Generics

Object Oriented Programming

Version 2.7.2 - April 2016

Motivation

- Often the same operations has to be performed on objects from unrelated classes
  - Typical solution is to use Object references to accommodate any object type
- The use of Object references induces cumbersome code
- Solution
  - Use Generic classes and methods

Example

- We may need to represent values of different types (e.g. **`int`**, **`String`**, etc.)

Use of **`Object`** for any value type

```
public class Pair {
  Object a,b;
  public Pair(Object a, Object b )
  {  this.a=a; this.b=b; }
  Object first(){ return a; }
  Object second(){ return b; }
}
```

NOTE: No primitive types, only wrappers allowed

Example

- You can use it with different types

```
Pair sp = new Pair("One","Two");
Pair ip = new Pair(1,2);
```

- You need down casts..

```
String a = (String) sp.second();
```

- ..that may be dangerous

```
String b = (String) ip.second();
```

ClassCastException at run-time

Generic class

```
public class Pair<T> {
  T a,b;

  public Pair(T a, T b) {
    this.a=a; this.b=b;
  }

  public T first(){ return a; }
  public T second(){ return b; }
}
```

Generics use

- Declaration is longer but…

```
Pair<String> sp = new Pair<>("One","Two");
Pair<Integer> ip = new Pair<>(1,2);
```

- ..use is more compact and safer

```
String a = sp.second();
int b = ip.second();
String bs = ip.second();
```

No down cast required

`Integer` can be auto-unboxed

Compiler error: type mismatch

Generic type declaration

- Syntax:

`(class|interface) Name <P`$_1$` {,P`$_2$`}>`

- Type parameters, e.g. **P**$_1$:
  - Represent classes
  - Conventionally uppercase letter
  - Usually: T(ype), E(lement), K(ey), V(alue)

Generic Interfaces

- All standard interfaces and classes have been defined as generics
  - since Java 5
- Use of generics lead to code that is
  - safer
  - more compact
  - easier to understand
  - equally performing

Generic `Comparable`

- Interface **`java.lang.Comparable`**

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

- Semantics: returns
  - a negative integer if `this` precedes `obj`
  - 0, if `this` equals `obj`
  - a positive integer if `this` succeeds `obj`

Generic `Comparable`

- Without generics:

```
public class Student
        implements Comparable {
   int id;
   public int compareTo(Object o){
     Student other = (Student)o;
     return this.id - other.id;
   }}
```

- With generics:

```
public class Student
        implements Comparable<Student> {
   int id;
   public int compareTo(Student other){
     return this.id - other.id;
   }}
```

Generic `Iterable` and `Iterator`

```
public interface List<E>{
  void add(E x);
  Iterator<E> iterator();
}
```

```
public interface Iterator<E>{
  E next();
  boolean hasNext();
}
```

Iterable example

```
class Random implements Iterable<Integer> {
  private int[] values;
  public Random(int n, int min, int max){ … }
  public Iterator<Integer> iterator() {
    return new Iterator<Integer>(){
        private int position=0;
        public boolean hasNext() {
           return position < values.length;
        }
        public Integer next() {
           return values[position++];
        }
     };
}}
```

`Iterable` example

- Without generics:

```
Random seq = new Random(10,5,10);
for(Object e : seq){
  int v = ((Integer)e).intValue();
  System.out.println(v);
}
```

- With generics:

```
Random seq = new Random(10,5,10);
for(int v : seq){
  System.out.println(v);
}
```

Diamond operator

- Reference type parameter must match the class parameter used in instantiation
  - E.g.

```
List<String> l=new LinkedList<String>();
```

- The Java compiler can infer the type using the diamond operator:

```
List<String> l = new LinkedList<>();
```

  - Since Java 7

Unbounded type

- The type parameters used in generics are unbounded by default
  - I.e. there are no constraints on the types that can be substituted to the type parameters
- The safe assumption is
  `T extends Object`
  - References of a type parameter T at least provide members that are defined in class Object

Unbounded example

- A point with different precisions

```
public class Point<T> {
  T x; T y;
  public Point(T x, T y){
    this.x = x; this.y = y;
  }
  public double length(){
    return Math.sqrt(
     Math.pow(x.doubleValue(),2)
     + Math.pow(y.doubleValue(),2) );
  }
}
```

method undefined
for some type T

Bounded types

- Express constraints on type parameters

```
class C< T extends B1 { & B2 } >
```

  - class C can be instantiated only with types extending from B1 (and B2, etc.) including B1
    - B1 is an upper bound

```
class C< T super D >
```

  - class C can be instantiated only with types that are super classes of D, including D
    - D is a lower bound

Bounded example

- T must be bounded to allow the compiler know which methods are available

```
public class Point<T extends Number> {
  T x; T y;
  public Point(T x, T y){
    this.x = x; this.y = y;
  }
  public double length(){
   return Math.sqrt(
    Math.pow(x.doubleValue(),2)
    + Math.pow(y.doubleValue(),2) );
  }
}
```

Generics subtyping

- We must be careful about inheritance when generic types are involved
  - `Integer` is a subtype of `Number`
  - `Point<Integer>` is NOT a subtype of `Point<Number>`

```
Point<Integer> pi = new Point<Integer>(0,1);
Point<Number> pn = pi;
pn.x = new Double(0.5);
Integer i = pi.x;
```

ERROR

if this were legal then...

.. we could end up assigning a Double to an Integer reference

Type invariance

- Type variance: it is always possible substitute a more general (super class) reference to a specific one

```
Integer i;
Object o = i;
```

- Generics type parameters are invariant

```
Point<Integer> pi;
Point<Number> pn = pi;
```

Type mismatch

Array covariance

- Arrays are type co-variant
  - The compiler assumes

    `SubClass[] extends BaseClass[]`
  - As a consequence run-time type clashes are possible

```
String[] as = new String[10];
Object[] ao;
ao = as; // this is ok!!!
ao[1] = new Integer(1);
```

java.lang.ArrayStoreException

Invariance limitations

- An attempt to have a generic method:

```
void printCoord(Point<Number> p) {
  System.out.println("("+p.x+","+p.y+")");
}
```

- Won't work with e.g. **`Point<Integer>`**

```
  Point<Integer> p = new Point<>(7,4);
  printCoord(p);
```

Method is not applicable

Wildcard

- An attempt to have a generic method:

Point of unknown

```
void printCoord(Point<?> p) {
   System.out.println("("+p.x+","+p.y+")");
}
```

- The type is literally unknown therefore it is treated in the safest possible way:
  - Only method from Object are allowed
  - Assignment to an unknown reference is illegal

Wildcards

- Allow to express (lack of) constraints when *using* generic types
- **`G<?>`**
  - G of unknown, unbounded
- **`G<? extends B>`**
  - upper bound: only sub-types of B
    - Including B
- **`G<? super D>`**
  - lower bound: only super-types of D
    - Including D

Generic method

- Syntax:

  modifiers **<T> *type* name(pars)**

- pars can be:
  - as usual
  - **T**
  - **type<T>**

Bounded wildcard – example

Cannot be invoked with `Pair<Integer>`

```
double sum(Pair<Number> p){
  return p.a.doubleValue()+p.b.doubleValue();
}
```

Defines an upper bound for the type parameter

```
<T extends Number> double sumB(Pair<T> p)
{…}
```

Unknown with upper bound
Equivalent but more compact

```
double sumUB(Pair<? extends Number> p)
{…}
```

Sort method

- On Comparable objects:

```
static <T extends Comparable<? super T>>
void sort(T[] list)
```

  - For backward compatibility, actually in class **Arrays** sort is defined as:
  - `public static void sort(Object[] a)`
  - No compile time check is performed.

- Using a Comparator object:

```
static <T> void
sort(T[] a, Comparator<? super T> cmp)
```

Sort generic

~~T~~ extends Comparable<~~?~~ super ~~T~~>

MasterStudent Student MasterStudent

- Why `<? super T>` instead of just `<T>` ?
  - Suppose you define
    - `MasterStudent extends Student { }`
  - Intending to inherit the Student ordering
    - It does not implement `Comparable<MasterStudent>`
    - But `MasterStudent` extends (indirectly) `Comparable<Student>`

TYPE ERASURE

Generic classes

- All the invocations of a generic class are expressions of that single class
  - Just one raw class is generated by the compiler

```
Person<Integer> a = new Person<Integer>
           ("Al","A",new Integer(123));
Person<String> b = new Person<String>
                     ("Pat","B","s32");
boolean same=(a.getClass()==b.getClass());
```

believe it or not
same is *true*

Type erasure

- Classes corresponding to generic types are generated by type erasure
  - The erasure of a generic class is a raw type
    - where any reference to the parameters is substituted with the parameter erasure
  - Erasure of a parameter is the erasure of its first constraint
    - If no constraint then erasure is **`Object`**
  - The erasure of a non-generic type is the type itself

Type erasure – examples

- In: `<T>`
  - `T` ==> `Object`
- In: `<T extends Number>`
  - `T` ==> `Number`
- In: `<T extends Number & Comparable>`
  - `T` ==> `Number`

Type erasure – consequences I

- Compiler applies checks only when a generic type is used, not within it.
- Whenever a generic or a parameter is used a cast is added to its erasure
- To avoid inconsistencies and wrong expectations
  - **`instanceof`** and **`.class`** cannot be used on generic types
  - valid for G<?> equivalent to the raw type

Type erasure – consequences II

- It is not possible to instantiate an object of the generic's parameter type from within the class

```
class G<T> {
    T[] toArray(){
    T[] res = new T[n];
    T t = new T();
  }}
```

The compiler cannot instantiate these objects

- It is not possible to substitute the erasure in an instantiation statement

Type erasure– consequences III

- Overload and ovverride must be considered after type erasure

```
class Base<T> {
  void m(int x){}
          Object
  void m(T t){}
  void m(String s){}
                                      Number
  <N extends Number> void m(N x){}
  void m(List<?> l){}
}
```

Type erasure– consequences IV

- Inheritance together with generic types leads to several possibilities
- It is not possible to implement two generic interfaces instantiated with different types

```
class Value implements Comparable<Value>
class ExtValue extends Value
              implements Comparable<ExtValue>
```

FUNCTIONAL INTERFACES WITH GENERICS

Functional Interfaces

- An interface with exactly one method
- The semantics is purely functional
  - The result of the method depends solely on the arguments
  - There are no side-effects on attributes
- Can be implemented as lambda expressions
- Predefined interfaces are defined in
  - **`java.util.function`**

Standard Functional Interfaces

Interface	Method
`Function <T,R>`	`R apply(T t)`
`BiFunction <T,U,R>`	`R apply(T t, U u)`
`BinaryOperator <T>`	`T apply(T t, T u)`
`UnaryOperator <T>`	`T apply(T t)`
`Predicate <T>`	`boolean test(T t)`
`Consumer <T>`	`void accept(T t)`
`BiConsumer <T,U>`	`void accept(T t, U u)`
`Supplier <T>`	`T get()`

Primitive specializations

- Functional interfaces handle references
- Specialized versions are defined for primitive types (**`int`**, **`long`**, **`double`**, **`boolean`**)
- Functions
  - **`To`*`Type`*`Function`**
  - **`Type1ToType2Function`**
- Suppliers: **`TypeSupplier`**
- Predicate: **`TypePredicate`**
- Consumer: **`TypeConsumer`**

Generic Comparator

- Interface **java.util.Comparator**

```
public interface Comparator<T>{
    int compare(T a, T b);
}
```

```
Arrays.sort(sv, (a,b) -> a.id - b.id );
```

```
Arrays.sort(sv, new Comparator<Student>(){
 public void compare(Student a, Student b){
   return a.id - b.id
});
```

Comparator factory

- Most comparators take some information out of the objects to be compared
  - Typically through a getter

```
static <T,U extends Comparable<U>>
Comparator<T>
    comparing(Function<T,U> keyGetter)
```

  - Accepts a function that extracts a sort key from a type T, and returns a Comparator<T> that compares by that sort key using the specified Comparator

Comparator.comparing

```
Arrays.sort(sv,comparing(Student::getId));
```

Requires:
`import static java.util.Comparator.*`

```
static <T,U extends Comparable<? super U>>
Comparator<T>
  comparing(Function<T,U> keyGetter){
  return (a,b) -> keyGetter.apply(a).
                    compareTo(keyGetter.apply(b));
}
```

Comparator historical perspective

```
Arrays.sort(sv,new Comparator(){
 public int compare(Object a, Object b){
  return ((Student)a).id-((Student)b).id;
 }});
```

Java ≥ 2

```
Arrays.sort(sv,new Comparator(){
 public int compare(Student a, Student b){
  return a.getId() - b.getId();
 }});
```

Java ≥ 5, Generics

Java ≥ 8, Lambda

```
Arrays.sort(sv,(a,b)->a.getId()-b.getId());
```

```
Arrays.sort(sv,comparing(Student::getId));
```

Java ≥ 8, Method reference

Comparator composition

- Reverse order
  - Default method Comparator.reversed()

```
default <T> Comparator<T> reversed(){
   return (a,b) -> - this.compare(a,b);
}
```

```
Arrays.sort(sv,
     comparing(Student::getId).reversed());
```

Comparator composition

- Multiple criteria
  - Default method Comparator.thenComparing()

```
default <T> Comparator<T>
        thenComparing(Comparator<T> other){
  return (a,b) -> {
        int r = this.compare(a,b);
        if(r!=0) return r;
        else return other.compare(a,b);
}
```

Comparator composition

- Multiple criteria

```
default <U extends Comparable<U>
Comparator<T> thenComparing(Function<T,U> ke){
  return (a,b) -> {
   int r = this.compare(a,b);
   if(r!=0) return r;
   return ke.apply(a).compareTo(ke.apply(b));
}
```

```
 Arrays.sort(sv,
        comparing(Student::getLast).
             thenComparing(Student::getFirst));
```

Performance

- Comparing
  - Anonymous Inner Class or Lambda Expression

```
Arrays.sort(sv,(a,b)->b.getId()-a.getId());
```

  - Comparator.comparing + reversed

```
Arrays.sort(sv,
     comparing(Student::getId).reversed());
```

    – Requires 50% to 60% more time

Functional Interfaces Composition

- Predicate
  - `default Predicate<T> and(Predicate<T> o)`
  - `default Predicate<T> or(Predicate<T> o)`
  - `default Predicate<T> negate()`
- Function
  - `default Function<V,R> compose(Function<V,T> before)`

Wrap-up

- Generics allow defining type parameter for methods and classes
- The same code can work with several different types
  - Primitive types must be replaced by wrappers
- Generics containers are type invariant
  - Wildcard, ? (read as unknown)
- Generics are implemented by type erasure
  - Checks are performed at compile time