

Computer Network Programming

A Review on Object-Orientation, Concurrency and Java

Dr. Petru Florin Mihancea

V20180301

1

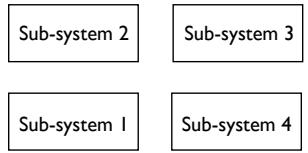
Object-Oriented Decomposition

Dr. Petru Florin Mihancea

Decomposition

Software system

Decomposition



Software system

Decomposition

Sub-system I

Decomposition

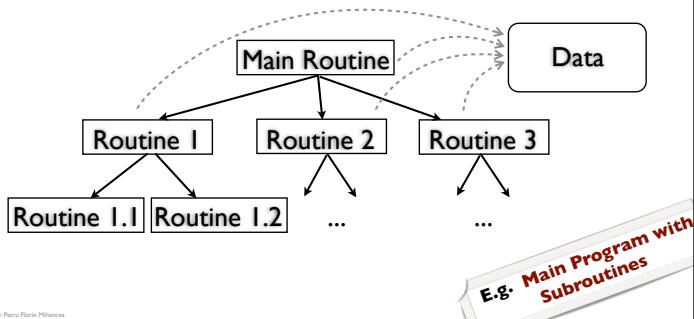


What are these
“entities” ?

Handwriting practice lines for the right side of the page.

Algorithmic Decomposition

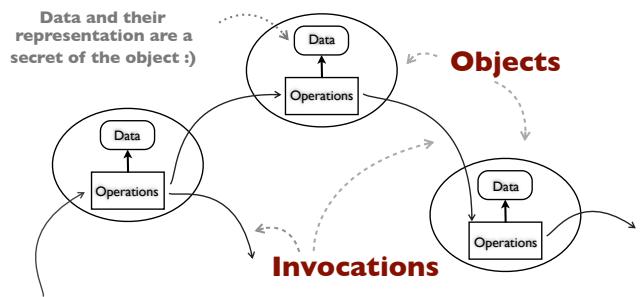
each “entity” in the system denotes a major step in some overall process



Dr. Peter Pohl-Milanes

Object-Oriented Decomposition

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



Dr. Peter Pöhl / Helmut

2

Classes and Objects

Dr. Petru Florin Mihăncescu

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

Booch - OO Analysis and Design

Dr. Peter Pohl-Milanesco

Class vs. Object



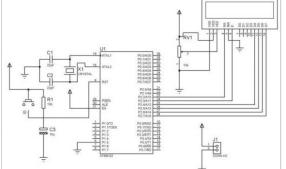
<http://www.timetools.co.uk>



Objects



Clock project



<http://hubpages.com/hub/digital-clock-using-microcontroller-89C5289S52>

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

GOE 1995

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. Peter Pohl-Milanes

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**

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;  
    }  
    //Constructors  
    //access_modifiers Class_Name(type par_name, ...) {}  
    public Clock() {  
        hour = minute = 0;  
    }  
    public Clock(int h, int m) {  
        setTime(h,m);  
    }  
}
```

Constructors

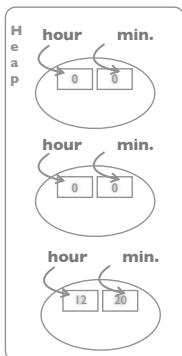
- special methods executed only once at the object creation time
- used to initialize the object state
- always has the same name as the class and does not have return type (neither void)
- if no constructor, the compiler generates one with an empty parameter list (no-arg constructor)

Dr. Peter Hirschmann

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);  
    }  
}
```

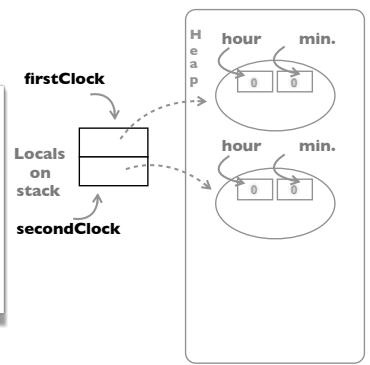


What about destruction ?
Don't worry ! The garbage collector
takes care of this

Dr. Peter Hirschmann

Referring an Object in Java

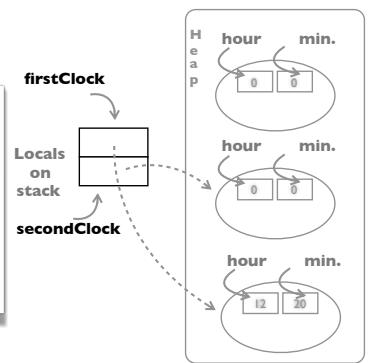
```
class Main {  
    public static void main(String args[]) {  
        ...  
        //Reference variables  
        //Here local variables but they can also  
        //be instance variables, etc.  
        Clock firstClock, secondClock;  
        firstClock = new Clock();  
        secondClock =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



Dr. Peter Pach Milanes

Referring an Object in Java

```
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();  
        firstClock = new Clock(12,20);  
        ...  
    }  
}
```

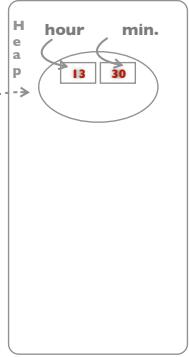


Dr. Peter Hirschmann

Invoking an Object in Java

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

firstClock
Locals
on
stack
secondClock



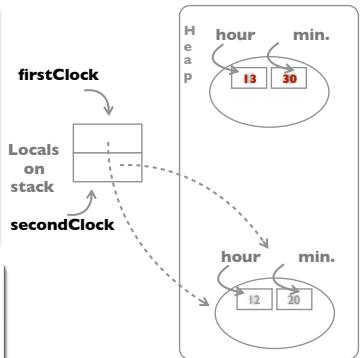
```
class Clock {  
    ...  
    public void setTime(int h, int m) {  
        hour = (h >= 0) && (h < 24) ? h : 0;  
        minute = (m >= 0) && (m < 60) ? m : 0;  
    }  
    ...  
}
```

Dr. Guru Ravi Pillai

Invoking an Object in Java

```
class Main {  
    public static void main(String argv[]) {  
        ...  
        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;  
    }  
    ...  
}
```

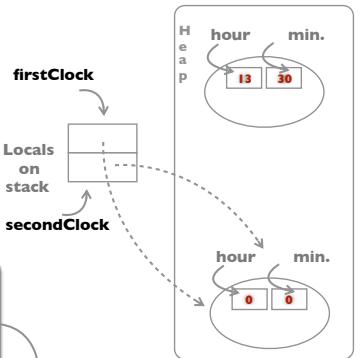


Dr. Hemant Rane, Mihimachi

Invoking an Object in Java

```
class Main {  
    public static void main(String argv[]) {  
        ...  
        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;  
    }  
    ...  
}
```



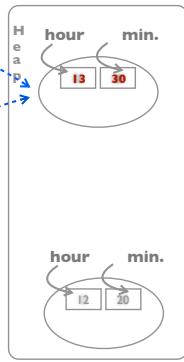
Dr. Guru Ravi Pillai

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

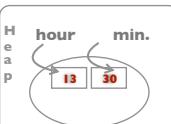
```
...  
Clock firstClock, secondClock;  
firstClock = new Clock();  
secondClock = new Clock(12,20);  
firstClock.setTime(13,30);  
secondClock.setTime(0,0);  
...
```



Dr. Peter Hirschmann

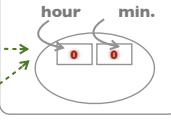
The Special **this** Reference

```
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);  
...
```

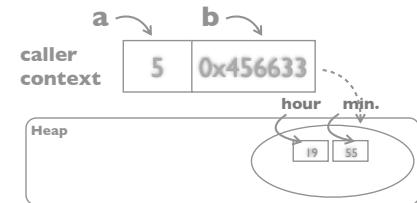


Dr. Peter Hirschmann

Parameter Passing in Java

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

passed by value

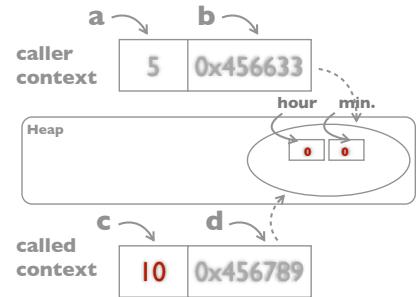


Parameter Passing in Java

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);  
}
```



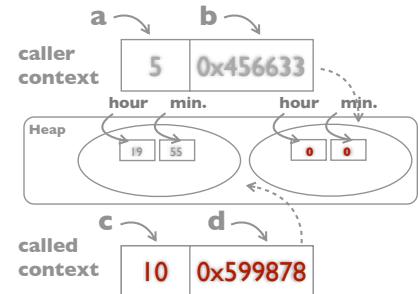
Dr. Peter Hirschmann

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);  
}
```



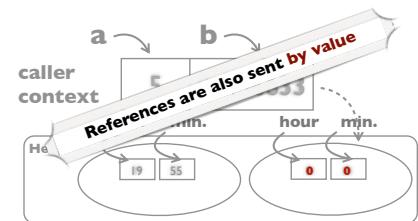
Dr. Peter Hirschmann

Parameter Passing in Java

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

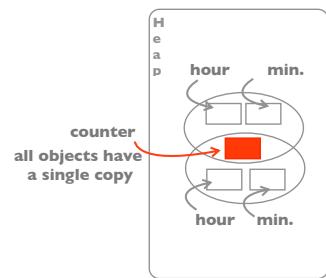
passed by value



Dr. Peter Pötsch-Millenbach

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);  
    }  
}
```

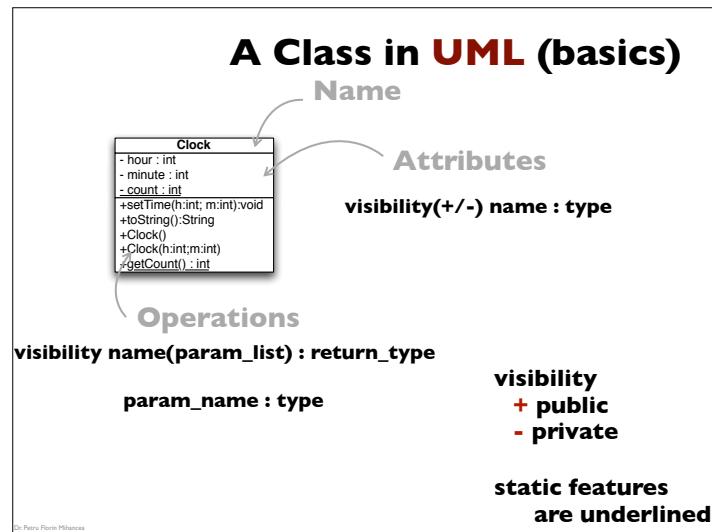


static methods do not execute on an object

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

Dr. Peter Hirschmann

A Class in UML (basics)



Dr. Peter Pöhl / Münster

3

Inheritance and Polymorphism

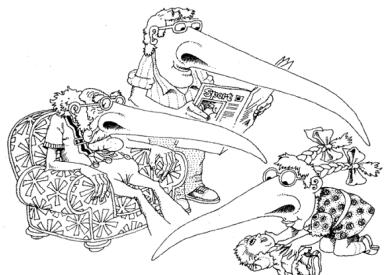
Dr. Petru Florin Mihancea

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. Peter Hirschman

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

Class inheritance in Java

Superclass

```
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);  
    }  
}
```

A simple code
factorization (reuse)
mechanism

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;  
    }  
}
```



Dr. Peter Hirschmann

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

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, 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

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 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. Peter Hirschmann

Some Important Details

Constructors

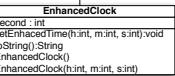
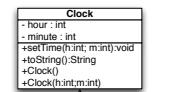
- the first instruction in a constructor must be a call to a constructor from superclass

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

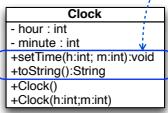


In UML

Dr. Peter Pöhl, Münster

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. Peter Payer, Münster

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 I : 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)

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

Type Inheritance

Refers to an inheritance relations between types

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

Type Inheritance in Java

Case I Using interfaces

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

```
<<interface>>  
ClockInterface  
<use h:int,m:int>>  
<use toString():String>
```

```
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);  
    }  
}
```

```
Clock  
+hour : int  
+minute : int  
+setTime(h:int,m:int):void  
+toString():String  
+Clock()  
+Clock(h:int,m:int)
```

Dr. Peter Hirschmann

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. Peter Hirschmann

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

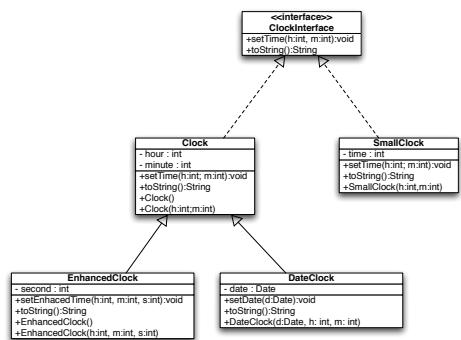
- a class can implement multiple interfaces (multiple types)

Cons

- a little more difficult to factor common implementation

Dr. Peter Pötsch-Milanesi

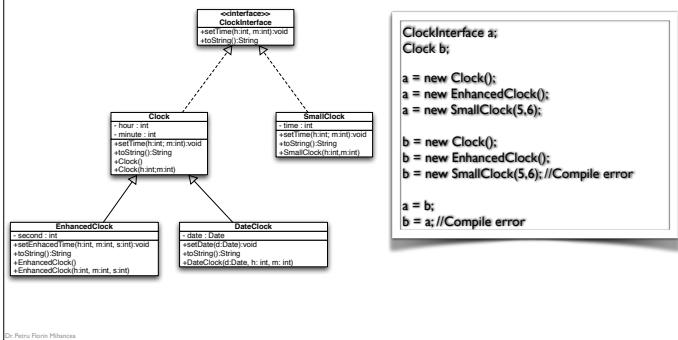
Class Hierarchy Example



Dr. Peter Pock-Hilzenrath

The effect of type inheritance

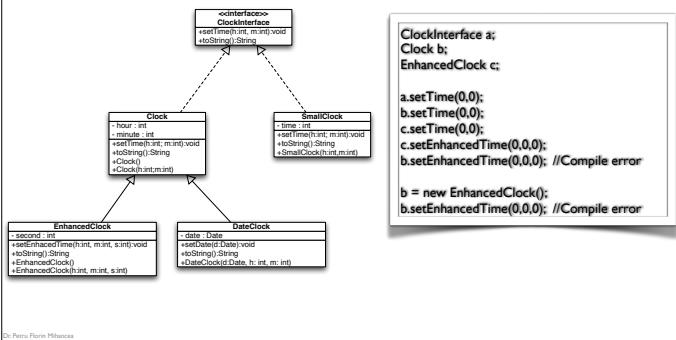
Polymorphism: A reference variable of a declared type (class) can refer objects of that type (class) and of any of its subtypes (subclasses)



Dr. Peter Hirschmann

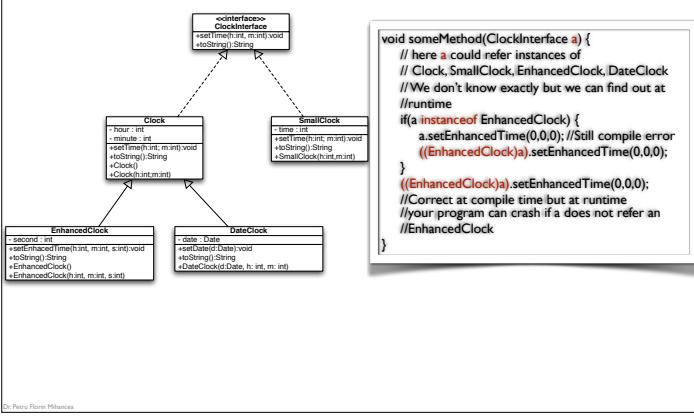
Correct Invocations

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



Dr. Timo Rönsch-Milanes

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); // Still compile error  
        ((EnhancedClock)a).setEnhancedTime(0,0);  
    }  
    ((EnhancedClock)a).setEnhancedTime(0,0);  
    // Correct at compile time but at runtime  
    // your program can crash if a does not refer an  
    // EnhancedClock  
}
```

Dr. Peter Hirschheydt

```
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();  
    ...  
}
```

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
then goto toString
...

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

© 2010 Pearson Education, Inc.

Object Class

toString():String

```
Object
...
+toString():String
+equals(o:Object):boolean
+hashCode():int
...
```

- 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

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

Dr. Peter Pötsch-Millenbach

Examples

```
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"
```

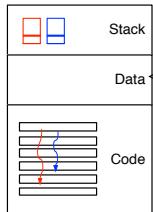
4

Threads and Synchronization Mechanisms in Java

Dr. Punit Rathi, Mihiana

Execution Thread

A single sequential flow of control
within a process



... the data
section of the
process is shared
between its
threads,
THUS ...

```
//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 :) ...

Dr. Peter Pöhl-Milanes

A BIG Concern

```
class Square {  
    private int l, L;  
    public void set(int a) {  
        l = a; l = 5  
        l = 10 (a) L = 5 (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(5)).start();  
    }  
}
```

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

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. Peter Hirschmann

Synchronized Blocks

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

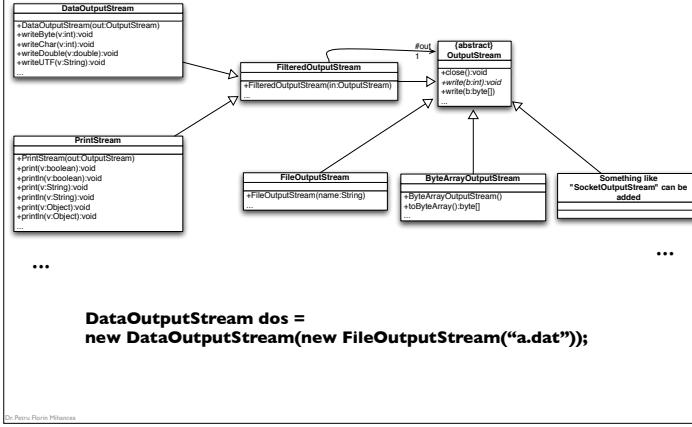
Similarly, it is
not possible
to interleave the
execution of such
blocks (or sync methods)
that are
synchronized using
the same obj
instance

5

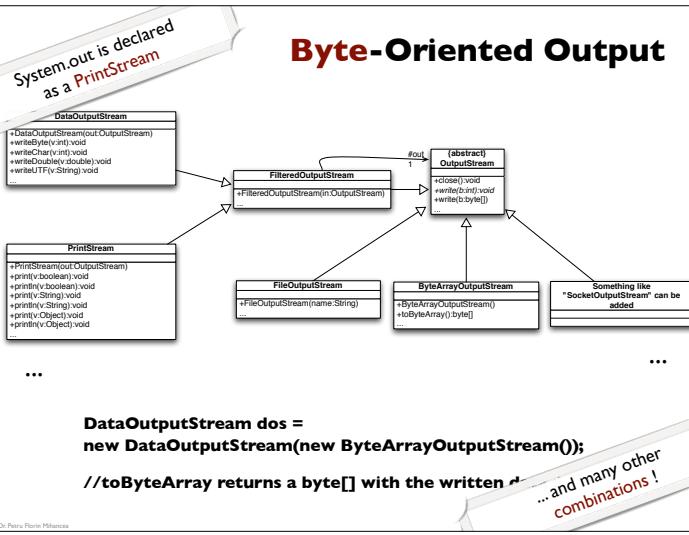
A Simplified View on java.io System

Dr. Petru Florin Mihăncescu

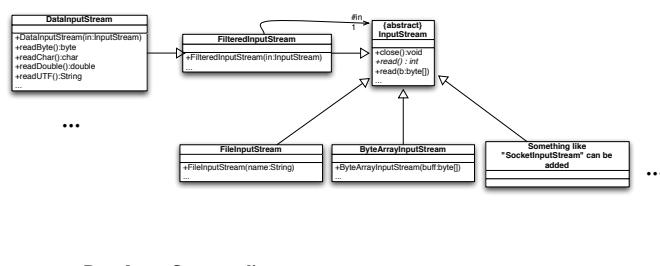
Byte-Oriented Output



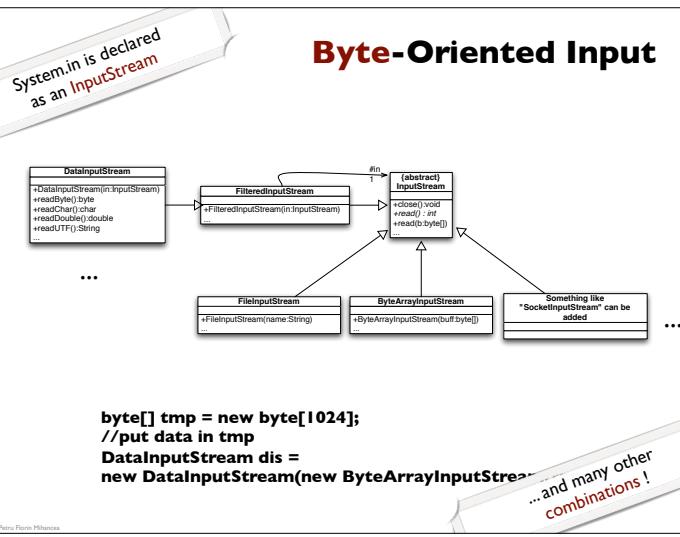
Dr. Bernhard Hilfinger



Byte-Oriented Input

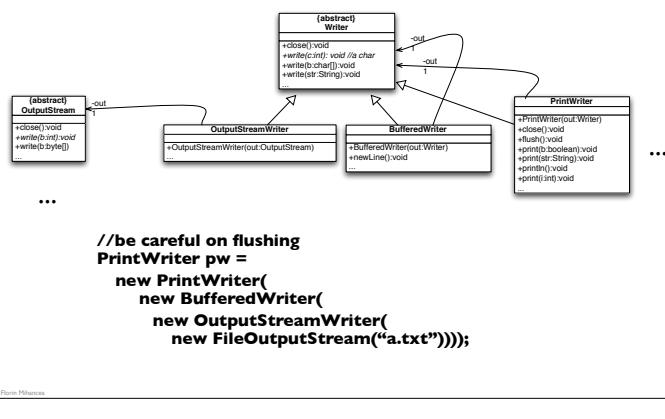


```
DataInputStream dis =  
new DataInputStream(new FileInputStream("a.dat"));
```



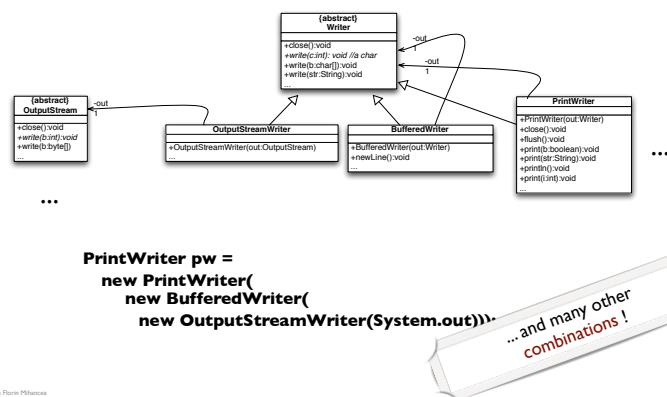
Dr. Peter Hirschmann

Char-Oriented Output



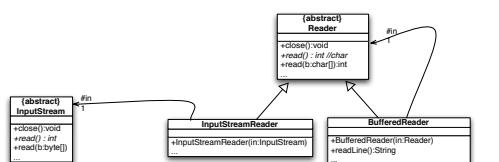
Dr. Peter Hirschmann

Char-Oriented Output



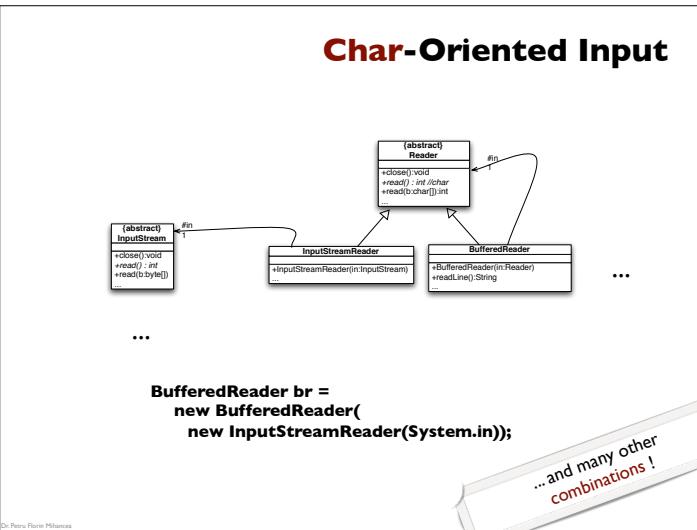
Dr. Peter Hinsch-Hilken

Char-Oriented Input



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

Char-Oriented Input

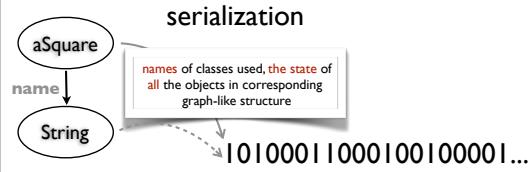


Dr. Peter Hinsch-Hilkenius

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

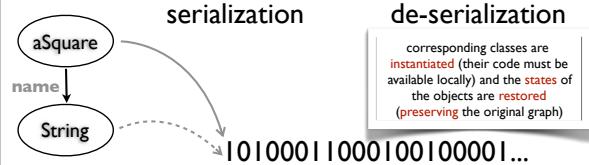


Dr. Perla Flores-Milanes

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

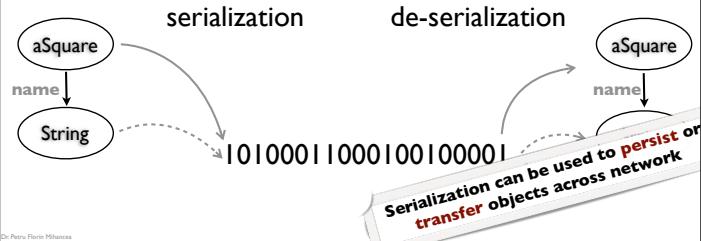


Dr. Peter Hirschmann

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. Perin, Flora Milanesi

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();
```

6

Miscellaneous

Dr. Petru Florin Mihancea

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  
}
```

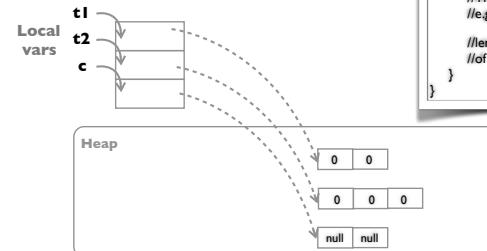
Normal execution

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

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



Dr. Peter Pachl, Münster

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;
```

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)

Dr. Peter Hirschmann