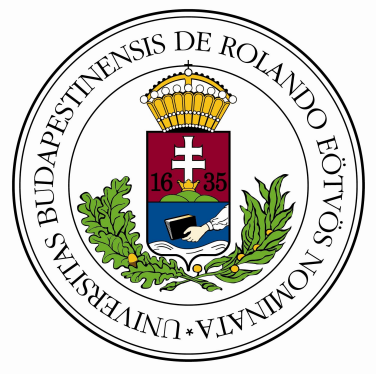
Programozási nyelvek – Java

Típushierarchia



Kozsik Tamás

ELTE Eötvös Loránd Tudományegyetem

Statikus és dinamikus típus: összefoglalás

Változók, paraméterek, kifejezések esetén

Statikus

- Deklarált
- Osztály/interface
- Állandó
- Fordítási időben ismert
- Általánosabb
- Statikus típusellenőrzéshez
- Biztonságot ad

Dinamikus

- Tényleges
- Osztály
- Változhat futás közben
- Futási időben derül ki
- Speciálisabb
- Dinamikus típusellenőrzéshez
- Rugalmasságot ad

Dinamikus kötés: csak példánymetódusra

- *Felüldefiniálni* csak példánymetódust lehet
  - ha nem `final`
- *Megvalósítani* `abstract`-ot, pl. `interface`-ből

Kell a kitüntetett paraméter (dinamikus típusa)

Öröklődéssel definiált interface

Adatszerkezetek bejárásához

```
package java.util;
public interface Iterator<E> {
    boolean hasNext();
    E next();
}
```

Új műveletekkel való kibővítés

```
package java.util;
public interface ListIterator<E> extends Iterator<E> {
    boolean hasPrevious();
    E previous();
    ...
}
```

Típusok közötti származtatás

- Interface `extends` interface
- Osztály `implements` interface
- Osztály `extends` osztály

Outline

Többszörös öröklődés

(Multiple inheritance)

- Egy típust több más típusból származtatunk
- Javában: több interface-ből
- Problémákat vet fel

Példák

OK

```
package java.util;
public class Scanner implements Closeable, Iterator<String> { ... }
```

OK

```
interface PoliceCar extends Car, Emergency { ... }
```

Hibás

```
    class PoliceCar extends Car, Emergency { ... }
```

Hipotetikusan

```
class Base1 {
    int x;
    void setX( int x ){ this.x = x; }
    ...
}

class Base2 {
    int x;
    void setX( int x ){ this.x = x; }
    ...
}

class Sub extends Base1, Base2 { ... }
```

Hipotetikusan: diamond-shaped inheritance

```
class Base0 {
    int x;
    void setX( int x ){ this.x = x; }
    ...
}

class Base1 extends Base0 { ... }

class Base2 extends Base0 { ... }

class Sub extends Base1, Base2 { ... }
```

Különbség class és interface között

- Osztályt lehet példányosítani
  - `abstract class`?
- Osztályból csak egyszeresen örökölhetünk
  - `final class`?
- Osztályban lehetnek példánymezők
  - interface-ben: `public static final`

Többszörös öröklés interfészekből

```
interface Base1 {
    abstract void setX( int x );
    ...
}

interface Base2 {
    abstract void setX( int x );
    ...
}

class Sub implements Base1, Base2 {
    public void setX( int x ){ ... }
    ...
}
```

Outline

abstract class

- Részlegesen implementált osztály
  - Tartalmazhat `abstract` metódust
- Nem példányosítható
- Származtatással konkretizálhatjuk

```
package java.util;
public abstract class AbstractList<E> implements List<E> {
    ...
    public abstract E get( int index );  // csak deklarálva
    public Iterator<E> iterator(){ ... } // implementálva
    ...
}
```

Részleges megvalósítás

```
public abstract class AbstractCollection<E> ... {
    ...
    public abstract int size();
    public boolean isEmpty(){
        return size() == 0;
    }
    public abstract Iterator<E> iterator();
    public boolean contains( Object o ){
        Iterator<E> iterator = iterator();
        while( iterator.hasNext() ){
            E e = iterator.next();
            if( o==null ? e==null : o.equals(e) ) return true;
        }
        return false;
    }
}
```

Konkretizálás

```
public abstract class AbstractCollection<E> implements Collection<E> {
    ...
    public abstract int size();
}
```

```
public abstract class AbstractList<E> extends AbstractCollection<E>
                                              implements List<E> {
    ...
    public abstract E get( int index );
}
```

```
public class ArrayList<E> extends AbstractList<E> {
    ...
    public int size(){ ... }              // implementálva
    public E get( int index ){ ... }      // implementálva
}
```

Outline

Öröklődésre tervezés

- Könnyű legyen származtatni belőle
- Ne lehessen elrontani a típusinvariánst

protected láthatóság

```
package java.util;
public abstract class AbstractList<E> implements List<E> {
    ...
    protected int modCount;
    protected AbstractList(){ ... }
    protected void removeRange(int fromIndex, int toIndex){ ... }
    ...
}
```

- Ugyanabban a csomagban
- Más csomagban csak a leszármazottak

`private` ⊆ félnyilvános (package-private) ⊆ `protected` ⊆ `public`

A private tagok nem hivatkozhatók a leszármazottban!

```
class Counter {
    private int counter = 0;
    public int count(){ return ++counter; }
}

class SophisticatedCounter extends Counter {
    public int count( int increment ){
        return counter += increment;   // fordítási hiba
    }
}
```

„Javítva”

```
class Counter {
    private int counter = 0;
    public int count(){ return ++counter; }
}

class SophisticatedCounter extends Counter {
    public int count( int increment ){
        if( increment < 1 ) throw new IllegalArgumentException();
        while( increment > 1 ){
            count();
            --increment;
        }
        return count();
    }
}
```

protected

```
package my.basic.types;
public class Counter {
    protected int counter = 0;
    public int count(){ return ++counter; }
}
```

```
package my.advanced.types;
class SophisticatedCounter extends my.basic.types.Counter {
    public int count( int increment ){
        return counter += increment;
    }
}
```

Outline

Öröklődés $\Rightarrow$ altípusosság

class A implements I

$A\ \Delta_{ci}\ I \Rightarrow A <: I$

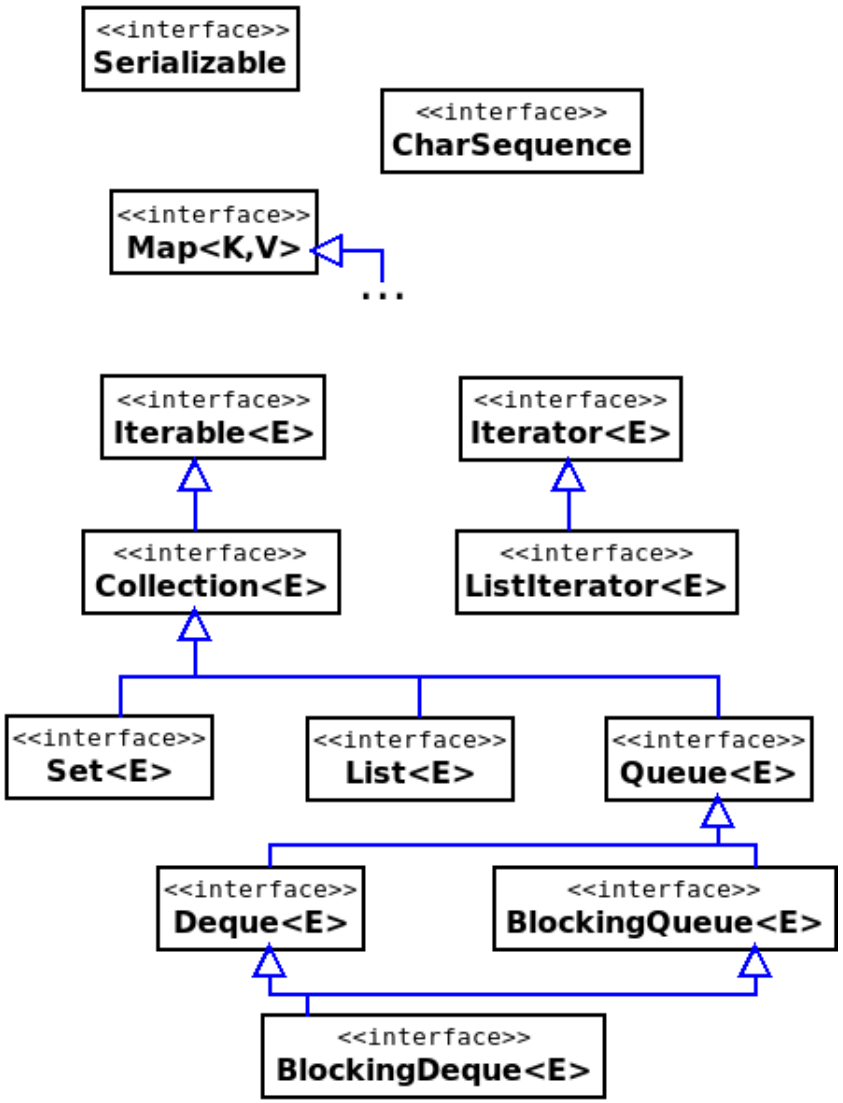
class A extends B

$A\ \Delta_{c}\ B \Rightarrow A <: B$

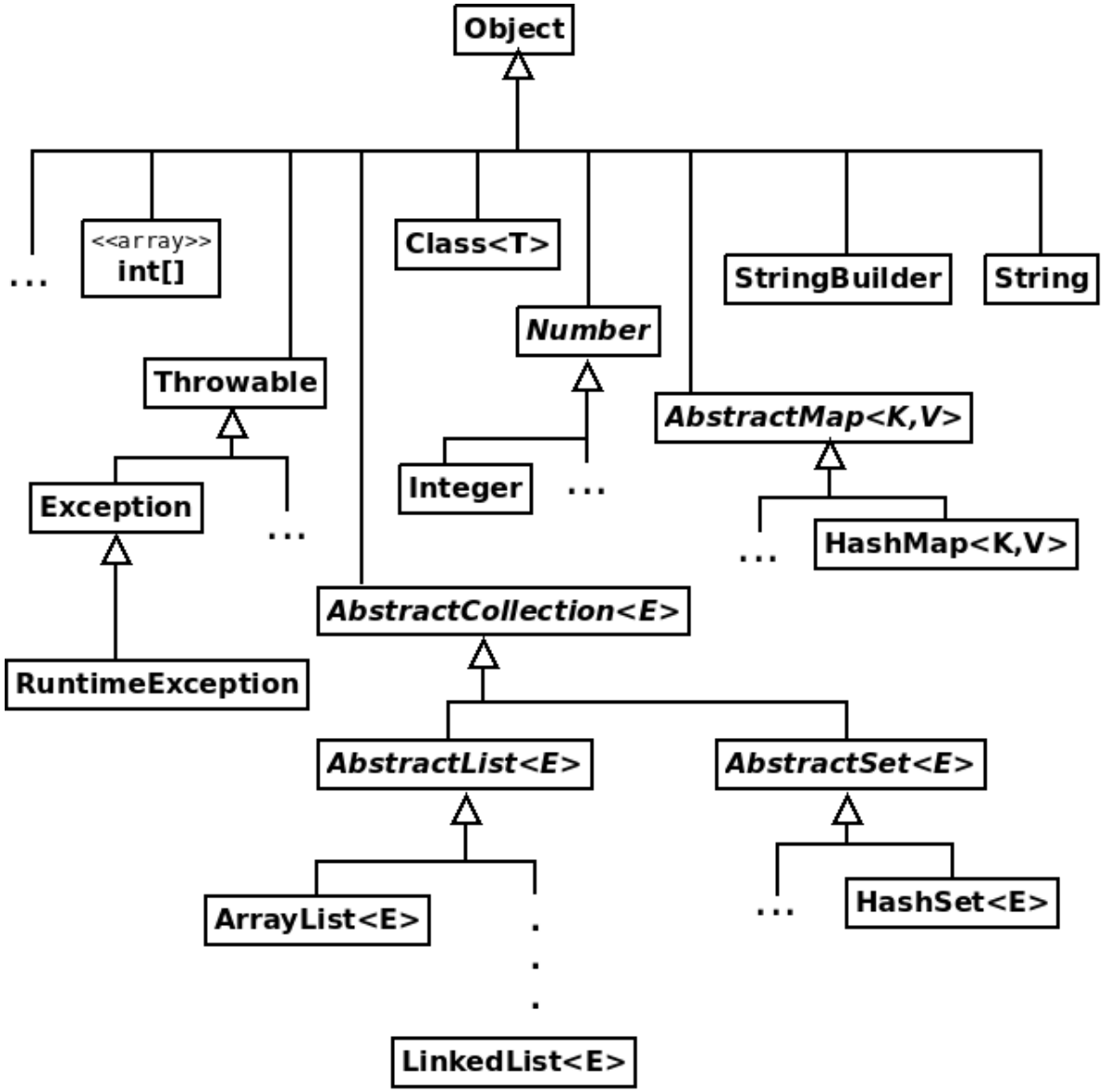
interface I extends J

$I\ \Delta_{i}\ J \Rightarrow I <: J$

Interface-ek hierarchiája a Javában (részlet)



Osztályok hierarchiája a Javában (részlet)



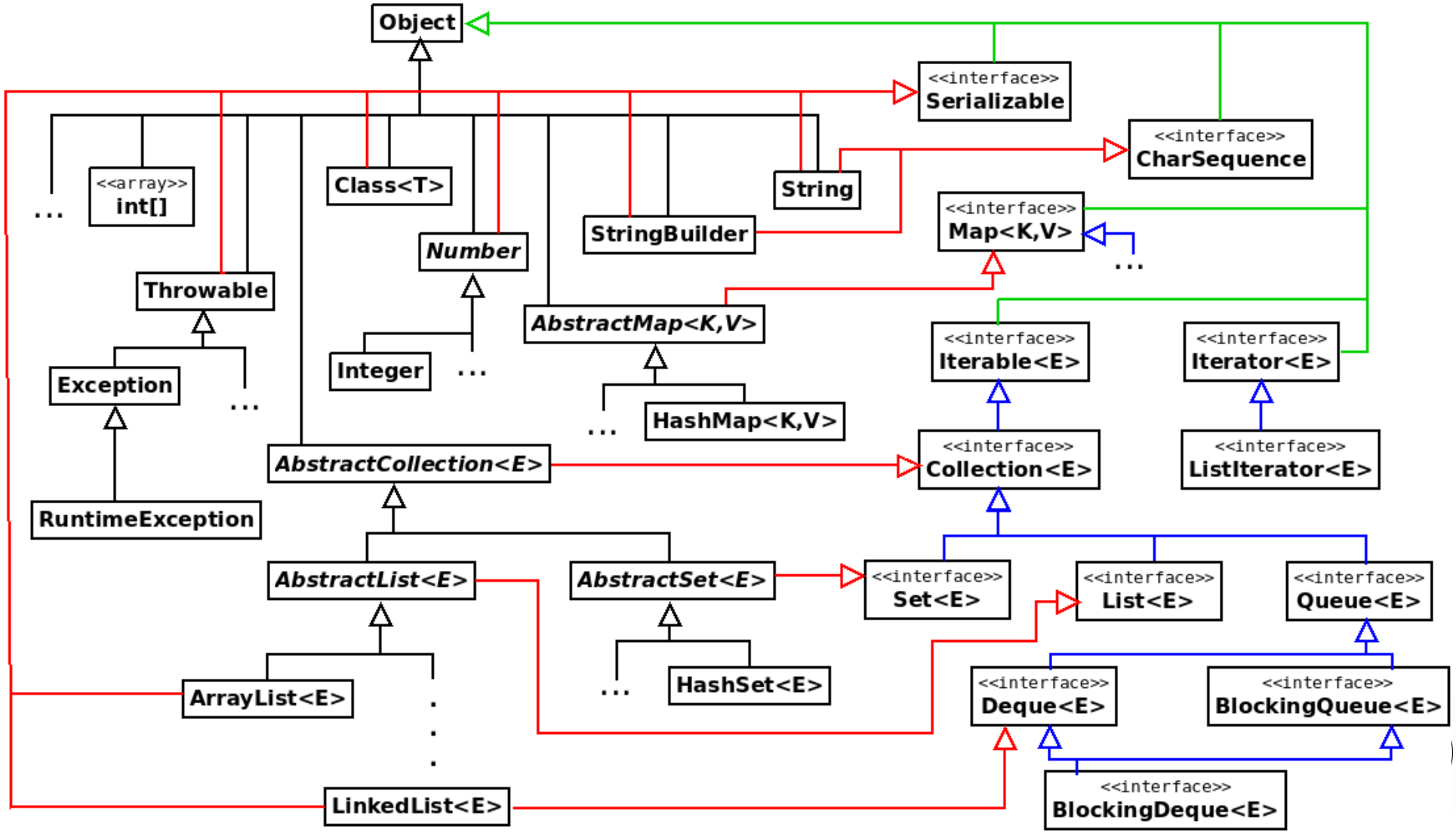
java.lang.Object

Minden osztály belőle származik, kivéve önmagát!

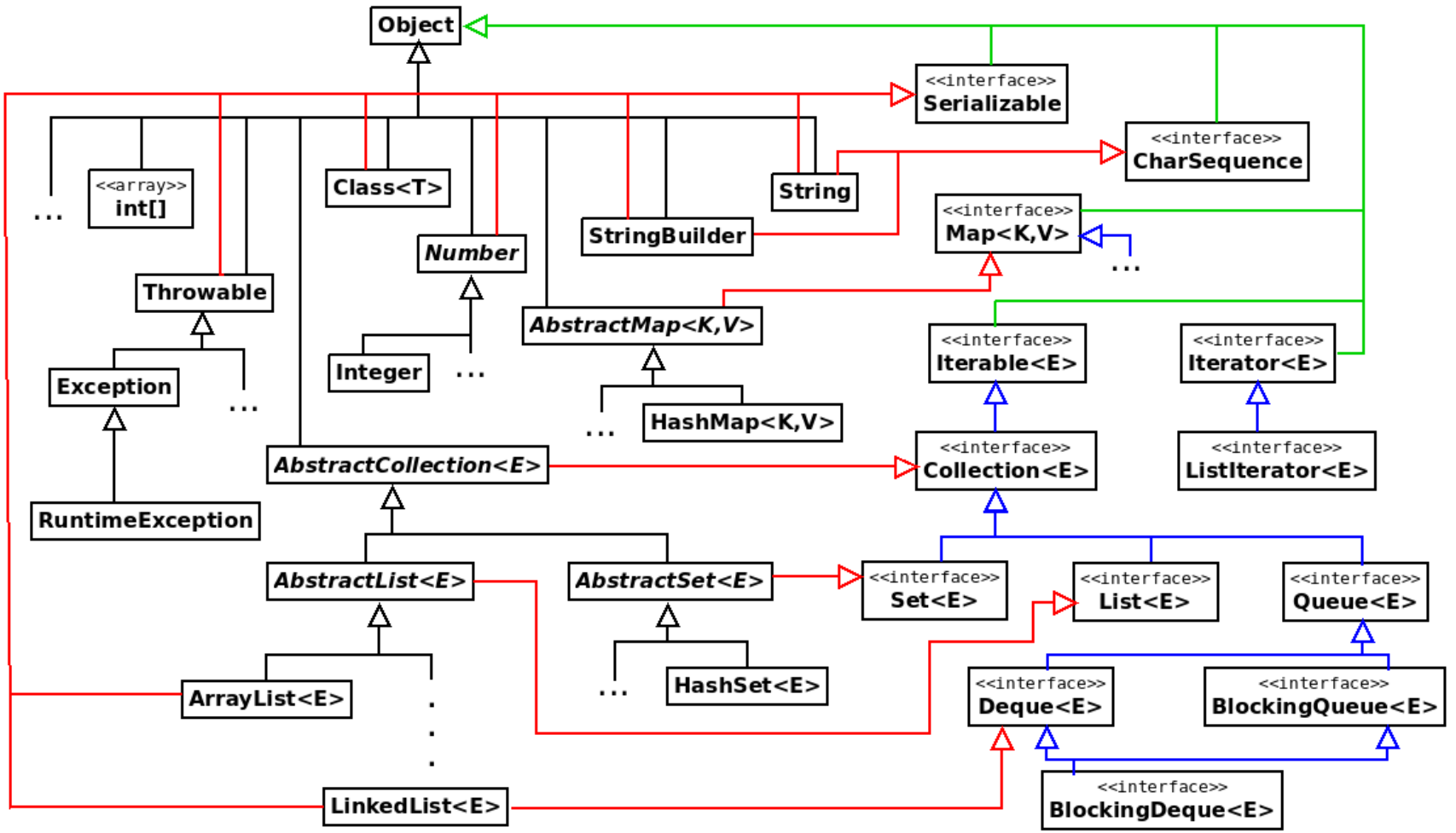
```
package java.lang;
public class Object {
    public Object(){ ... }
    public String toString(){ ... }
    public int hashCode(){ ... }
    public boolean equals( Object that ){ ... }
    public Class getClass(){ ... }
    ...
}
```

Referenciatípusok hierarchiája a Javában (részlet)

körmentes irányított gráf (DAG: directed acyclic graph)



Típusok hierarchiája a Javában (részlet)



boolean
char
byte
short
int
long
float
double

Outline

Altípus reláció

$$<: \quad = \quad (\Delta_c \cup \Delta_i \cup \Delta_{ci} \cup \Delta_o)^*$$

- Δ_o jelentése: minden a `java.lang.Object`-ből származik
- ϱ^* jelentése: ϱ reláció reflexív, tranzitív lezártja
  - Ha `A` ϱ `B`, akkor `A` ϱ^* `B`
  - Reflexív lezárt: `A` ϱ^* `A`
  - Tranzitív lezárt: ha `A` ϱ^* `B` és `B` ϱ^* `C`, akkor `A` ϱ^* `C`

Ez egy parciális rendezés (RAT)!

A dinamikus típus a statikus típus altípusa

Ha `A` $<:$ `B`, akkor

- `B v = new A();` helyes
- `void m( B p )...` esetén `m(new A())` helyes
- `A a; B b; ... b = a;` helyes

Altípusos polimorfizmus (subtype polymorphism)

Ha egy kódbázist megírtunk, újrahasznosíthatjuk speciális típusokra!

- Általánosabb típusok helyett használhatunk altípusokat
- Több típusra is működik a kódbázis: polimorfizmus

Újrafelhasználhatóság!

Specializálás

- Az altípus „mindent tud”, amit a bázistípus
- Az altípus speciálisabb lehet
- Ez az *is-egy* reláció
  - `Car` *is-a* `Vehicle`
  - `Boat` *is-a* `Vehicle`
- Emberi gondolkodás, OO modellezés

Többszörös altípusképzés

- Egy fogalom több általános fogalom alá tartozhat
  - `PoliceCar` *is-a* `Car` **és** *is-a* `EmergencyVehicle`
  - `FireBoat` *is-a* `Boat` **és** *is-a* `EmergencyVehicle`
- Összetett fogalmi modellezés Javában: `interface`

Többszörös kódöröklés?

- Kódöröklés: osztályok mentén
  - csak egyszeres öröklődés

Többszörös kódöröklés?

- Kódöröklés: osztályok mentén
  - csak egyszeres öröklődés
- `interface`-ekből
  - többszörös öröklődés
  - Korlátozott mértékű kódöröklés: `default` implementációjú példánymetódusok

Outline

Kivételosztályok hierarchiája

java.lang.Throwable

- java.lang.Exception
  - java.sql.SQLException
  - java.io.IOException
    - java.io.FileNotFoundException
  - ...
  - saját kivételek általában ide kerülnek
  - java.lang.RuntimeException
    - java.lang.NullPointerException
    - java.lang.ArrayIndexOutOfBoundsException
    - java.lang.IllegalArgumentException
    - ...
- java.lang.Error
  - java.lang.VirtualMachineError
  - ...

Nem ellenőrzött kivételek

- `java.lang.RuntimeException` és leszármazottjai
- `java.lang.Error` és leszármazottjai

Egyes alkalmazási területen akár ezek is kezelendők!

Kivételkezelő ágak

```
try {
    ...
} catch( FileNotFoundException e ){
    ...
} catch( EOFException e ){
    ...
} // nem kezeltük a java.net.SocketException-t
```

Speciálisabb-általánosabb kivételkezelő ágak

```
try {
    ...
} catch( FileNotFoundException e ){
    ...
} catch( EOFException e ){
    ...
} catch( IOException e ){  // minden egyéb IOException
    ...
}
```

Fordítási hiba: elérhetetlen kód

```
try {
    ...
} catch( FileNotFoundException e ){
    ...
} catch( IOException e ){  // minden egyéb IOException
    ...
} catch( EOFException e ){ // rossz sorrend!
    ...
}
```

Outline

Motiváló példa

```
package java.lang;
public interface CharSequence {
    int length();
    char charAt( int index );
    ...
}
```

Implementáló osztályok

- java.lang.String
- java.lang.StringBuilder
- java.lang.StringBuffer
- ...

Írjunk lexikografikus összehasonlítást!

```
static boolean less( CharSequence left, CharSequence right ){ ... }
```

Elegendő az altípusos polimorfizmus

```
static boolean less( CharSequence left, CharSequence right ){ ... }
```

```
less( "cool", "hot" )

StringBuilder sb1 = new StringBuilder(); ...
StringBuilder sb2 = new StringBuilder(); ...
less( sb1, sb2 )

less( "cool", sb1 )
```

Nem elegendő az altípusos polimorfizmus

```
static boolean less( CharSequence left, CharSequence right ){ ... }

static CharSequence min( CharSequence left, CharSequence right ){
    return less(left,right) ? left : right;
}
```

Nem elegendő az altípusos polimorfizmus

```
static boolean less( CharSequence left, CharSequence right ){ ... }

static CharSequence min( CharSequence left, CharSequence right ){
    return less(left,right) ? left : right;
}
```

OK

```
CharSequence cs = min( "cool", "hot" );
```

Fordítási hiba

```
String str = min( "cool", "hot" );
```

Parametrikus polimorfizmus

```
static boolean less( CharSequence left, CharSequence right ){ ... }

static <T> T min( T left, T right ){
    return less(left,right) ? left : right;
}
```

Fordítási hiba: `less`

Korlátozott univerzális kvantálás

```
static boolean less( CharSequence left, CharSequence right ){ ... }

static <T extends CharSequence> T min( T left, T right ){
    return less(left,right) ? left : right;
}
```

Korlátozott univerzális kvantálás

```
static boolean less( CharSequence left, CharSequence right ){ ... }

static <T extends CharSequence> T min( T left, T right ){
    return less(left,right) ? left : right;
}
```

```
String str = min( "cool", "hot" );
StringBuilder sb = min( new StringBuilder(), new StringBuilder() );
CharSequence cs = min( "cool", new StringBuilder() );
```

Korlátozott univerzális kvantálás

```
static boolean less( CharSequence left, CharSequence right ){ ... }

static <T extends CharSequence> T min( T left, T right ){
    return less(left,right) ? left : right;
}
```

```
String str = min( "cool", "hot" );
StringBuilder sb = min( new StringBuilder(), new StringBuilder() );
CharSequence cs = min( "cool", new StringBuilder() );
```

- constrained genericity
- bounded universal quantification
- bounded parametric polymorphism
- $\forall T$-re, amely a `CharSequence`-ből származik, definiáljuk a `min` függvényt úgy, hogy...
- felső korlát (upper bound)

„Természetes rendezés” (natural ordering)

```
java.util.Arrays.sort(args)
```

„Természetes rendezés” (natural ordering)

```
java.util.Arrays.sort(args)
```

```
public final class String ... {
    ...
    public int compareTo( String that ){ ... }
}
```

```
public final class Integer ... {
    ...
    public int compareTo( Integer that ){ ... }
}
```

Természetes rendezés - interface

3-way comparison

```
package java.lang;
public interface Comparable<T> {    // negative: this < that
    int compareTo( T that );        //     zero: this = that
}                                   // positive: this > that
```

Természetes rendezés - interface

3-way comparison

```
package java.lang;
public interface Comparable<T> {    // negative: this < that
    int compareTo( T that );        //     zero: this = that
}                                   // positive: this > that
```

```
package java.lang;
public final class String implements Comparable<String> { ... }
// public int compareTo( String that ){ ... }
```

```
package java.lang;
public final class Integer implements Comparable<Integer> { ... }
// public int compareTo( Integer that ){ ... }
```

Saját természetes rendezés

3-way comparison

```
package java.lang;
public interface Comparable<T> {    // negative: this < that
    int compareTo( T that );        //     zero: this = that
}                                   // positive: this > that
```

```
public class Rational implements Comparable<Rational> {
    ...
    public int compareTo( Rational that ){
        /* class invariant: denominator > 0 */
        return numerator * that.denominator -
               that.numerator * denominator;
    }
}
```

Rendezhetőség öröklése

```
public class Rational implements Comparable<Rational> {
    ...
    public int compareTo( Rational that ){
        return numerator * that.denominator -
               that.numerator * denominator;
    }
}
```

```
public class CanonicalRational extends Rational { ... }
```

```
(new Rational(3,6)).compareTo(new Rational(5,9))
(new Rational(3,6)).compareTo(new CanonicalRational(5,9))
(new CanonicalRational(3,6)).compareTo(new CanonicalRational(5,9))
(new CanonicalRational(3,6)).compareTo(new Rational(5,9))
```

Probléma az öröklődéssel

Egy osztály nem implementálhatja ugyanazt a generikus interface-t többször, különböző típusparaméterekkel.

```
public class Time implements Comparable<Time> {
    ...
    public int compareTo( Time that ){ ... }
}
```

Fordítási hiba

```
public class ExactTime extends Time
                       implements Comparable<ExactTime> {
    ...
    public int compareTo( ExactTime that ){ ... }
}
```

Más összehasonlításhoz

```
@FunctionalInterface
public interface Comparator<T> {
    int compare( T left, T right );   // 3-way
}
```

Más összehasonlításhoz

```
@FunctionalInterface
public interface Comparator<T> {
    int compare( T left, T right );   // 3-way
}
```

```
class StringLengthComparator implements Comparator<String> {
    public int compare( String left, String right ){
        return left.length() - right.length();
    }
}
```

```
java.util.Arrays.sort( args, new StringLengthComparator() );
```

Más összehasonlításhoz

```
@FunctionalInterface
public interface Comparator<T> {
    int compare( T left, T right );   // 3-way
}
```

```
class StringLengthComparator implements Comparator<String> {
    public int compare( String left, String right ){
        return left.length() - right.length();
    }
}
```

```
java.util.Arrays.sort( args, new StringLengthComparator() );
```

```
java.util.Arrays.sort( args, (a,b) -> a.length()-b.length() );
```

Rendezés

Nyilvános műveletek a `java.util.Arrays` osztályban:

```
static <T> void parallelSort( T[] a, Comparator<? super T> cmp )
```

- `cmp`: létezik olyan S típus, amelynek altípusa a T, és ilyeneket tud összehasonlítani
  - egzisztenciális kvantálás (existential quantification)
  - alsó korlát (lower bound)

Rendezés

Nyilvános műveletek a `java.util.Arrays` osztályban:

```
static <T> void parallelSort( T[] a, Comparator<? super T> cmp )
```

- `cmp`: létezik olyan S típus, amelynek altípusa a T, és ilyeneket tud összehasonlítani
  - egzisztenciális kvantálás (existential quantification)
  - alsó korlát (lower bound)

```
static <T extends Comparable<? super T>> void parallelSort(T[] a)
```

- A T olyan típus legyen, amelynek van olyan bázistípusa, amely rendelkezik természetes rendezéssel

Altípus reláció paraméterezett típusokon

```
public class ArrayList<T> ... implements List<T> ...
```

$\forall$ T-re: `ArrayList<T>` $<:$ `List<T>`

- `ArrayList<String>` $<:$ `List<String>`
- `ArrayList<Integer>` $<:$ `List<Integer>`

Típusparaméter altípusossága?

Szabály

`List<Integer>` $\not<:$ `List<Object>`

Típusparaméter altípusossága?

Szabály

`List<Integer>` $\not<:$ `List<Object>`

Indirekt tegyük fel, hogy `List<Integer>` $<:$ `List<Object>`

```
List<Integer> nums = new ArrayList<Integer>();
nums.add( 42 );                  // Integer.valueOf(42)
List<Object> things = nums;      // indirekt feltevés
things.add( "forty-two" );       // String <: Object
Integer n = nums.get(1);         // hiba!
```

Tömbökre gyengébb szabály vonatkozik

A Java megengedi, hogy `Integer[]` $<:$ `Object[]` legyen!

Tömbökre gyengébb szabály vonatkozik

A Java megengedi, hogy `Integer[]` $<:$ `Object[]` legyen!

Paraméterezett típusokra

```
List<Integer> nums = new ArrayList<Integer>();
nums.add( 42 );                  // Integer.valueOf(42)
List<Object> things = nums;      // fordítási hiba
things.add( "forty-two" );       // String <: Object
Integer n = nums.get(1);         // hiba lenne
```

Tömbökre gyengébb szabály vonatkozik

A Java megengedi, hogy `Integer[]` <: `Object[]` legyen!

Paraméterezett típusokra

```
List<Integer> nums = new ArrayList<Integer>();
nums.add( 42 );                  // Integer.valueOf(42)
List<Object> things = nums;      // fordítási hiba
things.add( "forty-two" );       // String <: Object
Integer n = nums.get(1);         // hiba lenne
```

Tömb típusokra

```
Integer[] nums = new Integer[2];
nums[0] =  42 ;                  // Integer.valueOf(42)
Object[] things = nums;          // szabályos
things[1] = "forty-two" ;        // ArrayStoreException
Integer n = nums[1];             // hiba lenne
```

Outline

Öröklődéssel definiált osztály

- A szülőosztály tagjai átöröklődnek
- Újabb tagokkal bővíthető (Java: `extends`)
- Megörökölt példánymetódusok újradefiniálhatók
  - ... és **újradeklarálhatók**

Példa: klónozás

```
package java.lang;
public interface Cloneable {}
```

```
package java.lang;
public class CloneNotSupportedException extends Exception { ... }
```

```
package java.lang;
public class Object {
    public boolean equals( Object that ){ return this == that; }
    ...
    protected Object clone() throws CloneNotSupportedException {
        if( this instanceof Cloneable ) return /* shallow copy */
        else throw new CloneNotSupportedException();
    }
}
```

Sekély másolat – első próbálkozás

```
public class Time implements Cloneable {
    private int hour, minute;
    public Time( int hour, int minute ){ ... }
    public void setHour( int hour ){ ... }
    ...
    @Override public String toString(){ ... }
    @Override public int hashCode(){ return 60*hour + minute; }
    @Override public boolean equals( Object that ){
        if( that != null && getClass().equals(that.getClass()) ){
            Time t = (Time)that;
            return hour == t.hour && minute == t.minute;
        } else return false;
    }
}
```

Kényelmetlen!

```
package java.lang;
public class Object {
    ...
    protected Object clone() throws CloneNotSupportedException ...
}
```

Nem hívható akárhol!

```
public class Time implements Cloneable {
    ...
    public static void main( String[] args ){
        Time t = new Time(12,30);
        try { Object o = t.clone(); }
        catch( CloneNotSupportedException e ){ assert false; }
    }
}
```

Újradeklarálva jó!

```
public class Time implements Cloneable {
    private int hour, minute;
    ...
    @Override public Time clone(){
        try { return (Time)super.clone(); }
        catch( CloneNotSupportedException e ){
            assert false; return null;
        }
    }
}
```

Szabályos felüldefiniálás

- Láthatóság bővíthető
- Visszatérési típus szűkíthető
- Bejelentett ellenőrzött kivételek szűkíthetők

Klónozás szabályai

`implements Cloneable`

- Tegyük nyilvánossá a `clone()`-t
  - felüldefiniálás
  - újradeklarálás

Klónozás szabályai

`implements Cloneable`

- Tegyük nyilvánossá a `clone()`-t
  - felüldefiniálás
  - újradeklarálás
- Mindig használjuk a `super.clone()`-t!
  - megőrzi a dinamikus típust
  - megörökölt `clone()` is jót ad vissza

Klónozás szabályai

`implements Cloneable`

- Tegyük nyilvánossá a `clone()`-t
  - felüldefiniálás
  - újradeklarálás
- Mindig használjuk a `super.clone()`-t!
  - megőrzi a dinamikus típust
  - megörökölt `clone()` is jót ad vissza
- Sekély másolat: maradhat ugyanaz az implementáció
  - mély(ebb) másolat: új implementáció
  - immutable tagot nem kell másolni

Mély másolás

```
public class Interval implements Cloneable {
    private Time from, to;
    public void setFrom( Time from ){
        this.from.setHour( from.getHour() );
        this.from.setMinute( from.getMinute() );
    }
    @Override public Interval clone(){
        try {   Interval that = (Interval)super.clone();
                that.from = that.from.clone();
                that.to = that.to.clone();
                return that;
        } catch( CloneNotSupportedException e ){
                assert false; return null;
        }
    }
    ...
```