Using a generic class to perform basic arithmetic operations

Asked 7 years, 1 month ago Active 3 months ago Viewed 9k times

11

I want to perform basic arithmetic operations like addition, subtraction, multiplication and division using only one generic method per operation for wrapper types like `Integer`, `Float`, `Double` ... (excluding `BigDecimal` and `BigInteger`).

I have tried to do something like the following (for addition) using a generic class.

```
public final class GenericClass<E extends Number> {

    public E add(E x, E y) {
        return x + y; // Compile-time error
    }
}
```

It issues a compile-time error,

> operator + cannot be applied to E,E

Is there a way to use such a generic version to achieve such operations?

java generics

edited Dec 23 '15 at 9:30

asked Dec 26 '12 at 21:00

Tiny
22.8k 81 274 542

1 this sounds like overengineering the problem domain. – Woot4Moo Dec 26 '12 at 21:05

10 Answers

12

No, there isn't a way to do this, or else it would be built into Java. The type system isn't strong enough to express this sort of thing.

answered Dec 26 '12 at 21:00

Louis Wasserman
157k 21 283 349

6

No, you can't do that because the + operator is not part of the Number class. What you can do is to create an abstract base class and extends from it:

```
static void test() {

    MyInteger my = new MyInteger();
    Integer i = 1, j = 2, k;
    k = my.add(i, j);
    System.out.println(k);
}

static public abstract class GenericClass<E extends Number> {
    public abstract E add(E x, E y);
}

static public class MyInteger extends GenericClass<Integer> {
    @Override
    public Integer add(Integer x, Integer y) {
        return x + y;
    }
}
```

(I made these classes static in order to facilitate the testing but you can remove this modifier.)

You could also add an abstract function that will take and return parameters and return value of type Number and override it and the subclasses but the required casting for the return value will defeat its usefulness.

answered Dec 26 '12 at 22:31

SylvainL
3,768 3 16 24

Regarding the scenario in the question, there is no meaning to use a generic class in this way. Additionally, declaring a static class in this way is not allowed in Java (this is however the case with C#). A static class must have at lest one none-static **enclosing** class (outermost classes cannot be static) i.e. a static class must be contained by at least one outermost none-static class. – Lion Dec 27 '12 at 0:31

For the static classes, I did not say that I was not enclosing them into another class. I made these static simply because I was instanciating them from a static function but you're right, it would have been simpler to just remove this word everywhere. As to your other statement, "Regarding the scenario in the question, there is no meaning to use a generic class in this way", I'm not totally sure to understand what you are saying but lets me just say that's simply impossible in Java to solve the original scenario exactly as described. – SylvainL Dec 27 '12 at 1:03

2

All I can think of is receiving the type as an unknown type and use instance of.

Something like:

```
public class GenericClass<? extends Number>{
    public Integer add(? x, ? y){
        if(x instance of Integer && y instance of Integer){
            //Do your math with Integer class methods help
            //Return here
        }
        return (Interger)null;
    }

}
```

But not sure :/

answered Dec 26 '12 at 23:27

Afonso Tsukamoto

1,086 11 16

1

Generally you don't need this as it's simpler and more efficient to use a "super" type like double or BigDecimal which can represent any value of any type.

Note: a `double` uses less than half the space of an `Integer` with a reference to it.

answered Dec 26 '12 at 21:05

Peter Lawrey

473k 66 635 1019

1

Number is a class type, so the primitive type operators do not apply. Additionally, generics in Java don't support primitive types, so you won't be able to bind E to allow primitive types only.

One way around this would be to handle each primitive type that Number supports separately in the Add method, but I think that defeats what you're trying to accomplish.

answered Dec 26 '12 at 21:08

Fls'Zen

4,369 1 25 36

1

As suggested by Afonso, another possibility would be to create a function and write down all the required possibilities. Here are some variations:

```
      // Examples with static functions:
        Integer i = addNumbers (1, 2); // OK
        Double d = addNumbers (3.0, 4.0); // OK
        String s = addObjects ("a", "b"); // OK
//
      // Example with a raw type for a class with both method
      // (or member function) and static functions:
GenericClass gc = new GenericClass();  // Raw type
//
      // Note the error if we don't add a cast:
```

```
// i = gc.add(1, 2); // Error: Cannot convert from Number to Integer
//
      // Now OK with a cast but with a Type safety warning:
        i = (Integer) gc.add (1, 2); // Type safety warning.
        i = GenericClass.add2 (1, 2); // OK
        i = GenericClass.add3 (1, 2); // OK
//
      // Example with an instanciated type for the same class:
        GenericClass<Integer> gc1 = new GenericClass<Integer>();
//
      // Note that we don't need a cast anymore:
        i = gc1.add(1, 2); // Now OK even without casting.
//
        i = GenericClass2.add2 (1, 2); // OK
        s = GenericClass2.add2 ("a", "b"); // OK
        i = GenericClass2.<Integer>add2 (1, 2); // OK.
        d = GenericClass2.<Double>add2 (1.0, 2.0); // OK
        s = GenericClass2.<String>add2 ("a", "b"); // OK
//
    public static<T extends Number> T addNumbers(T x, T y) {

        if (x instanceof Integer && y instanceof Integer){
            return (T) (Integer) ((Integer)x + (Integer)y);
        } else if (x instanceof Double && y instanceof Double){
            return (T) (Double) ((Double)x + (Double)y);
        } else
            return (T)null;
    }
//
    public static<T> T addObjects(T x, T y) {

        if (x instanceof Integer && y instanceof Integer) {
      //
      // We must add an (Integer) cast because the type of the operation
      // "((Integer)x + (Integer)y))" is "int" and not "Integer" and we
      // cannot directly convert from "int" to "T".  Same thing for Double
      // but not for the type String:
      //
            return (T) (Integer) ((Integer)x + (Integer)y);
        } else if (x instanceof Double && y instanceof Double) {
            return (T) (Double) ((Double)x + (Double)y);
      } else if (x instanceof String && y instanceof String) {
            return (T) ((String)x + (String)y);
      } else
            return (T)null;
    }
 //
    static class GenericClass<T extends Number> {

        public T add(T x, T y) {
            if (x instanceof Integer && y instanceof Integer) {
                return (T) (Integer) ((Integer)x + (Integer)y);
            } else if (x instanceof Double && y instanceof Double) {
                return (T) (Double) ((Double)x + (Double)y);
            } else
                return (T)null;
        }
 //
        // The type <T> here is NOT the same as the one for the class.
        // We should rename it in order to make this clearer. See add3()
        // for an example of this.
        public static<T> T add2(T x, T y) {
            if (x instanceof Integer && y instanceof Integer) {
                return (T) (Integer) ((Integer)x + (Integer)y);
```

```
            } else if (x instanceof Double && y instanceof Double) {
                return (T) (Double) ((Double)x + (Double)y);
            } else if (x instanceof String && y instanceof String) {
                return (T) ((String)x + (String)y);
            } else
                return (T)null;
        }
//
        // The type here is not the same as the one for the class
        // so we have renamed it from <T> to <N> to make it clearer.
        public static<N extends Number> N add3(N x, N y) {
            if (x instanceof Integer && y instanceof Integer) {
                return (N) (Integer) ((Integer)x + (Integer)y);
            } else if (x instanceof Double && y instanceof Double) {
                return (N) (Double) ((Double)x + (Double)y);
            } else
                return (N)null;
        }
    }
```

answered Dec 27 '12 at 15:26

SylvainL
3,768 3 16 24

1

As Louis Wasserman pointed out, there is no way to do this in Java. However, a solution can be presented by using some not-so-tricky programming. Let's start with a solution that I like: SylvainL's answer to the question. However, I believe we could go one step back and handle **every type** of `Number`. If you look at the Java API, you can note that a subclass of `Number` has to override a couple of abstract methods; namely, intValue() (as well as others). Using these methods, we can utilize polymorphism to its true potential. Taken the class we have from SylvainL's answer, we can produce a new class as such:

```
public final class EveryNumberClass<E extends Number>
{
    public static int add(E x, E y)
    {
        return x.intValue() + y.intValue();
    }

    public static int subract(E x, E y)
    {
        return x.intValue() - y.intValue();
    }
}
```

These operations could be expanded to multiplication and division, and while not limited to `Integer`s, could be used to take in **any** `Number` and express the appropriate behavior given an operation. And while the behavior of the `intValue()` method **may not** return an integer representation of the `Number`, it most certainly does (I looked in the source code for most numbers, including the atomic ones and the math ones). The only issue will occur when unexpected behavior is returned from `intValue()`, which may happen with atomic numbers, user defined `Number`s, or when big numbers are forced to shrink. If those are concerns for your project (or for your library), I would consider using `long` values, or referring to SylvainL's answer.

edited May 23 '17 at 12:08

Community ♦

1 1

answered Jun 23 '16 at 19:11

astrogeek14

214 1 13

0

Hope this might help someone who is interested to work on Arithmetic operations using Generics

```
  public < T > T add( T a, T b)
  {
      return (T) (String.valueOf((Integer.parseInt(a.toString()) +
Integer.parseInt(b.toString()))));
  }
  public < T > T sub(T a, T b)
  {
      return (T) (String.valueOf((Integer.parseInt(a.toString()) -
Integer.parseInt(b.toString()))));
  }
  public < T> T mul(T a,T b)
  {
      return (T) (String.valueOf((Integer.parseInt(a.toString()) *
Integer.parseInt(b.toString()))));
  }
  public < T  >  T  div (T a, T b)
  {
      return (T) (String.valueOf((Integer.parseInt(a.toString()) /
Integer.parseInt(b.toString()))));
  }
  public < T  > T mod(T a, T b)
  {
      return (T) (String.valueOf((Integer.parseInt(a.toString()) %
Integer.parseInt(b.toString()))));
  }
  public < T  > T leftShift(T a, T b)
  {
      return (T) (String.valueOf((Integer.parseInt(a.toString()) <<
Integer.parseInt(b.toString()))));
  }
  public < T  > T rightShift(T a, T b)
  {
      return (T) (String.valueOf((Integer.parseInt(a.toString()) >>
Integer.parseInt(b.toString()))));
  }
```

answered Jan 10 '17 at 4:47

BHANUTEJA REDDY KONTAM

1

0

```
enum Operator {
    ADD, SUBTRACT, DIVIDE, MULTIPLY;
}

public class GenericArithmeticOperation {
```

```
    public GenericArithmeticOperation() {
        // default constructor
    }

    public <E> Integer doArithmeticOperation(E operand1, E operand2, Operator operator)
{
        if (operand1 instanceof Integer && operand2 instanceof Integer) {
            switch (operator) {
            case ADD:
                return (Integer) ((Integer) operand1 + (Integer) operand2);
            case SUBTRACT:
                return (Integer) ((Integer) operand1 - (Integer) operand2);
            case MULTIPLY:
                return (Integer) ((Integer) operand1 * (Integer) operand2);
            case DIVIDE:
                return (Integer) ((Integer) operand1 / (Integer) operand2);
            }
        }
        throw new RuntimeException(operator + "is unsupported");
    }

    public static void main(String[] args) {
        GenericArithmeticOperation genericArithmeticOperation = new
GenericArithmeticOperation();
        System.out.println(genericArithmeticOperation.doArithmeticOperation(10, 6,
Operator.ADD));
        System.out.println(genericArithmeticOperation.doArithmeticOperation(10, 6,
Operator.SUBTRACT));
        System.out.println(genericArithmeticOperation.doArithmeticOperation(4, 6,
Operator.MULTIPLY));
        System.out.println(genericArithmeticOperation.doArithmeticOperation(21, 5,
Operator.DIVIDE));
    }
}
```

edited Jan 18 '17 at 3:06
user3559349

answered Jan 18 '17 at 2:44

Shiv
1 1

0

This may help you. Just create the different operations with the types you want to support. And voilà, you can use arithmetic operations on generics now. Without expensive `String` parsing.

```
static <V extends Number> V add(V lhs, V rhs)
{
    if (lhs.getClass() == Integer.class && rhs.getClass() == Integer.class)
        return (V) Integer.valueOf(((Integer) lhs) + ((Integer) rhs));
    if (lhs.getClass() == Double.class && rhs.getClass() == Double.class)
        return (V) Double.valueOf(((Double) lhs) + ((Double) rhs));
    if (lhs.getClass() == Float.class && rhs.getClass() == Float.class)
        return (V) Float.valueOf(((Float) lhs) + ((Float) rhs));
    if (lhs.getClass() == Long.class && rhs.getClass() == Long.class)
        return (V) Long.valueOf(((Long) lhs) + ((Long) rhs));
    throw new IllegalArgumentException("unsupported type: " + lhs.getClass());
}
```

`valueOf()` wraps the result into an `Integer` object, which then can be casted to V.

For a performance increase change the order descending by the amount of use.

A switch don't works because you would either have to give in a generic or a `Class<?>`, which are not acceptable for switch statements.

answered Nov 1 '19 at 20:37

jackjuni

69 6