Generics

Slide Series 8

Content

Generic types and methods, type inference
(Bounded) Type parameters
Parameterized types
Variance for Generics
(Bounded) Wildcards
Generic type compatibility
Restrictions
Reifiable types and type erasure
Visitor DP, Generic Iterator
Reflection

Generic Programming

"...to operate on objects of various types [but same] while providing compile-time type safety."

// Wikipedia

Very familiar from Haskell

```
// Doesn't mention any specific element type
data List a = Nil | Cons a (List a)


length :: List a -> Int
length Nil         = 0
length (Cons _ t) = 1 + length t
```

Generics in Java

No generics in original Java, added to Java 1.5

- Possible have a mixture of non-generic and generic code (avoid, warning)
- Java collection framework 1.5 completely rewritten to generic types, List<String>, List<Integer>
- Have generic classes, interfaces, constructors and methods (no generic enum types, anonymous inner classes or exception classes)
- Interesting effects on type system (may affect your mental health) …  more to come…

Usage Generics

Normally

- Use of Java generic (and other) API's in particular the Collections framework. Easy

Advanced topics

- Implement flexible reusable collections and API's
- Implement frameworks (semi-implemented applications). Sometimes in combination **reflection** (meta programming, upcoming ...), hard

Type Variable

"A type variable is an unqualified identifier used as a type in class, interface, method, and constructor bodies."

// JLS 4.4

Type variable traditionally single upper case; T, E, ...

Generic Class*

"A class [interface] is generic if it declares one or more type variables" // JLS 8.1.2

```
// Type variables
public class MyClass<T, E> {   // Type parameter section
    private final T t1;    // Attribute of type T
    public T doIt( E t ){ // Parameter type E return type T
    }
}
```

Combination static and T not permitted in generic class (but possible for generic method, upcoming...)

Parameterized Type

"A generic class or interface declaration C (§8.1.2, §9.1.2) with one or more type parameters A1 ...defines a set of parameterized types, one for each possible invocation of the type parameter section."

*"A **parameterized type** is written as a ClassType or InterfaceType that contains at least one type declaration specifier immediately followed by a type argument list . The type argument list denotes a particular invocation of the type parameters of the generic type indicated by the type declaration specifier".//* JLS 4.5

```
// Parameterized type (concrete instantiation of
// generic type)
List<String> strs = ....
// Parameterized type
List<List<String>> strs = ....
// Bad, primitive types not allowed
List<int> ints = ....
```

Generic Interface*

```
// Generic interface
public interface IA<T> {
    public T doIt();
}

// Non-generic implementation (using parameterized type)
public class MyClass implements IA<String> {
    public String doIt();
}

// Generic implementation
public class MyClass<T> implements IA<T> {
    public T doIt(){ ...}
}
```

Generic and Inheritance*

Many possibilities/variations, ..

- Should subclasses be generic too?
- See code samples

Generic Methods*

Generic methods are methods that introduce their own type parameters before the return type.

Type parameter's scope is limited to the method where it is declared

```
// Non-generic class (generic also possible)
public class MyClass {
    public <A> void doIt(A a) { ... }
    public static <A> A max(Collection<A> xs) { ... }
}
```

Invocation of Generic Methods*

Type arguments need not be provided explicitly, almost always automatically inferred

```
// Generic method (note: static ok)
public static <A> A max (Collection<A> xs) {...}

// Invocation
List<Long> list = ...;
list.add(0L);
list.add(1L);
Long y = SomeClass.max(list);  // Type inferred

(If problem supply type: SomeClass.<theType>max(list))
```

Type Inference for Generic Methods*

Compiler tries to infer the most specific types that makes the call type safe

- Using parameter or return types (or both)
- Sometimes counterintuitive!

Generic Constructor

Constructors can be generic (declare their own formal type parameters) in both generic and non-generic classes

```
// Generic class
public class MyClass<X> {
    // Generic constructor
    <T> MyClass(T t) {
      ...
   }
}

// Will infer X as Integer and T as String (Java 7)
MyClass<Integer> myObject = new MyClass<>("")
```

Remainder: Type Parameter vs Parameterized Type

Type parameter

- Used at declaration of class, interface , method or constructor

```
public class MyClass<T>
public <T> int count( T t )
```

Parameterized type

- Used for variables, methods arguments and return types

```
public List<Integer> getList();
public void print(List<String> strs);
```

Bounded Type Parameter

"*Every type variable declared as a type parameter has a bound. If no bound is declared for a type variable, Object is assumed. If a bound is declared, it consists of either:*

- *a single type variable T , or*
- *a class or interface type T possibly followed by interface types I1 & ... & In.' //* JLS 4.4

A bounds is a restriction on the type parameter

Bound set by keyword **extends**

Examples: Bounded Type Parameter*

```
// T is a subtype of Number (Number is the bound)
public class Calculator<T extends Number> { ... }

// T must implement Comparable<T>
public interface IMinMax<T extends Comparable<T>> { ... }

// Any subclass to MyClass that implements Serializble
public <T extends MyClass & Serializable>
```

Variance More Formally

Covariance

- if A :> B and and f(A) :> f(B) for some conversion operator f (for example [ ]) then f is covariant (preserves ordering of types)

Contra variance

- if A :> B and and f(A) <: f(B) for some conversion operator f (for example []) then f is covariant (reverses ordering of types)

Invariance, neither of above

Wildcards

Parameterize types may have "wildcards" as type argument

- Wildcard written as: **?** (questionmark)
- Wildcard represents the unknown type (**not** any type)
- Wildcards may have a bound (one only)
- A type variable T denotes the same but not known type, wildcard does not denote the same (more ? will possible stand for different types)

```
// Parameterized type with wildcard as type argument
Collection<?> coll = new ArrayList<>();
```

Usage: Wildcards

Wildcards are useful in situations where only partial knowledge about the type parameter is required."

// JLS 4.5.1

```
// Using the wildcard '?', a Collection of unknown
public void printAny(Collection<?> c) {
    for (Object o : c) { // This is what we can assume
        System.out.println(o);  // Call Object.toString()
    }
}
```

Note: This is not the same as Collection<Object>, Object is a known type!

Q: Why not just use subtypes?...

Answer: Generics and Variance

Generics are not covariant (they are invariant)!

if S :> T then C<S> **not** :> C<T> (C any type)

```
// Assume method
public void printAny(List<Object> c) {...}
// Can't use anything but List<Object>
// even though Object:> String
List<String> s = ...
printAny( s ); // No! List<String> not compatible!
```

Bounded Wildcards*

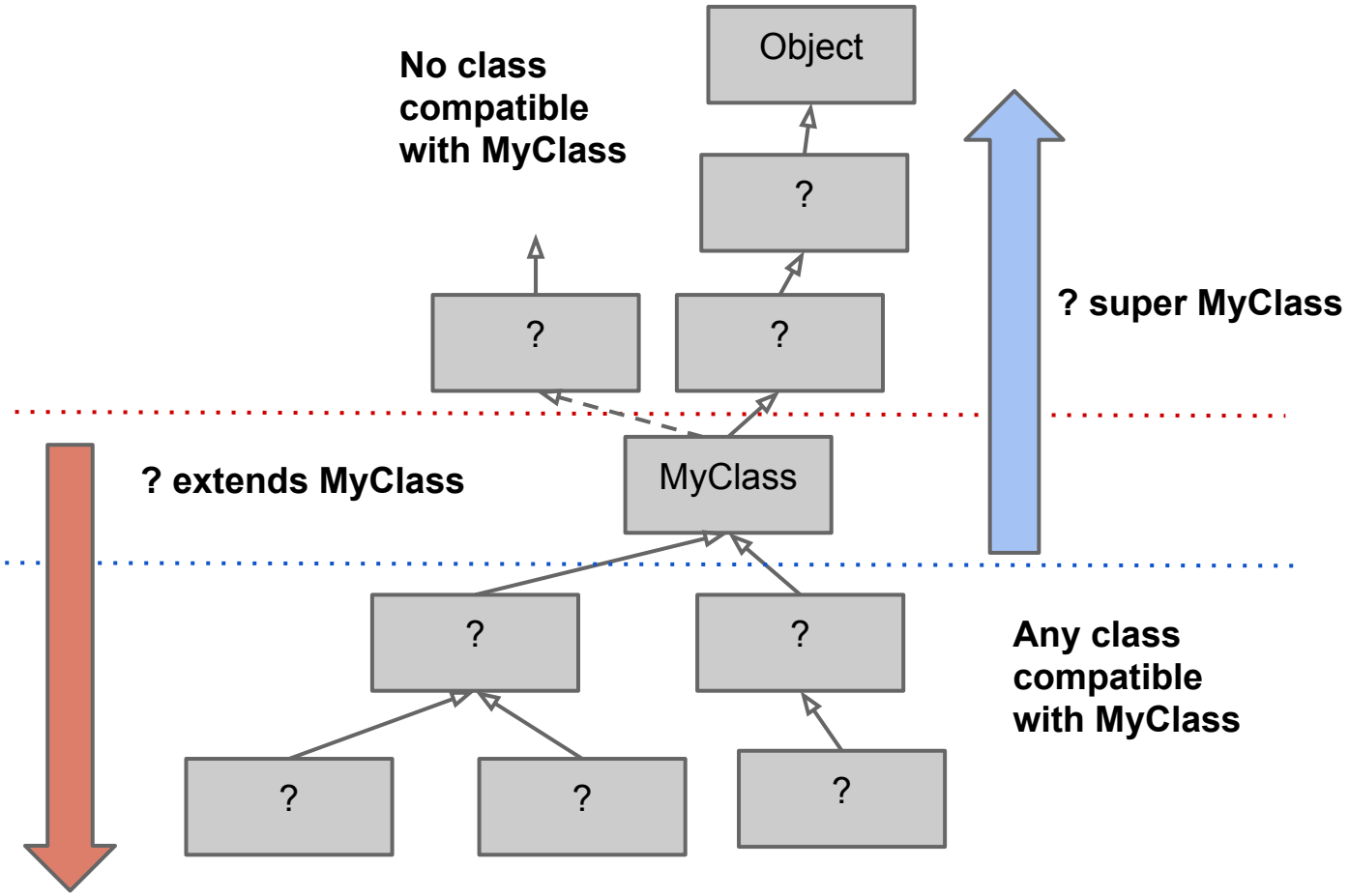
Bound set by keyword **extends** or **super** (can't have both at the same time)

```
// List contains unknown subtype of number. Upper bound
List<? extends Number> nlist;

// List contains unknown supertype of Integer. Lower bound
List<? super Integer> ilist;
```

super is highly unintuitive...

Wildcard Extends vs Super



Bounded wildcards gives type safe variance*

```
// Array loop hole, Object[] :> String[] ([] is covariant)
String[] strings = new String[1];
Object[] objects = strings;
objects[0] = new Integer(1);   // Oh, oh
String s = strings[0].substring(0);  // The array loophole!!

// No hole with generic types List<Object> not :> List<String>
// (no covariance)
List<String> slist = new ArrayList<>();
//List<Object> olist = slist;   // No!

// Ok, ...
List<? extends Object> olist2 = slist;
// ... but can just treat elements as Objects
// s = olist2.get(0);    // No!
Object o = olist2.get(0);  // Ok!
```

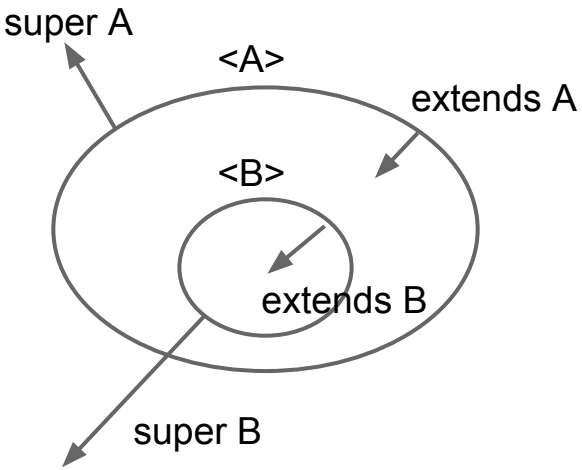
Super-subset relationships for Wildcards bounds

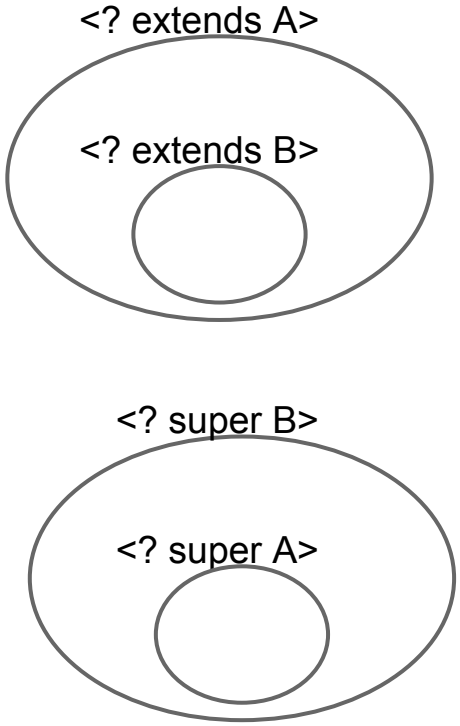
A super-subtype relationship between upper bounds leads to a super-subset relationship between the resulting upper bound wildcards, and vice versa for a lower bounds

- The unbounded wildcard "?" denotes the set of all types and is the superset of all sets denoted by bounded wildcards.
- A wildcard with an upper bound (extends) A denotes a superset of another wildcard with an upper bound B, if A is a super type of B.
- A wildcard with a lower bound A denotes a superset of another wildcard with a lower bound B, if A is a subtype of B.
- A concrete type denotes a set with only one element, namely the type itself.

Super-subset relationships for Wildcards bounds, cont.

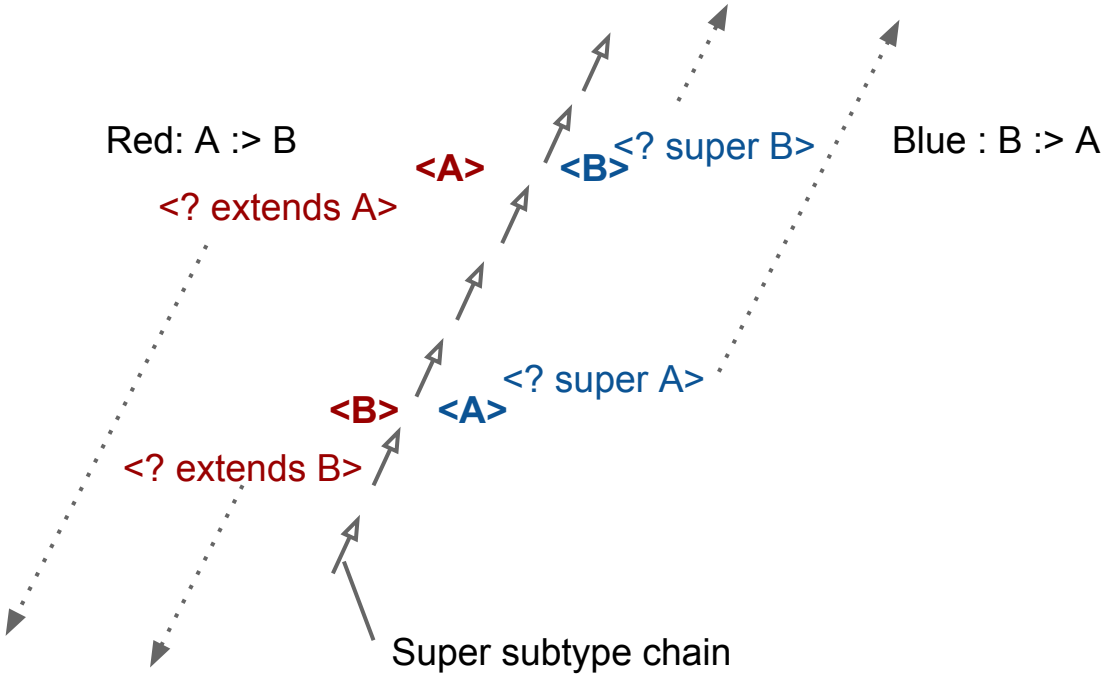
Illustration of previous slide A :> B





Super-subset relationships for Wildcards bounds? cont. cont.

A second illustration of previous, previous slide



Example Super-subset

```
<?> is superset of all bounded wildcard sets

<? extends Serializable>  (subset of the above)

// Relation of bounds: Serializable :> Number ...
<? extends Number> (... so subset of above)

<Number> (subset of above)
```

Generics and Type Compatibility*

Thanks to Angelika Langer

1. As long as the type arguments are identical, the inheritance relationship among generic types leads to a super- subtype relationship among corresponding parameterized types (Collection<Long> :> List<Long>)
2. Super-subset relationship based on the type arguments (at least one of the involved type arguments is a wildcard)

   Collection<? extends Number> ⊃ Collection<Long>

Both effects - the super-subtype relationship due to inheritance and the super-subset relationship due to type arguments - are combined and lead to a two-dimensional super-subtype relationship table see Angelika Langer

Note: There are possible more type parameters and also multi level wildcards, yum, yum, ...

Raw Type

Mixing non-generic and generic code, should be avoided!

```
// Using generic type List<T> in non-parameterized fashion
List l = ...    // List as raw type

// Compatibility, raw type super type to any parameterized
List :>  List<...>

// And also...
List<?> :> List
```

Examples: Type Compatibility

```
// Extends. Assume A and B implements Able
Collection<? extends Able> :> Collection<A>
Collection<? extends Able> :> Collection<B>

// Super
List<? super Integer> :> List<Number>
List<? super Integer> :> List<Object>
```

This lead to amazing restrictions ...

Restrictions on Collections with bounded Wildcards*

? extends T

- Can't add anything except null!
- Can read type of bound (T, so no problem)

? super T

- Can only add type of bound (T), subclass of bound or null
- Can only read Object

Bounded Wildcards as Return Type

"It is sometimes tempting to use a bounded wildcard in the return type of a method. But this temptation is best avoided because returning bounded wildcards tends to "pollute" client code. If a method were to return a Box<? extends T>, then the type of the variable receiving the return value would have to be Box<? extends T>, which pushes the burden of dealing with bounded wildcards on your callers. Bounded wildcards work best when they are used in APIs, not in client code."

// Someone

Capture of Wildcard

"What is the capture of a wildcard?
*An anonymous type variable that represents the particular unknown type that the wildcard stands for. The compiler uses the capture internally for evaluation of expressions and the term "**capture of ?"** occasionally shows up in error message." //* Angelika Langer

```
error: equalTo(Box<capture#1 of ?>) in Box<capture#2 of ?>
cannot be applied to (Box<capture#3 of ?>)
equal = unknownBox.equalTo(unknownBox)
```

This is a type error, compiler tries to find types for ? but fails. You have a type error!

Wildcard Capture*

This is different from previous slide!

Allowing a type variable to be instantiated to a wildcard

- I.e. Set<?> accepted for Set<T>
- Normally an unsafe conversion but compiler can judge if it's safe in some special cases

Generic Method vs Generic Class

Generic methods allow type parameters to be used to express dependencies among the types of one or more arguments to a method and/or its return type. If there isn't such a dependency, a generic method should not be used

```
// Ok, express dependencies between params
public class Collections {
    public static <T> void copy(List<T> dest, List<?
                                              extends T> src){
       ...

    }
   ...
```

Type Erasure

During compilation generic types are erasured

- The erasure of a parameterized type (§4.5) G is |G|.
- The erasure of a nested type T.C is |T|.C.
- The erasure of an array type T[] is |T|[].
- The erasure of a type variable (§4.4) is the erasure of its leftmost bound.
- The erasure of every other type is the type itself.

So List<String> becomes List (there's just one class shared by all parameterized types)

Reifiable Types

"Because some type information is erased during compilation, not all types are available at run time. Types that are completely available at run-time are known as ***reifiable types****.*

A type is reifiable if and only if one of the following holds

- *It refers to a non-generic class or interface type declaration*
- *...."* // JLS 4.7

Baaad news! Generics gone after compilation

What's Happening?*

A lot happens in background

- All type parameters replaced by their bounds
- Compiler inserts cast to preserve type safety
- Compiler possible generates bridge methods to preserve polymorphism in extended generic types

Many restriction for the programmer...

- Java has got a lot of criticism for the implementation of generics
- Cause: Backward compatibility

Type Erasure: Consequences*

1. Cannot create Instances of type parameters
2. Cannot declare static fields whose types are type parameters
3. Cannot use casts and instanceof with parameterized types
4. Cannot create arrays of parameterized types
5. Cannot create, catch, or throw objects of parameterized types
6. Cannot overload a method where the formal parameter types of each overload erase to the same raw type

Arrays and Generics*

Generic arrays forbidden (except <?>)

Doesn't work well together

- Arrays covariant, generics not
- Arrays reifiable, generic not
- .. don't mix (i.e. use Collection, List, Set...)

Override and Overload*

Overload possible but have to check erasure

- Possible same parametertypes after erasure, so same signature, compile error
- Some subtleties ...

Override possible

- Subtleties ...

Canonical Form*

Equals no problems

Clone must use reflection (upcoming...)

Pattern Matching

Pattern matching is a powerful feature

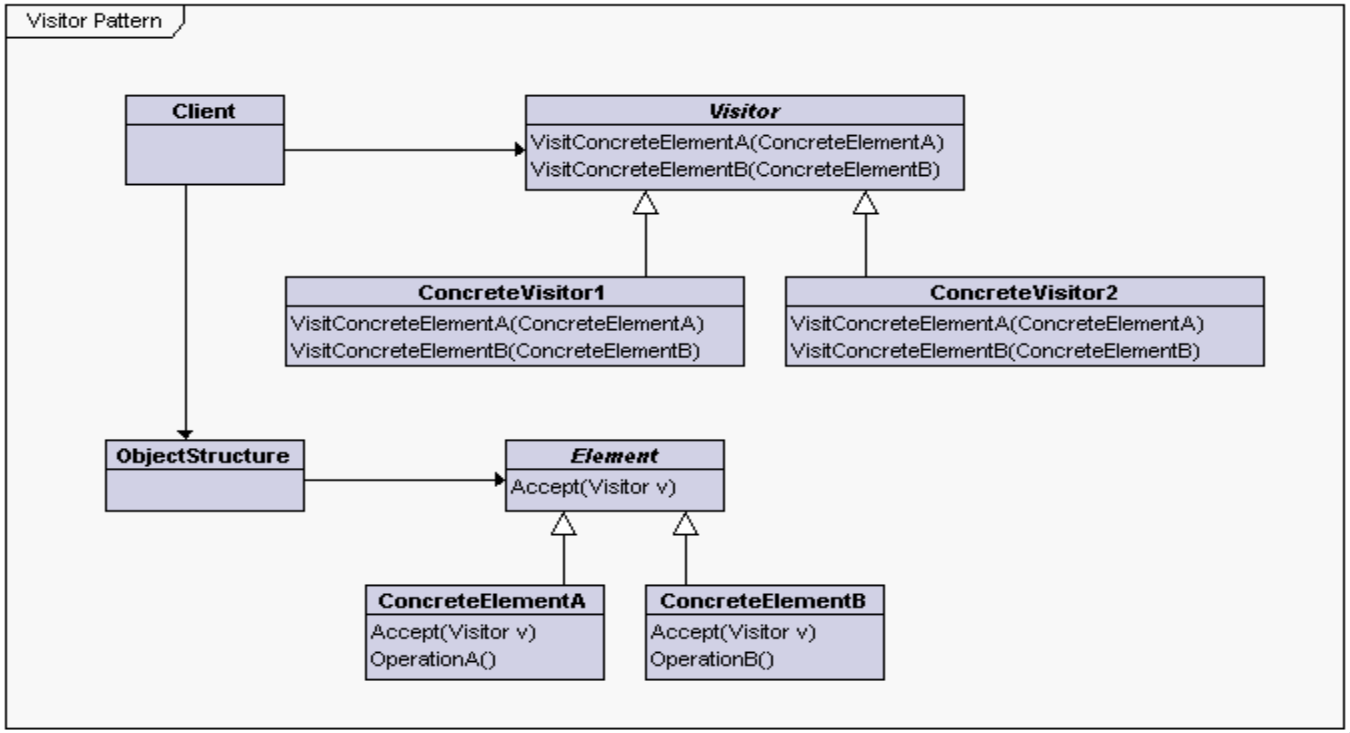
```
// Have this in Haskell, how to in Java?
data Tree t = Leaf | Node t (Tree t) (Tree t)
     deriving (Eq,Ord,Show)


myTree = (Node 12 (Node 11 Leaf Leaf) (Leaf))


sumTree:: Tree t -> int
sumTree Leaf = 0
sumTree (Node a t1 t2) = a + sumTree(t1) + sumTree(t2)
```

Visitor*

One possible implementation of pattern matching (separating an algorithm from an object structure on which it operates)



Reflection

*"In computer science, **reflection** is the ability of a computer program to examine (see type introspection) and modify the structure and behavior (specifically the values, meta-data, properties and functions) of an object at runtime". //* Wikipedia

Java Reflection API

The java.lang.Class<T> object

Class has no public constructor. Instead Class objects are constructed automatically by the Java Virtual Machine as classes are loaded and by calls to the defineClass method in the class loader [we will not talk about class loaders].

Classes from java.lang.reflect package

- Field, Method, Constructor<T>, AnnotatedElement,...

Usage of Reflection

Pros

- Extensibility Features: An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names (i.e. given a String create some object) and more...

Cons

- Performance Overhead, Security Restrictions, Exposure of Internals

Reflection is special, avoid!

Examples Reflection

- Inspecting objects*
- Modify objects (change access temporarily)*
- Instantiation without constructor*

Usage of Reflection in Frameworks*

Downloading plugins (*.class-files) from anywhere (web) and update (running) application with new functionality

Summary

- Generics very comfortable with Collections

Normal usage non problematic

- Generics complicates the type system, (bounded) wildcards, compatibility,...

- Wildcards imposes restrictions on collections

- Reflection is an advanced technique for modifying a running program