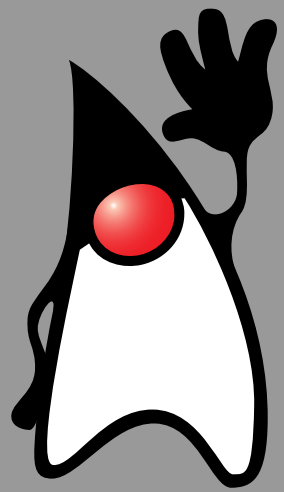
EPITA Première Année Cycle Ingénieur
Atelier Java - J2

Marwan Burelle

marwan.burelle@lse.epita.fr
http://www.lse.epita.fr



Plan

Plan

A Visitor

- Example of a classical visitor pattern
- We need to break some circular dependencies
- We heavily rely on interfaces

Visitor: Example

Example:

```
public interface Elmts {
    public String getName();
}

public interface VisitorInterface {
    void visit(IntElmts e);
    void visit(FloatElmts e);
    void visit(ConcreteElmts e);
}
```

Visitor: Example

Example:

```
public class ConcreteElmts implements Elmts {
    private String      _name;
    public String getName() { return _name; }

    public void Accept(VisitorInterface v) {
        v.Visit(this);
    }
}

public class IntElmts extends ConcreteElmts {
  private int        f;
  public String getName() {return Int.toString(f);}
}

public class FloatElmts extends ConcreteElmts {
  private float        f;
  public String getName(){return Float.toString(f);}
}
```

Abstract Classes

- Abstract classes are a mix between interfaces and classes
- Abstract classes contain complete methods definition as do classes
- Abstract classes define *contract*: abstract methods that are describe but not implemented
- Somehow an interface is an abstract class with only abstract methods.
- Abstract classes can't be instantiated
- Abstract classes can be inherited as other classes (but derived classes must implement abstract methods or be abstract themselves.)

Usage of Abstract Classes

- Abstract classes are used to provide standard concrete code relying on not yet known specific behavior
- Most of the time, the abstract class will (concretely) defines some public methods implementing a generic activity (display, serialization . . .) that need specific *properties* to work (such as extracting object state in format suitable for the activity . . .)
- In Java API abstract classes are used (in conjunction with interfaces) to provide skeleton code for design patterns or any other *ready-to-use* generic classes.

Abstract Class (example)

Example:

```
public abstract class AbstractInfo {
    abstract String get_name();
    abstract String get_rev_date();
    abstract String get_authors();

    public void print() {
        System.out.println("Object "
            + this.get_name()
            + "information:"
        );
        System.out.println("  revision date: "
            + this.get_rev_date()
        );
        System.out.println("  authors: "
            + this.get_authors()
        );
    }
}
```

Abstract Class (example)

Example:

```
public class ConcreteInfo extends AbstractInfo {
    String                    authors;
    String                    date;
    String                    name;
    String get_name()      { return name;    }
    String get_authors()   { return authors; }
    String get_rev_date()  { return date;    }
}
```

Plan

Plan

Cast is Evil

- When we need generic containers, we often use the fact that any class is a subclass of `Object`:

Example:

```
public class Box {
  private Object obj;
  public void add(Object o){
    this.obj = o;
  }
  public Object get(){
    return obj;
  }
}
```

Cast is Evil

- When using the previous generic container, we need to do some cast ... guess what happen ?

Example:

```
Box integerBox = new Box();

integerBox.add(new Integer(10));
Integer someInteger = (Integer)integerBox.get();
System.out.println(someInteger);

integerBox.add("10");
Integer someInteger2 = (Integer)integerBox.get();
System.out.println(someInteger2);
```

Plan

A Solution: Generics

- Generics are type variables explicitly introduced in class or method declarations;
- Generics allow a solution to our previous example:

Example:

```
public class Box<T>{
  private T obj;
  public void add(T o){
    this.obj = o;
  }
  public T get(){
    return obj;
  }
}
```

A Solution: Generics

Example:

```
Box<Integer> integerBox = new Box<Integer>();

integerBox.add(new Integer(10));
Integer someInteger = integerBox.get();// No cast!
System.out.println(someInteger);

integerBox.add("10");//Type Error Here!
Integer someInteger2 = integerBox.get();
System.out.println(someInteger2);
```

- **Now the miss-use is detected at compile time !**

Generics

- Enhancement of types system (fully static;)
- Pure typing syntax: no rewriting of class nor dynamic type checking;
- **Type Erasure Semantics:** Every type variables are eliminated after type checking;
- Enforce typing on generic containers (basis of the Java Collection;)
- **Well founded type discipline:** *well typed programs can not go wrong* (at least for that part;)

Plan

Relation to ML Polymorphism

- Generics are very similar to polymorphic types of ML;
- While ML type inference automatically provides polymorphism, Generics have to be explicit;
- Generics provides similar generic structured types as in ML:

Example:

An OCaml example of list type:

```
type 'a myList = Nil | Cons of 'a * 'a myList
```

A similar definition using Java Generics:

```
public class myList<T> {
  public T elem;
  public myList<T> next;
}
```

Plan

Generic Methods

- Methods can also be generic:

Example:

```
public class myBoxGen<T> {
  private T t;
  public void add(T o){
    t = o;
  }
  public <U> U print(U u){
    System.out.println("T: " + t);
    System.out.println("U: " + u);
    return u;
  }
}
```

Generic Methods

Example:

```
public class myBoxGen<T> {

 // see previous slide

 public static void main(String[] args){
   MyBoxGen<Integer> mbox = new MyBoxGen<Integer>();
   mbox.add(new Integer(10));
   String s = mbox.print("Some text.");
 }
}
```

Plan

System F

- System F is the first polymorphic variant of $\lambda-$calculus;
- Polymorphism is expressed using type level abstractions:

$$\lambda X.\ \lambda x{:}X.x$$

- The previous expression has type:

$$\forall X.X \to X$$

- System F can easily be extended with subtyping by adding a well founded partial order on types: $t \leq s$
- Generics just introduce F like polymorphism in Java type system: the generic method
`<T> void add(T t)` can be expressed in F with:

$$\texttt{let add} = \lambda T\ .\ \lambda t{:}T\ .\ \texttt{this.o} \leftarrow t$$

Constraints on Generics

Example:

```
public class NBox<T extends Number>{
  private T t;
  public void add(T o){
    t = o;
  }
  public int intProd(T s){
    return(t.intValue() * s.intValue());
  }
  public static void main(String[] args){
    NBox<Integer> intBox = new NBox<Integer>();
    intBox.add(new Integer(10));
    int x = intBox.intProd(new Integer(5));
    System.out.println(x);
  }
}
```

Bounded Polymorphism

- Generics in Java is related to system *F bounded*;
- In F Bounded:

$$\lambda\ T{<}t.\ \lambda\ x{:}T.\ x$$

has type

$$\forall T[T{<}t].T \to T$$

- Bounded Polymorphism offers genericity with constraints, enforcing properties on type variables (existence of methods or fields for instance;)
- The subtyping relation is general: `extends`, in this context, applies to class and interface;
- Bounded Polymorphism offers the same safety as without bounds;

Bounded Polymorphism

Example:

```
public class SomeClass{
  public static <T extends Number> T print(T t){
    System.out.println("T: " + t);
    return t;
  }
  public static void main(String[] args){
    // No cast needed !
    Float f = SomeClass.print(new Float(1.0));
    // Type error !
    String s = SomeClass.print("Some text.");
  }
}
```

```
SomeClass.java:8: <T>print(T) in SomeClass cannot be applied to (java.lang.String)
    String s = SomeClass.print("Some text.");
```

Plan

Subtyping: contravariance is back !

- Contravariance expresses reverse subtyping constraint;
- The question: is `Box<T>` $\leq$ `Box<S>` when `T` $\leq$ `S` ?
- The answer is **NO** !
- The contravariance rule gives us the reason: `Box<X>` can be seen as a system F expression λ `X.Box<X>` having *type* $\forall$`X. X` $\rightarrow$ `Box<X>`
  Contravariance says:

$$t \leq s \Leftrightarrow s \rightarrow u \leq t \rightarrow u$$

- Thus we have:

$$\texttt{S} \leq \texttt{T} \Leftrightarrow \texttt{Box<T>} \leq \texttt{Box<S>}$$

Contravariance is back !

- Why contravariance ?
- A function is of type $t \rightarrow u$ if it accepts any value of type t (and return a value of type u when given a value of type t;)
- If $s \rightarrow u \leq t \rightarrow u$, then any function of type $s \rightarrow u$ can be used where a function of type $t \rightarrow u$ is expected;
- What is expected of a function is defined by call-site context, namely what kind of arguments can be passed to a function: so saying that $s \rightarrow u$ can be used where a function of type $t \rightarrow u$ can used means that argument passed to that function can be of type t;
- Thus a function of type $s \rightarrow u$ is expected to accept any value of type t, which means that $t \leq s$!
- While this result seems surprising, it is important when dealing with subtyping and function (or, in our case, type function;)

Contravariance for the Dummies

- Let's have a class `Instrument` for music instruments and two subclass `Guitar` and `Drum`;
- We define a generic class `FlyCase<I extends Instrument>` for instruments' fly-cases;
- We can define a generic fly-case `FlyCase<Instrument>` and two specific fly-cases `FlyCase<Guitar>` and `FlyCase<Drum>`;
- If `FlyCase<Guitar>` were a subclass of `FlyCase<Instrument>`, then we it will be possible to put anything we can put in generic fly-case in it, say . . . a drum . . .

Contravariance for the Dummies(2)

Example:

```
public interface Instrument {
    public void play(char note, int scale);
}
public class Guitar implements Instrument {
    public void play(char note, int scale) {
        System.out.println("Playing: "+note+scale);
    }
}
public class Drum implements Instrument {
    public void play(char note, int scale) {
        System.out.println("BOOM!");
    }
}
public class FlyCase<I extends Instrument> {
    I instrument;
    public void put(I i) { instrument = i; }
    public I take() { return (instrument); }
}
```

Contravariance for the Dummies(3)

Example:

```
public class Roadie {
    FlyCase<Instrument> box;

    void takeBox(FlyCase<Instrument> b) { box = b; }
    void pack(Instrument i) { box.put(i); }
    Instrument unpack() { return (box.take()); }

    public static void main(String[] args) {
        Roadie r = new Roadie();
        Guitar lesPaul = new Guitar();
        Drum   pearl = new Drum();
        FlyCase<Guitar> fcg = new FlyCase<Guitar>();
        r.takeBox(fcg); // Type error !
        r.pack(pearl);  // This should work
    }
}
```

```
Roadie.java:13: takeBox(FlyCase<Instrument>) in Roadie cannot
be applied to (FlyCase<Guitar>)
       r.takeBox(fcg); // Type error !
        ^
```

Plan

Wildcards

- Sometimes we need to keep the type parameter of a class *unknown* (*i.e.* we don't need to know it;)
- Wildcards offers a way to do so without contradiction to the contravariance rule;

Example:

```
// Work only for Collection of Object
public void printColl(Collection<Object> c){
  for (Object e : c)
    System.out.println(e);
}
// With wildcard, we can have a real generic method
public void printColl(Collection<?> c){
  for (Object e : c)
    System.out.println(e);
}
```

Wildcards

- `?` stands for unknown type, thus we can use it for *reading* (as in previous example) but not for *writing*:

```
Collection<?> c = new ArrayList<String>();
c.add(new Object()); // Compile time error
```

- Bounded wildcards are possible:
  `<? extends Number>` represents any *unknown* sub-type of `Number`;
- Bounded wildcards can be used also to give lower bound: `<? super Integer>` represents any *unknown* super-type of `Integer`;

Plan

Plan

Collections ?

- **Collections** is a tool-box for managing objects aggregates;
- Java Collection provide a set of generic **interfaces**, a set of **implementations** of these interfaces and a set of generic **algorithms** on collections;
- Java Collection heavily uses **Generics**;

From Java Tutorial:

*A **collection** - sometimes called a container - is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data.*

Java Collection

From Java Tutorial:

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

- **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
- **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be *polymorphic:* that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

Plan

Collection Core Interface

```
public interface Collection<E> extends Iterable<E> {
 // Basic operations
 int size();
 boolean isEmpty();
 boolean contains(Object element);
 boolean add(E element);          //optional
 boolean remove(Object element); //optional
 Iterator<E> iterator();

 // Bulk operations
 boolean containsAll(Collection<?> c);
 boolean addAll(Collection<? extends E> c); //optional
 boolean removeAll(Collection<?> c);        //optional
 boolean retainAll(Collection<?> c);        //optional
 void clear();                              //optional

 // Array operations
 Object[] toArray();
 <T> T[] toArray(T[] a);
}
```

Collection Core Interface

- **for-each traversal:**

```
for (Object o : collection)
  { // for each o in collection }
```

- **Iterators:**

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

- `containsAll` - returns true if the target Collection contains all of the elements in the specified Collection.
- `addAll` - adds all of the elements in the specified Collection to the target Collection.
- `removeAll` - removes from the target Collection all of its elements that are also contained in the specified Collection.
- `retainAll` - removes from the target Collection all its elements that are not also contained in the specified Collection.
- `clear` - removes all elements from the Collection.

Plan

The Set Interface

- `Set` models the mathematical set abstraction (*it is meant to . . .* ;)
- A `Set` is a Collection that cannot contains duplicate elements;
- It only contains methods from Collection Interface;
- `Set` has a stronger contract on the behavior of the `equals` and `hashCode` operations, so that `Set` can be compared regardless of the chosen implementation;
- Two `Set` instances are equal if they contain the same elements.

The List Interface

- A `List` is an *ordered* collection and may contains duplicate elements;
- Adds *Positional Access*, *Search*, *Iteration* and *Range View*

```
public interface List<E> extends Collection<E> {
 // Positional access
 E get(int index);
 E set(int index, E element);
 boolean add(E element);
 void add(int index, E element);
 E remove(int index);
 boolean addAll(int index, Collection<? extends E> c);
 // Search
 int indexOf(Object o);
 int lastIndexOf(Object o);
 // Iteration
 ListIterator<E> listIterator();
 ListIterator<E> listIterator(int index);
 // Range-view
 List<E> subList(int from, int to);
}
```

The Queue Interface

- A Queue is a collection for holding elements prior to processing.
- Queue typically, but not necessarily, orders elements in a FIFO (first-in-first-out) manner.
- Queue provides additional insertion, removal, and inspection operations with versions throwing exceptions;

```
public interface Queue<E> extends Collection<E> {
  E element();          // Throws exception
  void add(E e);        // Throws exception
  E remove();           // Throws exception
  boolean offer(E e); // Returns special values
  E peek();             // Returns special value
  E poll();             // Returns special value
}
```

The Map Interface

A Map is an object that maps keys to values and cannot contains duplicate keys.

```
public interface Map<K,V> {
    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();
    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();
    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();
    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

Plan

Implementations

- **General-purpose implementations** are the most commonly used implementations, designed for everyday use.
- **Special-purpose implementations** are designed for use in special situations and display nonstandard performance characteristics, usage restrictions, or behavior.
- **Concurrent implementations** are designed to support high concurrency, typically at the expense of single-threaded performance.
- **Wrapper implementations** are used in combination with other types of implementations, often the general-purpose ones, to provide added or restricted functionality.
- **Convenience implementations** are mini-implementations, typically made available via static factory methods, that provide convenient, efficient alternatives to general-purpose implementations for special collections (for example, singleton sets).
- **Abstract implementations** are skeletal implementations that facilitate the construction of custom implementations.

General-purpose Implementations

- The following table lists the general-purpose implementations which are the most useful ones. Each Implementations is based on classical data representation like arrays, hash tables, trees or linked lists.
- `Queue` implementations are not listed here, but there are two main implementations: `LinkedList` (which are also `List`) and `PriorityQueue`. The former implements FIFO queues and the later queues ordered *w.r.t.* their values.

Interfaces	**Implementations**				
	Hash table	**Array**	**Tree**	**Linked list**	**Hash + Linked**
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap		LinkedHashMap

Plan

Algorithms

- Algorithms are polymorphic static functions that take a collection as first argument and operate classical and useful operation on that operation;
- Since many algorithms provided deal with order, they are focused on `List` rather than `Collection`, but some are also applicable to `Collection`.
- Various kind of algorithms are provided: sorting, shuffling, searching, composition, bound finding and routine data manipulation (reverse, fill, copy, swap, addAll).

Examples

```
public class DemoAlgo {
    public List<String> sl;
    public DemoAlgo(String[] args) {
        sl = Arrays.asList(args);
    }
    public void sort() {
        Collections.sort(sl);
    }
    public String max() { return Collections.max(sl); }
    public String min() { return Collections.min(sl); }
    public int freq(String s) { return Collections.frequency(sl, s); }

    public static void main(String[] args) {
        DemoAlgo        d = new DemoAlgo(args.clone());
        System.out.println(d.sl);
        d.sort();
        System.out.println(d.sl);
        System.out.println("max: " + d.max());
        System.out.println("min: " + d.min());
        System.out.println("freq(" + args[0] + "): " + d.freq(args[0]));
    }
}
```

Examples


```
> java DemoAlgo www zzz fff aaa bbb aaa www ggg
[www, zzz, fff, aaa, bbb, aaa, www, ggg]
[aaa, aaa, bbb, fff, ggg, www, www, zzz]
max: zzz
min: aaa
freq(www): 2
```

Examples

```
public class BinTree<T> {
    protected T                 key;
    protected BinTree<T>        lson;
    protected BinTree<T>        rson;
    public BinTree(T k) {
        key = k;
        lson = null;
        rson = null;
    }
    public BinTree(T k, BinTree<T> rs, BinTree<T> ls) {
        key = k;
        lson = ls;
        rson = rs;
    }
    public void bfPrint() {
        Queue<BinTree<T>> q = new LinkedList<BinTree<T>>();
        q.add(this);
        q.add(null);
        while (!q.isEmpty()){
            BinTree<T>  cur = q.remove();
            if (cur != null) {
                System.out.print(cur.key+" ");
                if (cur.lson != null)
                    q.add(cur.lson);
                if (cur.rson != null)
                    q.add(cur.rson);
            } else {
                System.out.println("");
                if (!q.isEmpty())
                    q.add(null);
            }
        }
    }
```

Examples

```
    public static <T> BinTree<T> build(int depth, Queue<T> k) {
        BinTree<T>      cur;
        if (depth > 0)
            cur = new BinTree<T>(k.remove(),
                                 build(depth - 1,k),
                                 build(depth - 1, k));
        else
            cur = new BinTree<T>(k.remove());
        return cur;
    }
    public static void main(String[] args) {
        Queue<Integer>  k = new LinkedList<Integer>();
        for (int i = 0; i<32; ++i)
            k.add(new Integer(i));
        BinTree<Integer> t = build(4,k);
        t.bfPrint();
    }
}
```

Examples

```
> java BinTree
0
16 1
24 17 9 2
28 25 21 18 13 10 6 3
30 29 27 26 23 22 20 19 15 14 12 11 8 7 5 4
```