanceof T) { … } // Compiler error!
        T anotherItem = new T(); // Compiler error!
        T[] itemArray = new T[10]; // Compiler error!
    }
}
```

Using Legacy Code in Generic Code

- Say I have some generic code dealing with Fruits, but I want to call this legacy library function:

```
Smoothie makeSmoothie(String name, List fruits);
```

- I can pass in my generic List<Fruit> for the fruits parameter, which has the *raw type* List. But why? That seems unsafe… makeSmoothie() could stick a Vegetable in the list, and that would taste nasty!

Raw Types and Generic Types

- List doesn't mean List<Object>, because then we couldn't pass in a List<Fruit> (subtyping, remember?)
- List doesn't mean List<?> either, because then we couldn't assign a List to a List<Fruit> (which is a legal operation)
- We need both of these to work for generic code to interoperate with legacy code
- Raw types basically work like wildcard types, just not checked as stringently
  - These operations generate an unchecked warning

The Problem with Legacy Code

- “Calling legacy code from generic code is inherently dangerous; once you mix generic code with non-generic legacy code, all the safety guarantees that the generic type system usually provides are void. However, you are still better off than you were without using generics at all. At least you know the code on your end is consistent.” – Gilad Bracha, Java Generics Developer

My Advice on Generics

- Don’t try to think about generic code abstractly; make an example instantiation in your head and run through scenarios using it
- Generics are a valuable tool to ensure type safety, so use them! Let the compiler help you
- However, generics also complicate syntax, and they can generate some nasty errors that are a pain to understand and debug

An Analogy: Functions

- Problem: I want to perform the same computation on many different input values without writing the computation over and over.
- Solution: Write a function! Use a variable to represent the input value, and write your code to perform the computation on this variable in a way that does not depend on its value. Now you can call the function many times, passing in different values for the variable. Easy stuff.

Generics Provide Another Abstraction

- Problem: I want to use the same class (or method) with objects of many different types without writing the class over and over or sacrificing type safety.
- Solution: Generify the class! Use a variable T to represent the input type, and write your code to operate on objects of type T in a way that does not depend on the actual value of T. Now you can instantiate the class many times, passing in different types for T.
- See? It's not so bad. Generics just allow you to abstract over types instead of values.

Exercises I

Will the following class compile?

```
public final class Algorithm {
    public static T max(T x, T y) {
        return x > y ? x : y;
    }
}
```

Exercises II

Will the following method compile?

```
public static void print(List<? extends Number> list) {
        for (Number n : list)
            System.out.print(n + " ");
        System.out.println();
    }
```

Exercise

Source: Washington University, CS 341

- J.M. defines an interface `Appendable` with an append method
- He defines two classes, `MyString` and `MyList`, which both implement `Appendable`
- He wants to allow a `MyString` to be appended to a `MyString`, and a `MyList` to a `MyList`
- But **not** a `MyString` to a `MyList` or vice-versa.
- Here is his definition of Appendable:

```
interface Appendable {
    Appendable append(Appendable a);
}
```

What is wrong with this definition? What is a correct one?

Literature

Java Tutorials on Generics
http://docs.oracle.com/javase/tutorial/java/generics/

Sierra, K. & Bates, B.
Head First Java. (Chapter 16)
O'Reilly Media, 2005.

Ullenboom, Ch.
Java ist auch eine Insel. (Chapter 9)
Galileo Computing, 2012.