

# Generics in Java

Marc Framvig-Antonsen & Jens Svensson

- Introduced in JDK1.5
- Classes, Class methods and Interfaces can be generic
- Generic types are erased by type erasure in the compiled program
- At runtime there exist only one implementation of the generic code(class,method,interface)
- Multiple generic parameters
- Wildcard generics arguments.
- Bounding generic parameter both upper and lower
- Default upper bound is Object.
- Type correctness is checked at compile time, using the upper bound of the generic parameters

## Generic class

---

```
public class Holder<T> {  
    private T value;  
    public Holder(T a_value) {  
        value=a_value;  
    }  
    public T getValue() {  
        return value;  
    }  
}
```

## Usage

---

```
Holder<String> h=new Holder<String>("String");  
Holder<Integer> hi=new Holder<Integer>(45);  
System.out.println(h.getValue());  
System.out.println(hi.getValue());  
System.out.println(h.getClass());  
System.out.println(hi.getClass());
```

## Output

---

```
String  
45  
class Holder  
class Holder
```

- Generic parameters come after the name of the class
- The class of the variables h and hi are Holder.
- Generic Classes don't infer types from constructor arguments.
- Have to specify the type two places
- The code of the class must valid for the bound of the generic parameter

## Generic class multiple generic parameters

---

```
public class _2Parameters<T,P>{
    T value1;
    P value2;
    public _2Parameters(T a_value,P a_value2){
        value1=a_value;
        value2=a_value2;
    }
    public T first(){
        return value1;
    }
    public P second(){
        return value2;
    }
}
```

## Usage

---

```
_2Parameters<String,Integer> _2par=new _2Parameters<String,Integer>("test",45);
System.out.println(_2par.first());
System.out.println(_2par.second());
```

## Output

---

```
test
45
```

- The Generic Parameter list is comma separated.

## Generic classes may not be direct or indirect subclass of Throwable

---

```
class Throwable_1<T> extends Throwable{  
  
}
```

### Usage

---

```
Throwable_1<Integer> Test=m.new Throwable_1<Integer>();
```

### Output

---

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:  
    The generic class Main.Throwable_1<T> may not subclass java.lang.Throwable  
  
    at Main$Throwable_1.<init>(Main.java:78)  
    at Main.main(Main.java:93)
```

- The catch mechanism only works with non generic types

## Generic method

---

```
public class Normal{  
    public <T> void printValue(Holder<T> a_value){  
        System.out.println("In Normal Class:"+a_value.getValue());  
    }  
}
```

## Usage

---

```
Normal n=new Normal();  
  
n.printValue(hi);  
n.printValue(h);
```

## Output

---

```
In Normal Class:45  
In Normal Class:String
```

- Generic Parameters comes after scope but before return value of the method
- Generic Methods do infer the generic types from passed values.

## Generic method

---

```
public class Normal{  
    public <T> void printValue(T a_value){  
        System.out.println("In Normal Class:"+a_value.getValue());  
    }  
}
```

## Usage

---

```
Normal n=new Normal();  
n.printValue(hi);  
n.printValue(h);
```

## Output

---

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:  
    The method getValue() is undefined for the type T  
  
    at Normal.printValue(Normal.java:6)  
    at Main.main(Main.java:91)
```

- The Generic Parameter T is unbounded so it defaults to Object. Object don't have a getValue function
- It does not matter that we only send objects of Holder to it

# Effects of Type Erasure

## Generic method Overloading

---

```
public class Normal{
    public void printHolder(Holder<String> a_value){
        System.out.println("In Normal Class:"+a_value.getValue()+" Its a string");
    }
    public void printHolder(Holder<Integer> a_value){
        System.out.println("In Normal Class:"+a_value.getValue()*10);
    }
}
```

## Output

---

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
    Duplicate method printHolder(Holder<String>) in type Normal
    Duplicate method printHolder(Holder<Integer>) in type Normal
```

- The Generic arguments <String> and <Integer> are both removed in the compiled code so the 2 functions are identical.

# Generic method Overloading on return type

---

```
public class Normal{
    public void printHolder(Holder<String> a_value) {
        System.out.println("In Normal Class:"+a_value.getValue()+" Its a string");
    }
    public int printHolder(Holder<Integer> a_value) {
        System.out.println("In Normal Class:"+a_value.getValue()*10);
        return 0;
    }
}
```

## Usage

---

```
Normal n=new Normal();
n.printHolder(hi);
n.printHolder(h);
```

## Output

---

```
In Normal Class:450
In Normal Class:String Its a string
```

- The correct method gets called.
- Has to return a value

## Generic method Overloading on dummy parameter

---

```
public class Normal{
    public void printHolder(Holder<String> a_value,String dummy){
        System.out.println("In Normal Class:"+a_value.getValue()+" Its a string");
    }
    public void printHolder(Holder<Integer> a_value,Integer dummy){
        System.out.println("In Normal Class:"+a_value.getValue()*10);
    }
}
```

## Usage

---

```
Normal n=new Normal();
n.printHolder(hi,new Integer(42));
n.printHolder(h,new String(""));
```

## Output

---

```
In Normal Class:450
In Normal Class:String Its a string
```

- The correct method gets called.
- Has to send in a dummy value

## Only one generic class

---

```
public class Specialisation<T>{}
```

```
public class Specialisation<T,P>{}
```

- The type erasure erases the type parameters, and there can't be 2 implementations of the same type.

## Shared static members

---

```
public class StatTest<T> {  
    private static int id=0;  
    public StatTest() {  
        id++;  
    }  
    public int getId() {  
        return id;  
    }  
}
```

## Usage

---

```
StatTest<Integer> s1=new StatTest<Integer>();  
StatTest<String> s2=new StatTest<String>();  
System.out.println(s1.getId());  
System.out.println(s2.getId());
```

## Output

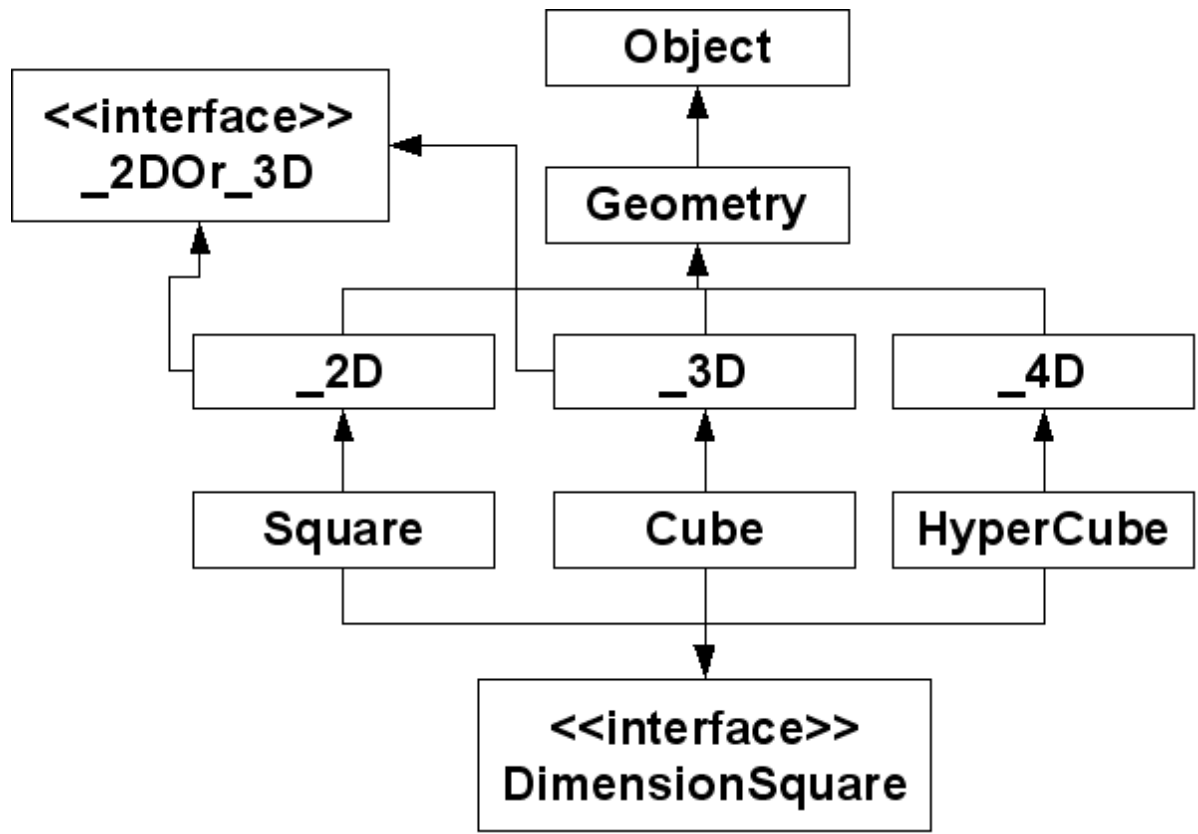
---

2  
2

- The field `id` are static and are therefore shared by all instances of `StatTest`.
- The Generics are type erased so eventhough the generic arguments are different the end type is the same `StatTest`

Bounding

Example Class Hirachy.



## Only `_2D` and its children

```
private <T extends _2D> void TestExtends(T value) {  
    System.out.println(value.getClass());  
}
```

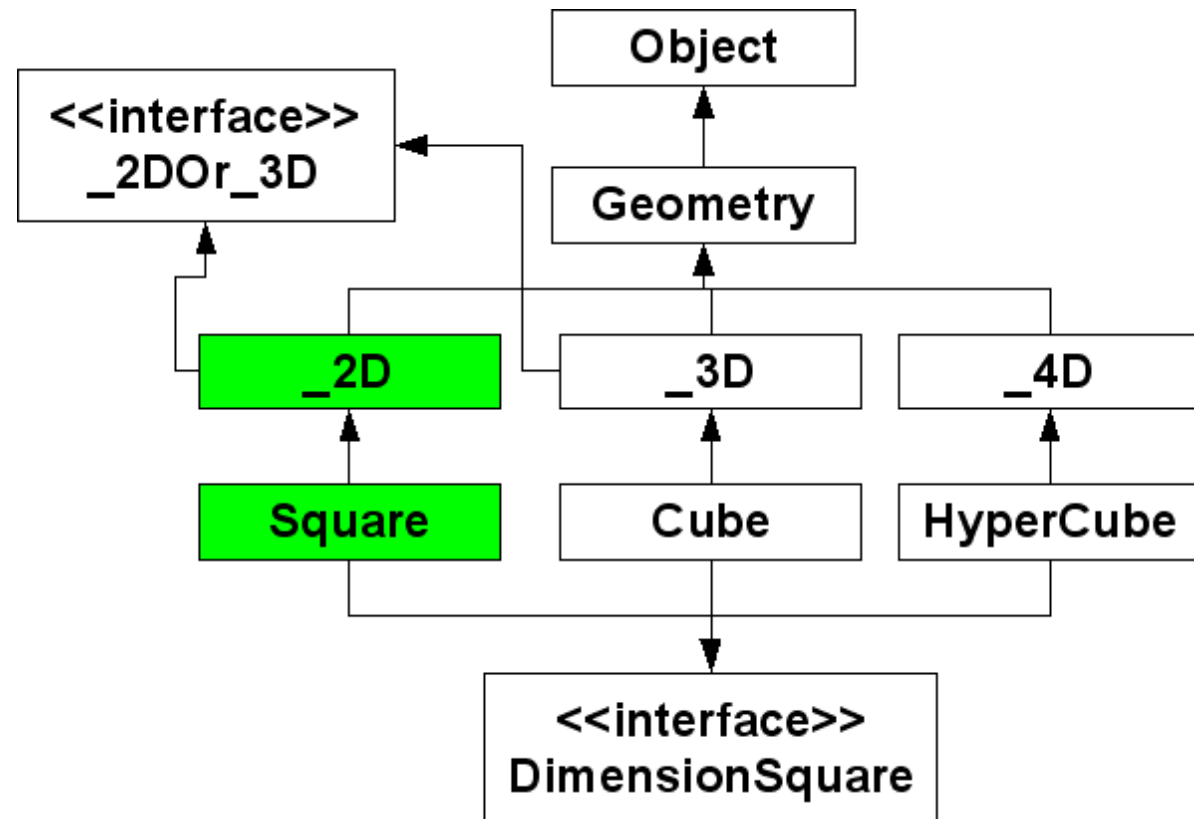
## Usage

```
Geometry geo=new Geometry();  
_2D _2d=new _2D();  
Square square=new Square();  
TestExtends(_2d);  
TestExtends(square);  
//TestExtends(geo);Geometry is not a subclass of _2D
```

## Output

```
class _2D  
class Square
```

- `T` extends `ClassType`, allows all classes that are subclasses and the class itself.



## Only childs of Geometry that implements `_2DOR_3D` interface

```
private <T extends Geometry & _2DOR_3D> void Test2D_OR_3D(T value) {  
    System.out.println(value.getClass());  
}
```

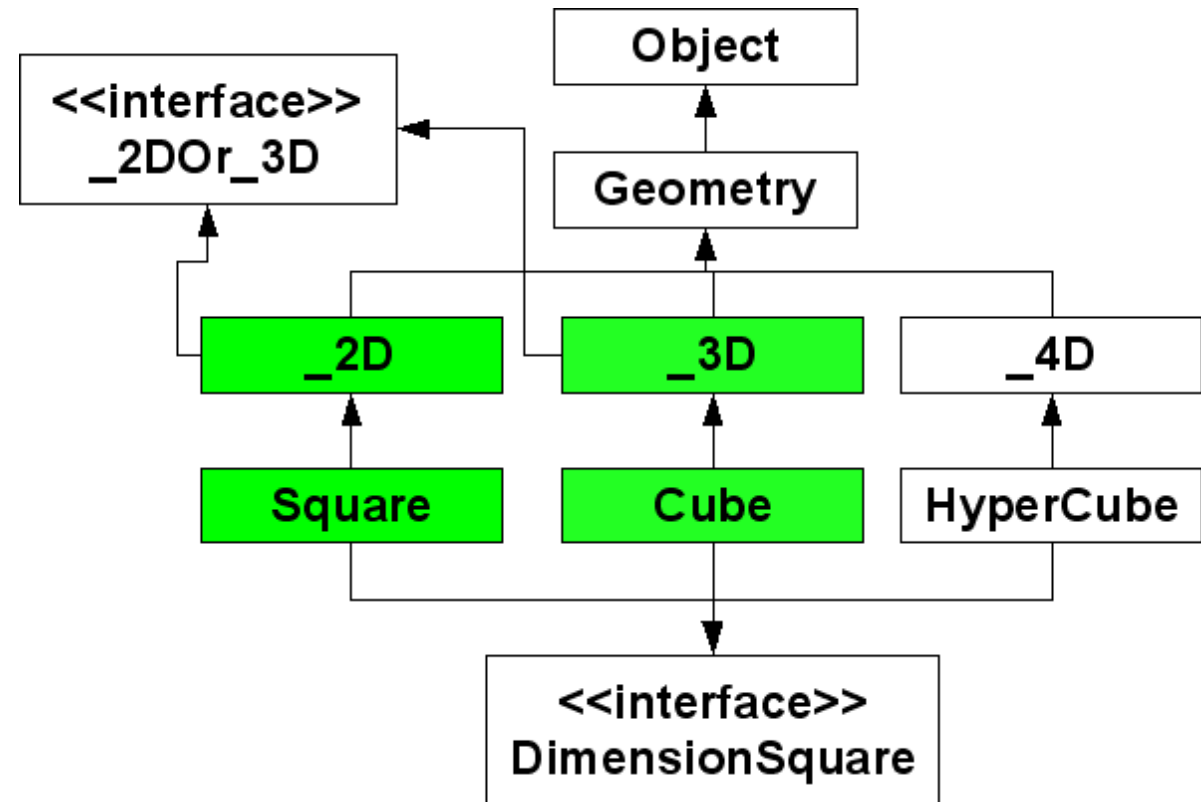
## Usage

```
Test2D_OR_3D(cube);  
Test2D_OR_3D(square);
```

## Output

```
class Cube  
class Square
```

- T extends `ClassType` & `InterfaceType`, allows all subclasses of `ClassType` and the `ClassType` itself that are subclass or the class itself that of a class that implements `InterfaceType`.
- More interfaces can be specified separated by &



# Only childs of classes that implements `_2DOR_3D` and `DimensionSquare` interfaces

```
private <T extends _2DOR_3D&DimensionSquare> void TestCube(T value) {  
    System.out.println(value.getClass());  
}
```

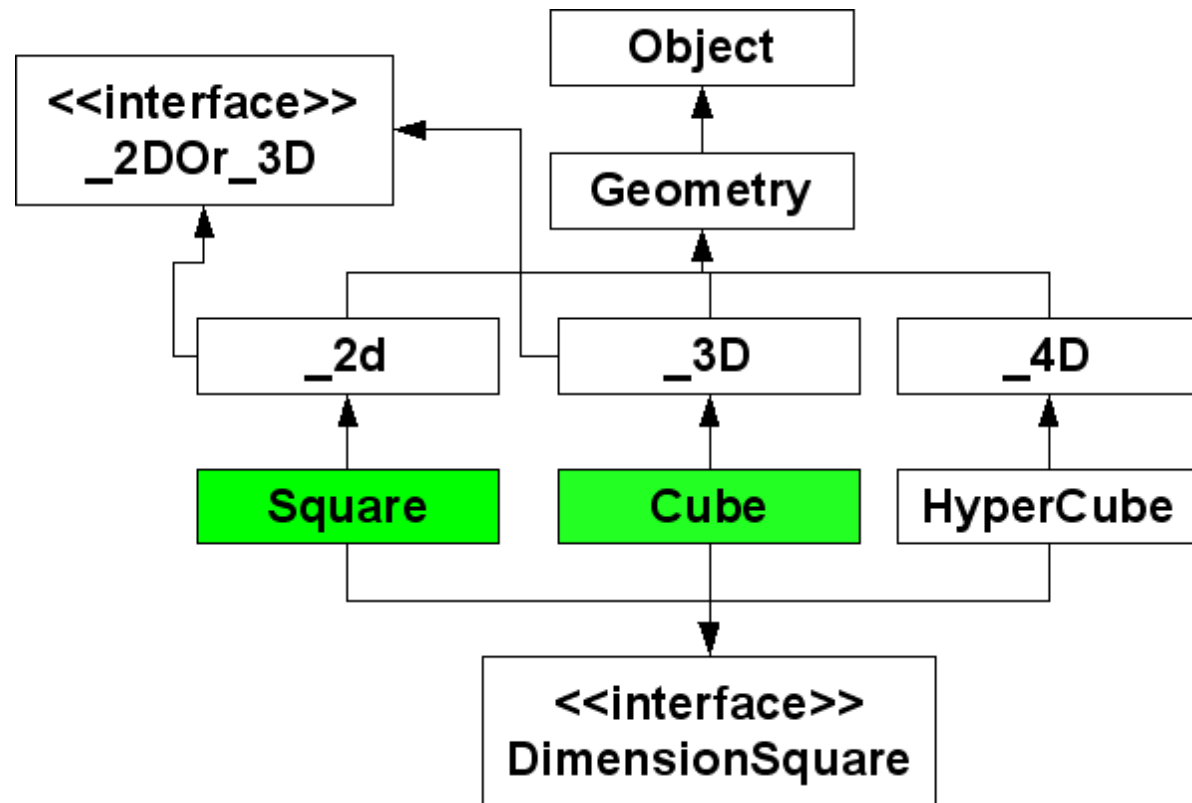
## Usage

```
TestCube(cube);  
TestCube(square);
```

## Output

```
class Cube  
class Square
```

- T extends `InterfaceType` & `InterfaceType2`, T is a class subclass of a class that implements `InterfaceType1` and `InterfaceType2`



Generic arguments

TypeArguments:  
    < ActualTypeArgumentList >

ActualTypeArgumentList:  
    ActualTypeArgument  
    ActualTypeArgumentList , ActualTypeArgument

ActualTypeArgument:  
    ReferenceType  
    Wildcard

Wildcard:  
    ? WildcardBoundsOpt

WildcardBounds:  
    extends ReferenceType  
    super ReferenceType

## Distinct Type Argument

```
public void DistinctGenericArgument (Holder<Geometry> value) {  
    System.out.println (value.getValue ().getClass ());  
}
```

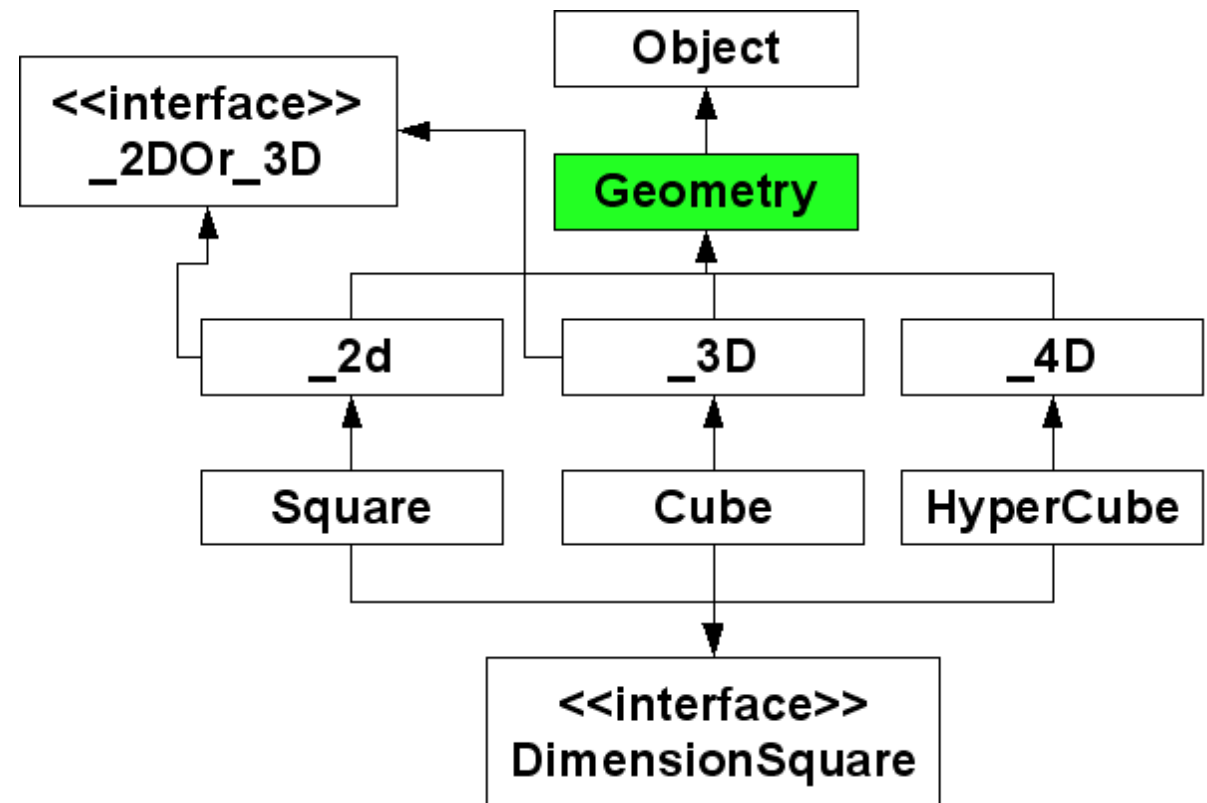
## Usage

```
DistinctGenericArgument (new Holder<Geometry> (geo));
```

## Output

```
class Geometry
```

- The type of the given variable must be the exact same as the argument type.



## Childed wild card

```
public void WildChild(Holder<? extends Geometry> value) {  
    System.out.println(value.getValue().getClass());  
}
```

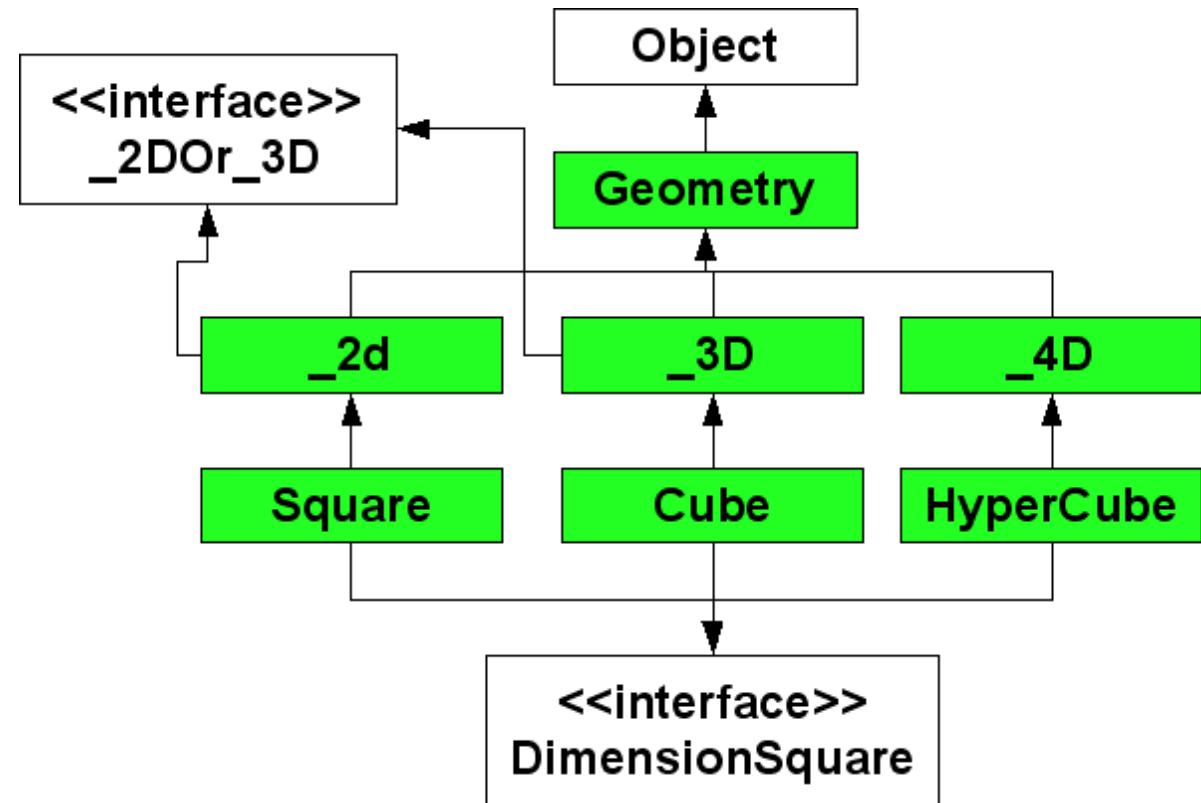
## Usage

```
WildChild(new Holder<Geometry>(geo));  
WildChild(new Holder<Cube>(cube));  
WildChild(new Holder<Square>(square));
```

## Output

```
class Geometry  
class Cube  
class Square
```

- The generic type of the given variable must be a child of the generic argument type or it.



## Parent wild card

```
public void WildParent(Holder<? super Square> value) {  
    System.out.println(value.getValue().getClass());  
}
```

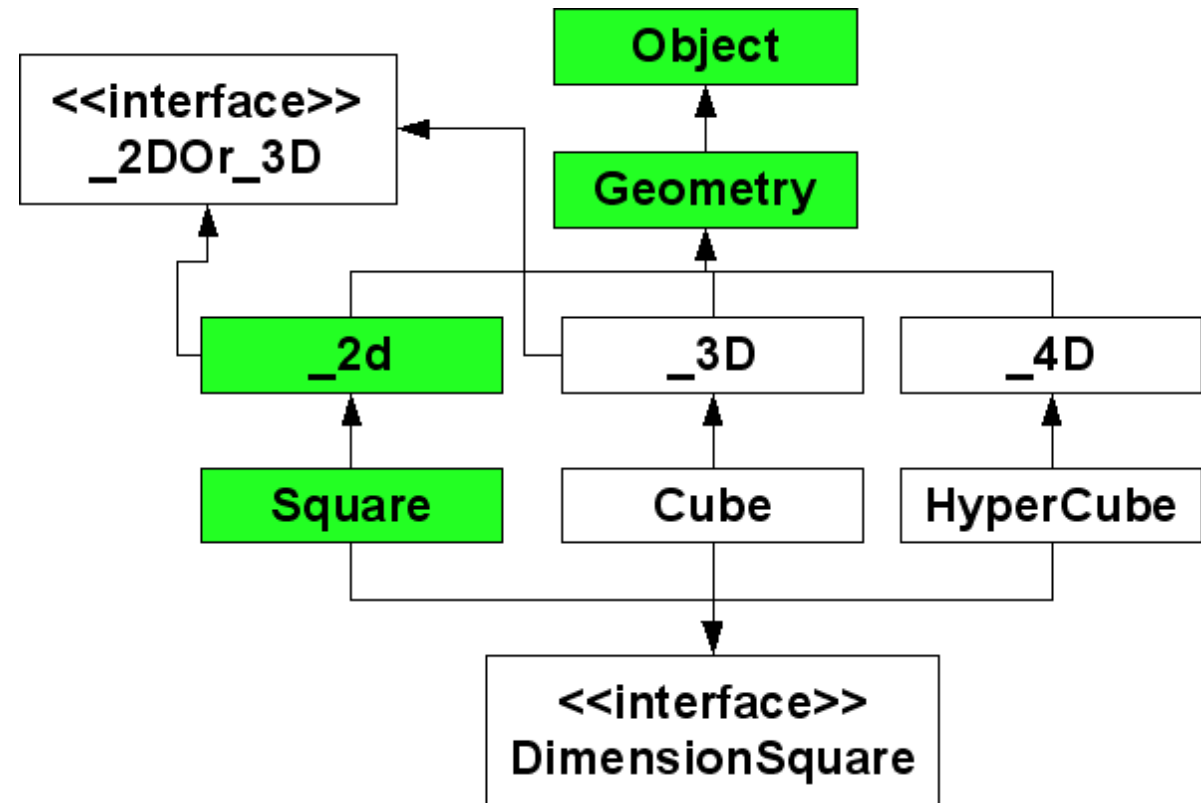
## Usage

```
WildParent(new Holder<Geometry>(geo));  
WildParent(new Holder<Object>(new Object()));  
WildParent(new Holder<_2D>(_2d));  
WildParent(new Holder<_Square>(square));
```

## Output

```
class Geometry  
class java.lang.Object  
class _2D  
class _Square
```

- The generic type of the given variable must be a parent of the generic argument type or it.



## Upper bounded parent wild card

```
public class GeoHolder<T extends Geometry> extends Holder<T>{
    public GeoHolder(T value){
        super(value);
    }
}

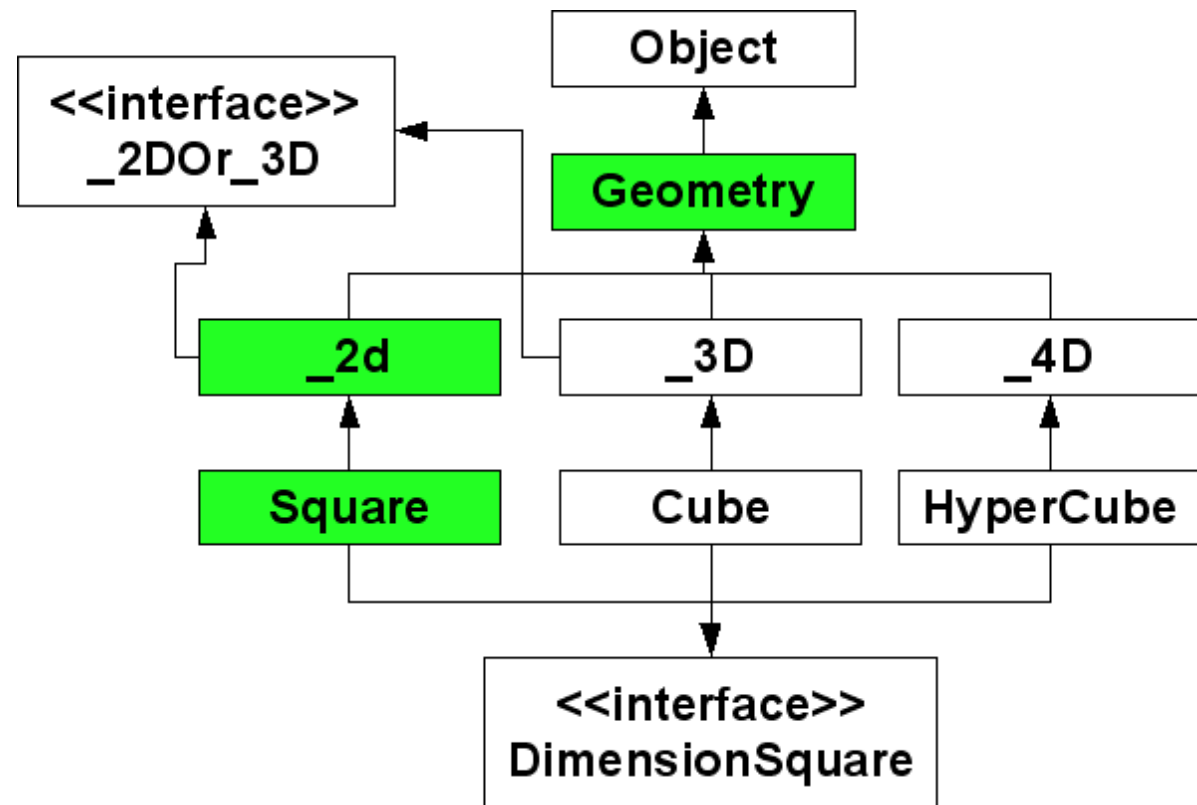
public void WildParentUpperBounded(GeoHolder<? super Square> value){
    UsageSystem.out.println(value.getValue().getClass());
}
```

```
WildParentUpperBounded(new GeoHolder<Geometry>(geo));
WildParentUpperBounded(new GeoHolder<_2D>(_2d));
WildParentUpperBounded(new GeoHolder<_Square>(square));
```

## Output

```
class Geometry
class _2D
class _Square
```

- To upper bound a wild card the generic parameter can define the upper bound



## Read only Wildcards

---

```
public class Write<T>{
    private T name;
    public Write(T value){
        name=value;
    }
    public void setName(T a_name){
        name=a_name;
    }
    public T getName(){
        return name;
    }
}
public void WriteName(Write<? extends String> value){
    value.setName("Test2");
    System.out.println(value.getName());
}
```

## Usage

---

```
WriteName(new Write<String>("Ha"));
```

## Output

---

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
    The method setName(capture#8-of ? extends String) in the type
    Main.Write<capture#8-of ? extends String> is not applicable for the
    arguments (String)

    at Main.WriteName(Main.java:35)
    at Main.main(Main.java:174)
```

- Can't write to values affected by the wildcard.

## Generic arguments may not be primitive types

---

```
public class Holder<T> {  
}
```

## Usage

---

```
Holder<int> h_int=new Holder<int>(45);
```

## Output

---

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:  
    Syntax error on token "int", Dimensions expected after this token  
    Syntax error on token "int", Dimensions expected after this token  
  
    at Main.main(Main.java:90)
```

- Primitive types are boolean, byte, short, int, long, char, float and double
- Use Boolean, Byte, Short, Integer, Long, Character, Float and Double as generic argument instead.

The Java Language Specification, Third Edition

[http://java.sun.com/docs/books/jls/third\\_edition/html/j3TOC.html](http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html)

**Generics in the Java Programming Language**

<http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>