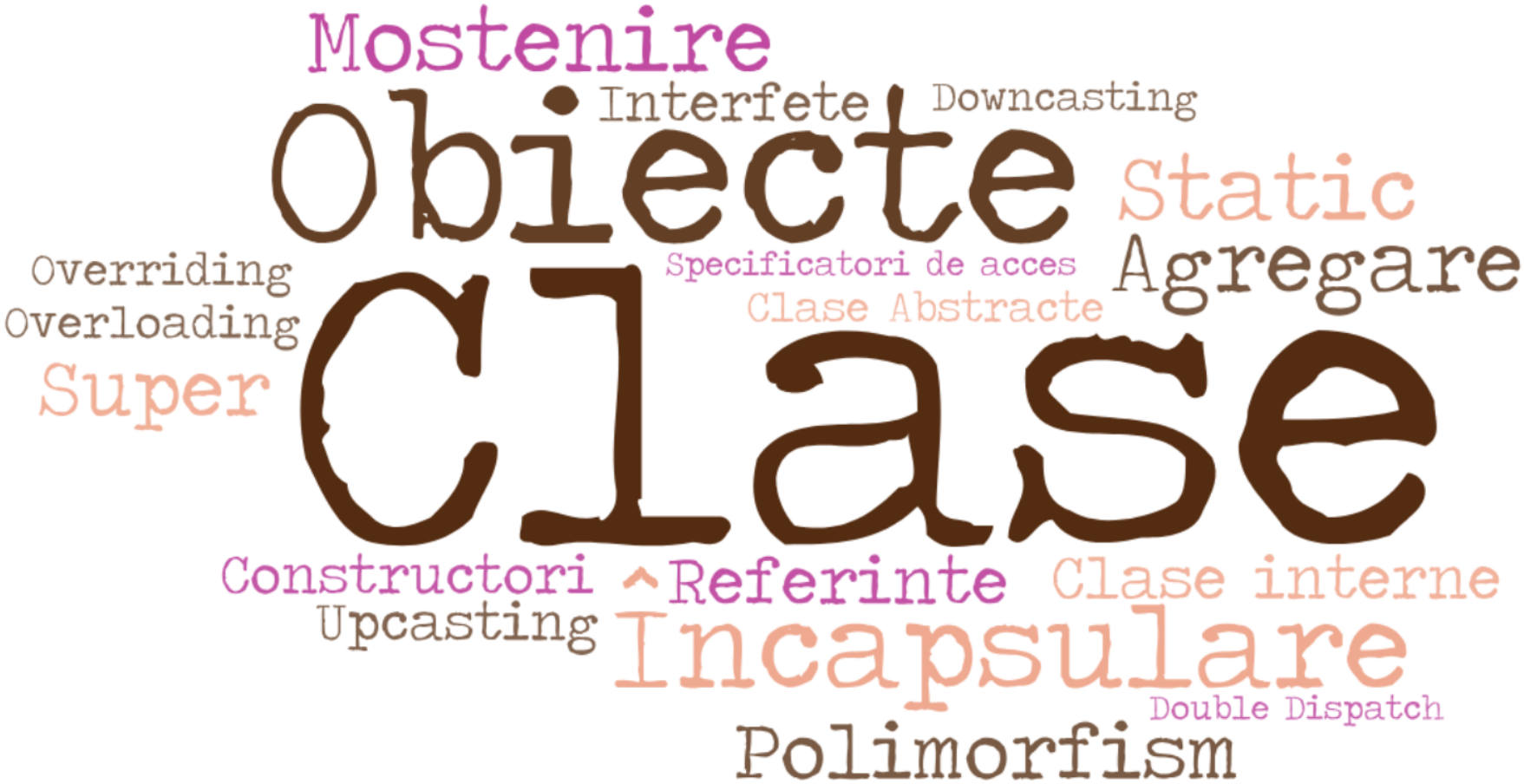
Genericitate (Generics / Templates)

Alexandru Olteanu

Universitatea Politehnica Bucuresti
Facultatea de Automatică si Calculatoare, Departamentul Calculatoare
alexandru.olteanu@upb.ro

OOP, 2020

Partea 1: Bazele teoretice ale OOP



Genericitate

Nevoia pentru genericitate

o singura implementare (structura de date, algoritm) pentru mai multe tipuri de date

```
List<Player> team = new ArrayList<Player>();
List<Integer> grades = new ArrayList<Integer>();
```

Nevoia pentru genericitate

o singura implementare (structura de date, algoritm) pentru mai multe tipuri de date

```
List<Player> team = new ArrayList<Player>();
List<Integer> grades = new ArrayList<Integer>();

Collections.sort(team);
Collections.sort(grades);
```

Nevoia pentru genericitate

daca as folosi un tip de baza cat mai general (void in C, Object in Java) pot aparea erori la rulare si nici codul nu este foarte clar:

```
List myIntList = new LinkedList();
myIntList.add(new Integer(0));
myIntList.add(new Player()); // which is bad
Integer x = (Integer) myIntList.iterator().next();
```

Nevoia pentru genericitate

daca as folosi un tip de baza cat mai general (void in C, Object in Java) pot aparea erori la rulare si nici codul nu este foarte clar:

```
List myIntList = new LinkedList();
myIntList.add(new Integer(0));
myIntList.add(new Player());
Integer x = (Integer) myIntList.iterator().next();
```

```
List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
myIntList.add(new Player());    //Eroare!
Integer x = myIntList.iterator().next(); // nici nu mai e
    nevoie de cast
```

Nevoia pentru genericitate

pentru metode, daca as supraincarca metoda cu fiecare tip de date, as duplica cod:

```
public class Collections {
  static boolean replaceAll(List<Integer> list,
      Integer oldVal, Integer newVal) {
    ... // o implementare
  }
  static boolean replaceAll(List<Player> list,
      Player oldVal, Player newVal) {
    ... // o alta implementare
  }
}
```

Nevoia pentru genericitate

pentru metode, daca as supraincarca metoda cu fiecare tip de date, as duplica cod:

```
public class Collections {
    static boolean replaceAll(List<Integer> list,
        Integer oldVal, Integer newVal) {
        ... // o implementare
    }
    static boolean replaceAll(List<Player> list,
        Player oldVal, Player newVal) {
        ... // o alta implementare
    }
    static <T>
    boolean replaceAll(List<T> list, T oldVal, T newVal) {
        ... // o singura implementare
    }
}
```

Definitia genericitatii

Genericitatea este un mecanism prin care tipurile folosite in definirea claselor / interfetelor si metodelor sa fie parametrizate.

Definitia genericitatii

Genericitatea este un mecanism prin care tipurile folosite in definirea claselor / interfetelor si metodelor sa fie parametrizate.

Acest mecanism se intalneste, ca principiu, in mai multe limbaje OOP: template in C++, generics in Java, C# si Objective-C.

Parametric polymorphism / Generics

Parametric polymorphism (...), allows a single piece of code to be typed "generically", using variables in place of actual types, and then instantiated with particular types as needed. Parametric definitions are uniform: all of their instances behave the same. (...)
B. Pierce, "Types and Programming Languages", MIT Press
Why is C++ said not to support parametric polymorphism?

Tipuri formale

La definire se foloseste tipul formal (e.g. T mai jos)

```
public class Erasure<T> {
    private T obj;
    Erasure(T o) { obj = o; }
    T getObj() { return obj; }
}
...
public static void main(String[] args) {
    Erasure<Integer> test = new Erasure<Integer>(10);
    System.out.println(test.getObj());
}
```

Type Erasure

La compilare se produce un singur cod pentru o clasa, tipul formal este substituit cu Object (alte limbaje implementeaza genericitatea diferit)

```
public class Erasure<T> {
    private T obj;
    Erasure(T o) { obj = o; }
    T getobj() { return obj; }
}
...
public static void main(String[] args) {
    Erasure<Integer> test = new Erasure<Integer>(10);
    System.out.println(test.getObj());
}
```

Incercati sa decompilati o clasa cu javap (folosind -c)

Type Erasure

Restrictii in definirea claselor generice:

- Cannot Create Instances of Type Parameters
- Cannot Create Arrays of Parameterized Types
- ▸ Restrictii

```
public class Erasure<T> {
    private T obj;
    Erasure() { obj = new T(); } // does not compile
}

public class GenericsErasure<T> {
    private T[] objs;
    Erasure() { objs = new T[10]; } // does not compile
}
```

Se poate totusi prin Reflection

Bridge methods

Cand o clasa extinde o clasa generica sau implementeaza o interfata generica:

```
public static class A<T> {
    public T getT(T args) {
        return args;
    }
}
public static class B extends A<String> {
    public String getT(String args) {
        return args;
    }
}
...
A a = new B();
a.getT(new Object()); // ClassCastException la runtime
    pentru ca...
```

Bridge methods

compilatorul produce o metoda sintetica (care nu apare in cod si nu poate fi apelata explicit):

```
public class B extends A<java.lang.String> {
  ...
  public java.lang.String getT(java.lang.String);
    Code:
       0: aload_1
       1: areturn
  public java.lang.Object getT(java.lang.Object);
    Code:
       0: aload_0
       1: aload_1
       2: checkcast     #2                  // class java/
           lang/String
       5: invokevirtual #3                  // Method getT:(
           Ljava/lang/String;)Ljava/lang/String;
       8: areturn
}
```

▸ Java Bridge Methods explained

Type Erasure

Atentie insa! la folosirea unei clase generice:

```
List<String> list = new ArrayList<String>();
list.add("foo");
String x = list.get(0);
```

se va substitui cu

```
List list = new ArrayList();
list.add("foo");
String x = (String) list.get(0);
```

Tipuri formale

Pot avea mai multe tipuri formale intr-o definitie si pot avea tipuri formale imbricate:

```
public interface Map<K,V>{
    static interface Map.Entry<K,V>;
    Set<Map.Entry<K,V>> entrySet();
    ...
}
```

Tipuri formale

Some things may make you frown:

```
public interface Map<K,V>{
    static interface Map.Entry<K,V>;
    Set<Map.Entry<K,V>> entrySet();
    V get(Object key); // desi in C# este 'V Get(K k);'
    ...
}
```

Genericitatea in subtipuri

Este List de Strings o List de Objects?

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
```

Genericitatea in subtipuri

Nu, pentru ca ar duce la erori:

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;    // eroare de compilare
lo.add(new Object());
String s = ls.get(0);
```

Wildcards

Ce fac daca vreau sa definesc o metoda care sa ia ca parametru lista de orice fel de elemente? (nu pot folosi List<Object>)

```
void printList(List<?> l) {
    for (Object e : l) {        // e ok
        System.out.println(e);
    }
}
```

Wildcards

Ce fac daca vreau sa definesc o metoda care sa ia ca parametru lista de orice fel de elemente? (nu pot folosi List<Object>)

```
void printList(List<?> l) {
    for (Object e : l) {        // e ok
        System.out.println(e);
    }
    l.add(new Object()); // eroare de compilare
}
```

Bounded Type Parameters and Wildcards

Ce fac daca vreau sa limitez listele pe care le pot primi ca parametru la liste de elemente de un anumit fel?

```
void sayHello(List<? extends Man> l) {
    for (Object e : l) {
        System.out.println("Hello "+e.getName());
    }
}
```

Bounded Type Parameters and Wildcards

Ce fac daca vreau sa limitez listele pe care le pot primi ca parametru la liste de elemente de un anumit fel?

```
void sayHello(List<? extends Man> l) {
    for (Object e : l) {
        System.out.println("Hello "+e.getName());
    }
}
```

Tipurile formale si wildcards pot fi upper bounded (? extends T) si lower bounded (? super T):

```
public class Collections {
    static <T extends Comparable<? super T>>
    void sort(List<T> list) {
      ... // o singura implementare
    }
}
```

Bounded Type Parameters and Wildcards

Ce fac daca vreau sa limitez listele pe care le pot primi ca parametru la liste de elemente de un anumit fel?

```
void sayHello(List<? extends Man> l) {
    for (Object e : l) {
        System.out.println("Hello "+e.getName());
    }
}
```

Tipurile formale si wildcards pot fi upper bounded (? extends T) si lower bounded (? super T):

```
public class Collections {
    static <T extends Comparable<? super T>>
    void sort(List<T> list) {
      ... // o singura implementare
    }
}
```

▸ Explicatie pe StackOverflow

Type Erasure

Daca tipul formal este upper bounded, va fi substituit cu upper bound

```
public class Erasure<T extends Number> {
    private T obj;
    Erasure(T o) { obj = o; }
    T getobj() { return obj; }
}
...
public static void main(String[] args) {
    Erasure<Integer> test = new Erasure<Integer>(10);
    System.out.println(test.getObj());
}
```

Incercati sa decompilati o clasa cu javap

Metode generice

Metodele generice permit folosirea de parametrii formali pentru a exprima dependenta intre parametrii si/sau intre parametrii si rezultat:

```
// Metoda corecta
static <T> void arrayToCollection(T[] a, Collection<T> c) {
    for (T o : a) {
        c.add(o);
    }
}
```

Metode generice

Metodele generice permit folosirea de parametrii formali pentru a exprima dependenta intre parametrii si/sau intre parametrii si rezultat:

```
// Metoda corecta
static <T> void arrayToCollection(T[] a, Collection<T> c) {
    for (T o : a) {
        c.add(o);
    }
}
```

```
// Metoda incorecta: de ce?
static void arrayToCollection(Object[] a, Collection<?> c) {
    for (Object o : a) {
        c.add(o);
    }
}
```

Metode generice

Metodele generice permit folosirea de parametrii formali pentru a exprima dependenta intre parametrii si/sau intre parametrii si rezultat:

```
// Metoda corecta
static <T> void arrayToCollection(T[] a, Collection<T> c) {
    for (T o : a) {
        c.add(o);
    }
}
```

```
// Metoda incorecta: de ce?
static void arrayToCollection(Object[] a, Collection<?> c) {
    for (Object o : a) {
        c.add(o);  // eroare de compilare
    }
}
```

Metode generice

La folosire, compilatorul deduce automat tipul formal

```
static <T> void arrayToCollection(T[] a, Collection<T> c) {
    for (T o : a) {
        c.add(o); // Correct
    }
}

String[] sa = new String[100];
Collection<String> cs = new ArrayList<String>();

// T inferred to be String
arrayToCollection(sa, cs);
```

Metode generice

Putem sa avem metoda generica, chiar daca clasa nu este generica. Daca si clasa este generica, a nu se confunda tipul formal al metodei generice cu tipul formal al clasei

```
public interface Collection<E> extends Iterable<E> {
  boolean add(E e);
  boolean addAll(Collection<? extends E> c);
  <T> T[] toArray(T[] a); // programmer controls the type
}
```

Metode generice

Putem sa avem metoda generica, chiar daca clasa nu este generica. Daca si clasa este generica, a nu se confunda tipul formal al metodei generice cu tipul formal al clasei

```
public interface Collection<E> extends Iterable<E> {
  boolean add(E e);
  boolean addAll(Collection<? extends E> c);
  <T> T[] toArray(T[] a); // programmer controls the type
}
```

```
List<Integer> myList = new ArrayList<Integer>();
Integer[]myArray=myList.toArray(new Integer[myList.size()]);
Number[] myArray=myList.toArray(new Number[myList.size()]);
Object[] myArray=myList.toArray(new Object[myList.size()]);
```

Metode generice

Putem sa avem metoda generica, chiar daca clasa nu este generica. Daca si clasa este generica, a nu se confunda tipul formal al metodei generice cu tipul formal al clasei

```
public interface Collection<E> extends Iterable<E> {
  boolean add(E e);
  boolean addAll(Collection<? extends E> c);
  <T> T[] toArray(T[] a); // programmer controls the type
}
```

De ce nu <E> E[] toArray(E[] a);?

Metode generice

Putem sa avem metoda generica, chiar daca clasa nu este generica. Daca si clasa este generica, a nu se confunda tipul formal al metodei generice cu tipul formal al clasei

```
public interface Collection<E> extends Iterable<E> {
  boolean add(E e);
  boolean addAll(Collection<? extends E> c);
  <T> T[] toArray(T[] a); // programmer controls the type
}
```

De ce nu <E> E[] toArray(E[] a);?
pentru ca vreau sa pot scoate Number[] din List<Integer>

Metode generice

Putem sa avem metoda generica, chiar daca clasa nu este generica. Daca si clasa este generica, a nu se confunda tipul formal al metodei generice cu tipul formal al clasei

```
public interface Collection<E> extends Iterable<E> {
  boolean add(E e);
  boolean addAll(Collection<? extends E> c);
  <T> T[] toArray(T[] a); // programmer controls the type
}
```

De ce nu <T super E> T[] toArray(T[] a)?

Metode generice

Putem sa avem metoda generica, chiar daca clasa nu este generica. Daca si clasa este generica, a nu se confunda tipul formal al metodei generice cu tipul formal al clasei

```
public interface Collection<E> extends Iterable<E> {
  boolean add(E e);
  boolean addAll(Collection<? extends E> c);
  <T> T[] toArray(T[] a); // programmer controls the type
}
```

De ce nu <T super E> T[] toArray(T[] a)?
pentru ca as putea incerca sa scot un Integer[] din List<Number>

Limitari pentru generics

Why isn't Collection.remove(Object o) generic?
Josh Bloch and Bill Pugh (some of the guys who worked on generification) refer to this issue in:

- Java Puzzlers IV: The Phantom Reference Menace, Attack of the Clone, and Revenge of The Shift.

De citit

- Lab 8: Colectii
- Lab 9: Genericitate
- Why is C++ said not to support parametric polymorphism?