# A programozás alapjai 3.

## Lambda Java-ban

*Ez az oktatási segédanyag a Budapesti Műszaki és Gazdaságtudományi Egyetem oktatója által kidolgozott szerzői mű. Kifejezett felhasználási engedély nélküli felhasználása szerzői jogi jogsértésnek minősül.*

*Goldschmidt Balázs*
*balage@iit.bme.hu*

1

## *Interfészek bővítése*



2

## Interfészek újra

- Az interfészek nevesített metódusfejléc-halmazok
  - nincs implementáció
  - publikus a hozzáférés
- Interfészek között lehet többszörös öröklés
  - az osztályok is megvalósíthatnak több interfészt
- Céljuk a csatoltság csökkentése
  - elkülönül az interfész és az implementáció



## Interfész példa és feladat

- Definiáljunk egy *Stack* interfészt
  - *push*, *pop*, *top*

```
public interface Stack<T> {
    T top();
    T pop();
    void push(T t);
}
```

- Bővítsük a *Stack*-et új metódussal!
  - *dup*: a legfelső elemet duplikálja

```
public interface Stack2<T>
extends Stack<T> {
    void dup();
}
```

# Interfész bővítése *default*-tal

- Mit csinál a *dup*?

```
public void dup() {
    push(top());
}
```

- Mit tegyünk ha *StackImpl* már megvalósítja *Stack*-et?
  - a *StackImpl* implementálja mostantól *Stack2*-t
    - *Stack*-ként is használható
    - de meg kell valósítsa az új metódust
  - jó lenne, ha default implementációt adhatnánk *dup*-hoz



# Interfész bővítése *default*-tal

- Egy metódus alapértelmezett megvalósítása

```
public interface Stack<T> {
    T top();
    T pop();
    void push(T t);
    default void dup() {
        push(top());
    }
}
```

# Metódusok *default* módosítóval

- Előny
  - a megvalósító osztályban nem kell újra implementálni
  - a meglevő interfészek egyszerűen bővíthetők
    - meglevő implementációkat nem kell módosítani
- Hátrány
  - metódustörzs az interfészben ☹
  - nem igazán OO megoldás



# Default metódusok szabályai

- Leszármazott interfészben
  - ha nem említjük
    - megtartja az ős implementációját
  - ha újradeklaráljuk (nem default-ként)
    - asbtract metódus lesz
  - ha újradefiniáljuk (default-ként)
    - az új definíció lesz érvényes
- Megvalósító osztályban
  - a default metódusok felüldefiniálhatók

## Statikus metódus interfészben

- Segédfüggvények definiálásához

```
public interface X {
    static int foo(String s) {
        return s.length();
    }
}
```

  - only used with interface name
    - e.g. X.foo("hello")
  - nincs objektumpéldány, osztály, stb.
    - egyszerűsíti a utility osztályok létrehozását



## Példa: *Comparator* interfész

- `interface java.util.Comparator<T>`
  - `int compare(T o1, T o2)`
    *eredeti metódus, amit meg kell valósítani*
  - `default Comparator<T> reversed()`
    visszaad egy ellentétesen komparáló *Comparator*-t
  - `static <T extends Comparable<? super T>> Comparator<T> naturalOrder() / reverseOrder()`
    olyan komparátor, ami természetes vagy fordított rendezést ad *T*-re
  - `static <T> Comparator<T> nullsFirst(Comparator<? super T> cmp)`
    `static <T> Comparator<T> nullsLast(Comparator<? super T> cmp)`
    bővített komparátor, *null* értékek elöl/hátul, egyébként mint *cmp*

## Lambda alapjai



# Problémafelvetés

- Rendezzük hallgatók listáját!

```
public class Student {
   private String name;
   private LocalDate birthDate;
   private double average;
   // + getters-setters

   public Student(String na, LocalDate bd, double a) {
      name = na; birthDate = bd; average  = a;
   }
   public String toString() {
      return name+" "+birthDate.getYear()+", "+average;
   }
}
```

# Klasszikus megoldás

- Explicit komparáló osztály

```
public class NComp implements Comparator<Student> {
   public int compare(Student s0, Student s1) {
      return s0.getName().compareTo(s1.getName());
   }
}
```

```
List<Student> l = new ArrayList<Student>();
l.add(new Student("Kenneth McCormick",
             LocalDate.of(1987, 3, 13), 5.0));
l.add(new Student("Kyle Broflovski",
             LocalDate.of(1987, 7, 4), 4.1));
l.add(new Student("Eric Cartman",
             LocalDate.of(1987, 8, 13), 2.3));

Collections.sort(l, new NComp());
```



# Másik megoldás

- Anoním osztály (INKÁBB NE!)

```
Collections.sort(
  l,
  new Comparator<Student>() {
     public int compare(Student s0, Student s1) {
       return s0.getName().compareTo(s1.getName());
     }
  }
);
```

# Lambda alapú megoldás

- Java v8 óta
  - OO és más nyelvekben terjedőben
    - eredetileg funkcionális és deklaratív nyelvekben
    - C++1X, C#, python stb. támogat ilyet

```
Collections.sort(l,
    (s0,s1) -> s0.getName().compareTo(s1.getName())
);
```



# Java lambda alapjai

- *Függvény-interfészek (function interface)*
  - olyan interfész, aminek egyetlen abstract metódusa van
- Lambda-kifejezés
  - függvény-interfészű paraméterek helyén
  - szintaxis:
    - `(`*`paraméterek`*`) -> (`*`kifejezés`*`)`
    - `(`*`paraméterek`*`) -> {` *`metódustörzs`* `}`
  - közvetlenül hozzáfér a tartalmazó környezet változóihoz, metódusaihoz, attribútumaihoz

# Lambda fordítása

- Forráskód

```
void sort(List<T> l, Comparator<T> c);
```

```
Collections.sort(l,
    (s0,s1) -> s0.getName().compareTo(s1.getName())
);
```

- Generált kód (automatikus)

```
class LambdaX implements Comparator<Student> {
    public int compare(Student s0, Student s1) {
        return s0.getName().compareTo(s1.getName());
    }
);
```

```
Collections.sort(l, new LambdaX());
```



# Lambda & Swing: OO

```
final class MyActionListener implements ActionListener {
  JTextField t;
  public MyActionListener(JTextField tt) { t = tt;}
  public void actionPerformed(ActionEvent ae) {
    if (ae.getActionCommand().equals("date")) {
      t.setText((new Date()).toString());
    }
 }
}
```

```
...
JTextField t = new JTextField("Type here!");
JButton b = new JButton("Click Me!");
b.setActionCommand("date");
ActionListener al = new MyActionListener(t);
b.addActionListener(al);
...
```

# Lambda & Swing: lambda

```
final class MyActionListener implements ActionListener {
  JTextField t;
  public MyActionListener(JTextField tt) { t = tt;}
  public void actionPerformed(ActionEvent ae) {
  ...
```

(struck through in the original)

```
...
JTextField t = new JTextField("Type here!");
JButton b = new JButton("Click Me!");
b.setActionCommand("date");
ActionListener al = new MyActionListener(t);
b.addActionListener(
       ae -> t.setText((new Date()).toString())
);
...
```

(the line `ActionListener al = new MyActionListener(t);` is struck through in the original)

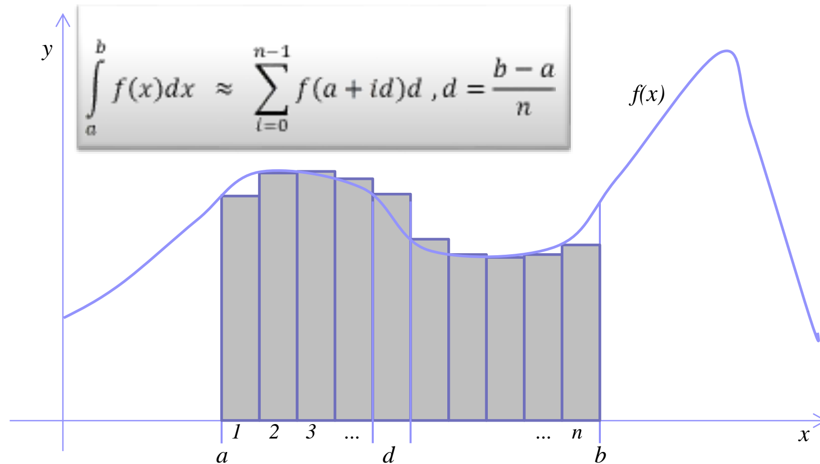
eléri a lokális változót



# Lambda és a metódushívások

- Integráljunk!
  - Számoljunk határozott integrált
    - téglalap-módszerrel
  - A függvények legyen objektumok
    - az elvárt metódus a *double apply(double d)*
    - vö. C++ functors (operator() felüldefiniálása)
  - Interfész kell hozzá
    - *Function*

# Integrál (téglalap-módszer)






# Integrál megvalósítása

```
// általános C implementációs ötlet
double sum = 0.0, d=(b-a)/n;
for (double x = a; x < b; x += d) {
   sum += f(x)*d; // a téglalap területe x-nél
}
```

```
// Java implementáció, utility class
public class Calculus {
  static double integrate(double a, double b,
                          int n, Function f) {
    double sum = 0.0, d = (b-a)/n;
    for (double x = a; x < b; x += d) {
      sum += f.apply(x)*d;
    }
    return sum;
  }
}
```

osztály vagy interfész kell

metódus neve bármi lehetne

# Integrálszámítás OO módon

```
public interface Function {
   double apply(double x);
}
```

elvárt interface

```
class Sin implements Function {
   public double apply(double x) {
      return Math.sin(x);
   }
}
```

példa implementáció

```
Sin s = new Sin();
double result =
   Calculus.integrate(0, Math.PI, 1000, s);
System.out.println(result); // 1.9999983550656881
```

objecktum-referencia



# Integrálszámítás OO módon


```
class Sin implements Function {
   public double apply(double x) {
      return Math.sin(x);
   }
}
```

példa implementáció

```
Sin s = new Sin();
double result =
   Calculus.integrate(0, Math.PI, 1000, s);
System.out.println(result); // 1.9999983550656881
```

objecktum-referencia


```
double result1 =
   Calculus.integrate(0, Math.PI, 1000,
               x->Math.sin(x)
   );
```

lambda kifejezés

# Lambda és metódus-referencia

```
double result1 =
   Calculus.integrate(0, Math.PI, 1000,
             x->Math.sin(x)
   );
```

lambda kifejezés

```
double result2 =
   Calculus.integrate(0, Math.PI, 1000,
             Math::sin
   );
```

statikus metódus referenciája

```
double result3 =
   Calculus.integrate(0, Math.PI, 1000,
             s::apply
   );
```

példány metódusának referenciája



# Miért "lambda"?

- Lambda kalkulus (λ-kalkulus)
  - Alonzo Church, 1936.
  - függvényeken végzett absztrakt műveletek
    - $\lambda$ jelzi a kifejezés kötött változóját, kb. fv.-paraméter pl.: $\lambda x[x+2]$
    - függvények paraméterei lehetnek más függvények
- Programozási nyelvekben
  - deklaratív és funkcionális nyelvekben jelent meg
    - lambda: anonim függvény, pl.: x->x+2
  - függvények és műveletek mint operandusok
    - paraméterként, visszatérési értékként

## *Haladó lambda*





# Függvény-kompozíció

- Hogyan építhetünk függvényeket?
  - *Math.sin* és *Math.abs* egyenként rendben vannak
    - Math::sin, Math::abs
  - *f(x) = abs(sin(x)) ???*

```
double result1 =
   Calculus.integrate(0, Math.PI, 1000,
             x->Math.abs(Math.sin(x))
   );
```

```
double result1 =
   Calculus.integrate(0, Math.PI, 1000,
             Math::abs # Math::sin
   );
```

# Függvény-kompozíció

- Hogyan építhetünk függvényeket?

```
double result1 =
   Calculus.integrate(0, Math.PI, 1000,
              Math::abs # Math::sin
   );
```

```
public class Compose implements Function {
   Function f1, f2;
   public Compose(Function f1, Function f2) {
      this.f1=f1; this.f2=f2;
   }
   public double apply(double x) {
      return f1.apply(f2.apply(x));
   }
}
```



# Függvény-kompozíció példa

```
// kombináció
Compose as = new Compose(Math::abs, Math::sin);
double result4 =
   Calculus.integrate(0,  Math.PI*2, 1000, as);
System.out.println(result4); // 3.9999868405188863
```

|sin(x)|

```
// lambda kifejezés használatával
Compose as2 = new Compose(x->x*x, Math::sin);
double result5 =
   Calculus.integrate(0,  Math.PI*2, 1000, as2);
System.out.println(result5); // 3.1415926535898726
```

$\sin^2(x)$

## Student rendezése ismét

- Rendezés lambdával

```
Collections.sort(l,
    (s0,s1) -> s0.getName().compareTo(s1.getName())
);
```

Komparálás lambda kif.-sel

- Rendezés getter metódussal?

```
Collections.sort(l,
    Student.comparing(Student::getName)
);
```

Getter-alapú komparálás

```
Collections.sort(l,
    Student.comparing(Student::getBirthDate)
);
```



## Getter-alapú rendezés

- Student bővítése...

A ***double getAverage()*** esetén is működik!

```
public class Student {
    public static
    Comparator<Student> comparing(Getter s) {
      return (Comparator<Student>)
         (o1,o2) -> s.get(o1).compareTo(s.get(o2));
    } // s.get(o) == o.getXXX()
    ...
}
```

itt hívjuk a gettert

```
interface Getter {
    Comparable get(Student s);
}
```

getter metódus interfésze

## Komparátorok kombinálása

- Rendezzünk *name*, *date*, *average* alapján!
  - az eddigi megoldás csak eggyel működik ☹
  - kaszkád rendezés (ha egy szinten egyenlőség lenne)

```
public int compare(T o1, T o2) {
   int x = cmp1.compare(o1, o2); // első comparator
   if (x!=0) return x;
   else return cmp2.compare(o1,o2);
                                // második comparator
}
```

  - ha lehetne komparátorokból listát építeni, az lenne az igazi!



## Komparátorok kombinálása

```
public class CComparator<T> implements Comparator<T> {
  Comparator<? super T> cmp1,cmp2;
  public CComparator(Comparator<? super T> c1,
                      Comparator<? super T> c2) {
      cmp1 = c1; cmp2 = c2;
  }
  public int compare(T o1, T o2) {
      int x = cmp1.compare(o1, o2);
      if (x!=0) return x;
      else return cmp2.compare(o1,o2);
  }
  public CComparator<T> then(Comparator<? super T> c){
      return new CComparator(this, c);
  }
}
```

kaszkád compare

builder metódus

## Komparátorok kombinálása

```
// sorting on a single field
Collections.sort(l,
        Student.comparing(Student::getAverage));
```

egyetlen mező

```
// sorting on name, then birthdate, then average
Collections.sort(l,
   new CComparator(
        Student.comparing(Student::getName),
        Student.comparing(Student::getBirthDate)
   ).then(Student.comparing(Student::getAverage)));
```

kombináljuk a 2 komp.-t

adunk egy harmadikat



## Általános komparátor, 1. lépés

- Legyen a komparátor generikus!

```
public static Comparator<Student>
comparing(Function f) {
   return (Comparator<Student>)
     (o1,o2) -> f.apply(o1).compareTo(f.apply(o2));
}
```

ez legyen generikus

generikus típusok

generikus komparátor

```
public static <T,U> Comparator<T>
comparing(Function<T,U> f) {
   return (Comparator<T>)
     (o1,o2) -> f.apply(o1).compareTo(f.apply(o2));
}
```

generikus függvény ($T \rightarrow U$)

# Általános komparátor, 2. lépés

- Legyen a komparátor generikus!

```
public static <T,U>
Comparator<T>
comparing(Function<T,U> f) {
   return (Comparator<T>)
     (o1,o2) -> f.apply(o1).compareTo(f.apply(o2));
}
```

*U* legyen *Comparable*

A függvény ősön hívva leszármazottat ad vissza

```
public static <T, U extends Comparable<U>>
Comparator<T>
comparing(Function<? super T, ? extends U> f) {
   return (Comparator<T>)
     (o1,o2) -> f.apply(o1).compareTo(f.apply(o2));
}
```





# Általános komparátor, 3. lépés

- Legyen a komparátor generikus!

```
public static <T, U extends Comparable<U>>
Comparator<T>
comparing(Function<T, ? extends U> f) {
   return (Comparator<T>)
     (o1,o2) -> f.apply(o1).compareTo(f.apply(o2));
}
```

*U* lehet leszármazott is

```
public static <T, U extends Comparable<? super U>>
Comparator<T>
comparing(Function<T, ? extends U> f) {
   return (Comparator<T>)
     (o1,o2) -> f.apply(o1).compareTo(f.apply(o2));
}
```

# Általános komparátor, teljes

- A generikus megoldás

```
public static <T, U extends Comparable<? super U>>
Comparator<T>
comparing(Function<? super T, ? extends U> f) {
   return (Comparator<T>)
     (o1,o2) -> f.apply(o1).compareTo(f.apply(o2));
}
```

- *T* típust rendezzük *Comparator*-ral
- *U* típusú mező alapján, aminek őse implementálja *Comparator*-t
- *Function* metódust *T*-ben (vagy ősében) definiáltuk
  és visszaad egy értéket, ami *U* vagy leszármazottja lehet



# Function interfész, Java API

értelmezési tartomány
értékkészlet

- `interface java.util.function.Function<T,R>`
  - `R apply(T x)`
    - a függvény törzse ( *f(x): T→R* )
  - `static <T> Function<T,T> identity()`
    - függvény, ami visszaadja *x*-et: *f(x) = x*
  - `default <V> Function<V,R> compose(Function<? super V,? extends T> before)`
    - *before*, aztán *this*: *f(x) = this.apply(before.apply(x))*

```
Function <Double,Double> f = Math::sin;
f = f.compose(Math::abs);
```

sin(|x|)

# Function interfész, Java API

- `interface java.util.function.Function<T,R>`
  - `R apply(T t)`
    - a függvény törzse ( *f(x)* )
  - `default <V> Function<T,V> andThen(Function<? super R,? extends V> after)`
    - *this*, aztán *after:* *f(x) = after.apply(this.apply(x))*

```
Function <Double,Double> f = Math::sin;
f = f.andThen(Math::abs);
```

|sin(x)|



# Comparator interface in Java 8

- `interface java.util.Comparator<T>`
  - `int compare(T o1, T o2)`

    *a megvalósítandó összehasonlító metódus*
  - `static <T,U extends Comparable<? super U>> Comparator<T> comparing(Function<? super T,? extends U> getter)`

    getter alapján készít komparátort (vö. *Student.cmp*)
  - `default Comparator<T> thenComparing(Comparator<? super T> other)`
  - `default <U extends Comparable<? super U>> Comparator<T> thenComparing(Function<? super T,? extends U> getter)`
  - `default <U> Comparator<T> thenComparing(Function<? super T,? extends U> getter, Comparator<? super U> cmp)`

    olyan komparátor lesz, ami *this-t* kombinálja *cmp-vel* vagy *getter-rel*

# A komparátor API használata

```
// Comparator API
Collections.sort(l,
     Comparator.comparing(Student::getAverage));
```

komparátor készítése

egyetlen mező

```
// sort on average (decr), then birthdate, then name
Collections.sort(l,
   Comparator.comparing(Student::getAverage).reversed()
             .thenComparing(Student::getBirthDate)
             .thenComparing(Student::getName)
);
```

sorrend megfordítása

kaszkád komparátor készítése



# További metódusok *(válogatás)*

- Collection
  - `default boolean removeIf(Predicate<? super E> f)`
    - minden olyan *x* kitörlése, amelyre *f.test(x)* igaz

```
l.removeIf(s->s.getAverage()<2.5);
```

- Iterator
  - `default void forEachRemaining(Consumer<? super T> action)`
    - meghívja *action.accept(x)* metódust minden tárolt *x* elemre

```
l.iterator().forEachRemaining(System.out::println);
```

**_Köszönöm a figyelmet!_**