# Programming languages Java

## More on types

Kozsik Tamás

ELTE
IK


ELTE
IK

# Abstraction: encapsulation and information hiding

```
public class Rational {
    private final int numerator, denominator;
    private static int gcd(int a, int b) { ... }
    private void simplify() { ... }
    public Rational(int numerator, int denominator) { ... }
    public Rational(int value) { super(value,1); }
    public int getNumerator() { return numerator; }
    public int getDenominator() { return denominator; }
    public Rational times(Rational that) { ... }
    public Rational times(int that) { ... }
    public Rational plus(Rational that) { ... }
    ...
}
```

ELTE
IK

# The interface of a class

```
public Rational(int numerator, int denominator)
public Rational(int value)
public int getNumerator()
public int getDenominator()
public Rational times(Rational that)
public Rational times(int that)
public Rational plus(Rational that)
...
```

ELTE
IK

# An interface-definition

```
public interface Rational {
    public int getNumerator();
    public int getDenominator();
    public Rational times(Rational that);
    public Rational times(int that);
    public Rational plus(Rational that);
    ...
}
```

# abstract method: only declared, not defined

```
public interface Rational {
    abstract public int getNumerator();
    abstract public int getDenominator();
    abstract public Rational times(Rational that);
    abstract public Rational times(int that);
    abstract public Rational plus(Rational that);
    ...
}
```

# interface: all members are automatically public

```
public interface Rational {
    int getNumerator();
    int getDenominator();
    Rational times(Rational that);
    Rational times(int that);
    Rational plus(Rational that);
    ...
}
```

# Contents of an `interface`-definition

Declaration of instance methods: specification terminated with ;

```
int getNumerator();
```

# Contents of an `interface`-definition

Declaration of instance methods: specification terminated with ;

```
int getNumerator();
```

···but there's more

- Instance methods can optionally have a `default` implementation
- Definition of "constants" : `public static final`
- Static methods
- Nested (member) type definitions

# Implementation of an `interface`

## Rational.java

```
public interface Rational {
  int getNumerator();
  int getDenominator();
  Rational times(Rational that);
}
```

## Fraction.java

```
public class Fraction implements Rational {
  private final int numerator, denominator;
  public Fraction(int numerator, int denominator) { ... }
  public int getNumerator() { return numerator; }
  public int getDenominator() { return denominator; }
  public Rational times(Rational that) { ... }
}
```

# Multiple implementations

## Fraction.java

```
public class Fraction implements Rational {
  private final int numerator, denominator;
  public Fraction(int numerator, int denominator) { ... }
  public int getNumerator() { return numerator; }
  public int getDenominator() { return denominator; }
  public Rational times(Rational that) { ... }
}
```

## Simplified.java

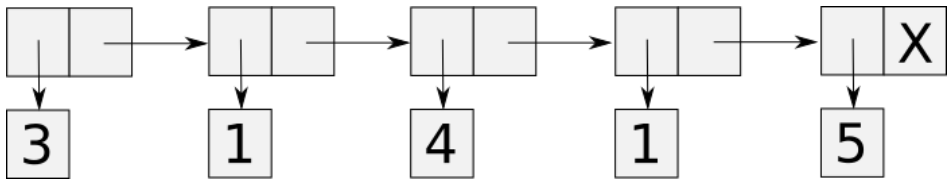
```
public class Simplified implements Rational {
    ...
    public int getNumerator() { ... }
    public int getDenominator() { ... }
    Rational times(Rational that) { ... }
}
```

# Types representing sequences

- `int[]`
- `java.util.ArrayList<Integer>`
- `java.util.LinkedList<Integer>`

# Linked representation

```
public class LinkedList<T> {
  private T head;
  private LinkedList<T> tail;
  public LinkedList() { ... }
  public T get(int idx) { ... }
  public void set(int idx, T item) { ... }
  public void add(T item) { ... }
  ...
}
```




ELTE IK

# Generic interface

## java/util/List.java

```
package java.util;
public interface List<T> {
  T get(int idx);
  void set(int idx, T item);
  void add(T item);
  ...
}
```

## java/util/ArrayList.java

```
package java.util;
public class ArrayList<T> implements List<T> {
  public ArrayList() { ... }
  public T get(int idx) { ... }
  ...
}
```

# Subtyping

```
class Fraction implements Rational { ... }

class ArrayList<T> implements List<T> { ... }
class LinkedList<T> implements List<T> { ... }
```

- Fraction <: Rational
- Simplified <: Rational
- For all T: ArrayList<T> <: List<T>
- For all T: LinkedList<T> <: List<T>

ELTE
IK

# LSP: Liskov Substitution Principle

Type $A$ is a subtype of (base) type $B$ if instances of $B$ can be substituted with instances of $A$ without causing any trouble.

# Instantiation

An `interface` is a type

```
List<String> names;

static List<String> noDups(List<String> names) {
    ...
}
```

# Instantiation

An `interface` is a type

```
List<String> names;

static List<String> noDups(List<String> names) {
    ...
}
```

- Not possible to instantiate

```
List<String> names = new List<String>(); // compilation error
```

ELTE
IK

# Instantiation

An `interface` is a type

```
List<String> names;

static List<String> noDups(List<String> names) {
    ...
}
```

- Not possible to instantiate

```
List<String> names = new List<String>(); // compilation error
```

- A `class` is a type, and it is allowed to be instantiated

```
ArrayList<String> names = new ArrayList<String>();
ArrayList<String> nicks = new ArrayList<>();
```

# Usually: variable type is an `interface`

This is possible

```
ArrayList<String> names = new ArrayList<>();
```

# Usually: variable type is an `interface`

This is possible

```
ArrayList<String> names = new ArrayList<>();
```

But usually, this is better

```
List<String> names = new ArrayList<>();
```

ELTE
IK

# Usually: variable type is an `interface`

This is possible

```
ArrayList<String> names = new ArrayList<>();
```

But usually, this is better

```
List<String> names = new ArrayList<>();
```

Can change the instance's class later without modifying other code

```
List<String> names = new LinkedList<>();
```

ELTE
IK

# Usually: variable type is an `interface`

This is possible

```
ArrayList<String> names = new ArrayList<>();
```

But usually, this is better

```
List<String> names = new ArrayList<>();
```

Can change the instance's class later without modifying other code

```
List<String> names = new LinkedList<>();
```

Note: in Java 10+, there is a shortcut

```
var names = new LinkedList<String>();  // LinkedList<String>
```

# Static and dynamic type

```
List<String> names = new ArrayList<>();
```

- `List<String>`: **static** (declared) type of `names`
  - ◇ can be an interface or a class
- `ArrayList<String>`: **dynamic** type, the type of the bound value
  - ◇ can only be an instance of a class

```
static List<String> noDups(List<String> names) {
    ... names ...
}


List<String> shortList = noDups(names);
```

IK

# Interfaces with specific meanings

```
// supports enhanced for-loop
class MyDataStructure<T> implements java.lang.Iterable<T>

// supports try-with-resources
class MyResource implements java.lang.AutoCloseable

// supports shallow copying
class MyRational implements java.lang.Cloneable

// supports object serialization
class MyData implements java.io.Serializable
```

ELTE
IK

# Iterable and Iterator

- `Iterable` (e.g. a data structure): provides an iterator
- `Iterator`: visit the elements of a data structure one by one

## java.lang.Iterable

```
public interface Iterable<T> {
  java.util.Iterator<T> iterator();
  ...
}
```

## java.util.Iterator

```
public interface Iterator<T> {
  boolean hasNext();
  T next();
  ...
}
```

# Implementation of Iterator (simplified)

```
package java.util;
public class ArrayList<T> implements Iterable<T> {
  Object[] data;
  int size = 0;
  ...
  public Iterator<T> iterator() {return new IterImpl<>(this);}
}
class IterImpl<T> implements Iterator<T> {
  private final ArrayList<T> storage;
  private int idx = 0;
  IterImpl(ArrayList<T> al) { storage = al; }
  public boolean hasNext() { return idx < storage.size; }
  @SuppressWarnings("unchecked") public T next() {
    return (T)storage.data[idx++];
  }
}
```

# Iterable and Iterator – polymorphism

```
long sum(Iterable<Integer> is) {
  long sum = 0L;
  Iterator<Integer> it = is.iterator();
  while (it.hasNext()) {
    sum += it.next();
  }
  return sum;
}
```

```
List<Integer> list = new LinkedList<>();
...
long sum = sum(list);
```

ELTE IK

# Iteration with a loop (*enhanced for-loop*)

```
long sum(Iterable<Integer> items) {
  long sum = 0L;

  for (Integer item: items) {
    sum += item;
  }
  return sum;
}
```

```
List<Integer> list = new LinkedList<>();
...
long sum = sum(list);
```

ELTE
IK

# Multiple simultaneous iterations

```
List<Pair<Integer,Integer>> pairs(List<Integer> ns) {
  List<Pair<Integer,Integer>> ps = new LinkedList<>();
  for (Iterator<Integer> it = ns.iterator(); it.hasNext(); ) {
    Integer item = it.next();
    for (var it2 = ns.iterator(); it2.hasNext(); ) {
      ps.add(new Pair<Integer,Integer>(item, it2.next()));
    }
  }
  return ps;
}
```

ELTE
IK

# Multiple simultaneous iterations

```
List<Pair<Integer,Integer>> pairs(List<Integer> ns) {
  List<Pair<Integer,Integer>> ps = new LinkedList<>();
  for (Iterator<Integer> it = ns.iterator(); it.hasNext(); ) {
    Integer item = it.next();
    for (var it2 = ns.iterator(); it2.hasNext(); ) {
      ps.add(new Pair<Integer,Integer>(item, it2.next()));
    }
  }
  return ps;
}
```

No built-in `Pair` type; can use `Map.Entry` type from `java.util.Map`

```
List<Map.Entry<Integer,Integer>> ps = new LinkedList<>();
ps.add(Map.entry(item, it2.next()));
```

ELTE IK

# Object identity

```
Time t1 = new Time(13,30);
Time t2 = new Time(13,30);
System.out.println(t1 == t2);

t2 = t1;
System.out.println(t1 == t2);
```

ELTE
IK

# More general equality on reference types?

```
Time t1 = new Time(13,30);
Time t2 = new Time(13,30);

System.out.println(t1 == t2);

System.out.println(t1.equals(t2));
```

ELTE
IK

# More general equality on reference types

```
ArrayList<Integer> seq1 = new ArrayList<>();
seq1.add(1984); seq1.add(2001);

ArrayList<Integer> seq2 = seq1;
System.out.println(seq1 == seq2);

seq2 = new ArrayList<>();
seq2.add(1984); seq2.add(2001);

System.out.println(seq1 == seq2);

System.out.println(seq1.equals(seq2));
```

ELTE
IK

# "The" equals method

```
package java.lang;
public class Object {
    ...
    public boolean equals(Object that) { ... }
}
```

- Can be overridden (e.g. for `Time`)
- A single method with many (partial) implementations
- The implementations form a compound implementation

ELTE
IK

# "The" equals method

```
package java.lang;
public class Object {
    ...
    public boolean equals(Object that) { ... }
}
```

- Can be overridden (e.g. for `Time`)
- A single method with many (partial) implementations
- The implementations form a compound implementation

- ···if done properly!

ELTE
IK

# Obey the contract of `equals`!

- Deterministic
- Equivalence relation (RST: reflexive, symmetric, transitive)
- If `a != null` then `!a.equals(null)`
  - ◇ Note that `null.equals(a)` → `NullPointerException`
- Consistent with the `hashCode()` method
  - ◇ `a.equals(b)` → `a.hashCode() == b.hashCode()`
  - ◇ [it's good if non-equal objects have different `hashCodes`]

# Default behaviour

```
package java.lang;
public class Object {
  ...

  public boolean equals(Object that) {
    return this == that;
  }

  public int hashCode() { ... }
}
```

ELTE
IK

# Correct overriding

```
public class Time {
  ...

  @Override
  public boolean equals(Object that) {
    if (that == null)                          return false;
    if (this.getClass() != that.getClass())    return false;
    Time t = (Time)that;
    return hour == t.hour && min == t.min;
  }

  @Override
  public int hashCode() { return 60*hour + min; }
}
```

# Correct overriding using the name of the class

```
public class Time {
  ...

  @Override
  public boolean equals(Object that) {
    if (that == null)                    return false;
    if (that.getClass() != Time.class)   return false;
    Time t = (Time)that;
    return hour == t.hour && min == t.min;
  }

  @Override
  public int hashCode() { return 60*hour + min; }
}
```

# Correct overriding + "fast lane"

```
public class Time {
  ...

  @Override
  public boolean equals(Object that) {
    if (this == that)                    return true;
    if (that == null)                    return false;
    if (that.getClass() != Time.class)   return false;
    Time t = (Time)that;
    return hour == t.hour && min == t.min;
  }

  @Override public int hashCode() { return 60*hour + min; }
}
```

# Typical error

```
package java.lang;
public class Object {
    ...
    public boolean equals(Object that) { return this == that;
    public int hashCode() { ... }
}
```

## Compilation error due to @Override

```
public class Time {
    ...
    @Override public boolean equals(Time that) {
        return that != null && hour == that.hour && ...
    }
    @Override public int hashCode() { return 60*hour + min; }
}
```

# Bad design leads to wrong result

```
public class Time {
    ...
    public boolean equals(Time that) {  // no @Override here
        return that != null && hour == that.hour && ...
    }
    @Override public int hashCode() { return 60*hour + min; }
}
```

This is over**load**ing, not over**rid**ing

```
Time t1 = new Time(12, 34);
Time t2 = new Time(12, 34);
```

- No dynamic binding: `t1.equals(t2)` calls equals from `Object`
  - ◇ Therefore, equivalent to `t1 == t2` → `false`

ELTE IK

# Importance of `@Override`

- Makes programmer's intention explicit
- Compiler detects failed overriding

**Use it!**

ELTE IK

# Overloading on subtypes

```
static void connect(Employee e, Manager m) {
    m.addUnderling(e);
}
static void connect(Manager m, Employee e) {
    m.addUnderling(e);
}
```

ELTE
IK

# Overloading on subtypes

```
static void connect(Employee e, Manager m) {
    m.addUnderling(e);
}
static void connect(Manager m, Employee e) {
    m.addUnderling(e);
}

Employee eric = new Employee("Eric", 12000);
Manager mary = new Manager("Mary", 14000);

connect(eric, mary);        connect(mary, eric);
```

ELTE
IK

# Overloading on subtypes

```
static void connect(Employee e, Manager m) {
    m.addUnderling(e);
}
static void connect(Manager m, Employee e) {
    m.addUnderling(e);
}

Employee eric = new Employee("Eric", 12000);
Manager mary = new Manager("Mary", 14000);

connect(eric, mary);        connect(mary, eric);

Manager mike = new Manager("Mike",13000);
connect(mike, mary);
```

ELTE
IK

# Overloading on subtypes

```
static void connect(Employee e, Manager m) {
    m.addUnderling(e);
}
static void connect(Manager m, Employee e) {
    m.addUnderling(e);
}

Employee eric = new Employee("Eric", 12000);
Manager mary = new Manager("Mary", 14000);

connect(eric, mary);        connect(mary, eric);

Manager mike = new Manager("Mike",13000);
connect(mike, mary);

connect(mike, (Employee)mary);
```

ELTE
IK

# Rule of thumb

**Never overload methods on subtypes!**

# Rule of thumb

**Never overload methods on subtypes!**

```
class Object {
    public boolean equals(Object that) { ... }
    ...
}
class Time {
    public boolean equals(Time that) { ... }
    ...
}
```

ELTE
IK

# Rule of thumb

**Never overload methods on subtypes!**

```
class Object {
    public boolean equals(Object that) { ... }
    ...
}
class Time {
    public boolean equals(Time that) { ... }
    ...
}

Time t = new Time(11,22);
Object o = new Time(11,22);

t.equals(t)   t.equals(o)   o.equals(t)   o.equals(o)
```

# Inheritance and equals

```
public class Time {
  ...
  @Override public boolean equals(Object that) { ... }
}
```

```
public class ExactTime extends Time {
  ...
  @Override public boolean equals(Object that) {
    return super.equals(that) && sec == ((ExactTime)that).sec;
  }
}
```

ELTE
IK

# Inheritance and equals

```
public class Time {
  ...
  @Override public boolean equals(Object that) { ... }
}
```

```
public class ExactTime extends Time {
  ...
  @Override public boolean equals(Object that) {
    return super.equals(that) && sec == ((ExactTime)that).sec;
  }
}
```

Subtyping?

```
new Time(11,22).equals(new ExactTime(11,22,33))
```

# instanceof + "fast lane"

```
public class Time {
    ...
    @Override public boolean equals(Object that) {
        if (this == that) return true;
        if (that instanceof Time) {
            Time t = (Time)that;
            return hour == t.hour && min == t.min;
        }
        return false;
    }
    @Override public int hashCode() { return 60*hour + min; }
}
```

```
new Time(11,22).equals(new ExactTime(11,22,33))
```

# Java 14+: enhanced `instanceof`

```
public class Time {
    ...
    @Override public boolean equals(Object that) {
        if (this == that) return true;
        if (that instanceof Time) {
            // no need for a downcast anymore
            return hour == time.hour && min == time.min;
        }
        return false;
    }
    @Override public int hashCode() { return 60*hour + min; }
}
```

ELTE
IK

# "Proper" equality?

```
public class Time {
    ...
    @Override public boolean equals(Object that) {
        if (that instanceof Time) {
            Time t = (Time)that;
            return hour == t.hour && min == t.min;
        } else return false;
    }
    @Override public int hashCode() { return 60*hour + min; }
}
```

Let us suppose that there is no override for equals in ExactTime

```
new ExactTime(11,22,44).equals(new ExactTime(11,22,33))
```

# Symmetry?

```
public class Time {
    @Override public boolean equals(Object that) {
        if (that instanceof Time) { ...
            return hour == that.hour && min == that.min;
        ...
```

```
public class ExactTime extends Time {
    @Override public boolean equals(Object that) {
        if (that instanceof ExactTime) { ...
            return super.equals(that) && sec == that.sec;
        ...
```

ELTE
IK

# Symmetry?

```
public class Time {
    @Override public boolean equals(Object that) {
        if (that instanceof Time) { ...
            return hour == that.hour && min == that.min;
        ...
```

```
public class ExactTime extends Time {
    @Override public boolean equals(Object that) {
        if (that instanceof ExactTime) { ...
            return super.equals(that) && sec == that.sec;
        ...
```

```
Time t1 = new Time(11,22), t2 = new ExactTime(11,22,33);
t1.equals(t2)        t2.equals(t1)
```

# Transitivity?

```
public class ExactTime extends Time {
    @Override public boolean equals(Object that) {
        if (that instanceof ExactTime) { ...
            return super.equals(that) && sec == t.sec;
        } else if (that instanceof Time) {
            return that.equals(this);
        } else {
            return false;
        }
    ...
```

ELTE
IK

# Transitivity?

```
public class ExactTime extends Time {
    @Override public boolean equals(Object that) {
        if (that instanceof ExactTime) { ...
            return super.equals(that) && sec == t.sec;
        } else if (that instanceof Time) {
            return that.equals(this);
        } else {
            return false;
        }
    ...
```

```
Time t = new Time(11,22);
ExactTime e1 = new ExactTime(11,22,33);
ExactTime e2 = new ExactTime(11,22,44);
e1.equals(t)        t.equals(e2)        e1.equals(e2)
```

# `final` methods

- overriding disallowed

# final methods

- overriding disallowed

```
public class Time {
    ...
    @Override public final boolean equals(Object that) {
        if (that instanceof Time) {
            return hour == that.hour && min == that.min;
        }
        return false;
    }
}
```

# final methods

- overriding disallowed

```
public class Time {
    ...
    @Override public final boolean equals(Object that) {
        if (that instanceof Time) {
            return hour == that.hour && min == that.min;
        }
        return false;
    }
}
```

## Compilation error

```
public class ExactTime extends Time {
    @Override public boolean equals(Object that) {
...
```

# final methods

- overriding disallowed

```
public class Time {
    ...
    @Override public final boolean equals(Object that) {
        if (that instanceof Time) {
            return hour == that.hour && min == that.min;
        }
        return false;
    }
}
```

```
ExactTime e1 = new ExactTime(11,22,44);
ExactTime e2 = new ExactTime(11,22,33);

e1.equals(e2)  // RST, but not a "real" equality
```

# final class

```
package java.lang;
public final class String implements ... { ... }
```

- Subclassing is disallowed
- Cannot specialize it, cannot change it by overriding, cannot spoil it
- Very useful for immutable design
- `java.lang.Class`, `java.lang.Integer` and other wrapper classes, `java.math.BigInteger` etc.

ELTE
IK

# final class: final definition for equality

```
public final class Time {
    ...
    @Override public boolean equals(Object that) {
        if (that instanceof Time) {
            Time t = (Time)that;
            return hour == t.hour && min == t.min;
        } else return false;
    }
}
```

# final class: final definition for equality

```
public final class Time {
    ...
    @Override public boolean equals(Object that) {
        if (that instanceof Time) {
            Time t = (Time)that;
            return hour == t.hour && min == t.min;
        } else return false;
    }
}
```

## Compilation error

```
public class ExactTime extends Time { ... }
```

ELTE IK

# Design for inheritance

- Want a class to be subclassed? Design it that way!
  - ◇ `equals`
  - ◇ `protected` visibility
  - ◇ document well how to subclass from it
  - ◇ it should not evolve (change over time)
- Want a class to *never* be subclassed? Make it `final`!

ELTE
IK

# Using composition instead of inheritance

```
public class ExactTime {
    private final Time time;
    private int sec;
    public ExactTime(int hour, int min, int sec) {
        time = new Time(hour,min);
        if (0 <= sec && sec < 60) this.sec = sec;
        else throw new IllegalArgumentException();
    }
    public int getSec() { return sec; }
    public int getMin() { return time.getMin(); }
    public void aMinPassed() { time.aMinPassed(); }
    ...
}
```

# Using composition instead of inheritance: equality

```
public final class ExactTime {
    private final Time time;
    private int sec;
    ...
    @Override public boolean equals(Object that) {
        if (this == that) return true;
        if (that instanceof ExactTime) {
            return time.equals(that.time) && sec == that.sec;
        }
        return false;
    }
}
```

ELTE
IK

# Equality of Strings

```
String verb = "ring";
String noun = "ring";

verb.equals(noun)
verb == noun
```

# Equality of Strings

```
String verb = "ring";
String noun = "ring";

verb.equals(noun)
verb == noun

String mathematical = new String("ring");
noun.equals(mathematical)
noun == mathematical
```

# Equality of Strings

```
String verb = "ring";
String noun = "ring";

verb.equals(noun)
verb == noun

String mathematical = new String("ring");
noun.equals(mathematical)
noun == mathematical
```

*Always use `equals()` on `Strings`!*

# Equality of Integers

```
Integer nineteen = 19;
Integer twentyButOne = 20-1;

nineteen.equals(twentyButOne)
nineteen == twentyButOne
```

ELTE
IK

# Equality of Integers

```
Integer nineteen = 19;
Integer twentyButOne = 20-1;

nineteen.equals(twentyButOne)
nineteen == twentyButOne

Integer dog = -123456;
Integer pup = -123456;

dog.equals(pup)
dog == pup
```

ELTE
IK

# Equality of Integers

```
Integer nineteen = 19;
Integer twentyButOne = 20-1;

nineteen.equals(twentyButOne)
nineteen == twentyButOne

Integer dog = -123456;
Integer pup = -123456;

dog.equals(pup)
dog == pup
```

*Always use `equals()` on `Integers`!*

ELTE
IK

# Comparing objects for equality

*Always use `equals()` on reference types!*

# Comparing objects for equality

*Always use `equals()` on reference types!*

- ···except for special ones.
- == works well on enum elements

## Enumeration types

```
enum Colour { RED, WHITE, GREEN }
...
if (colour1 == colour2) ...
```

# Comparing objects for equality

*Always use `equals()` on reference types!*

- ···except for special ones.
- == works well on enum elements

## Enumeration types

```
enum Colour { RED, WHITE, GREEN }
...
if (colour1 == colour2) ...
```

- Cannot instantiate
- Cannot subclass
- Can be used in `switch` statements

ELTE IK

# Equality of arrays

*Always use `equals()` on reference types!*

# Equality of arrays

*Always use `equals()` on reference types!*

- ···except when even that is not good enough.

```
int[] x = {1,2}; int[] y = {1,2};
! x.equals(y)
```

ELTE
IK

# Equality of arrays

*Always use `equals()` on reference types!*

- ···except when even that is not good enough.

```
int[] x = {1,2}; int[] y = {1,2};
! x.equals(y)
```

```
static boolean isEqualTo(int[] x, int[] y) {
    if (x == y)                          return true;
    if (x == null || y == null)          return false;
    if (x.length != y.length)            return false;
    for (int i = 0; i < x.length; ++i) {
        if (x[i] != y[i])  return false;
    }
    return true;
}
```

# Equality of arrays

*Always use `equals()` on reference types!*

- ···except when even that is not good enough.

```
int[] x = {1,2}; int[] y = {1,2};
! x.equals(y)
```

```
static boolean isEqualTo(int[] x, int[] y) {
    if (x == y)                          return true;
    if (x == null || y == null)          return false;
    if (x.length != y.length)            return false;
    for (int i = 0; i < x.length; ++i) {
        if (x[i] != y[i])  return false;
    }
    return true;
}
```


```
java.util.Arrays.equals(x,y)
```

# Equality of arrays of references

```
Integer[] x = {1,2}; Integer[] y = {1,2};
! x.equals(y)
```

# Equality of arrays of references

```
Integer[] x = {1,2}; Integer[] y = {1,2};
! x.equals(y)
```

```
static boolean isEqualTo(Integer[] x, Integer[] y) {
    if (x == y)                          return true;
    if (x == null || y == null)          return false;
    if (x.length != y.length)            return false;
    for (int i = 0; i < x.length; ++i) {
        if (!x[i].equals(y[i]))          return false;
    }
    return true;
}
```

ELTE
IK

# Equality of arrays of references – more precisely

```
Integer[] x = {1,2}; Integer[] y = {1,2};
! x.equals(y)
```

```
static boolean isEqualTo(Integer[] x, Integer[] y) {
    if (x == y)                          return true;
    if (x == null || y == null)          return false;
    if (x.length != y.length)            return false;
    for (int i = 0; i < x.length; ++i) {
        if (x[i] == y[i])                continue;
        if (x[i] == null)                return false;
        if (!x[i].equals(y[i]))          return false;
    }
    return true;
}
```

# Equality of arrays of references – more precisely

```
Integer[] x = {1,2}; Integer[] y = {1,2};
! x.equals(y)
```

```
static boolean isEqualTo(Integer[] x, Integer[] y) {
    if (x == y)                          return true;
    if (x == null || y == null)          return false;
    if (x.length != y.length)            return false;
    for (int i = 0; i < x.length; ++i) {
        if (x[i] == y[i])                continue;
        if (x[i] == null)                return false;
        if (!x[i].equals(y[i]))          return false;
    }
    return true;
}
```


```
java.util.Arrays.equals(x,y)
```

# Equality of arrays of arrays

```
int[][] x = {{1,2}};
int[][] y = {{1,2}};
! x.equals(y)
! java.util.Arrays.equals(x,y)
java.util.Arrays.deepEquals(x,y)
```

ELTE
IK

# Equality of arrays of arrays

```
int[][] x = {{1,2}};
int[][] y = {{1,2}};
! x.equals(y)
! java.util.Arrays.equals(x,y)
java.util.Arrays.deepEquals(x,y)
```

```
int[][][] x = {{{1,2}}};
int[][][] y = {{{1,2}}};
java.util.Arrays.deepEquals(x,y)
```

ELTE
IK

# Equality for fields of primitive types

```
public class Time {
  ...

  @Override public boolean equals(Object that) {
    if (that != null && getClass().equals(that.getClass())) {
      Time t = (Time)that;
      return hour == t.hour && min == t.min;
    }
    return false;
  }

  @Override public int hashCode() { return 60*hour + min; }
}
```

ELTE
IK

# Deep equality for fields of reference types

```
public class Interval {
  private Time from, to;
  ...
  @Override public boolean equals(Object that) {
    if (that != null && getClass().equals(that.getClass())) {
      Interval u = (Interval)that;
      return from.equals(u.from) && to.equals(u.to);
    }
    return false;
  }
  ...
}
```

- Only when `from` and `to` are guaranteed to be non-`null`!

ELTE
IK

# The class java.util.Objects

```
java.util.Objects.equals(Object a, Object b)
java.util.Objects.deepEquals(Object a, Object b)
java.util.Objects.hash(Object a...)
```

ELTE
IK

# equals tolerating nulls and good quality hashCode

```
import java.util.Objects;
public class Interval {
  private Time from, to;   // nulls are allowed
  ...
  @Override public boolean equals(Object that) {
    if (that != null && getClass().equals(that.getClass())) {
      Interval u = (Interval)that;
      return Objects.equals(from, u.from) &&
             Objects.equals(to, u.to);
    }
    return false;
  }
  @Override public int hashCode() {
    return Objects.hash(from, to);
  }
}
```

# Data structures

- `ArrayList<ElemType>`: sequence of elements
- `HashSet<ElemType>`: no duplicate elements
- `HashMap<KeyType, ValueType>`: dictionary with key→value lookup
- Need proper `equals` and `hashCode` on the element/key type

```
myArrayList.contains(item)

myHashSet.add(item)

myHashMap.put(key, value)
myHashMap.get(key)
```

ELTE
IK

# Heterogeneous equality

```
ArrayList<Integer> arrayList = new ArrayList<>();
LinkedList<Integer> linkedList = new LinkedList<>();

arrayList.add(19);
linkedList.add(20-1);

arrayList.equals(linkedList)
```

ELTE
IK

# Mutable object in a data structure

```
Time t = new Time(5,30);
HashSet<Time> set = new HashSet<>();
```

# Mutable object in a data structure

```
Time t = new Time(5,30);
HashSet<Time> set = new HashSet<>();

set.add(t); set.add(t); System.out.println(set);  // [5:30]
```

# Mutable object in a data structure

```
Time t = new Time(5,30);
HashSet<Time> set = new HashSet<>();

set.add(t); set.add(t); System.out.println(set);  // [5:30]

set.remove(new Time(5,30)); System.out.println(set);  // []
```

ELTE
IK

# Mutable object in a data structure

```
Time t = new Time(5,30);
HashSet<Time> set = new HashSet<>();

set.add(t); set.add(t); System.out.println(set);  // [5:30]

set.remove(new Time(5,30)); System.out.println(set);  // []

set.add(t);
t.setHour(6);
set.remove(new Time(5,30));
System.out.println(set);                          // [6:30]
```

ELTE
IK

# Mutable object in a data structure

```
Time t = new Time(5,30);
HashSet<Time> set = new HashSet<>();

set.add(t); set.add(t); System.out.println(set);  // [5:30]

set.remove(new Time(5,30)); System.out.println(set);  // []

set.add(t);
t.setHour(6);
set.remove(new Time(5,30));
System.out.println(set);                          // [6:30]

set.remove(new Time(6,30));
System.out.println(set);                          // [6:30]
```

ELTE
IK

# java Searching 42

```java
import java.util.ArrayList;
public class Searching {
  public static void main(String[] args) {
    ArrayList<String> seq = new ArrayList<>();
    seq.add("42");
    System.out.println(seq.contains("42"));
    System.out.println(seq.contains(args[0]));
  }
}
```

```
true
true
```

ELTE IK

# Searching in the data structure

```
public class ArrayList<T> {
  private Object[] data;
  private int size = 0;
  ...
  public boolean contains(T item) {
    for (int i = 0; i < size; ++i) {
      if (data[i] == item) return true;
    }
    return false;
  }
}
```

## java Searching 42

```
ArrayList<String> seq = new ArrayList<>();
seq.add("42");                          // true false
seq.contains("42") + " " + seq.contains(args[0]);
```

# By-value equality

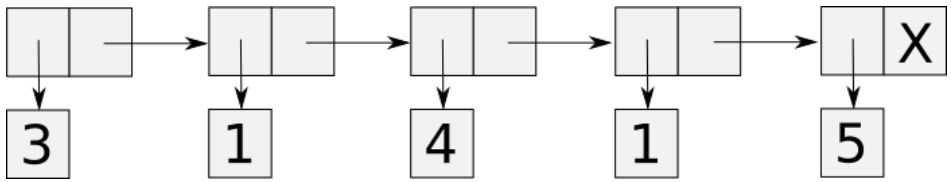
```
public class ArrayList<T> {
  private Object[] data;
  private int size = 0;
  ...
  public boolean contains(T item) {
    for (int i = 0; i < size; ++i) {
       if (Objects.equals(data[i],item)) return true;
    }
    return false;
  }
}
```

## java Searching 42

```
ArrayList<String> seq = new ArrayList<>();
seq.add("42");                          // true true
seq.contains("42") + " " + seq.contains(args[0]);
```

# User defined *Sequence* class

```
package datastructures;
public class Sequence<E> {
  public void insert(int index, E element) { ... }
  public E get(int index) { ... }
  public E remove(int index) { ... }
  public int length() { ... }
}
```




# Implementation with linked list

## datastructures/Sequence.java

```
package datastructures;
public class Sequence<E> {
  private int size = 0;
  private Node<E> first = null;
  ...
}
```

## datastructures/Node.java

```
package datastructures;
class Node<E> {
  E data;
  Node<E> next;
  Node(E data, Node<E> next) { ... }
}
```

# Several type definitions in the same compilation unit

- Too many classes have unnecessary access to the helper class

**datastructures/Sequence.java**

```
package datastructures;
public class Sequence<E> {
  private int size = 0;
  private Node<E> first = null;
  ...
}

class Node<E> {
    E data;
    Node<E> next;
    Node(E data, Node<E> next) { ... }
}
```

# Private static member class

datastructures/Sequence.java

```
package datastructures;

public class Sequence<E> {
  private int size = 0;
  private Node<E> first = null;
  ...

  private static class Node<E> {
    E data;
    Node<E> next;
    Node(E data, Node<E> next) { ... }
  }
}
```

# Generated names

- `datastructures.Sequence.Node`: byte code saved to file `datastructures/Sequence$Node.class`
- the first anonymous class of `datastructures.Sequence` has the generated name `datastructures/Sequence$1.class`

# Embedded class: class defined within another structure

- nested class: defined within a class
  - ◇ (non-static) inner class
  - ◇ static nested class
- local class: defined in code (most often, in a method):
  - ◇ with class name
  - ◇ anonymous inner class (class name is autogenerated)

```
public class OuterClass {
    class InnerClass { ... }
    static class StaticNestedClass { ... }
    public SomeClass f() {
        class LocalClass { ... }
        // anonymous class
        return new SomeClass(...) { ... };
    }
}
```

# Static type embedding: java.util.Map.Entry

```
package java.util;

public interface Map<K,V> {
  public static interface Entry<K,V> {
    ...
  }

  ...
}
```

# Lambda method (anonymous method)

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

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

ELTE
IK

# Lambda method (anonymous method)

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

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

```
java.util.Arrays.sort(
  args,
  new Comparator<String>() {
      public int compare(String left, String right) {
        return left.length() - right.length();
      }
    }
  );
```

# Iterator as static member class

```
...
public class Sequence<E> implements Iterable<E> {
  private static class Node<E> { ... }
  private Node<E> first = null;
  ...

  public Iterator<E> iterator() { return new SeqIt<>(this); }
  private static class SeqIt<E> implements Iterator<E> {
    private Node<E> current;
    SeqIt(Sequence<E> seq) { current = seq.first; }
    public boolean hasNext() { return current != null; }
    public E next() { ... }
  }
}
```

# Instance level embedding

```
...
public class Sequence<E> implements Iterable<E> {
  private static class Node<E> { ... }
  private Node<E> first = null;
  ...

  public Iterator<E> iterator() { return new SeqIt<>(this); }
  private static class SeqIt<E> implements Iterator<E> {
    private Node<E> current;
    SeqIt(Sequence<E> seq) { current = seq.first; }
    public boolean hasNext() { return current != null; }
    public E next() { ... }
  }
}
```

ELTE
IK

# Iterator as instance level member class

```
...
public class Sequence<E> implements Iterable<E> {
  private static class Node<E> { ... }
  private Node<E> first = null;
  ...

  public Iterator<E> iterator() { return new SeqIt(); }
  private class SeqIt implements Iterator<E> {
    private Node<E> current = first;

    public boolean hasNext() { return current != null; }
    public E next() { ... }
  }
}
```

ELTE
IK

# Iterator as instance level member class

```
...
public class Sequence<E> implements Iterable<E> {
  private static class Node<E> { ... }
  private Node<E> first = null;
  ...

  public Iterator<E> iterator() { return new SeqIt(); }
  private class SeqIt implements Iterator<E> {
    private Node<E> current = first;

    public boolean hasNext() { return current != null; }
    public E next() { ... }
  }
}
```


- Sequence.this.first

# Iterator as local class

```
...
public class Sequence<E> implements Iterable<E> {
  private static class Node<E> { ... }
  private Node<E> first = null;
  ...

  public Iterator<E> iterator() {
    class SeqIt implements Iterator<E> {
      private Node<E> current = first;
      public boolean hasNext() { return current != null; }
      public E next() { ... }
    };
    return new SeqIt();
  }
}
```

# Iterator as anonymous class

```
...
public class Sequence<E> implements Iterable<E> {
  private static class Node<E> { ... }
  private Node<E> first = null;
  ...

  public Iterator<E> iterator() {
    return new Iterator<E>() {
      private Node<E> current = first;
      public boolean hasNext() { return current != null; }
      public E next() { ... }
    };

  }
}
```

ELTE
IK