

Computer Network Programming

## A Review on Object-Orientation, Concurrency and Java

Dr. Petru Florin Mihancea

V20180301

1

## Object-Oriented Decomposition

Dr. Petru Florin Mihancea

### Decomposition

**Software system**

### Decomposition

Sub-system 2

Sub-system 3

Sub-system 1

Sub-system 4

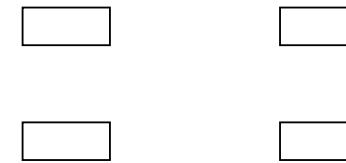
**Software system**

## Decomposition

Sub-system I

Dr. Petru Florin Mihăescu

## Decomposition

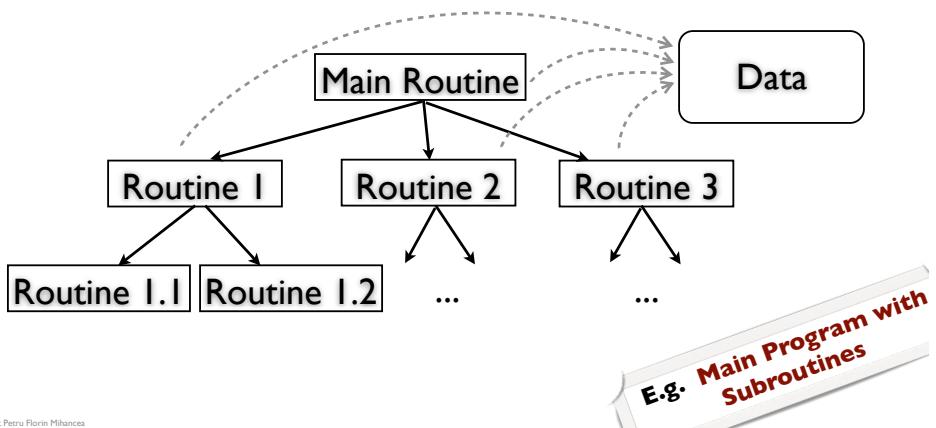


What are these  
“entities” ?

Dr. Petru Florin Mihăescu

## Algorithmic Decomposition

each “entity” in the system denotes a major step in some overall process

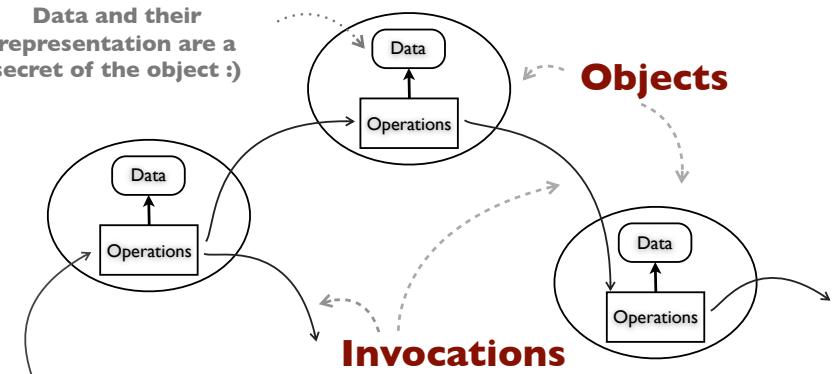


Dr. Petru Florin Mihăescu

## Object-Oriented Decomposition

the system is decomposed into a set of collaborating objects (and their classes)

Data and their representation are a secret of the object :)



Dr. Petru Florin Mihăescu

# 2

## Classes and Objects

Dr. Petru Florin Mihăescu



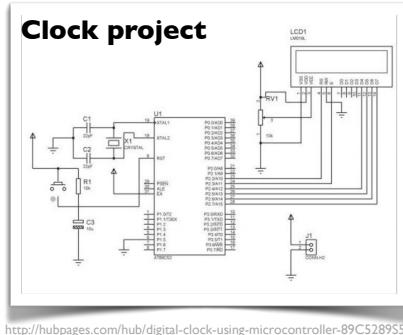
<http://www.timetools.co.uk>



Objects



### Class vs. Object



Class

An object's implementation is defined by its class. The class specifies the object's internal data and representation and defines object's operations

### Raw Definitions

A **class** is a set of objects that share a common structure and a common behavior

An **object** has state, behavior [...] the structure and behavior of similar objects are defined in their common class

A single object is simply an instance of a class

Bloch - OO Analysis and Design

Dr. Petru Florin Mihăescu

### Anatomy of a Java Class

```
class Clock {  
    //Instance variables / Instance Fields  
    //access_modifiers type var_name1, ...;  
    private int hour, minute;
```

#### Instance variables

- define object's state representation (the representation of the data the objects know)
- each object has its own copy (memory locations) for each instance variable
- access modifiers
  - private - can be accessed only in this class
  - public - can be accessed from everywhere
  - if missing, the field can be accessed from the same package

Dr. Petru Florin Mihăescu

# Anatomy of a Java Class

```
class Clock {  
    //Instance variables / Instance Fields  
    //access_modifiers type var_name1, ...;  
    private int hour, minute;  
    //Methods  
    //modifiers returned_type method_name(type par_name, ...) { ... }  
    public void setTime(int h, int m) {  
        hour = (h >= 0) && (h < 24) ? h : 0;  
        minute = (m >= 0) && (m < 60) ? m : 0;  
    }  
    public String toString() {  
        return hour + ":" + minute;  
    }  
}
```

## Methods

- define object's behavior (the implementation of an object operations)
- can use/change the state (current values of instance variables) of an object
- access modifiers are like in the case of fields

```
class Clock {  
    //Instance variables / Instance Fields  
    //access_modifiers type var_name1, ...;  
    private int hour, minute;  
    //Methods  
    //modifiers returned_type method_name(type par_name, ...) { ... }  
    public void setTime(int h, int m) {  
        hour = (h >= 0) && (h < 24) ? h : 0;  
        minute = (m >= 0) && (m < 60) ? m : 0;  
    }  
    public String toString() {  
        return hour + ":" + minute;  
    }  
    //Constructors  
    //access_modifiers Class_Name(type par_name, ...) {}  
    public Clock() {  
        hour = minute = 0;  
    }  
    public Clock(int h, int m) {  
        setTime(h,m);  
    }  
}
```

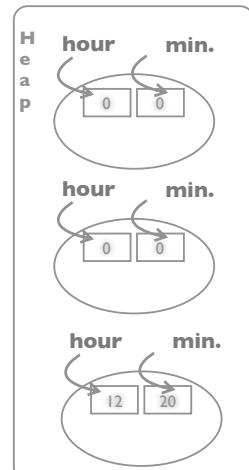
## Constructors

- special methods executed only once at the object creation time
- used to initialize the object state
- always has the same name as the class and does not have return type (neither void)
- if no constructor, the compiler generates one with an empty parameter list (no-arg constructor)

# Creating an Object in Java

created at runtime and allocated in the heap

```
class Main {  
    public static void main(String arg[]) {  
        ...  
        new Clock();  
        new Clock();  
        new Clock(12,20);  
    }  
}
```

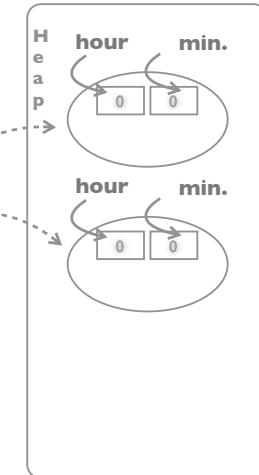


What about destruction ?  
Don't worry ! The garbage collector takes care of this

```
class Main {  
    public static void main(String arg[]) {  
        ...  
        //Reference variables  
        //Here local variables but they can also  
        //be instance variables, etc.  
        Clock firstClock, secondClock;  
        firstClock = new Clock();  
        secondClock = new Clock();  
    }  
}
```

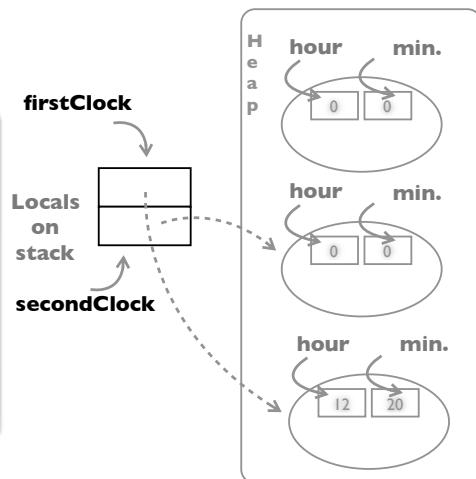
# Referring an Object in Java

firstClock  
Locals on stack  
secondClock



## Referring an Object in Java

```
class Main {
    public static void main(String args[]) {
        ...
        //Reference variables
        //Here local variables but they can also
        //be instance variables, etc.
        Clock firstClock, secondClock;
        firstClock = new Clock();
        secondClock = new Clock();
        firstClock = new Clock(12,20);
        ...
    }
}
```

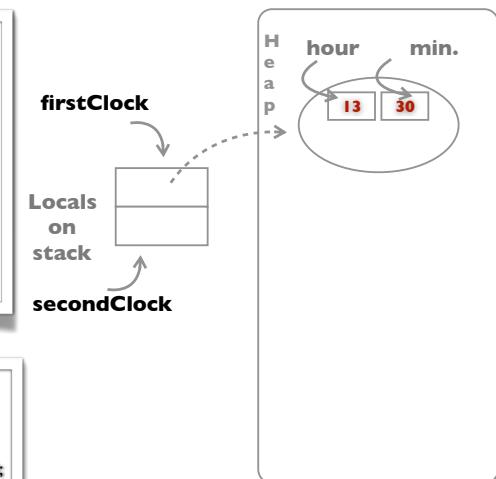


Dr. Petru Florin Mihăescu

## Invoking an Object in Java

```
class Main {
    public static void main(String args[]) {
        ...
        Clock firstClock, secondClock;
        firstClock = new Clock();
        firstClock.setTime(13,30);
    }
}
```

```
class Clock {
    ...
    public void setTime(int h, int m) {
        hour = (h >= 0) && (h < 24) ? h : 0;
        minute = (m >= 0) && (m < 60) ? m : 0;
    }
    ...
}
```

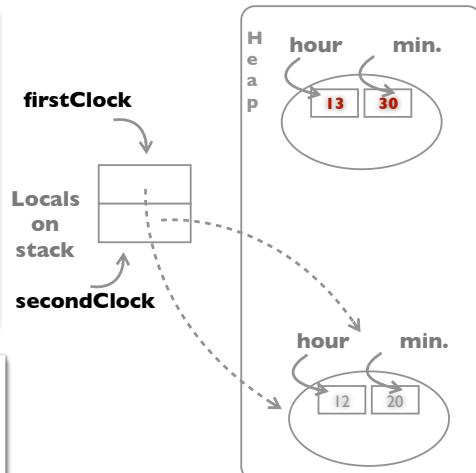


Dr. Petru Florin Mihăescu

## Invoking an Object in Java

```
class Main {
    public static void main(String args[]) {
        ...
        Clock firstClock, secondClock;
        firstClock = new Clock();
        firstClock.setTime(13,30);
        secondClock = new Clock(12,20);
        firstClock = secondClock;
    }
}
```

```
class Clock {
    ...
    public void setTime(int h, int m) {
        hour = (h >= 0) && (h < 24) ? h : 0;
        minute = (m >= 0) && (m < 60) ? m : 0;
    }
    ...
}
```

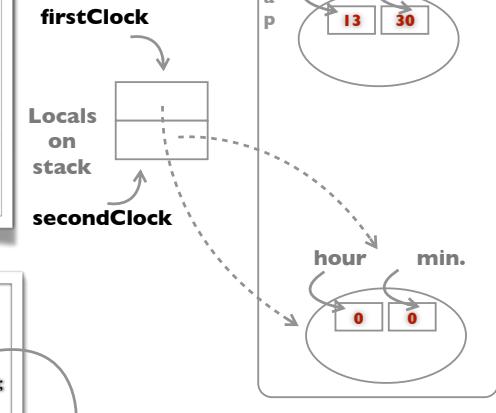


Dr. Petru Florin Mihăescu

```
class Main {
    public static void main(String args[]) {
        ...
        Clock firstClock, secondClock;
        firstClock = new Clock();
        firstClock.setTime(13,30);
        secondClock = new Clock(12,20);
        firstClock = secondClock;
        firstClock.setTime(0,0);
        //secondClock.toString() returns "0:0"
    }
}
```

```
class Clock {
    ...
    public void setTime(int h, int m) {
        hour = (h >= 0) && (h < 24) ? h : 0;
        minute = (m >= 0) && (m < 60) ? m : 0;
    }
    ...
}
```

## Invoking an Object in Java



Dr. Petru Florin Mihăescu

How does the method know on which copy of hour and minute should work?

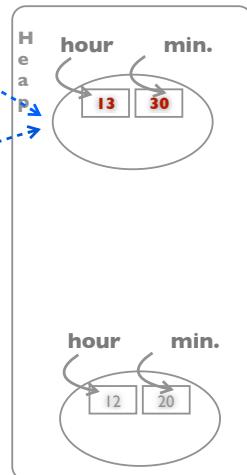
## The Special **this** Reference

```
class Clock {
    ...
    public void setTime(int h, int m) {
        this.hour = (h >= 0) && (h < 24) ? h : 0;
        this.minute = (m >= 0) && (m < 60) ? m : 0;
    }
}
...
```

during the execution of a method, **this** refers to the object in front of the call

```
...
Clock firstClock, secondClock;
firstClock = new Clock();
secondClock = new Clock(12,20);
firstClock . setTime(13,30);
secondClock . setTime(0,0);
...

```



Dr. Petru Florin Mihăescu

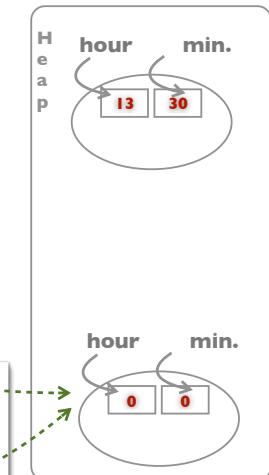
## The Special **this** Reference

```
class Clock {
    ...
    public void setTime(int h, int m) {
        this.hour = (h >= 0) && (h < 24) ? h : 0;
        this.minute = (m >= 0) && (m < 60) ? m : 0;
    }
}
...
```

during the execution of a method, **this** refers to the object in front of the call

```
...
Clock firstClock, secondClock;
firstClock = new Clock();
secondClock = new Clock(12,20);
firstClock . setTime(13,30);
secondClock . setTime(0,0);
...

```

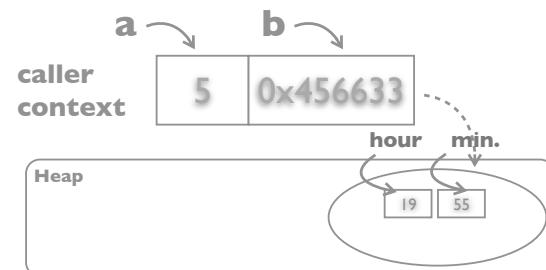


Dr. Petru Florin Mihăescu

## Parameter Passing in Java

**passed by value**

```
void caller() {
    int a = 5;
    Clock b = new Clock();
    b.setTime(19,55);
    called(a,b);
}
```



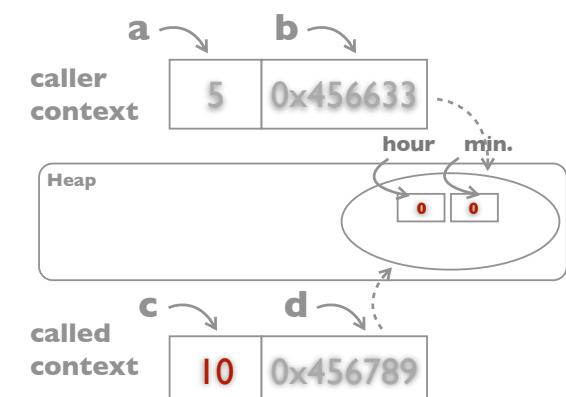
Dr. Petru Florin Mihăescu

## Parameter Passing in Java

**passed by value**

```
void caller() {
    int a = 5;
    Clock b = new Clock();
    b.setTime(19,55);
    called(a,b);
}
```

```
void called(int c, Clock d) {
    c = 10;
    d.setTime(0,0);
}
```



Dr. Petru Florin Mihăescu

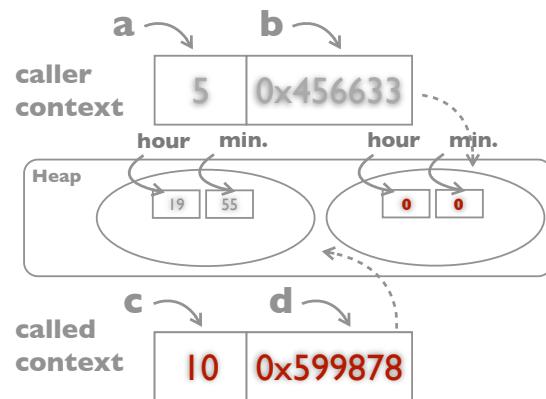
# Parameter Passing in Java

## passed by value

```
void caller() {  
    int a = 5;  
    Clock b = new Clock();  
    b.setTime(19,55);  
    called(a, b);  
}
```

```
void called(int c, Clock d) {  
    c = 10;  
    d.setTime(0,0);  
    d = new Clock(19,55);  
}
```

Dr. Petru Florin Mihăescu



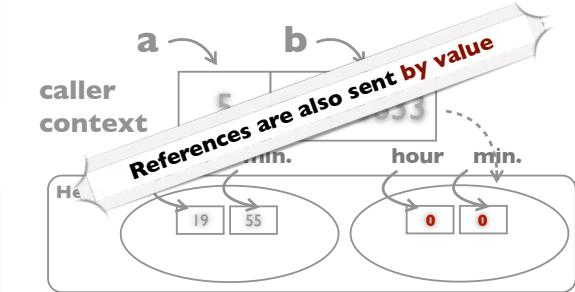
# Parameter Passing in Java

## passed by value

```
void caller() {  
    int a = 5;  
    Clock b = new Clock();  
    b.setTime(19,55);  
    called(a, b);  
    //a is still 5  
    //b refers to the same clock !  
    //its time indication is 0:0  
}
```

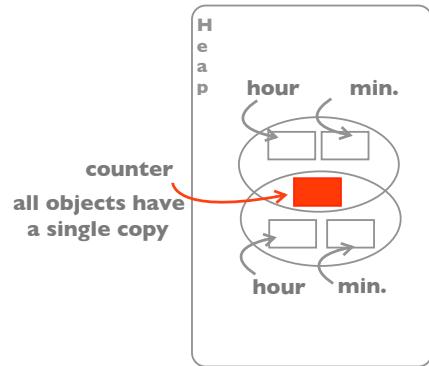
```
void called(int c, Clock d) {  
    c = 10;  
    d.setTime(0,0);  
    d = new Clock(19,55);  
}
```

Dr. Petru Florin Mihăescu



# The static Modifier

```
class Clock {  
    ...  
    //counts the number of clocks created  
    //static - class variable/field/method  
    private static int count;  
    public static int getCount() {  
        return count;  
    }  
    public Clock() {  
        count++;  
        hour = minute = 0;  
    }  
    public Clock(int h, int m) {  
        count++;  
        setTime(h,m);  
    }  
}
```

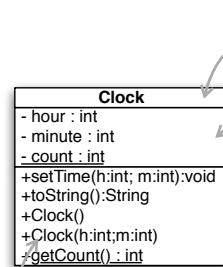


static methods do not execute on an object

- **this** does not make sense in static methods
- cannot access instance fields via **this** (implicitly or explicitly)
- usage: **ClassName.methodName(params)**

Dr. Petru Florin Mihăescu

# A Class in UML (basics)



Name  
Attributes  
visibility(+/-) name : type

Operations  
visibility name(param\_list) : return\_type

param\_name : type

visibility  
+ public  
- private

static features  
are underlined

# 3

## Inheritance and Polymorphism

Dr. Petru Florin Mihăescu

## Inheritance

**inheritance is a relationship among classes wherein one class shares the structure and/or behavior defined in other classes**

Two “flavors”

- **class inheritance**
- **type inheritance**



Booch - OO Analysis and Design

Dr. Petru Florin Mihăescu

## Class inheritance in Java

```
class Clock {
    private int hour, minute;
    public void setTime(int h, int m) {
        hour = (h >= 0) && (h < 24) ? h : 0;
        minute = (m >= 0) && (m < 60) ? m : 0;
    }
    public String toString() {
        return hour + ":" + minute;
    }
    public Clock() {
        hour = minute = 0;
    }
    public Clock(int h, int m) {
        setTime(h,m);
    }
}
```

```
class EnhancedClock {
    private int hour, minute, second;
    public void setTime(int h, int m) {
        hour = (h >= 0) && (h < 24) ? h : 0;
        minute = (m >= 0) && (m < 60) ? m : 0;
    }
    public String toString() {
        return hour + ":" + minute;
    }
    public EnhancedClock() {
        hour = minute = second = 0;
    }
    public EnhancedClock(int h, int m, int s) {
        setTime(h,m);
        second = (s >=0) && (s < 60) ? s : 0;
    }
    public void setEnhancedTime(int h, int m, int s) {
        setTime(h,m);
        second = (s >=0) && (s < 60) ? s : 0;
    }
}
```

A lot of duplication

Dr. Petru Florin Mihăescu

## Superclass

```
class Clock {
    private int hour, minute;
    public void setTime(in h, int m) {
        hour = (h >= 0) && (h < 24) ? h : 0;
        minute = (m >= 0) && (m < 60) ? m : 0;
    }
    public String toString() {
        return hour + ":" + minute;
    }
    public Clock() {
        hour = minute = 0;
    }
    public Clock(int h, int m) {
        setTime(h,m);
    }
}
```

Dr. Petru Florin Mihăescu

## Subclass

```
class EnhancedClock extends Clock {
    private int second;
    public String toString() {
        return super.toString() + ":" + second;
    }
    public EnhancedClock() {
        second = 0;
    }
    public EnhancedClock(int h, int m, int s) {
        super(h,m);
        second = (s >=0) && (s < 60) ? s : 0;
    }
    public void setEnhancedTime(int h, int m, int s) {
        setTime(h,m);
        second = (s >=0) && (s < 60) ? s : 0;
    }
}
```



**A simple code factorization (reuse) mechanism**

# Some Important Details

```
class EnhancedClock extends Clock {
    private int second;
    public String toString() {
        return super.toString() + ":" + second;
    }
    public EnhancedClock() {
        second = 0;
    }
    public EnhancedClock(int h, int m) {
        super(h,m);
        second = (s >=0) && (s < 60) ? s : 0;
    }
    public void setEnhancedTime(int h, int m, int s) {
        setTime(h,m);
        second = (s >=0) && (s < 60) ? s : 0;
    }
}
```

## Constructors

- the first instruction in a constructor must be a call to a constructor from superclass

If the superclass has a no-arg constructor, the compiler implicitly adds a call to it

Dr. Petru Florin Mihăescu

```
class EnhancedClock extends Clock {
    private int second;
    public String toString() {
        return super.toString() + ":" + second;
    }
    public EnhancedClock() {
        second = 0;
    }
    public EnhancedClock(int h, int m, int s) {
        super(h,m);
        second = (s >=0) && (s < 60) ? s : 0;
    }
    public void setEnhancedTime(int h, int m, int s) {
        setTime(h,m);
        second = (s >=0) && (s < 60) ? s : 0;
    }
}
```

## Constructors

- the first instruction in a constructor must be a call to a constructor from superclass

## Visibility

- subclass cannot access private members from superclass
- it can access them if they are declared using the protected access modifier (# in UML)

## More details

- an inherited method can be redefined (overridden)
- a class can extend only one class
- if a class is used only to factor code (does not have objects of itself) make it abstract (see later)
- Object is superclass for all classes

Dr. Petru Florin Mihăescu

# Some Important Details

```
class EnhancedClock extends Clock {
    private int second;
    public String toString() {
        return super.toString() + ":" + second;
    }
    public EnhancedClock() {
        second = 0;
    }
    public EnhancedClock(int h, int m) {
        super(h,m);
        second = (s >=0) && (s < 60) ? s : 0;
    }
    public void setEnhancedTime(int h, int m, int s) {
        setTime(h,m);
        second = (s >=0) && (s < 60) ? s : 0;
    }
}
```

## Constructors

- the first instruction in a constructor must be a call to a constructor from superclass

### Overriding example

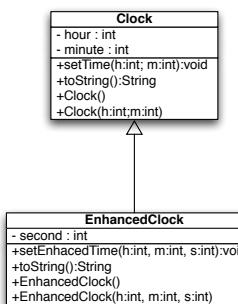
- the method has the same signature as the inherited method
- the old implementation can be invoked using super.methodName(...)

- it can access them if they are declared using the protected access modifier (# in UML)

## More details

- an inherited method can be redefined (overridden)
- a class can extend only one class
- if a class is used only to factor code (does not have objects of itself) make it abstract (see later)
- Object is superclass for all classes

Dr. Petru Florin Mihăescu



# In UML

## Constructors

- the first instruction in a constructor must be a call to a constructor from superclass

## Visibility

- subclass cannot access private members from superclass
- it can access them if they are declared using the protected access modifier (# in UML)

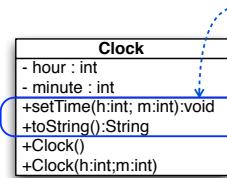
## More details

- an inherited method can be redefined (overridden)
- a class can extend only one class
- if a class is used only to factor code (does not have objects of itself) make it abstract (see later)
- Object is superclass for all classes

Dr. Petru Florin Mihăescu

## Second flavor - Type vs. Class

### The interface of Clock objects



- the set of all **declarations (not the bodies)** of the public operations an object provides
- ~ this interface specifies the **type** of an object

### Conceptually class != type

- a class is actually the **implementation** of a type
- a type is ~ the **declarations** of the public operations of a class

### Source of interchangeable use

- a class also defines the interface (operation declarations) of an object and thus also specifies the object type

Dr. Petru Florin Mihăescu

## Pure Type Declaration in Java

```
<<interface>>
ClockInterface
+setTime(h:int, m:int):void
+toString():String
```

```
interface ClockInterface {
    void setTime(int h, int m);
    String toString();
}
```

### Case 1 : Using an interface

- all members are **implicitly public**
- methods **do not have bodies** (are abstract)
  - interfaces cannot be instantiated
- we cannot specify any implementation detail of an object
  - we cannot have instance fields
  - fields are implicitly static and final (like a constant)

Dr. Petru Florin Mihăescu

## Pure Type Declaration in Java

```
{abstract}
ClockInterface
+setTime(h:int, m:int):void
+toString():String
```

```
abstract class ClockInterface {
    public abstract void setTime(int h, int m);
    public abstract String toString();
}
```

### Case 2 : Using a pure abstract class

- a class having only abstract methods
- abstract methods do not have bodies
- a class with abstract methods must be declared abstract
- abstract classes cannot be instantiated

Dr. Petru Florin Mihăescu

## Type Inheritance

### Refers to an inheritance relations between types

a **type (subtype)** inherits some operation declarations from another type (**supertype**)

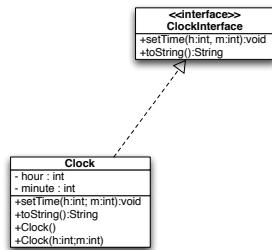
Dr. Petru Florin Mihăescu

# Type Inheritance in Java

## Case 1 Using interfaces

```
interface ClockInterface {  
    void setTime(int h, int m);  
    String toString();  
}
```

```
class Clock implements ClockInterface {  
    private int hour, minute;  
    public void setTime(int h, int m) {  
        hour = (h >= 0) && (h < 24) ? h : 0;  
        minute = (m >= 0) && (m < 60) ? m : 0;  
    }  
    public String toString() {  
        return hour + ":" + minute;  
    }  
    public Clock() {  
        hour = minute = 0;  
    }  
    public Clock(int h, int m) {  
        setTime(h,m);  
    }  
}
```



Dr. Petru Florin Mihăescu

# Type Inheritance in Java

## Case 2 Using classes

```
abstract class ClockInterface {  
    public abstract void setTime(int h, int m);  
    public abstract String toString();  
}
```

**Between classes,  
extends means  
both class and  
type inheritance**

```
class Clock extends ClockInterface {  
    private int hour, minute;  
    public void setTime(int h, int m) {  
        hour = (h >= 0) && (h < 24) ? h : 0;  
        minute = (m >= 0) && (m < 60) ? m : 0;  
    }  
    public String toString() {  
        return hour + ":" + minute;  
    }  
    public Clock() {  
        hour = minute = 0;  
    }  
    public Clock(int h, int m) {  
        setTime(h,m);  
    }  
}
```

Dr. Petru Florin Mihăescu

# Type Declaration in Java

## 1. Using (abstract) classes

### Pros

- we can combine type declaration with common implementation of subtypes (not all methods must be abstract in a Java abstract class, it can have instance fields)

### Cons

- a class can extends only one other class

## 2. Using interfaces

### Pros

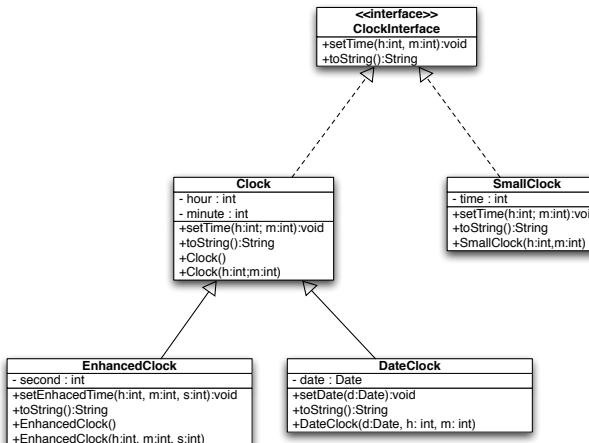
- a class can implement multiple interfaces (multiple types)

### Cons

- a little more difficult to factor common implementation

Dr. Petru Florin Mihăescu

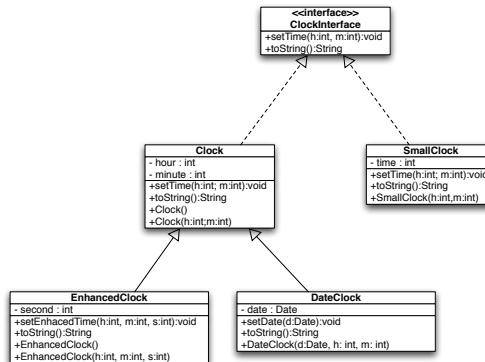
# Class Hierarchy Example



Dr. Petru Florin Mihăescu

# The effect of type inheritance

**Polymorphism:** A reference variable of a declared type (class) can refer objects of that type (class) and of any of its subtypes (subclasses)



```

ClockInterface a;
Clock b;

a = new Clock();
a = new EnhancedClock();
a = new SmallClock(5,6);

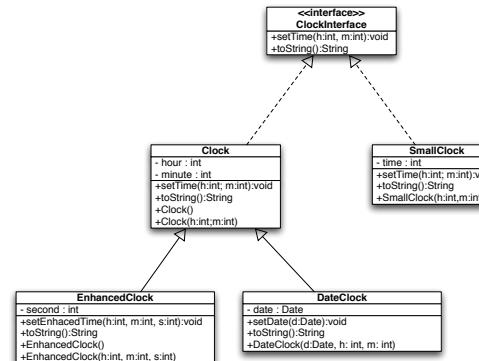
b = new Clock();
b = new EnhancedClock();
b = new SmallClock(5,6); //Compile error

a = b;
b = a; //Compile error
  
```

Dr. Petru Florin Mihăescu

# Correct Invocations

On a reference variable we can invoke any instance operation specified by its declared type (class) or by one of its supertypes (superclasses)



```

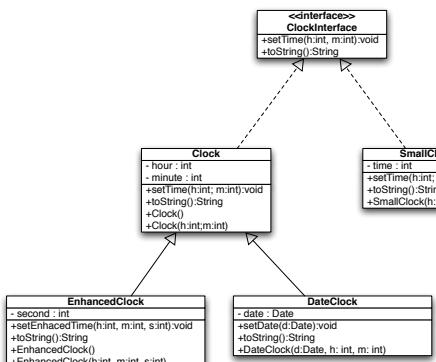
ClockInterface a;
Clock b;
EnhancedClock c;

a.setTime(0,0);
b.setTime(0,0);
c.setTime(0,0);
c.setEnhancedTime(0,0,0); //Compile error
b.setEnhancedTime(0,0,0); //Compile error

b = new EnhancedClock();
b.setEnhancedTime(0,0,0); //Compile error
  
```

Dr. Petru Florin Mihăescu

# Runtime Type Check and Casts



```

void someMethod(ClockInterface a) {
    // here a could refer instances of
    // Clock, SmallClock, EnhancedClock, DateClock
    // We don't know exactly but we can find out at
    // runtime
    if(a instanceof EnhancedClock) {
        a.setEnhancedTime(0,0,0); //Still compile error
        ((EnhancedClock)a).setEnhancedTime(0,0,0);
    }
    ((EnhancedClock)a).setEnhancedTime(0,0,0);
    //Correct at compile time but at runtime
    //your program can crash if a does not refer an
    //EnhancedClock
}
  
```

Dr. Petru Florin Mihăescu

# Dynamic binding

```

interface ClockInterface {
    void setTime(int h, int m);
    String toString();
}

class Clock implements ClockInterface {
    ...
    public String toString() {
        return hour + ":" + minute;
    }
    ...
}

class EnhancedClock extends Clock {
    ...
    //Remember that this method overrides the
    //inherited toString
    public String toString() {
        return super.toString() + ":" + second;
    }
    ...
}

class SmallClock implements ClockInterface {
    ...
    public String toString() {
        return // bits operations to extract data
    }
    ...
}
  
```

Hmmm, what code is invoked ???

```
interface ClockInterface {
    void setTime(int h, int m);
    String toString();
}
```

```
class Clock implements ClockInterface {
    ...
    public String toString() {
        return hour + ":" + minute;
    }
    ...
}
```

```
class EnhancedClock extends Clock {
    ...
    //Remember that this method overrides the
    //inherited toString
    public String toString() {
        return super.toString() + ":" + second;
    }
    ...
}
```

```
class SmallClock implements ClockInterface {
    ...
    public String toString() {
        return // bits operations to extract data
    }
    ...
}
```

## Dynamic binding

```
void someMethod(ClockInterface a) {
    // here a could refer at runtime instances of
    // Clock, SmallClock, EnhancedClock, DateClock
    // so, multiple overridden versions of toString exists
    String s = a.toString();
    ...
}
```

### It depends

If **a** refers an instance of EnhancedClock class  
then goto `toString` from EnhancedClock  
else  
if **a** refers an instance of Clock class  
then goto `toString` from Clock  
...

**The binding between the call and the executed implementation is performed at runtime**

```
interface ClockInterface {
    void setTime(int h, int m);
    String toString();
}
```

```
class Clock implements ClockInterface {
    ...
    public String toString() {
        return hour + ":" + minute;
    }
    ...
}
```

```
class EnhancedClock extends Clock {
    ...
    //Remember that this method overrides the
    //inherited toString
    public String toString() {
        return super.toString() + ":" + second;
    }
    ...
}
```

```
class SmallClock implements ClockInterface {
    ...
    public String toString() {
        return // bits operations to extract data
    }
    ...
}
```

## Dynamic binding

```
void someMethod(ClockInterface a) {
    // here a could refer at runtime instances of
    // Clock, SmallClock, EnhancedClock, DateClock
    // so, multiple overridden versions of toString exists
    String s = a.toString();
    ...
}
```

### It depends

If **a** refers an instance of EnhancedClock class  
then goto `toString` from EnhancedClock  
else  
if **a** refers an instance of Clock class  
then goto `toString` from Clock  
...

**This “procedure” is done automatically by the execution environment for calls to instance methods (public/protected)**

**Between the call and the executed implementation is performed at runtime**

## Object Class

### `toString():String`

- used by many library methods to convert an object into a string representation
  - e.g. displaying purposes
- Object implementation returns object's class name + its hash code
- it can be excluded from the previous example interfaces and abstract classes

Object
<code>+toString():String</code> <code>+equals(o:Object):boolean</code> <code>+hashCode():int</code>

### `equals(Object):boolean`

- used by many library methods (e.g. collection system) to compare two objects for equality
- Object implementation returns true when `this == o`

You should use (override) them when you have to implement similar features in your classes

```
class MyInt {
    private int n;
    public MyInt(int n) {
        this.n = n;
    }
    public String toString() {
        return n + "";
    }
    public boolean equals(Object o) {
        if(o instanceof MyInt) {
            return this.n == ((MyInt)o).n;
        } else return false;
    }
}
```

```
MyInt a = new MyInt(9);
MyInt b = new MyInt(9);
```

```
System.out.println(a); //Prints on screen "9"
System.out.println(a==b); //Prints on screen "false"
//Two references are equal only when they refer
//to the same object
```

```
System.out.println(a.equals(b)); //Prints on screen "true"
```

## Examples

# 4

## Threads and Synchronization Mechanisms in Java

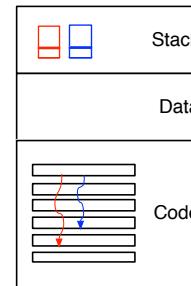
Dr. Petru Florin Mihăescu

```
class Square {  
    private int l, L;  
    public void set(int a) {  
        l = a; // l = 10 (a)  
        L = a;  
        if (l != L) {  
            //Can we get here?  
        }  
    }  
}  
  
class Task extends Thread {  
    private Square s;  
    private int l;  
    public Task(Square a, int b) { s = a; l = b;}  
    public void run() {  
        while (true) { s.set(l); }  
    }  
}  
  
class Main {  
    public static void main(String argv[]) {  
        Square s = new Square();  
        (new Task(s,10)).start();  
        (new Task(s,5)).start();  
    }  
}
```

A BIG Concern

Sometimes hazards may occur on shared data and thus, synchronisation/cooperation mechanisms are required

Many mechanisms exist to ensure thread safeness, but we discuss only one: monitors



Dr. Petru Florin Mihăescu

**Execution Thread**  
**A single sequential flow of control within a process**

```
//Defining a thread in Java  
class MyTask extends Thread {  
    public void run() {  
        //The "main" of the thread  
    }  
}
```

```
//Starting a new thread  
MyTask t = new MyTask();  
t.start();
```

All threads are executed in parallel :)

## Synchronized Methods

“It is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object”

<https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html>

```
class Square {  
    private int l, L;  
    public synchronized void set(int a) {  
        l = a;  
        L = a;  
        if (l != L) {  
            //Can we get here? No :).  
        }  
    }  
}
```

Dr. Petru Florin Mihăescu

## Synchronized Blocks

```
synchronized (obj) {  
    ...  
}
```

Similarly, it is  
**not possible**  
to interleave the  
execution of such  
blocks (or sync methods)  
that are  
synchronized using  
the **same obj**  
instance

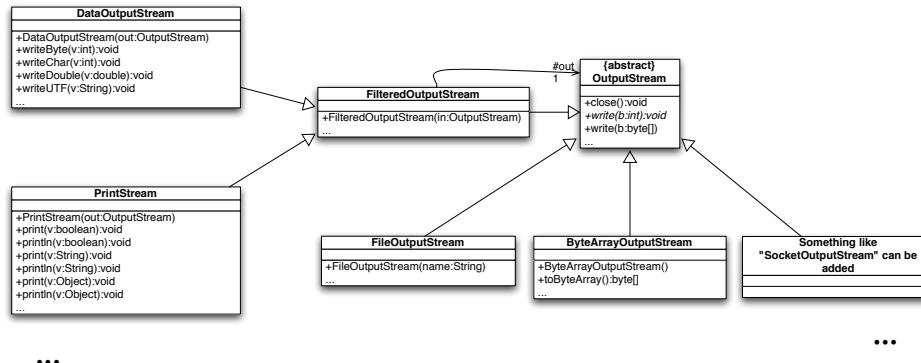
Dr. Petru Florin Mihăescu

5

## A Simplified View on java.io System

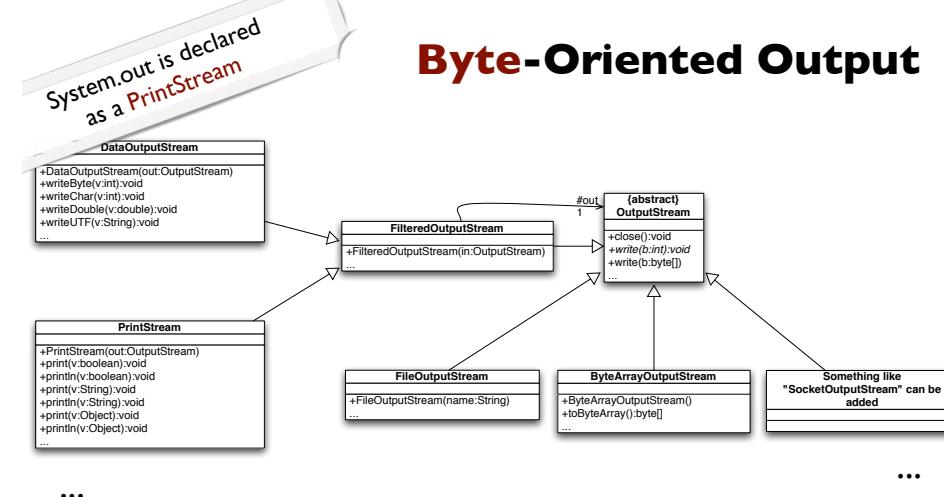
Dr. Petru Florin Mihăescu

## Byte-Oriented Output



```
DataOutputStream dos =  
new DataOutputStream(new FileOutputStream("a.dat"));
```

Dr. Petru Florin Mihăescu



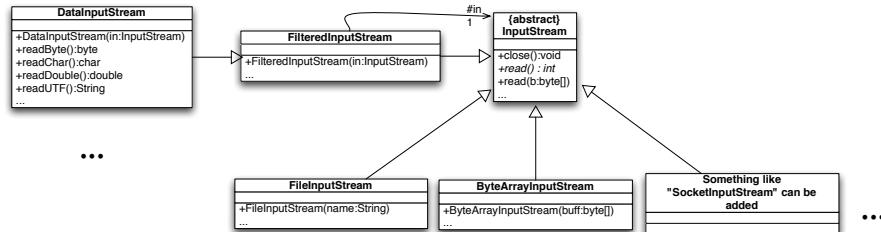
```
DataOutputStream dos =  
new DataOutputStream(new ByteArrayOutputStream());
```

//toByteArray returns a byte[] with the written data

...and many other  
combinations!

Dr. Petru Florin Mihăescu

## Byte-Oriented Input



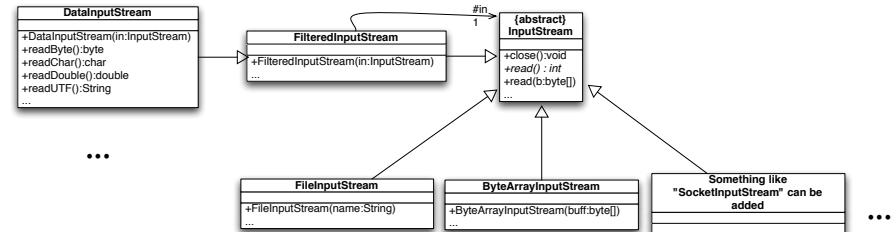
```

DataInputStream dis =
new DataInputStream(new FileInputStream("a.dat"));
  
```

Dr. Petru Florin Mihancea

System.in is declared  
as an *InputStream*

## Byte-Oriented Input



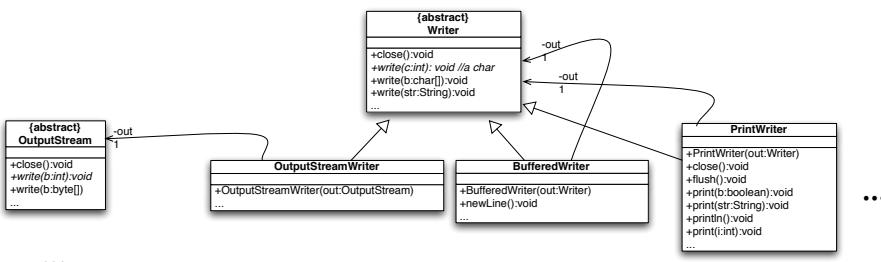
```

byte[] tmp = new byte[1024];
//put data in tmp
DataInputStream dis =
new DataInputStream(new ByteArrayInputStream(tmp));
  
```

... and many other combinations !

Dr. Petru Florin Mihancea

## Char-Oriented Output



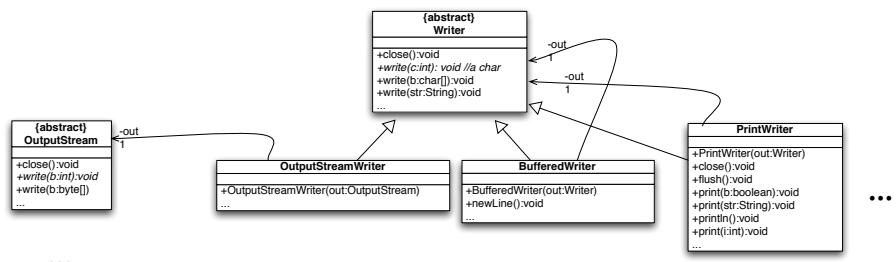
```

//be careful on flushing
PrintWriter pw =
new PrintWriter(
    new BufferedWriter(
        new OutputStreamWriter(
            new FileOutputStream("a.txt"))));
  
```

Dr. Petru Florin Mihancea

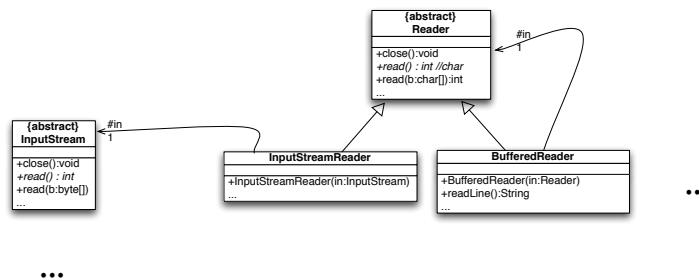
PrintWriter pw =
new PrintWriter(
 new BufferedWriter(
 new OutputStreamWriter(
 System.out)));

... and many other combinations !



Dr. Petru Florin Mihancea

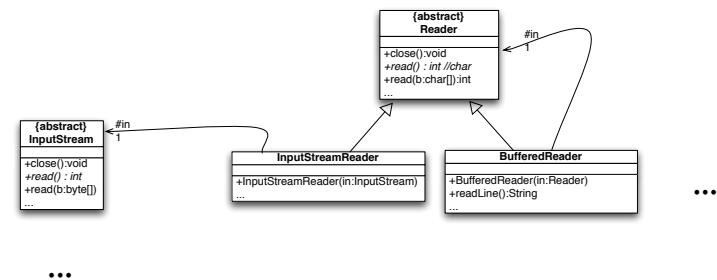
## Char-Oriented Input



```
BufferedReader br =
new BufferedReader(
    new InputStreamReader(
        new FileInputStream("a.txt")));
...
```

Dr. Petru Florin Mihăescu

## Char-Oriented Input



```
BufferedReader br =
new BufferedReader(
    new InputStreamReader(System.in));
...
```

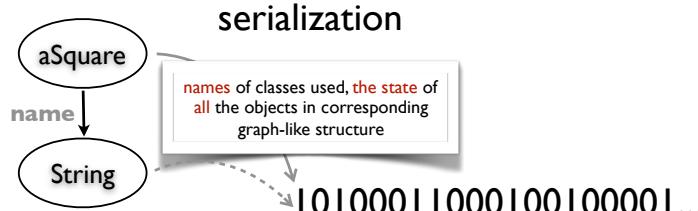
...and many other combinations !

Dr. Petru Florin Mihăescu

## Object Serialization/De-serialization

```
class Square implements Serializable {
    ...
    private int x;
    private String name;
    ...
}
```

**The class of a serializable object must be descendant of the `Serializable` interface**

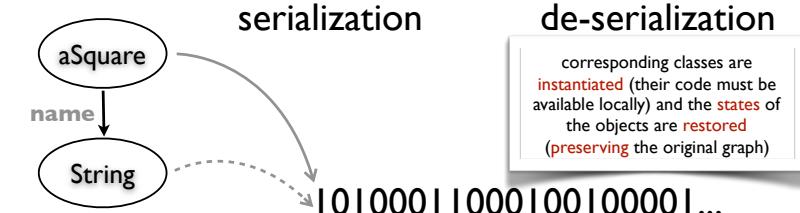


Dr. Petru Florin Mihăescu

## Object Serialization/De-serialization

```
class Square implements Serializable {
    ...
    private int x;
    private String name;
    ...
}
```

**The class of a serializable object must be descendant of the `Serializable` interface**

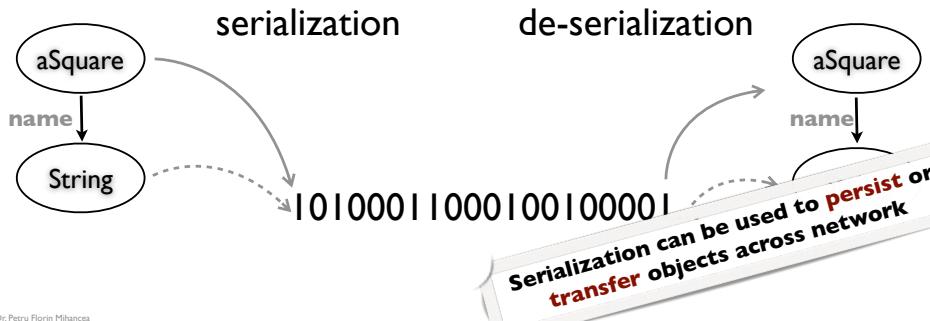


Dr. Petru Florin Mihăescu

## Object Serialization/De-serialization

```
class Square implements Serializable {  
    ...  
    private int x;  
    private String name;  
    ...  
}
```

**The class of a serializable object must be descendant of the Serializable interface**



Dr. Petru Florin Mihăescu

## Object Serialization/De-serialization

```
Square s = new Square();  
//Any kind of output stream  
//e.g. FileOutputStream, ByteArrayOutputStream, etc.  
OutputStream os = ...  
ObjectOutputStream oos = new ObjectOutputStream(os);  
oos.writeObject(s);
```

```
//Any kind of input stream  
//e.g. FileInputStream, ByteArrayInputStream, etc.  
InputStream is = ...  
ObjectInputStream ois = new ObjectInputStream(is);  
Square s = (Square)ois.readObject();
```

Dr. Petru Florin Mihăescu

# 6

## Miscellaneous

Dr. Petru Florin Mihăescu

## Intercepting Exceptions

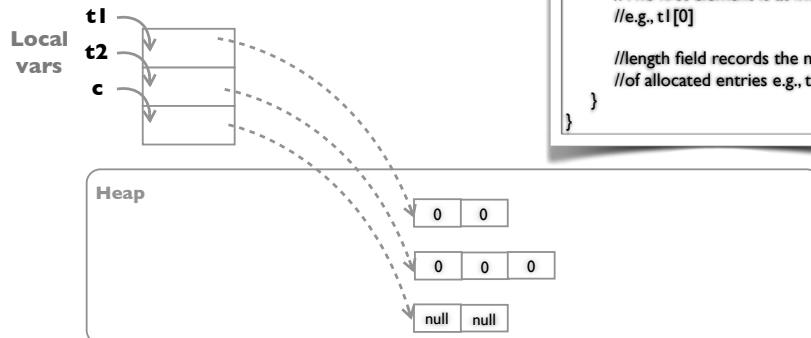
```
try {  
    //Code that may throw  
    //exceptions e.g. IO operations  
} catch(Type1 e) {  
    //Code executed when an  
    //exception of Type1 appears  
} catch(TypeN e) {  
    //We can handle multiple exceptions  
} finally {  
    //Always executed  
}
```

**Normal execution**  
1. Try block  
2. Finally block  
3. Next statement after try-catch-finally

**Exceptional execution**  
1. Try block  
2. When the exception occurs, the try block is abandoned  
3. First compatible catch  
4. Finally block  
5. Next statement after try-catch-finally

Dr. Petru Florin Mihăescu

**Dynamically created using `new`, are allocated in **heap** and are accessed via **reference variables****



Dr. Petru Florin Mihăescu

## Arrays

```
class Main {  
    public static void main(String argv[]) {  
        //EntryType[] refName;  
        int[] t1, t2;  
        String[] c;  
        //new EntryType[size]  
        t1 = new int[2];  
        t2 = new int[3];  
        c = new String[2];  
  
        //The first element is at index 0  
        //e.g., t1[0]  
  
        //length field records the number  
        //of allocated entries e.g., t1.length  
    }  
}
```

## Accessing Classes from a Package

**Usually, Java programs / libraries are structured in **packages** (containing classes)**

Outside its package, use the full name of a public class

```
java.io.BufferedReader br;
```

... or use an import clause at the beginning of the file

```
import java.io.*;  
or  
import java.io.BufferedReader;  
...  
BufferedReader br;
```

Dr. Petru Florin Mihăescu

*It is **NOT** an #include! It simply tells the compiler in what packages to search for classes when an "unknown" class name is found in the compilation unit (source file)*