

# **Advanced Programming Methods**

## **Lecture 1 – Java Basics**

# Course Overview

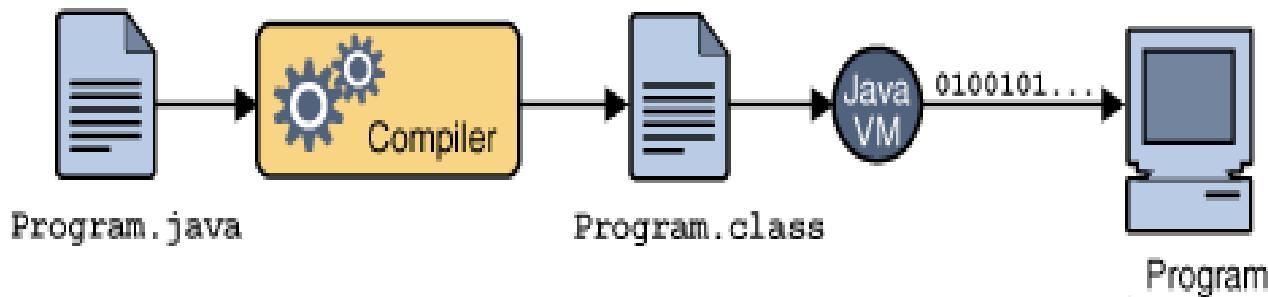
Object-oriented languages:

- Java (the first 9-10 lectures): Basics, Generics, Collections, IO, Functional Programming, Concurrency, GUI, Metaprogramming
- C# (about 3 lectures): Basics, Collections, IO, Linq
- Scala and Rust (1-2 lectures)

# Java References

- Bruce Eckel, *Thinking in Java*
- The Java Tutorials
- <https://docs.oracle.com/javase/tutorial/>
- Java 19, 2022
- <https://docs.oracle.com/en/java/javase/19/>
- Any Java book, tutorial,...

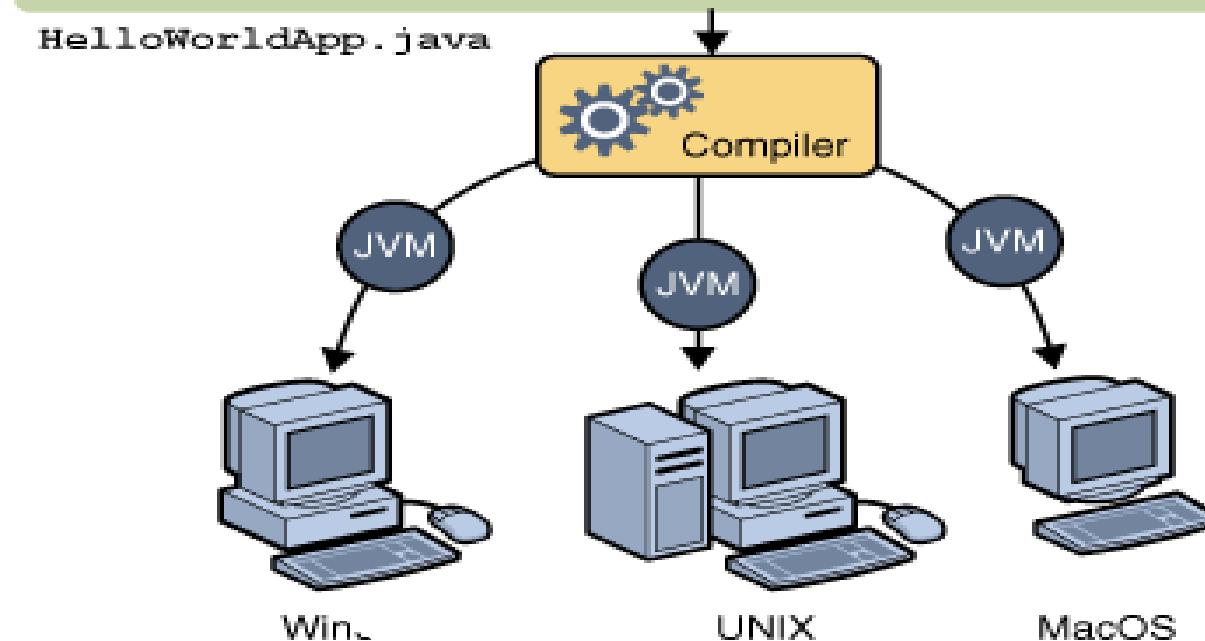
# Java Technology



Java Program

```
class HelloWorldApp {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

HelloWorldApp.java



# A simple Java program

```
//Test.java  
public class Test {  
    public static void main(String[] args) {  
        System.out.println("Hello");  
        for(String el : args)  
            System.out.println(el);  
    }  
}
```

## Compilation:

```
javac Test.java
```

## Execution:

```
java Test
```

```
java Test ana 1 2 3
```

```
!!! You can use int value=Integer.parseInt(args[i]) in order to  
transform a string value into an int value.
```

# Lecture Overview

1. Recap OO concepts
2. Java classes and objects
3. Java Primitive Data
4. Passing Arguments in Java
5. Java Arrays
6. Java Char and String
7. Java code reusing: composition and inheritance
8. Java polymorphism: abstract classes, interfaces

# Object-oriented programming Concepts

*Class:* represents a new data type

- Corresponds to an implementation of an ADT.

*Object:* is an instance of a class.

- The objects interact by messages.

*Message:* used by objects to communicate.

- A message is a method call.

*Encapsulation(hiding)*

- data (state)
- Operations (behaviour)

*Inheritance:* code reusing

*Polymorphism* – the ability of an entity to react differently depending on the context

# Java Classes and Objects

## ■ Class Declaration/Definition:

```
//ClassName.java  
[public] [final] class ClassName{  
[data (fields) declaration]  
[methods declaration and implementation]  
}
```

1. A class defined using `public` modifier is saved into a file with the class name `ClassName.java`.

2. A file `.java` may contain multiple class definitions, but only one can be public.

3. Java vs. C++:

■ No 2 different files (.h, .cpp).

■ Methods are implemented when declared.

■ A class declaration does not end with ;

# Java Classes and Objects

## ■ Examples:

```
//Persoana.java
public class Persoana{
//...
}
```

```
// Complex.java
class Rational{
//...
}
```

```
class Natural{
//...
}
```

```
public class Complex{
//...
}
```

# Java Classes and Objects

## ■ Class Members (Fields) declaration:

```
... class ClassName{  
    [access_modifier] [static] [final] Type name [=init_val];  
}
```

access\_modifier can be **public**, **protected**, **private**.

1. Class members can be declared anywhere inside a class.
2. Access modifier must be given for each field.
3. If the access modifier is missing, the field is visible inside the package (directory).

# Java Classes and Objects

## ■ Examples:

```
//Persoana.java
public class Persoana{
    private String nume;
    private int varsta;
//...
}
```

```
//Punct.java
public class Punct{
private double x;
private double y;
//...
}
```

# Java Classes and Objects

## ■ Initializing fields

- - at declaration-site:

```
private double x=0;
```

- in a special initialization block:

```
public class Rational{  
    private int numerator;  
  
    private int numitor;  
  
    {  
  
        numerator=0;  
  
        numitor=1;  
  
    }  
  
    //...  
}
```

- in constructor.

Any field that is not explicitly initialized will take the default value of its type.

# Constructors

- The constructor body is executed after the object memory space is allocated in order to initialize that space.

```
[...] class ClassName{  
    [access_modifier] ClassName([list_formal_parameters]) {  
        //body  
    }  
}  
  
access_modifier ∈ {public, protected, private}  
  
list_formal_parameters takes the following form:  
  
Type1 name1[, Type2 name2[,...]]
```

- 1.The constructor has the same name as the class name (case sensitive).
- 2.The constructor does not have a return type.
- 3.For a class without any declared constructor, the compiler generates an implicit public constructor (without parameters).

# Overloading Constructors

- A class can have many constructors, but they must have different signatures. . .

```
//Complex.java
public class Complex{
    private double real, imag;

    public Complex(){           //implicit constructor
        real=0;
        imag=0;
    }

    public Complex(double real){
        this.real=real;
        imag=0;
    }

    public Complex(double real, double imag){ //...
    }

    public Complex(Complex c){ //...
    }
}
```

# this

- It refers to the current (receiver) object.
- It is a reserved keyword used to refer the fields and the methods of a class.

```
//Complex.java
public class Complex{
    private double real, imag;
    //...
    public Complex(double real) {
        this.real=real;
        imag=0;
    }

    public Complex(double real, double imag) {
        this.real=real;
        this.imag=imag;
    }

    public Complex suma(Complex c) {
        //...
        return this;
    }
}
```

# Calling another constructor

- `this` can be used to call another constructor from a given constructor.

```
//Complex.java
public class Complex{
    private double real, imag;

    public Complex() {
        this(0,0);
    }

    public Complex(double real) {
        this(real,0);
    }

    public Complex(double real, double imag) {
        this.real=real;
        this.imag=imag;
    }
    //...
}
```

# Creating objects

## ■ Operator `new`:

```
Punct p=new Punct();      //the parentheses are compulsory  
Complex c1=new Complex();  
Complex c2=new Complex(2.3);  
Complex c3=new Complex(1,1.5);  
  
Complex cc; //cc=null, cc does not refer any object  
cc=c3;      //c3 si cc refer to the same object in the memory
```

- 1.The objects are created into the heap memory.
- 2.The operator new allocates the memory for an object;

# Static fields

```
public class Natural{  
    private long val;  
    public static long MAX=232.... //2^63-1  
//....  
}  
  
public class Produs {  
    private static long counter;  
    private final long id=counter++;  
    public String toString(){  
        return ""+id;  
    }  
//....
```

Static fields are shared by all class instances. They are allocated only once in the memory.

# Defining methods

```
[...] class ClassName{  
    [access_modifier] Result_Type methodName([list_formal_param]) {  
        //method body  
    }  
}  
  
access_modifier ∈ {public, protected, private}  
  
list_formal_param takes the form Type1 name1[, Type2 name2[, ...]]  
Result_Type poate can be any primitive type, reference type, array, or void.
```

1. If the access\_modifier is missing, that method can be called by any class defined in that package (director).
2. If the return type is not `void`, then each execution branch of that method must end with the statement `return`.

# Defining methods

```
//Persoana.java
public class Persoana{
    private byte varsta;
    private String nume;
    public Persoana() {
        this("",0);
    }
    public Persoana(String nume, byte varsta) {
        this.nume=nume;
        this.varsta=varsta;
    }
    public byte getVarsta() {
        return varsta;
    }
    public void setNume(String nume) {
        this.nume=nume;
    }
    public boolean maiTanara(Persoana p) {//...
    }
}
```

# Overloading methods

- A class may contain multiple methods with the same name but with different signature. A signature = return type and the list of the formal parameters

```
public class Complex{  
    private double real, imag;  
    // constructors ...  
    public void aduna (double real){  
        this.real+=real;  
    }  
    public void aduna(Complex c) {  
        this.real+=c.real;  
        this.imag+=c.imag;  
    }  
    public Complex aduna(Complex cc) {  
        this.real+=cc.real;  
        this.imag+=cc.imag;  
        return this;  
    }  
}  
//Errors?
```

- Java does not allow the operators overloading.
- Class String has overloaded operators `+` and `+=`.

```
String s="abc";
String t="EFG";
String r=s+t;
s+=23;
s+=' ';
s+=4.5;
//s="abc23 4.5";
//r="abcEFG"
```

- Destructor: In Java there is no any destructor.
- The garbage collector deallocates the memory .

# Static methods

- Are declared using the keyword **static**
- They are shared by all class instances

```
public class Complex{  
    private double real, imag;  
    public Complex(double re, double im){  
        //...  
    }  
    public static Complex suma(Complex a, Complex b){  
        return new Complex(a.real+b.real, a.imag+b.imag);  
    }  
    //...  
}
```

# Static methods

- They are called using the class name:

```
Complex a,b;  
//... initialization a and b  
Complex c=Complex.aduna(a, b);
```

1. A static method cannot use those fields (or call those methods) which are not static. It can use or call only the static members.

# Java Primitive Data Types

## Variables Declaration:

```
type name_var1 [=expr1] [, name_var2 [=expr2]...];
```

## Primitive Data Types

Type	Nr. byte	Values	Default value
boolean	-	true, false	false
byte	1	-128 ... +127	(byte)0
short	2	$-2^{15} \dots 2^{15}-1$	(short)0
int	4	$-2^{31} \dots 2^{31}-1$	0
long	8	$-2^{63} \dots 2^{63}-1$	0L
float	4	IEEE754	0.0f
double	8	IEEE754	0.0d
char	2	Unicode 0, Unicode $2^{16}-1$	'\u0000' (null)

# Passing arguments

- Primitive type arguments ( boolean, int, byte, long, double) are **passed by value**. Their values are copied on the stack.
- Arguments of reference type are **passed by value**. A reference to them is copied on the stack, but their content (fields for objects, locations for array) can be modified if the method has the rights to accces them.

1. There is not any way to change the passing mode( like & in C++).

```
class Parametrii{  
    static void interschimba(int x, int y){  
        int tmp=x;  
        x=y;  
        y=tmp;  
    }  
    public static void main(String[] args) {  
        int x=2, y=4;  
        interschimba(x,y);  
        System.out.println("x="+x+" y="+y); //?  
    }  
}
```

# Passing arguments

```
class B{
    int val;
    public B(int x) {
        this.val=x;
    }
    public String toString() {
        return ""+val;
    }
    static void interschimba(B x, B y) {
        B tmp=x;
        x=y;
        y=tmp;
        System.out.println("[Interschimba B] x="+x+" y="+y);
    }
    public static void main(String[] args) {
        B bx=new B(2);
        B by=new B(4);
        System.out.println("bx="+bx+" by="+by);
        interschimba(bx,by);
        System.out.println("bx="+bx+" by="+by); //?
    }
}
```

# Passing arguments

```
class B{
    int val;
    public B(int x) {
        this.val=x;
    }
    public String toString() {
        return ""+val;
    }
    static void interschimbaData(B x, B y) {
        int tmp=x.val;
        x.val=y.val;
        y.val=tmp;
        System.out.println("[InterschimbaData] x="+x+" y="+y);
    }
    public static void main(String[] args) {
        B bx=new B(2);
        B by=new B(4);
        System.out.println("bx="+bx+" by="+by);
        interschimbaData(bx,by);
        System.out.println("bx="+bx+" by="+by); //?
    }
}
```

# Java Arrays

# One dimension Array

## Array declaration

```
type[] name;  
type name[];
```

## Array allocation:

```
array_name=new type[dim]; //memory allocation  
//index 0 ... dim-1
```

Accessing an array element: `array_name[index]`

## Examples:

```
float[] vec;  
  
vec=new float[10];  
  
int[] sir=new int[3];  
  
float tmp[];  
  
tmp=vec;      //vec and tmp refer to the same array
```

# One dimension Array

*Built-in length:* returns the array dimension.

```
int[] sir=new int[5];  
int lung_sir=sir.length; //lung=5;  
sir[0]=sir.length;  
sir.length=2; //error  
  
int[] y;  
int lung_y=y.length; //error, y was not created  
  
double[] t=new double[0];  
int lung_t=t.length; //lung_t=0;  
t[0]=2.3 //error: index out of bounds
```

the shortcut syntax to create and initialize an array:

```
int[] dx={-1,0, 1};
```

# Rectangular multidimensional array

Declaration:

```
type name[][][]...[];  
type [][][]...[] name;
```

Creation:

```
name=new type[dim1][dim2]...[dimn];
```

Accessing an element:

```
name[index1][index2]...[indexn];
```

Examples:

```
int[][] a;  
a=new int[5][5];  
a[0][0]=2;  
int x=a[2][2];
```

# Non-Rectangular Multidimensional Array

Examples:

```
int[][] a=new int[3][];
for(int i=0;i<3;i++)
    a[i]=new int[i+1];
int x=a.length;          //x=?
int y=a[2].length;      //y=?
```

Declaration+creation+initialization:

```
char[][] lit={{'a'}, {'b'}};
int[][] b={{1,2},
           {2,5,8},
           {1}};
double[][] mat=new double[][]{{1.3, 0.5}, {2.3, 4.5}};
```

# Array of objects

- Each array element must be allocated and initialized.

```
public class TablouriObiecte {  
    static void genereaza(int nrElem, Complex[] t) {  
        t=new Complex[nrElem];  
        for(int i=0;i<nrElem;i++)  
            t[i]=new Complex(i,i);  
    }  
    static Complex[] genereaza(int nrElem) {  
        Complex[] t=new Complex[nrElem];  
        for(int i=0;i<nrElem;i++)  
            t[i]=new Complex(i,i);  
        return t;  
    }  
    static void modifica(Complex[] t) {  
        for(int i=0;i<t.length;i++)  
            t[i].suma(t[i]);  
    }  
//...
```

# Array of objects

```
static Complex suma(Complex[] t) {  
    Complex suma=new Complex(0,0);  
    for(int i=0; i<t.length;i++)  
        suma.aduna(t[i]);  
    return suma;  
}  
  
public static void main(String[] args) {  
    Complex[] t=genereaza(3);  
    Complex cs=suma(t);  
    System.out.println("suma "+cs);  
    Complex[] t1=null;  
    genereaza(3,t1);  
    Complex cs1=suma(t1);  
    System.out.println("suma "+cs1);  
    modifica(t);  
    System.out.println("suma dupa modificare "+suma(t));  
}  
}
```

# Java Char and String

```
char[] sir={'a','n','a'}; //comparison and printing  
//is done character by character  
sir[0]='A';
```

A constant Sequence of Chars :

```
"Ana are mere"; //object of type String
```

String class is immutable:

```
String s="abc";  
s=s+"123"; //concatenating strings  
String t=s; //t="abc123"  
t+="12"; //t=? , s=?
```

String content can not be changed:      **t[0]='A' ;**

```
char c=t.charAt(0);  
  
method length(): int lun=s.length();  
t.equals(s)  
/*Returns true if and only if the argument is a String object that  
represents the same sequence of characters as this object. */  
compareTo(): int rez=t.compareTo(s)  
/*Compares two strings lexicographically. Returns an integer indicating  
whether this string is greater than (result is > 0), equal to (result is =  
0), or less than (result is < 0) the argument.*/
```

# Java code reusing: composition and inheritance

# Code reusing

- *Composing*: The new class consists of instance objects of the existing classes
- *Inheritance*: A new class is created by extending an existing class (new fields and methods are added to the fields and methods of the existing class)

# Composing

The new class contains fields which are instance objects of the existing classes.

```
class Adresa{  
    private String nr, strada, localitate, tara;  
    private long codPostal;  
//...  
}
```

```
class Persoana{  
    private String nume;  
    private Adresa adresa;  
    private String cnp;  
//...  
}
```

```
class Scrisoare{  
    private String destinatar;  
    private Adresa adresaDestinatar;  
    private String expeditor;  
    private Adresa adresaExpeditor  
//...  
}
```

# Inheritance

- Using the keyword `extends`:

```
class NewClass extends ExistingClass{  
    //...  
}
```

- `NewClass` is called subclass, or child class or derived class.
- `ExistingClass` is called superclass, or parent class, or base class.
- Using inheritance, `NewClass` will have all the members of the class `ExistingClass`. However, `NewClass` may either redefine some of the methods of the class `ExistingClass` or add new members and methods.

# Inheritance

```
public class Punct{  
    private int x,y;  
    public Punct(int x, int y){  
        this.x=x;  
        this.y=y;  
    }  
    public void muta(int dx, int dy){  
        //...  
    }  
    public void deseneaza(){  
        //...  
    }  
}  
public class PunctColorat extends Punct{  
    private String culoare;  
    public PunctColorat(int x, int y, String culoare){...}  
    public void deseneaza(){...}  
    public String culoare(){...}  
}
```

# Inheritance

## ■ Notions

- **deseneaza** is an overridden method.
- **muta** is an inherited method.
- **culoare** is a new added method.

## ■ Heap memory:

```
Punct punct =new Punct(2,3);  
PunctColorat punctColorat=new PunctColorat(2,3,"alb");
```

# Method overriding

- A derived class may override methods of the base class

- Rules:

1. The class **B** overrides the method **mR** of the class **A** if **mR** is defined in the class **B** with the same signature as in the class **A**.
2. For a call **a.mR()**, where **a** is an object of type **A**, it is selected the method **mR** which correspond to the object referred by **a**.

```
A a=new A();  
a.mR();      //method mR from A  
a=new B();  
a.mR();      //method mR from B
```

3. The methods which are not overridden are called based on the receiver type.

# Method overriding

4. annotation `@Override` in order to force a compile-time verification

```
public class A{  
    public void mR(){  
        //...  
    }  
}  
  
public class B extends A{  
    @Override  
    public void mR(){  
    }  
}
```

4. The return type of an overriding method may be a subtype of the return type of the overridden method from the base class (*covariant return type*).

```
public class C{  
    public SuperA m(){...}  
}  
  
public class D extends C{  
    public SubB m(){...}  
}  
  
public SubB extends SuperA {...}
```

# Method overloading

- A subclass may overload a method from the base class.
- An instance object of a subclass may call all the overloaded methods including those from the superclass.

```
public class A{  
    public void f(int h){ //...  
    }  
    public void f(int i, char c){ //...  
    }  
}
```

```
public class B extends A{  
    public void f(String s, int i){  
        //...  
    }  
}
```

```
B b=new B();  
b.f(23);  
b.f(2, 'c');  
b.f("mere",5);
```

# Calling the superclass constructors

- A constructor of a subclass can call a constructor of the base class.
- It is used the keyword **super**.
- The call of the base class must be the first instruction of the subclass constructor.

```
public class Persoana{  
    private String nume;  
    private int varsta;  
    public Persoana(String nume, int varsta) {  
        this.nume=nume;  
        this.varsta=varsta;  
    }  
    //...  
}  
  
public class Angajat extends Persoana{  
    private String departament;  
    public Angajat(String nume, int varsta, String departament) {  
        super(nume, varsta);  
        this.departament=departament;  
    }  
    //...  
}
```

# The keyword super

- It is used in the followings:

- To call a constructor of the base class.
- To refer to a member of the base class which has been redefined in the subclass.

```
public class A{  
    protected int ac=3;  
    //...  
}
```

```
public class B extends A{  
    protected int ac=3;  
    public void f(){  
        ac+=2;  super.ac--;  
    }  
}
```

- To call the overridden method (from the base class) from the overriding method (from the subclass).

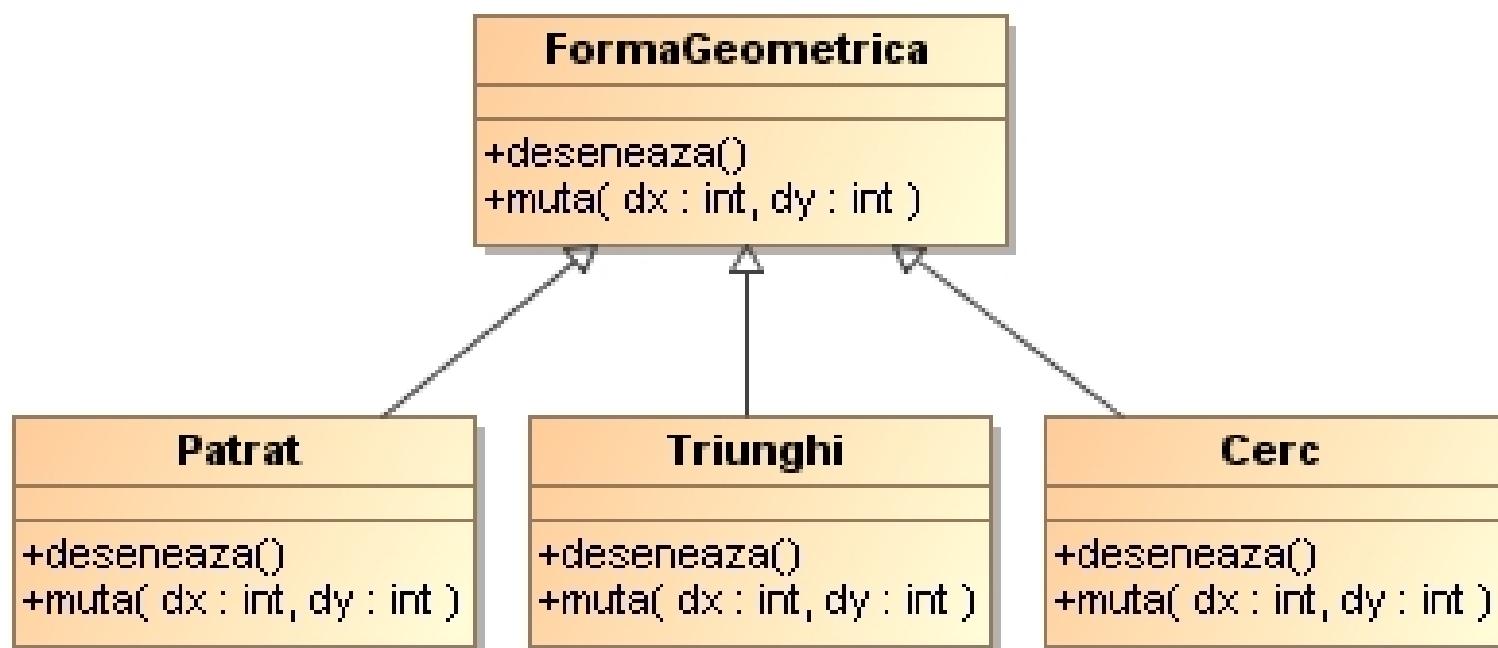
```
public class Punct{  
    //...  
    public void deseneaza(){  
    //...  
    }  
}
```

```
public class PunctColorat extends Punct{  
    private String culoare;  
    public void deseneaza(){  
        System.out.println(culoare);  
        super.deseneaza();  
    }  
}
```

# Java polymorphism: abstract classes, interfaces

# Polymorphism

- The ability of an object to have different behaviors according to the context.
- 3 types of polymorphism:
  - ad-hoc: method overloading.
  - Parametric: generics types.
  - inclusion: inheritance.



# Polymorphism

- *early binding*: the method to be executed is decided at compile time
- *late binding*: the method to be executed is decided at execution time
- Java uses late binding to call the methods. However there is an exception for static methods.

```
void deseneaza(FormaGeometrica fg) {  
    fg.deseneaza();  
}  
  
//...  
FormaGeometrica fg=new Patrat();  
deseneaza(fg); //call deseneaza from Patrat  
fg=new Cerc();  
deseneaza(fg); //call deseneaza from Cerc
```

# Polymorphic collections

```
public FiguraGeometrica[] genereaza(int dim) {  
    FiguraGeometrica[] fg=new FiguraGeometrica[dim];  
    Random rand = new Random(47);  
    for(int i=0;i<dim;i++){  
        switch(rand.nextInt(3)) {  
            case 0: fg[i]= new Cerc(); break;  
            case 1: fg[i]= new Patrat(); break;  
            case 2: fg[i]= new Triunghi(); break;  
            default:  
        }  
    }  
    return fg;  
}  
  
public void muta(FiguraGeometrica[] fg){  
    for(FiguraGeometrica f: fg)  
        f.muta(3,3);  
}
```

# Abstract classes

- An abstract method is declared but not defined. It is declared with the keyword **abstract**.

```
[modificator _ acces] abstract ReturnType nome([list _ param _ formal]);
```

- An abstract class may contain abstract methods.

```
[public] abstract class ClassName {  
    [fields]  
    [abstract methods declaration]  
    [methods declaration and implementation]  
}
```

```
public abstract class Polinom{  
    //...  
    public abstract void aduna(Polinom p);  
}
```

# Abstract classes

1. An abstract class cannot be instantiated.

`Polinom p=new Polinom();`

2. If a class contains at least one abstract method then that class must be abstract.

3. A class can be declared abstract without having any abstract method.

4. If a class extends an abstract class and does not define all the abstract methods then that class must also be declared abstract.

```
public abstract class A{  
    public A(){}  
    public abstract void f();  
    public abstract void g(int i);  
}
```

```
public abstract class B extends A{  
    private int i=0;  
    public void g(int i){  
        this.i+=i;  
    }  
}
```

# Java interfaces

- Are declared using keyword `interface`.

```
public interface InterfaceName{  
    [methods declaration];  
}
```

1. Only method declaration, no method implementation
2. No constructors
3. All declared methods are implicitly public.
4. It may not contain any method declaration.
5. It may contain fields which by default are `public`, `static` and constant (`final`).

```
public interface LuniAn{  
    int IANUARIE=1, FEBRUARIE=2, MARTIE=3, APRILIE=4, MAI=5,  
    IUNIE=6, IULIE=7, AUGUST=8, SEPTEMBRIE=9, OCTOMBRIE=10, NOIEMBRIE=11,  
    DECEMBRIE=12;  
}
```

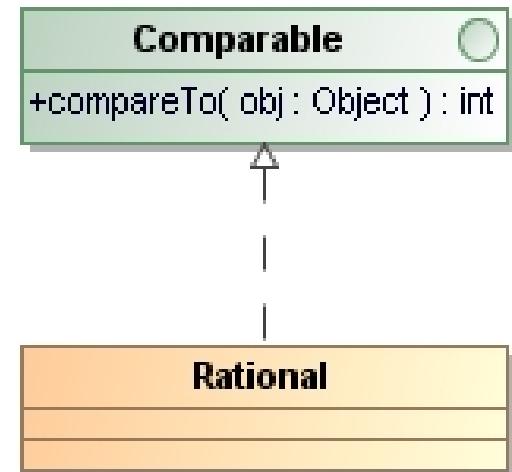
# Interface implementation

- A class can implement an interface, using `implements`.

```
[public] class ClassName implements InterfaceName{  
    [interface method declarations]  
    //other definitions  
}
```

1. The class must implement all the interface methods

```
public interface Comparable{  
    int compareTo(Object o);  
}  
  
public class Rational implements Comparable{  
    private int numerator, numitor;  
    //...  
    public int compareTo(Object o){  
        //...  
    }  
}
```



# Extending an interface

- An interface can inherit one or more interfaces

```
[public] interface InterfaceName extends Interface1[, Interface2[, ...]]  
{  
    [declaration of new methods]  
  
}
```

## 1. Multiple inheritance.

```
public interface A{  
    int f();  
}  
public interface B{  
    double h(int i);  
}  
public interface C extends A, B{  
    boolean g(String s);  
}
```

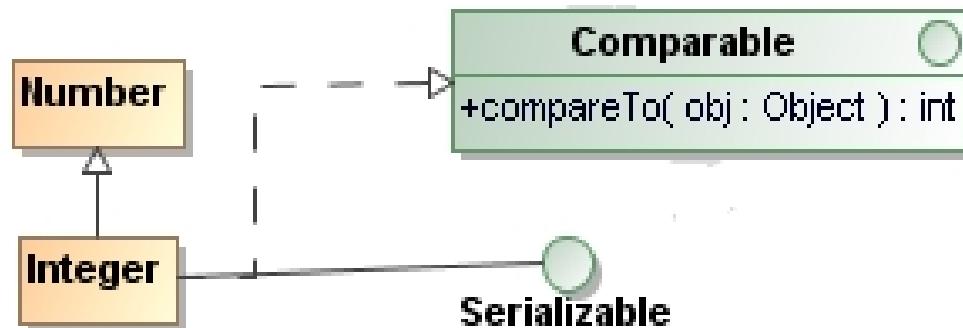
# Inheritance and interfaces

- A class can inherit one class but can implement multiple interfaces

```
[public] class NumeClasa extends SuperClasa implements Interfata1,  
    Interfata2, ..., Interfatan{  
    //...  
}
```

Example:

```
public class Integer extends Number implements Serializable, Comparable{  
    //...  
    public int compareTo(Object o) {  
        //...  
    }  
}
```



# Variables of type interface

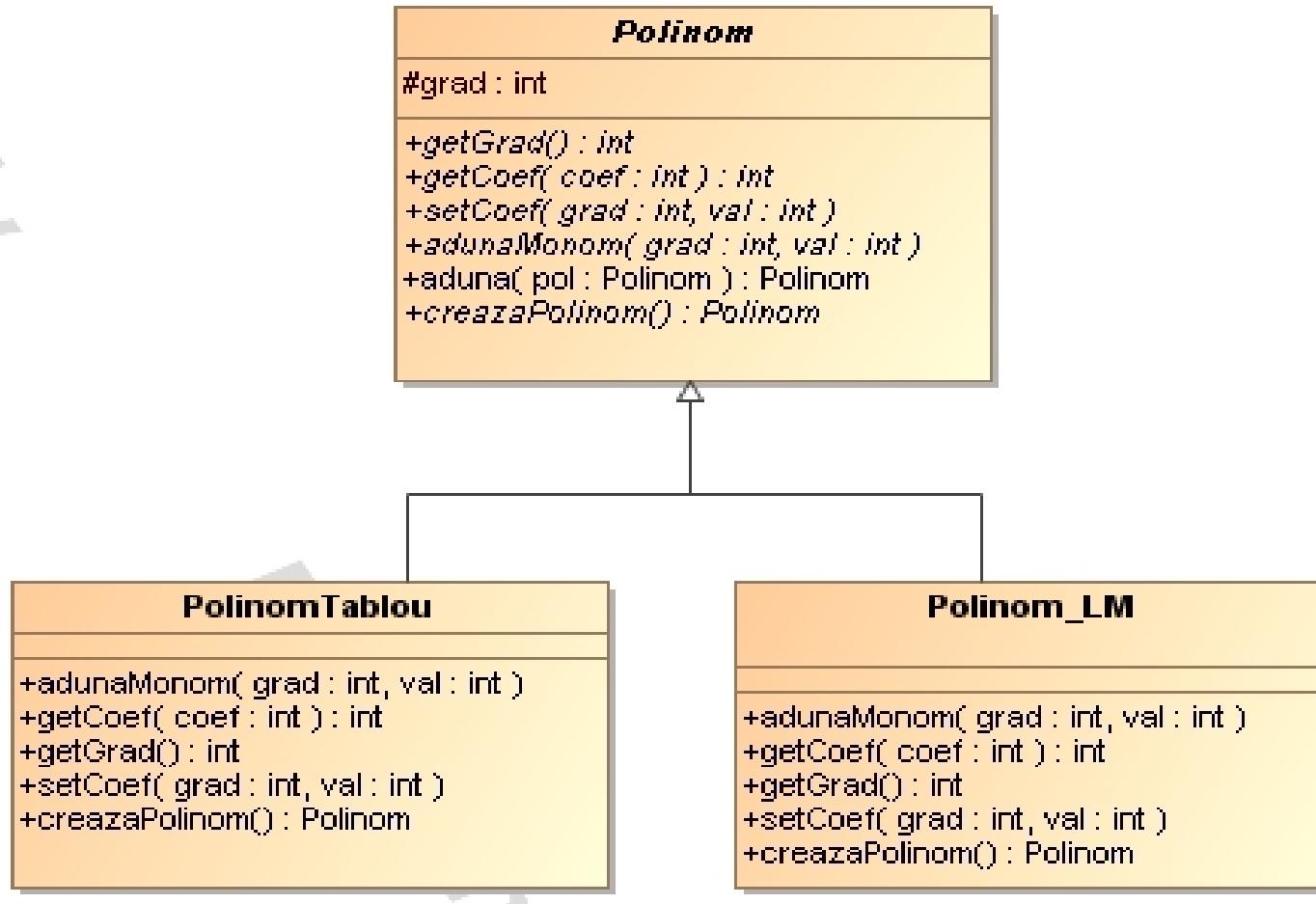
- An interface is a reference type
- It is possible to declare variables of type interface. These variables can be initialized with objects instances of classes which implement that interface. Through those variables only interface methods can be called

```
public interface Comparable{  
    //...  
}  
  
public class Rational implements Comparable{  
    //...  
}  
  
Rational r=new Rational();  
Comparable c=r;  
Comparable cr=new Rational(2,3);  
cr.compareTo(c);  
c.aduna(cr); //ERROR!!
```

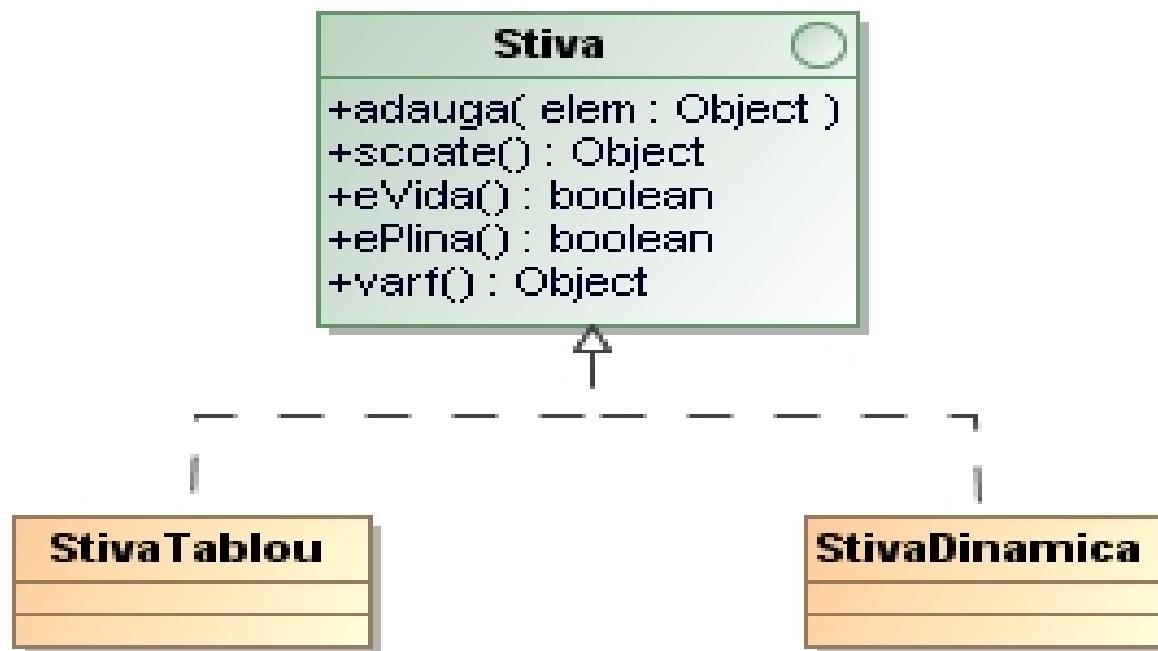
# Abstract Class vs Interface

Public, protected, private methods	only public methods.
Have fields	Can have only static and final fields
Have constructors	No constructors.
It is possible to have no any abstract method.	It is possible to have no any methods
Both do not have instance objects	

# Abstract classes vs Interfaces



# Abstract Classes vs Interfaces



# **Advanced Programming Methods**

## **Lecture 2 – Java Basics, Exceptions, Packages**

# Lecture Overview

1. Interfaces – completions
2. Downcasting
3. The operator instanceof
4. The class Object
5. Packages
6. Access Modifiers
7. Exceptions

# Interfaces

- Prior to java 8, interfaces are designed to define a contract. They had only abstract methods and final constants.
- With Java 8, interfaces can have default and static methods (definitions).
- Default methods can be overridden. Whereas, static methods can't be overridden.
- All the classes implementing interface should provide definition of all the methods in the interface.

# Interfaces

- if the designers had to change the interface, then all the implementing classes are affected. They all have to provide a definition for the new method.
- Java 8's interface default and static methods provide a way to overcome this problem.

(an example from

<https://examples.javacodegeeks.com/interface-vs-abstract-class-java-example/>)

```
public interface MobileInterface {  
  
    /**  
     * Java8 adds capability to have static method in interface.  
     * Static method in interface Can't be overridden  
     */  
    public static void printWelcomeMessage() {  
        System.out.println("****STATIC METHOD*** Welcome!!");  
    }  
    /*  
     * Java8 adds capability of providing a default definition of method in an interface.  
     * Default method can be overridden by the implementing class  
     */  
    default void makeACall(Long number) {  
        System.out.println(String.format("****DEFAULT METHOD*** Calling ..... %d", number));  
    }  
    /*  
     * Regular interface method, which every class needs to provide a definition  
     */  
    public void capturePhoto();  
}
```

```
/**  
 * BasicMobile class provides definition only for mandatory abstract method  
 * Need to provide definition for capturePhoto()  
 * Whereas, makeACall() takes the default implementation  
 */
```

```
class BasicPhone implements MobileInterface {  
    @Override  
    public void capturePhoto() {  
        System.out.println("****BASIC PHONE*** Cannot capture photo");  
    }  
}
```

```
/**  
 * SmartPhone class overrides both default method and abstract method  
 * Provides definition for both makeACall() and capturePhoto() methods  
 */
```

```
class SmartPhone implements MobileInterface {  
    @Override  
    public void makeACall(Long number) {  
        System.out.println(String.format("****SMART PHONE*** Can make  
                                         voice and video call to number .... %d", number));  
    }  
    @Override  
    public void capturePhoto() {  
        System.out.println("****SMART PHONE*** Can capture photo");  
    }  
}
```

```
/**  
 * MobileInterfaceDemo is the driver class  
 */  
public class MobileInterfaceDemo {  
    public static void main(String[] args) {  
        MobileInterface basicPhone = new BasicPhone();  
        MobileInterface smartPhone = new SmartPhone();  
  
        // Calls static method of interface  
        MobileInterface.printWelcomeMessage();  
        System.out.println("***** BASIC PHONE *****");  
        // Calls default implementation of interface  
        basicPhone.makeACall(1234567890L);  
        // Calls abstract method of interface  
        basicPhone.capturePhoto();  
  
        System.out.println("***** SMART PHONE *****");  
        // Calls overridden implementation of makeACall()  
        smartPhone.makeACall(1234567890L);  
        // Calls abstract method of interface  
        smartPhone.capturePhoto();  
    }  
}
```

# Downcasting

Class Base {...}

Class SubBase extends Base{...}

Base objB=new Base();

SubBase objSB1 = new SubBase();

SubBase objSB2;

objB = objSB1; //correct

objSB2 = objB; //compiler rejects it

objSB2 = **(SubBase)** objB; //OK

**//compiler does not verify it and trusts the  
programmer**

# Instanceof operator

- compares an object to a specified type.
- to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface.
- null is not an instance of anything.

# Instanceof operator

```
class Parent {}  
class Child extends Parent implements MyInterface {}  
interface MyInterface {}
```

Parent obj1 = new Parent();

Parent obj2 = new Child();

obj1 instanceof Parent: true

obj1 instanceof Child: false

obj1 instanceof MyInterface: false

obj2 instanceof Parent: true

obj2 instanceof Child: true

obj2 instanceof MyInterface: true

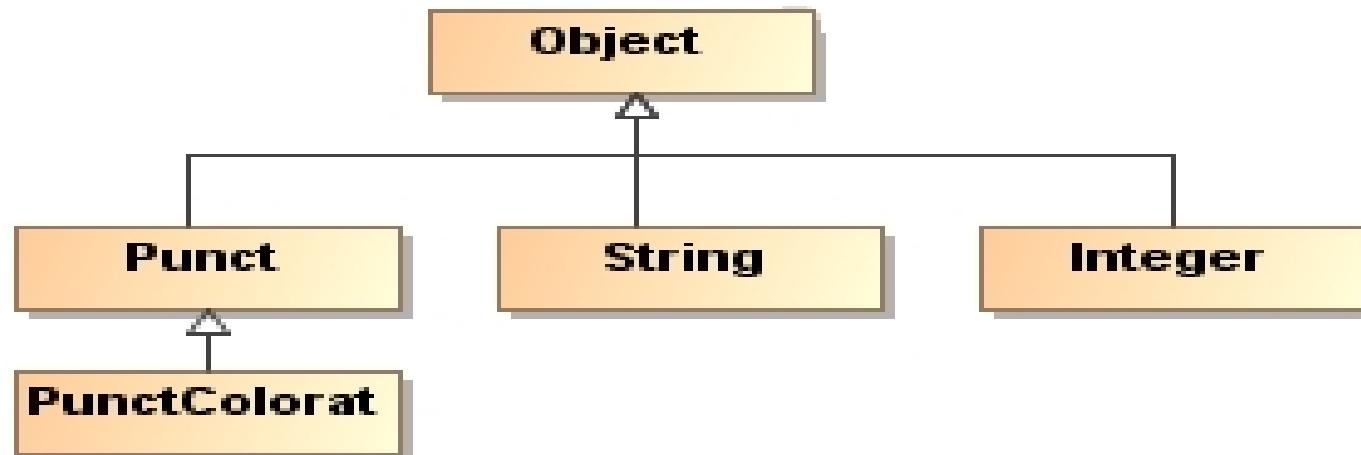
# Real use of instanceof operator

```
interface Printable{  
  
class A implements Printable{  
  
    public void a(){System.out.println("a method");}  
  
}  
  
class B implements Printable{  
  
    public void b(){System.out.println("b method");}  
  
}  
  
class Call{  
  
    void invoke(Printable p){  
  
        if(p instanceof A){  
  
            A a=(A)p;//Downcasting  
  
            a.a();  
  
        }  
  
        if(p instanceof B){  
  
            B b=(B)p;//Downcasting  
  
            b.b();  
  
        } } }
```

# The class Object

- It is the top of the java classes hierarchy.
- By default Object is the parent of a class if other parent is not explicitly defined

```
public class Punct{  
    //...  
}  
  
public class PunctColorat extends Punct{  
    //...  
}
```



# Class Object - methods

Object
+equals( o : Object ) : boolean
+toString() : String
+hashCode() : int
+notify()
+notifyAll()
+wait()
#clone() : Object
#finalize()

- **toString()** is called when a String is expected
- **equals()** is used to check the equality of 2 objects. By default it compares the references of those 2 objects.

```
Punct p1=new Punct(2,3);
Punct p2=new Punct(2,3);
boolean egale=(p1==p2);    //false;
egale=p1.equals(p2);      //true, Punct must redefine equals
System.out.println(p1);   //toString is called
```

# Overriding Class Object methods

```
public class Punct {  
    private int x,y;  
    public Punct(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    @Override  
    public boolean equals(Object obj) {  
        if (! (obj instanceof Punct))  
            return false;  
        Punct p=(Punct)obj;  
        return (x==p.x) && (y==p.y);  
    }  
  
    @Override  
    public String toString() {  
        return ""+x+' '+y;  
    }  
    //...  
}
```

# Packages

- Groups classes and interfaces
- Name space management
- ex. package `java.lang` contains the classes `System`, `Integer`, `String`, etc.
- A package is defined by the instruction `package`:

```
//structuri/Stiva.java
package structuri;
public interface Stiva{
    ...
}
```

Obs:

1. `package` must be the first instruction of the java file
2. The file is saved in the folder `structuri` (case-sensitive).

```
//structuri/liste/Lista.java
package structuri.liste;          //folder structuri/liste/Lista.java
public interface Lista{
    ...
}
```

# Using the classes declared in the packages

```
// structuri/ArboreBinar.java  
package structuri;  
public class ArboreBinar{  
    //...  
}
```

- The classes are referred using the following syntax:

```
[pac1.[pac2.[...]]]NuméClasa  
//TestStructuri.java  
public class TestStructuri{  
    public static void main(String args[]){  
        structuri.ArboareBinar ab=new structuri.ArboareBinar();  
        //...  
    }  
}
```

# Using the classes declared in the packages

- Instruction `import`:

- one class:

```
import pac1.[pac2.[...]]NumeClasa;
```

- All the package classes, but not the subpackages:

```
import pac1.[pac2.[...]]*;
```

- A file may contain multiple import instructions. They must be at the beginning before any class declaration.

```
//structuri/Heap.java
package structuri;
public class Heap{
    //...
}

//Test.java
//without import instructions
public class Test{
    public static void main(String args[]){
        structuri.ArboareBinar ab=new structuri.ArboareBinar();
        structuri.Heap hp=new structuri.Heap();
    }
}
```

# Using the classes declared in the packages

```
//Test.java
import structuri.ArboreBinar;
public class Test{
    public static void main(String args[]){
        ArboreBinar ab=new ArboreBinar();
        structuri.Heap hp=new structuri.Heap();
    }
}
//Test.java
import structuri.*;
import structuri.liste.*;
public class Test{
    public static void main(String args[]){
        ArboreBinar ab=new ArboreBinar();
        Heap hp=new Heap();
        Lista li=new Lista();
    }
}
```

# Package+import

- The instruction `package` must be before any instruction `import`

```
//algoritmi/Backtracking.java  
package algoritmi;  
import structuri.*;  
  
public class Backtracking{  
    //...  
}
```

- The package `java.lang` is implicitly imported by the compiler.

# Anonymous package

```
//Persoana.java  
public class Persoana{...}  
  
//Complex.java  
class Complex{...}  
  
//Test.java  
public class Test{  
    public static void main(String args[]){  
        Persoana p=new Persoana();  
        Complex c=new Complex();  
        //...  
    }  
}
```

If a file `.java` does not contain the instruction `package`, all the file classes are part of anonymous package.

# Name Collision

```
// unu/A.java
package unu;
public class A{
    //...
}
```

```
// doi/A.java
package doi;
public class A{
    //...
}
```

```
//Test.java
import unu.*;
import doi.*;
public class Test{
    public static void main(String[] args) {
        A a=new A(); //compilation error
        unu.A a1=new unu.A();
        doi.A a2=new doi.A();
    }
}
```

# Access modifiers

- 4 modifiers for the class members:

- **public**: access from everywhere

- **protected**: access from the same package and from subclasses

- **private**: access only from the same class

- : access only from the same package

- Classes (excepting inner classes) and interfaces can be public or nothing.

# Access modifiers

```
// structuri/Nod.java
package structuri;
class Nod{
    private Nod urm;
    public Nod getUrm(){...}
    void setUrm(Nod p){...}
    //...
}
```

//Test.java

```
import structuri.*;
class Test{
    public static void main(String args[]){
        Coada c=new Coada();
        Nod n=new Nod();      //class is not public
        Coada c2=new Coada(2); //constructor is not public
    }
}
```

```
// structuri/Coada.java
package structuri;
public class Coada{
    Nod cap;
    Coada(){ cap.urm=null; }
    Coada(int i){...}
    //...
}
```

# Access modifiers

```
package unu;
public class A{
    A(int c, int d){...}
    protected A(int c){...}
    public A(){...}
    protected void f(){...}
    void h(){...}
}
```

```
package unu;
class DA extends A{
    DA(int c){ super(c); }
}
```

```
package doi;
import unu.*;
class DDA extends A{
    DDA(int c){super(c);}
    DDA(int c, int d) {super(c,d);}
    protected void f(){
        super.h();
    }
}
```

# Protected

- The fields and methods which are declared **protected** are visible inside the class, inside the derived classes and inside the same package.

```
public class Persoana{  
    private String nume;  
    private int varsta;  
    public Persoana(String nume, int varsta) {  
        this.nume=nume;  
        this.varsta=varsta;  
    }  
    //...  
}  
  
public class Angajat extends Persoana{  
    private String departament;  
    public Angajat(String nume, int varsta, String departament) {  
        this.nume=nume;  
        this.varsta=varsta;  
        this.departament=departament;  
    }  
    //...  
}
```

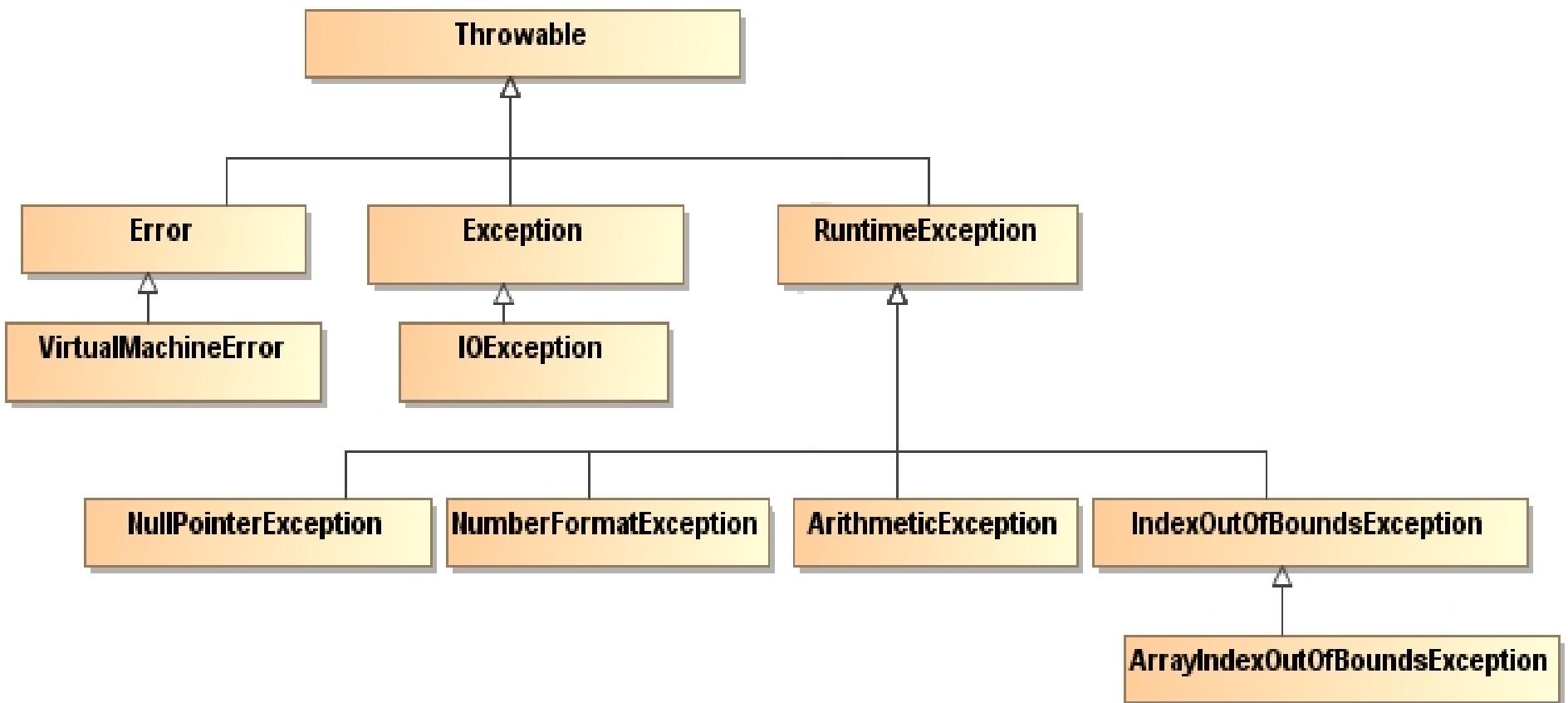
# Protected

```
public class Persoana{  
    protected String nume;  
    protected int varsta;  
    public Persoana(String nume, int varsta) {  
        this.nume=nume;  
        this.varsta=varsta;  
    }  
    //...  
}  
  
public class Angajat extends Persoana{  
    protected String departament;  
    public Angajat(String nume, int varsta, String departament) {  
        this.nume=nume;  
        this.varsta=varsta;  
        this.departament=departament;  
    }  
    //...  
}
```

# **JAVA EXCEPTIONS**

# Java Exceptions

Three types of exceptions: Errors (external to the application), Checked Exceptions (subject to try-catch), and Runtime Exceptions (correspond to some bugs)



# Example 1

Program for  $ax+b=0$ , where a, b are integers.

```
class P1{
    public static void main(String args[]){
        int a=Integer.parseInt(args[0]);      // (1)
        int b=Integer.parseInt(args[1]);      // (2)
        if (b % a==0)                      // (3)
            System.out.println("Solutie "+(-b/a));           // (4)
        else
            System.out.println("Nu exista solutie intreaga"); // (5)
    }
}

java P1 1 1 //-1
java P1 0 3 //exception, divide by 0
                //Lines 4 or 5 are not longer executed
```

# Example 1

Java VM creates the exception object corresponding to that abnormal situation and throws the exception object to those program instructions that generates the abnormal situation.

Thrown exception object can be caught or can be ignored (in our example the program P1 ignores the exception)

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at P1.main(P1.java:13)
```

# Catching exceptions

Using try-catch statement:

```
try{  
    //code that might generates abnormal situations  
}catch(TipExceptie numeVariabila){  
    //treatment of the abnormal situation  
}
```

Execution Flow:

- If an abnormal situation occurs in the block try(), JVM creates an exception object and throws it to the block catch.
- If no abnormal situation occurs, try block normally executes.
- If the exception object is compatible with one of the exceptions of the catch blocks then that catch block executes

# Example 2

```
class P2{  
    public static void main(String args[]) {  
        try{  
            int a=Integer.parseInt(args[0]);      // (1)  
            int b=Integer.parseInt(args[1]);      // (2)  
            if (b % a==0)                      // (3)  
                System.out.println("Solutie "+(-b/a)); // (4)  
            else  
                System.out.println("Nu exista solutie intreaga"); // (5)  
        }catch(ArithmeticeException e){  
            System.out.println("Nu exista solutie"); // (6)  
        }  
    }  
}  
  
java P2 1 1 //Solutie -1  
java P2 0 3 // Nu exista solutie  
          // (1), (2), (3), (6) are executed
```

# Multiple catch clauses

```
try{  
    //code with possible errors  
}catch(TipExceptione1 numeVariabila1) {  
    //instructions  
}catch(TipExceptione2 numeVariabila2) {  
    //instructions  
}...  
catch(TipExceptionen numeVariabilan) {  
    // instructions  
}
```

# Example 3

```
class P3{  
    public static void main(String args[]) {  
        try{  
            int a=Integer.parseInt(args[0]);      // (1)  
            int b=Integer.parseInt(args[1]);      // (2)  
            if (b % a==0)                      // (3)  
                System.out.println("Solutie "+(-b/a)); // (4)  
            else  
                System.out.println("Nu exista solutie intreaga"); // (5)  
        }catch(ArithmetricException e){  
            System.out.println("Nu exista solutie"); // (6)  
        }catch(ArrayIndexOutOfBoundsException e){  
            System.out.println("java P3 a b"); // (7)  
        }    }  
}  
java P3 1 1 //Solution -1  
java P3 0 3 // Nu exista solutie  
                           // (1), (2), (3), (6) are executed  
java P3 1 //java P3 a b
```

# Nested try statements

```
class P4{  
    public static void main(String args[]) {  
        try{  
            int a=Integer.parseInt(args[0]);      // (1)  
            int b=Integer.parseInt(args[1]);      // (2)  
            try{  
                if (b % a==0)                  // (3)  
                    System.out.println("Solutie "+(-b/a));           // (4)  
                else  
                    System.out.println("Nu exista solutie intreaga"); // (5)  
            }catch(ArithmeticException e){  
                System.out.println("Nu exista solutie"); // (6)  
            }  
            }catch(ArrayIndexOutOfBoundsException e){  
                System.out.println("java P4 a b");      // (7)  
            } }}  
  
java P4 1 1 //Solutie -1  
java P4 0 3 // Nu exista solutie  
java P4 1 //java P4 a b
```

# Finally clause

The finally clause is executed in any situation:

```
try{
    //...
} catch(TipExceptie1 numeVar1) {
    //instructiuni
} [catch(TipExceptien numeVarn) {
    // ...
}]
[finally{
    //instructiuni
}]
}
```

# Finally Clause

```
A  
try{  
    B  
}catch(TipException nume){  
    C  
}finally{  
    D  
}  
E
```

Block D executes:

- After A and B (before E) if no exception occurs in B. (A, B, D, E)
- After C, if an exception occurs in B and that exception is caught (A, a part of B, C, D, E).
- Before exit from the method:
  - » An exception occurs in B, but is not caught (A, a part of B, D).
  - » An exception occurs in B, it is caught but a return exists in C (A, a part of B, C, D).
  - » If a return exists in B (A, B, D).

# Finally Clause

```
public void writeElem(int[] vec) {  
    PrintWriter out = null;  
    try {  
        out = new PrintWriter(new FileWriter("fisier.txt"));  
        for (int elem:vec)  
            out.print(" "+elem);  
    } catch (IOException e) {  
        System.err.println("IOException: "+e);  
    } finally{  
        if (out != null)  
            out.close();  
    }  
}
```

# Defining exception classes

- By deriving from class `Exception`:

```
public class ExceptieNoua extends Exception{  
    public ExceptieNoua() {}  
    public ExceptieNoua(String mesaj) {  
        super(mesaj);  
    }  
}
```

# Exceptions Specification

- Use keyword `throws` in method signatures:

```
public class ExceptieNoua extends Exception{}
```

```
public class A{  
    public void f() throws ExceptieNoua{  
        //...  
    }  
}
```

- Many exceptions can be specified (their order does not matter):

```
public class Exceptie1 extends Exception{}  
public class B{  
    public int g(int d) throws ExceptieNoua, Exceptie1{  
        //...  
    }  
}
```

# Throwing exceptions

- Statement `throw` :

```
public class B{  
    public int g(int d) throws ExceptieNoua, Exceptie1{  
        if (d==3)  
            return 10;  
        if (d==7)  
            throw new ExceptieNoua();  
        if (d==5)  
            throw new Exceptie1();  
        return 0;  
    }  
    //...  
}
```

- Statement `throw` throws away the exception object and the method execution is interrupted.
- All exceptions thrown inside a method must be specified in the method signature.

# Calling a method having exceptions

- use try-catch to treat the exception:

```
public class C{  
    public void h(A a) {  
        try{  
            a.f();  
        } catch(ExceptieNoua e) {  
            System.err.println(" Exceptie "+e);  
        }  
    }  
}
```

- Throwing away an uncaught exception (uncaught exception must be specified in the signature):

```
public class C{  
    public void t(B b) throws ExceptieNoua {  
        try{  
            int rez=b.g(8);  
        } catch(Exceptie1 e){      //only Exceptie1 is caught  
            System.err.println(" Exceptie "+e);  
        }  
    }  
}
```

# Exception specification

- The subclass constructor must specify all the base class constructor (explicitly or implicitly called) in its signature.
- The subclass constructor may add new exceptions to its signature.

```
public class A{  
    public A() throws Exceptie1{  
    }  
    public A(int i){ }  
    //...  
}  
  
public class B extends A{  
    public B() throws Exceptie1{ }  
    public B(int i){  
        super(i);  
    }  
    public B(char c) throws Exceptie1, ExceptieNoua{  
    }  
    //...  
}
```

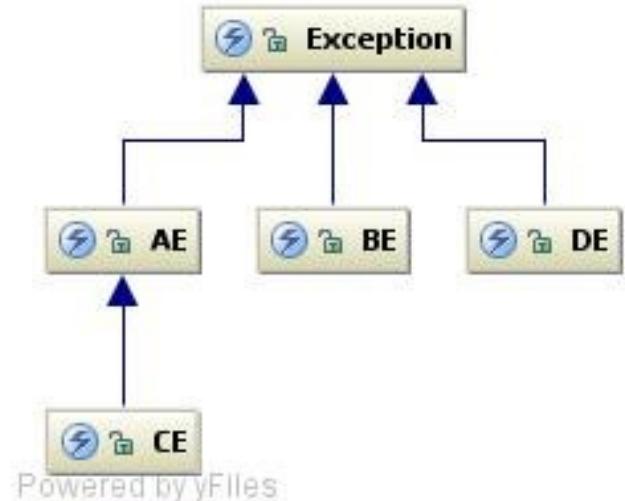
# Exceptions and method overriding

- An overriding method may declare a part of the exceptions of the overridden method.
- An overriding method may add only new exceptions which are inherited from the overridden method exceptions
- The same rules are applied for the interfaces.

```
public class AA {  
    public void f() throws Exceptie1, Exceptie2{ }  
    public void g(){ }  
    public void h() throws Exceptie1{ }  
}  
  
public class BB extends AA{  
    public void f() throws Exceptie1{ } //Exceptie2 is not declared  
    public void g() throws Exceptie2{ } //not allowed  
    public void h() throws Exceptie3{ }  
}  
public class Exceptie3 extends Exceptie1{...}
```

# Exceptions and method overriding

```
public class A {  
    public void f() throws AE, BE {}  
    public void g() throws AE {}  
}  
  
public class B extends A{  
    public void g() {}  
    public void f() throws AE, BE, CE{}  
    public void f() throws AE, BE, DE{}  
    public void g() throws DE{}  
}  
//?
```



# Exceptions order in catch clauses

- The order of catch clauses is important since the JVM selects the first catch clause on which the try block thrown exception matches.
- An exception A matches an exception B if A and B have the same class or A is a subclass of B.

```
public class C {  
    public void g(B b) {  
        try{  
            b.f();  
        } catch(Exception e) { ... }  
        } catch(CE ce) { ... }  
        } catch(AE ae) { ... }  
        } catch(BE be) { ... }  
    }  
}
```

```
public class C {  
    public void g(B b) {  
        try{  
            b.f();  
        } catch(CE ce) { ... }  
        } catch(AE ae) { ... }  
        } catch(BE be) { ... }  
        } catch(Exception e) { ... }  
    }  
}
```

# Re-throwing an exception

- A caught exception can be re-thrown

```
public class C {  
    public int h(A a) throws Exceptie4, BE {  
        try {  
            a.f();  
        } catch (BE be) {  
            System.out.println("Exceptie rearuncata "+be);  
            throw be;  
        } catch (AE ae) {  
            throw new Exceptie4("mesaj", ae);  
        }  
        return 0;  
    }  
}  
  
public class Exceptie4 extends Exception {  
    public Exceptie4() {}  
    public Exceptie4(String message) {  
        super(message);  
    }  
    public Exceptie4(String message, Throwable cause) {  
        super(message, cause);  
    }  
}
```

# **Advanced Programming Methods**

## **Lecture 3 – Java Generics and Collections**

# Lecture Overview

1. Enhanced For
2. Generics
3. Collections

# Enhanced FOR (EACH) statement

Syntax:

```
for(Type elemName : tableName)
    instr;

int[] x={1, 4, 6, 9, 10};
for(int el:x)
    System.out.println(el);

for(int i=0;i<x.length;i++)
    System.out.println(x[i]);
```

# JAVA GENERICS

# A non generic class

```
public class SingleBox {  
    private Object first;  
    public void setFirst(Object object) { this.first = object; }  
    public Object getFirst() { return first; }}
```

## Class usage:

```
SingleBox ob= new SingleBox();  
Integer i = new Integer(10);  
ob.setFirst(i);//information about Integer is lost
```

```
//we need a downcast to recover the specific information, namely the Integer  
Integer a = (Integer) ob.getFirst();  
//the cast may be wrong, it will pass the compiler but a runtime error will occur  
String s = (String) ob.getFirst();
```

# First generic version

```
public class SingleBox<T> {  
    // T stands for "Type"  
    protected T first;  
    public void setFirst(T t) { this.first = t; }  
    public T getFirst() { return first; }}
```

## Class usage:

```
SingleBox<Integer> ob= new SingleBox<Integer>();
```

```
Integer i = new Integer(10);
```

```
ob.setFirst(i);
```

```
Integer a = ob.getFirst();
```

**String s = ob.getFirst(); //an error will be generated at compile time**

# Inheritance Example

```
public class PairBox<T1, T2> extends SingleBox<T1> {  
    private T2 second;  
    public PairBox(T1 f, T2 s) {  
        this.first = f; this.second = s;  
    }  
  
    public T2 getSecond() { return second; }  
    public void setSecond(T2 t) { this.second=t; }  
}
```

# Generics

- Parameterized types
- Started with Java 1.5
- Different than C++ templates
- It does not generate a new class for each parameterized type
- The constraints can be imposed on the type variables of the parameterized types.

# Generic Class declaration

```
[access_mode] class ClassName <TypeVar1[, TypeVar2[, ...]]>{  
    TypeVar1 field1;  
    [declarations of fields]  
    [declarations and definitions of methods]  
}
```

Obs:

Type variables must be upper letters(for example E for element, K for key, V for value, T, U, S ...).

```
public class Stiva<E>{  
    private class Nod<T>{  
        T info;  
        Nod<T> next;  
        Nod(){info=null; next=null;}  
        Nod(T info, Nod next){  
            this.info=info;  
            this.next=next;  
        }  
    }//class Nod  
    Nod<E> top;  
    //...  
}
```

# Generic Classes Usage

```
public class Test{  
    public static void main(String[] args){  
        Stiva<String> ss=new Stiva<String>();  
        ss.push("Ana");  
        ss.push("Maria");  
        ss.push(new Persoana("Ana", 23));      //error at compile-time  
        String elem=ss.pop();                  //NO CAST  
  
        Stiva<Persoana> sp=new Stiva<Persoana>();  
        sp.push(new Persoana("Ana", 23));  
        sp.push(new Persoana("Maria", 10));  
  
        Dictionar<String, String> dic=new Dictionar<String, String>();  
        dic.add("abc", "ABC");  
        dic.add(23, "acc");   //error at compile-time  
        dic.add("acc", 23);  //error la compile-time  
    }  
}
```

# Autoboxing

Type variables can be instantiated only with reference types. Primitive types: int, byte, char, float, double,... are not allowed. Therefore the corresponding reference types are used.

```
Stiva<int> si=new Stiva<int>();  
//error at compile-time  
  
Stiva<Integer> si=new Stiva<Integer>();
```

primitive types	Corresponding reference types
boolean	Boolean
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

# Autoboxing

- Autoboxing: automatic conversion of a value of a primitive type to an object instance of a corresponding reference type when an object is expected, and vice-versa when a primitive value is expected.

```
Stiva<Integer> si=new Stiva<Integer>();  
si.push(23);           //autoboxing  
si.push(new Integer(23));  
  
int val=si.pop();
```

```
Character ch = 'x';  
char c = ch;
```

# Generic methods

## ■ Methods with type variables

```
class ClassName[<TypeVar ...>] {  
[access_mod] <TypeVar1[, TypeVar2[, ...]]> TypeR nameMethod([list_param]) {  
}  
//...  
}
```

Obs:

- Static methods cannot use the type variables of the class.
- A generic method can contain type variables different than those used by the generic class.
- A generic method can be defined in a non-generic class.

# Generic methods

```
public class GenericMethods {  
    public <T> void f(T x) {  
        System.out.println(x.toString());  
    }  
  
    public static <T> void copy(T[] elems, Stiva<T> st) {  
        for(T e:elems)  
            st.push(e);  
    }  
}
```

# Calling a generic method

- The compiler automatically infers the types which instantiate the type variables when a generic method is called.

```
public class A {  
    public <T> void print(T x) {  
        System.out.println(x);  
    }  
  
    public static void main(String[] args) {  
        A a=new A();  
        a.print(23);  
        a.print("ana");  
        a.print(new Persoana("ana",23));  
    }  
}
```

# Calling a generic method

- The instantiations of the type vars are explicitly given:

- Instance method:

```
a.<Integer>print(3);  
a.<Persoana>print(new Persoana("Ana", 23));
```

- Static method :

```
NameClass.<Typ>nameMethod([parameters]);  
//...  
Integer[] ielem={2,3,4};  
Stiva<Integer> st=new Stiva<Integer>();  
GenericMethods.<Integer>copy(ielem, st);  
//
```

- Non-static method in a class:

```
this.<Typ>nameMethod([parameters]);  
class A{  
    public <T> void print(T x){...}  
    public void g(Complex x){  
        this.<Complex>print(x);  
    }  
}
```

# Raw Types

```
SingleBox<String> stringBox = new SingleBox<>();
```

Raw types is the non-generic class SingleBox

```
SingleBox rawBox = stringBox; // OK
```

But if you assign a raw type to a parameterized type, you get a warning:

```
SingleBox rawBox = new SingleBox(); // rawBox is a raw type of SingleBox<T>
```

```
SingleBox<Integer> intBox = rawBox; // warning: unchecked conversion
```

# Raw Types

You also get a warning if you use a raw type to invoke generic methods defined in the corresponding generic type:

```
SingleBox<String> stringBox = new SingleBox<>();  
SingleBox rawBox = stringBox;  
rawBox.setFirst(new Integer(8));  
// warning: unchecked invocation to setFirst(T)
```

**The warnings show that raw types bypass generic type checks, deferring the catch of unsafe code to runtime.**

**Therefore, you should avoid using raw types!!!!**

# Generic arrays

- Cannot be created using new:

```
T[] elem=new T[dim]; //error at compile time
```

but we can use:

```
T[] elem=(T[])new Object[dim]; //warning at compile-time
```

- Alternatives:

- Using `Array.newInstance`

```
import java.lang.reflect.Array;  
  
public class Stiva <E>{  
  
    private E[] elems;  
    private int top;  
    @SuppressWarnings("unchecked")  
    public Stiva(Class<E> tip) {  
        elems= (E[])Array.newInstance(tip, 10);  
        top=0;  
    }  
    //...  
}  
  
Stiva<Integer> si=new Stiva<Integer>(Integer.class);
```

- Using `ArrayList` instead of array.

# Generic arrays

- Use an array of Object, but read operation requires an explicit cast:

```
public class Stiva <E>{  
    private Object[] elems;  
    private int top;  
    public Stiva() {  
        elems=new Object[10];  
        top=0;  
    }  
    public void push(E elem){  
        elems[top++]=elem;  
    }  
    @SuppressWarnings("unchecked")  
    public E pop(){  
        if (top>0)  
            return (E)elems[--top];  
        return null;  
    }  
    //...  
}
```

# Bounds

```
public class ListOrd<E> {  
    private class Nod<E>{  
        E info;  
        Nod<E> nxt;  
        public Nod(){ info=null; nxt=null; }  
        private Nod(E info, Nod<E> nxt) { this.info = info; this.nxt = nxt; }  
        private Nod(E info) { this.info = info; nxt=null; }  
    }  
    private Nod<E> head;  
    public ListOrd(){ head=null; }  
    public void add(E elem){  
        if (head==null){  
            head=new Nod<E>(elem);  
            return;  
        }  
        if /*compare elem to head.info*/{  
            head=new Nod<E>(elem,head);  
        }else {...}  
    }  
}
```

# *Bounds*

- Type variables can have constraints (namely bounds) using `extends`.

`T extends E`    //T is the type E or is a subtype of E.

- General form of the constraint:

`T extends [C &] I1 [& I2 &...& In]`

T inherits the class C and implements the interfaces I<sub>1</sub>, ... I<sub>n</sub>.

- If T has constraints then through T we can call any method from the class and interfaces specified as bounds.

# Bounds

```
public interface Comparable<E>{
    int compareTo(E e);
}

public class ListOrd<E extends Comparable<E>> {
    private class Nod<E>{...}
    private Nod<E> head;
    public ListOrd(){ head=null;}
    public void add(E elem){
        if (head==null){
            head=new Nod<E>(elem);
            return;
        }
        if (elem.compareTo(head.info)<0){
            head=new Nod<E>(elem,head);
        }else {...}
    }
    public E retElemPoz(int poz){
        //...
    }
}
```

# Motivation for Wildcards

```
ListOrd<String> ls=new ListOrd<String>();  
ListOrd<Object> lo=ls; //ASSUME this is CORRECT  
  
lo.add(23);  
String s=ls.retElemPoz(0); //ERROR
```

Our assumption is wrong!!!

# Motivation for Wildcards

## The correct rule:

If **SB** is a subclass (subtype) of **T** and **G** is a generic container class then **G<SB>** is not a subclass (subtype) of **G<T>**.

```
void printLista(ListOrd<Object> lo) {  
    for(Object o:lo)  
        System.out.println(o);  
}  
...  
ListOrd<String> ls=new ListOrd<String>();  
ls.add("mere");  
ls.add("pere");  
printLista(ls); //error at compile-time
```

# Wildcards

We use ? to denote any type (or unknown type)

```
void printLista(ListOrd<?> lo) {  
    for(Object o:lo)  
        System.out.println(o);  
}
```

Obs:

- When we use ?, the elements can be considered to be of type Object
- When we use ? to declare an instance, the instance elements cannot be read or write, the only allowed operations are to read Object and to write null.

```
ListOrd<String> ls=new ListOrd<String>();  
ls.add("mere");  
ls.add("pere");  
ListOrd<?> ll=ls;  
ll.add("portocale"); //error  
ll.update(1, "struguri");//error  
Object el=ll.retElemPoz(0);
```

# Wildcards

SingleBox<?>

- is the top of the hierarchy

```
SingleBox<?> wildbox= new SingleBox<Number>;
```

```
SingleBox<Integer> intbox= new SingleBox<Integer>;
```

```
wildbox = intbox; //OK
```

```
intbox.setFirst(new Integer(2)); //OK
```

```
wildbox.setFirst(new Integer(2)); //ERROR at compile time
```

```
Integer = intbox.getFirst(); //OK
```

```
Integer = wildbox.getFirst(); //ERROR at compile time
```

```
Object = wildbox.getFirst(); //OK
```

in fact SingleBox<?> means SingleBox<T> where bottom < T<Object

# Bounded Wildcards

We can specify bounds for ?:

■ Upper bound by `extends: ? extends C` or `? extends I`

■ Lower bound by `super: ? super C` (`any superclass of C`)

1. Upper bound means that we can read elements of the type (or of superclass of the type) given by the upper bound.
2. Lower bound means that we can write elements of type (or of subclasses of the type) given by the lower bound.

```
ListOrd<Angajat> la=new ListOrd<Angajat>();  
la.add(new Angajat(...));  
ListOrd<? extends Persoana> lp=la;  
lp.add(new Angajat()); //error at compile time  
Persoana p=lp.retElemPoz(0); //OK  
lp.retElemPoz(0).getNum();  
ListOrd<? super Angajat> linf=la;  
linf.add(new Angajat()); //correct
```

# *Upper Bounds—Covariant Subtyping*

- we can only read the content

```
SingleBox<? extends Integer> a = new SingleBox<Integer>(); //OK
SingleBox<? extends Number> b = a; // OK.
SingleBox<? extends Number> b =new SingleBox<Integer>(); //OK
SingleBox<? extends Integer> a = new SingleBox<Number>(); //ERROR
```

# *Upper Bounds—Covariant Subtyping*

Upper bound means that we can only read elements of the type (or of superclass of the type) given by the upper bound.

`SingleBox<? extends Number>` means `SingleBox<T>` where `bottom<T<Number` and we can read `Number` or a superclass of `Number`.

an example:

```
public Number processReadBox(Box<? extends Number> a) { return  
    a.getFirst(); }
```

and can be called by `processReadBox(v)` where

`v` has the type `Box<? extends Integer>`, `Integer` is derived from `Number` or `v` can have the type `Box<Integer>`

# *Lower Bounds—Contravariant Subtyping*

```
SingleBox<? super Number> a = new SingleBox<Number>(); //OK
SingleBox<? super Integer> b = a; // OK.
SingleBox<? super Number> b =new SingleBox<Integer>(); //ERROR
SingleBox<? super Integer> a = new SingleBox<Number>(); //OK
```

# *Lower Bounds—Contravariant Subtyping*

Lower bound means that we can write elements of type (or of subclasses of the type) given by the lower bound.

`SingleBox<? super Integer>` means `SingleBox<T>` where `Integer<T<Object` and we can write `Integer` or a subclass of `Integer`.

an example:

```
public static void processWriteBox(Box<? super Integer> a, Integer x)
{ a.setFirst(x); }
```

and can be called by `processWriteBox(v)` where

`v` has the type `Box<? super Number>, Integer` is derived from `Number` or `v` can have the type `Box<Integer>`

# *Wildcards*

**SingleBox<?>**

- is the top of the hierarchy

```
SingleBox<?> wildbox;
```

```
SingleBox<? extends Number> nrbox;
```

```
SingleBox<? super Integer> intbox;
```

```
wildbox=nrbox; //OK
```

```
wildbox=intbox; //OK
```

# *Erasur*

- Java does not create a new class for each new instantiation of the type variables in case of the generic classes.
- The compiler erases all type variables and replaces them with their upper bounds (usually Object) and explicit casts are inserted when it is necessary

```
public class A {  
    public String f (Integer ix){  
        Stiva<String> st=new Stiva<String>();  
        Stiva sts=st;  
        sts.push(ix);  
        return st.top();  
    }  
}
```

```
public class A {  
    public String f (Integer ix){  
        Stiva st=new Stiva();  
        Stiva sts=st;  
        sts.push(ix);  
        return (String)st.top();  
    }  
}
```

compilation



- Reason: backward compatibility with the non-generic Java versions

- The generic class is not recompiled for each new instantiation of the type variables like in C++.

# JAVA COLLECTIONS

# Java Collections Framework (JCF)

A *collection* is an object that maintains references to other objects  
JCF is part of the `java.util` package and provides:

## Interfaces

- Each defines the operations and contracts for a particular type of collection (List, Set, Queue, etc)
- Idea: when using a collection object, it's sufficient to know its interface

## Implementations

- Reusable classes that implement above interfaces (e.g. LinkedList, HashSet)

## Algorithms

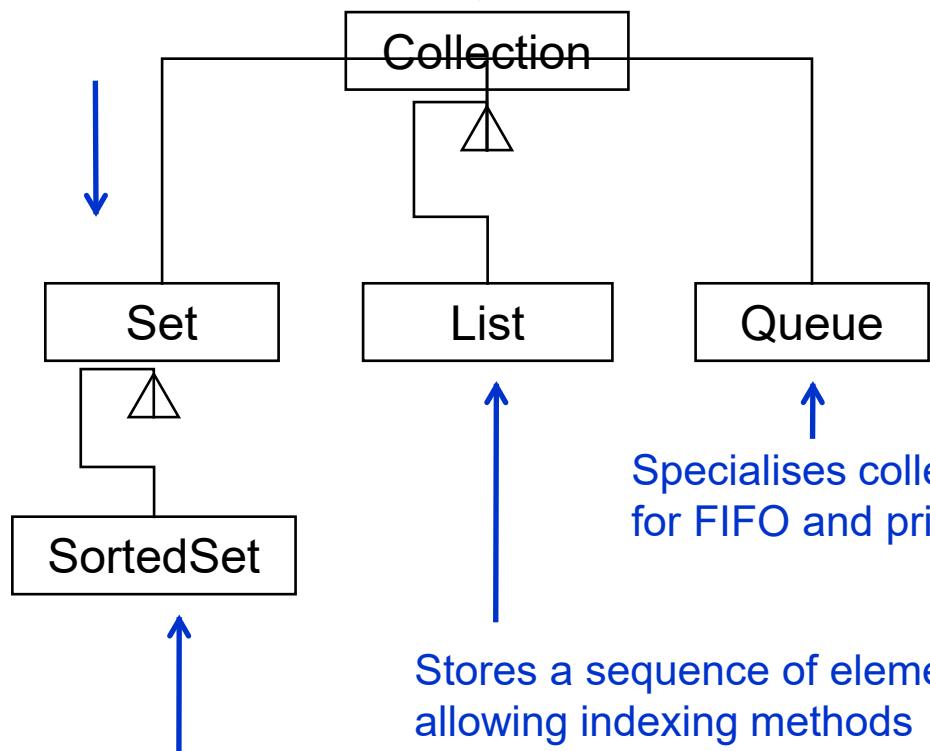
- Useful polymorphic methods for manipulating and creating objects whose classes implement collection interfaces

• Sorting, index searching, reversing, replacing, etc

# Generalisation

# Interfaces

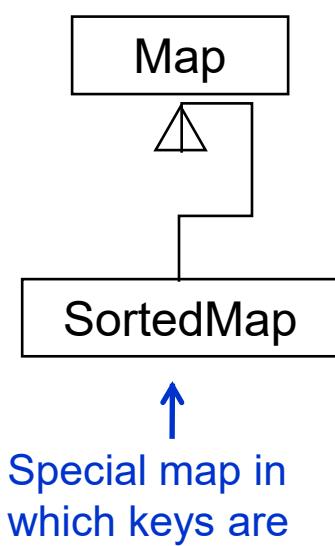
A special Collection that cannot contain duplicates.



Special Set that retains ordering of elements.

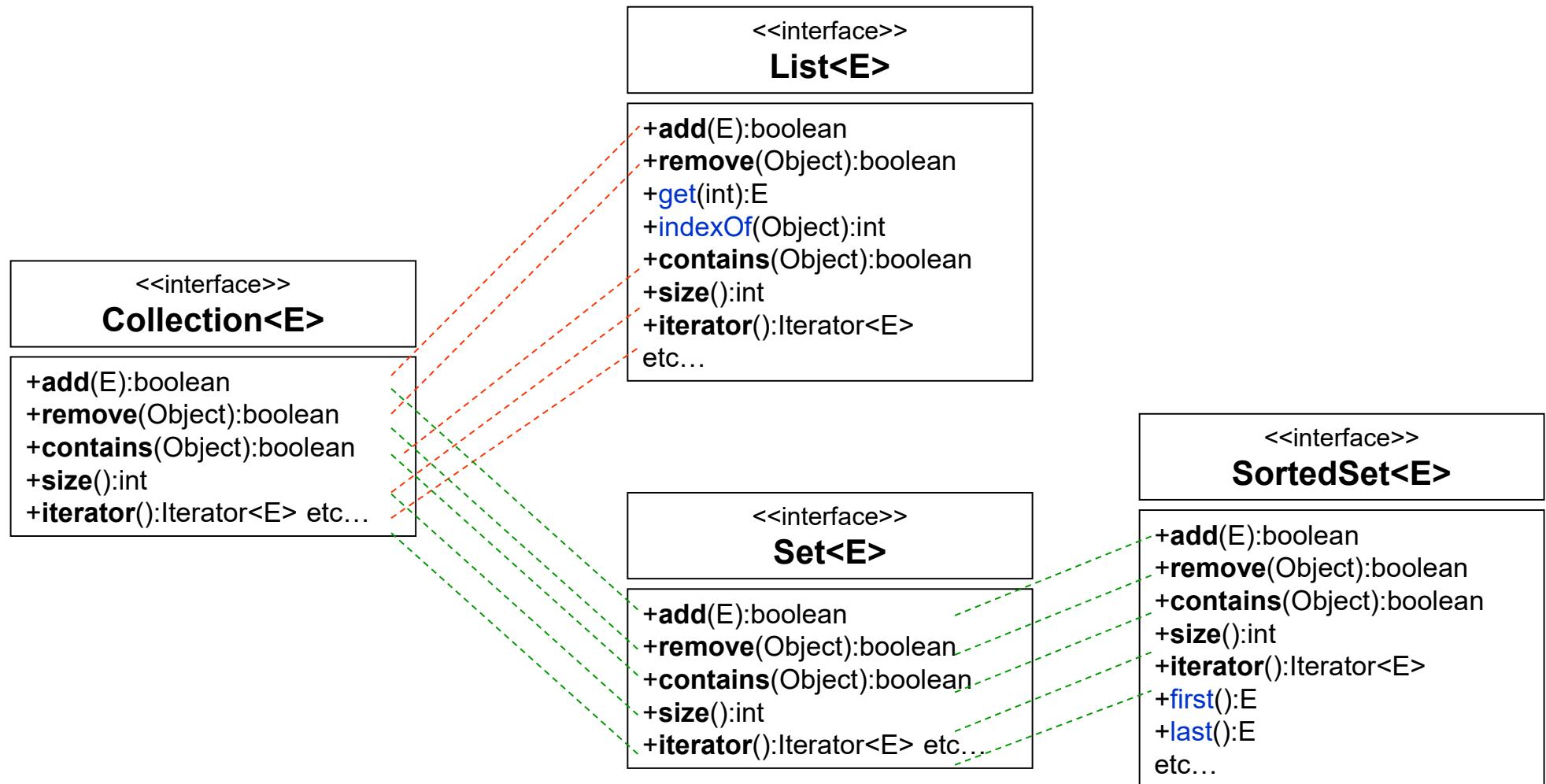
Root interface for operations common to all types of collections

Stores mappings from keys to values



# Specialisation

# Expansion of contracts



# The Collection Interface

---

- The Collection interface provides the basis for List-like collections in Java. The interface includes:

```
boolean add(Object)
boolean addAll(Collection)
void clear()
boolean contains(Object)
boolean containsAll(Collection)
boolean equals(Object)
boolean isEmpty()
Iterator iterator()
boolean remove(Object)
boolean removeAll(Collection)
boolean retainAll(Collection)
int size()
Object[] toArray()
Object[] toArray(Object[])
```

# List Interface

---

- Lists allow duplicate entries within the collection
  - Lists are an ordered collection much like an array
  - Lists grow automatically when needed
  - The list interface provides accessor methods based on index
- 
- The List interface extends the Collections interface and add the following method definitions:

```
void add(int index, Object)
boolean addAll(int index, Collection)
Object get(int index)
int indexOf(Object)
int lastIndexOf(Object)
ListIterator listIterator()
ListIterator listIterator(int index)
Object remove(int index)
Object set(int index, Object)
List subList(int fromIndex, int toIndex)
```

# Set Interface

---

- The Set interface also extends the Collection interface but does not add any methods to it.
- Collection classes which implement the Set interface have the add stipulation that Sets CANNOT contain duplicate elements
- Elements are compared using the equals method
- NOTE: exercise caution when placing mutable objects within a set. Objects are tested for equality upon addition to the set. **If the object is changed after being added to the set, the rules of duplication may be violated.**

## SortedSet Interface

---

- SortedSet provides the same mechanisms as the Set interface, except that SortedSets maintain the elements in ascending order.
- Ordering is based on natural ordering (Comparable) or by using a Comparator.

# java.util.Iterator<E>

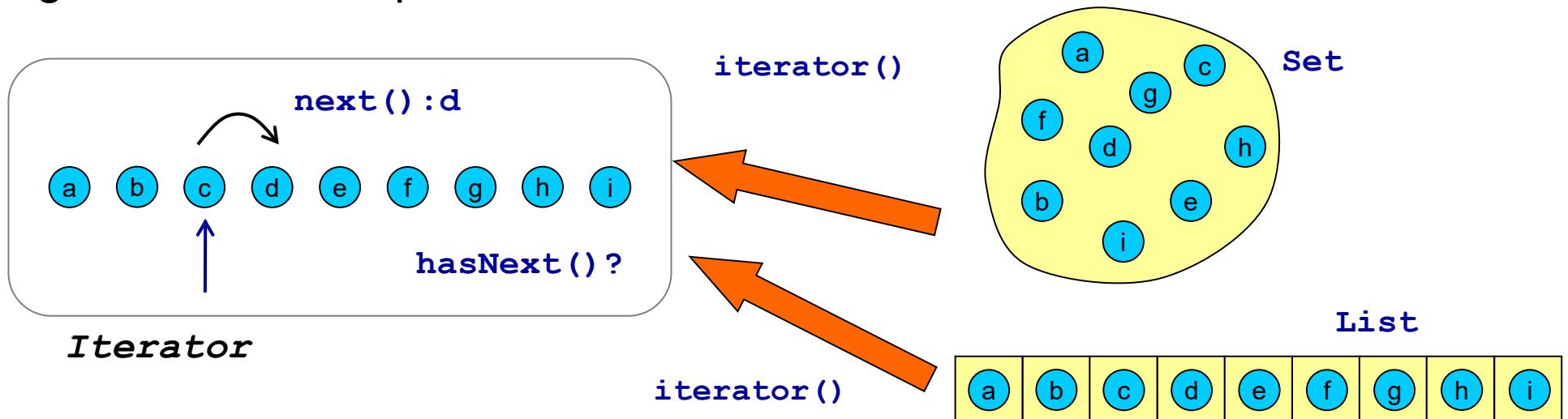
Think about typical usage scenarios for Collections

**Retrieve** the list of all patients

**Search** for the lowest priced item

More often than not you would have to traverse every element in the collection – be it a List, Set, or your own datastructure

Iterators provide a generic way to traverse through a collection regardless of its implementation



# Using an Iterator

Quintessential code snippet for collection iteration:

```
public void list(Collection<T> items) {  
    Iterator<T> it = items.iterator();  
    while(it.hasNext()) {  
        T item = it.next();  
        System.out.println(item.toString());  
    }  
}
```

<<interface>>  
**Iterator<E>**

+hasNext():boolean  
+next():E  
+remove():void

Design notes:

- Above method takes in an object whose class implements Collection
- List, ArrayList, LinkedList, Set, HashSet, TreeSet, Queue, MyOwnCollection, etc
- We know any such object can return an Iterator through method iterator()
- We don't know the exact implementation of Iterator we are getting, but **we don't care**, as long as it provides the methods next() and hasNext()
- Good practice: **Program to an interface!**

# java.lang.Iterable<T>

```
for (Item item : items) {  
    System.out.println(item);  
}
```

=

```
Iterator<Item> it = items.iterator();  
while(it.hasNext()) {  
    Item item = it.next();  
    System.out.println(item);  
}
```

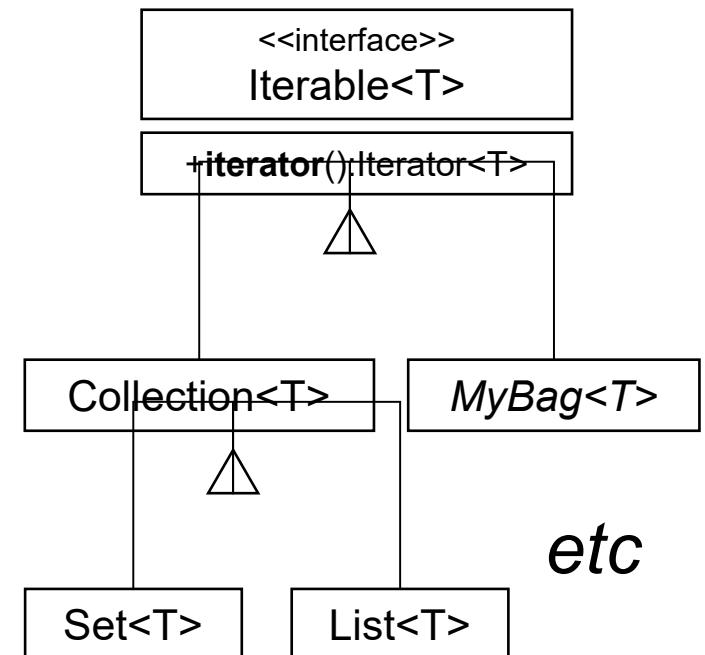
This is called a “**for-each**” statement  
For each **item** in **items**

This is possible as long as **items** is of type **Iterable**  
-Defines single method **iterator()**

**Collection** (and hence all its subinterfaces) implements  
**Iterable**

You can do this to your own implementation of **Iterable**  
too!

To do this you may need to return your own implementation of  
**Iterator**



# java.util.Collections

Offers many very useful utilities and algorithms for manipulating and creating collections

## Sorting lists

Index searching

Finding min/max

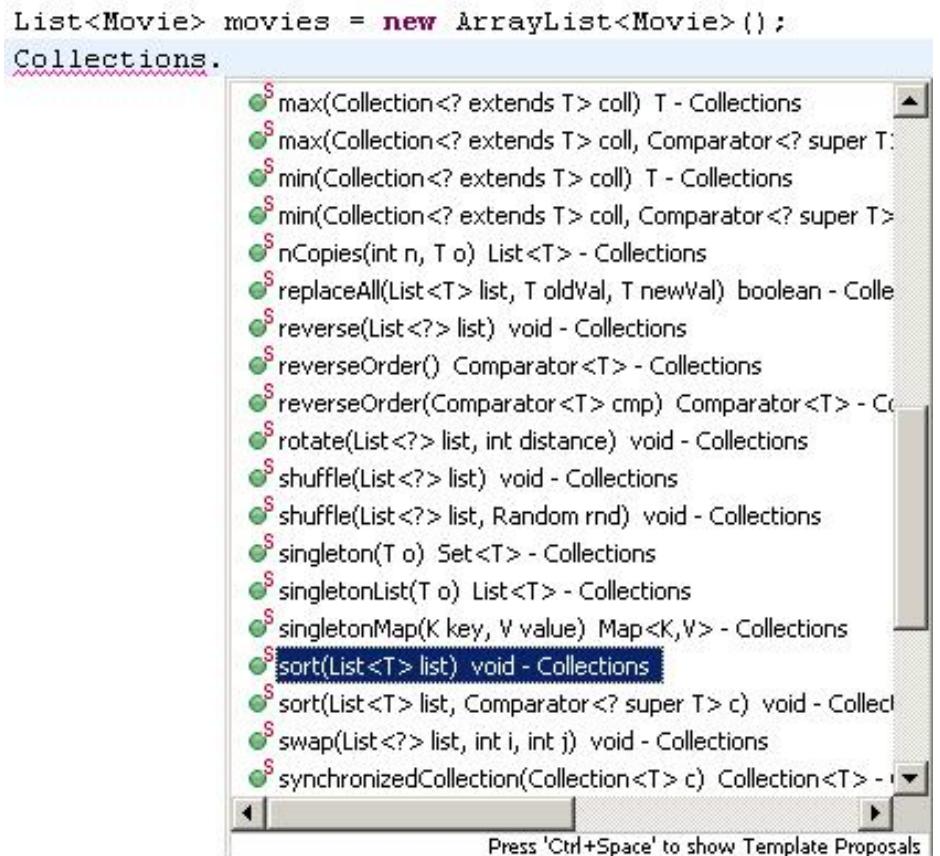
Reversing elements of a list

Swapping elements of a list

Replacing elements in a list

Other nifty tricks

Saves you having to implement them yourself → **reuse**



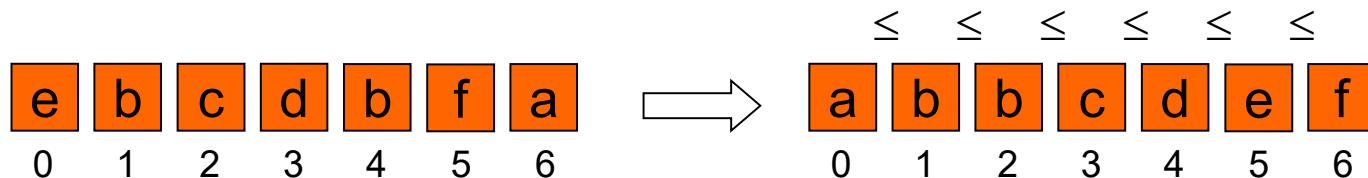
# Comparable and Comparators

---

- You will have noted that some classes provide the ability to sort elements.
- How is this possible when the collection is supposed to be de-coupled from the data?
- Java defines two ways of comparing objects:
  - The objects implement the Comparable interface
  - A Comparator object is used to compare the two objects
- If the objects in question are Comparable, they are said to be sorted by their "natural" order.
- Comparable object can only offer one form of sorting. To provide multiple forms of sorting, Comparators must be used.

# Collections.sort()

Java's implementation of merge sort – ascending order



- What types of objects can you sort? Anything that has an **ordering**
- Two sort() methods: sort a given List according to either 1) *natural ordering* of elements or an 2) externally defined ordering.

1) `public static <T extends Comparable<? super T>> void sort(List<T> list)`

2) `public static <T> void sort(List<T> list, Comparator<? super T> c)`

Translation:

- Only accepts a List parameterised with type implementing **Comparable**
- Accepts a List parameterised with any type as long as you also give it a **Comparator** implementation that defines the ordering for that type

# java.lang.Comparable<T>

A **generic interface** with a single method: `int compareTo(T)`

Return 0 if this = other

Return **any +ve integer** if this > other

Return **any -ve integer** if this < other

Implement this interface to define **natural ordering** on objects of type T

```
public class Money implements Comparable<Money> {  
    ...  
    public int compareTo( Money other ) {  
        if( this.cents == other.cents ) {  
            return 0;  
        }  
        else if( this.cents < other.cents ) {  
            return -1;  
        }  
        else {  
            return 1;  
        }  
    }  
}
```

m1 = new Money(100,0);  
m2 = new Money(50,0);  
m1.compareTo(m2) returns 1;

A more concise way of doing this? (hint: 1 line)

```
return this.cents - other.cents;
```

# Natural-order sorting

```
List<Money> funds = new ArrayList<Money>();  
funds.add(new Money(100,0));  
funds.add(new Money(5,50));  
funds.add(new Money(-40,0));  
funds.add(new Money(5,50));  
funds.add(new Money(30,0));  
  
Collections.sort(funds);  
System.out.println(funds);
```

What's the output?  
[-40.0,  
 5.50,  
 5.50,  
 30.0,  
 100.0]

```
List<CD> albums = new ArrayList<CD>();  
albums.add(new CD("Street Signs", "Ozomatli", 2.80));  
//etc...  
Collections.sort(albums);
```

CD does not implement a Comparable interface

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

# java.util.Comparator<T>

Useful if the type of elements to be sorted is not Comparable, or you want to define an alternative ordering

Also a generic interface that defines methods

**compare (T , T)** and **equals (Object)**

Usually only need to define **compare (T , T)**

Define ordering by CD's getPrice() → Money

Note: PriceComparator implements a Comparator parameterised with CD → T “becomes” CD

```
<<interface>>
Comparator<T>
+compare(T o1, T o2):int
+equals(Object other):boolean
```

```
CD
+getTitle():String
+getArtist():String
+getPrice():Money
```

```
public class PriceComparator
implements Comparator<CD> {
    public int compare(CD c1, CD c2) {
        return c1.getPrice().compareTo(c2.getPrice());
    }
}
```

Comparator and Comparable  
going hand in hand ☺

# Comparator sorting

```
List<CD> albums = new ArrayList<CD>();  
albums.add(new CD("Street Signs", "Ozomatli", new Money(3,50)));  
albums.add(new CD("Jazzinho", "Jazzinho", new Money(2,80)));  
albums.add(new CD("Space Cowboy", "Jamiroquai", new Money(5,00)));  
albums.add(new CD("Maiden Voyage", "Herbie Hancock", new Money(4,00)));  
albums.add(new CD("Here's the Deal", "Liquid Soul", new Money(1,00)));  
  
Collections.sort(albums, new PriceComparator());  
System.out.println(albums);
```

implements Comparator<CD>

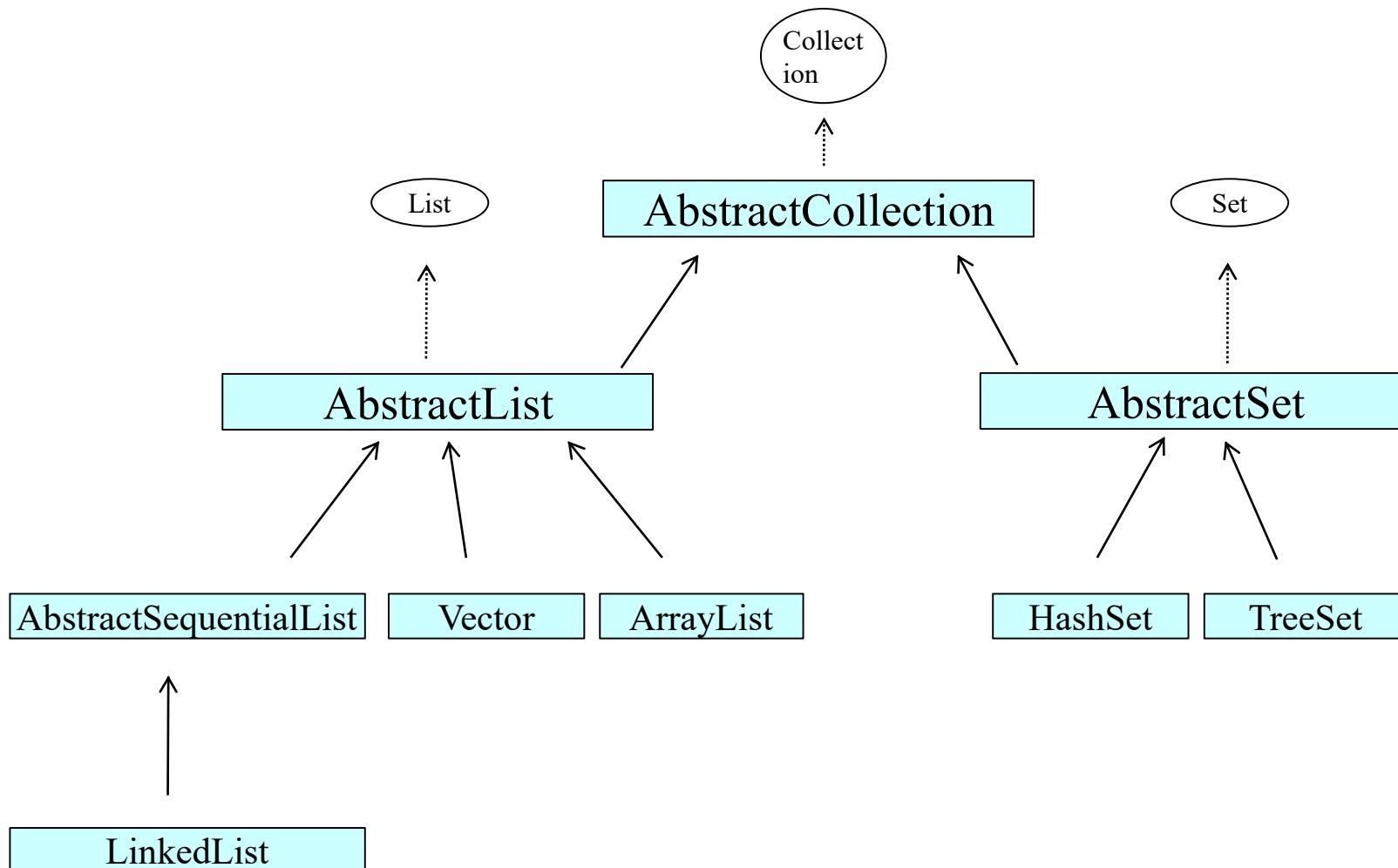
Note, in sort(), Comparator overrides natural ordering  
i.e. Even if we define natural ordering for CD, the given  
comparator is still going to be used instead  
(On the other hand, if you give null as Comparator, then natural  
ordering is used)

```
public static <T> void sort(List<T> list, Comparator<? super T> c)
```

# The Class Structure

---

- The Collection interface is implemented by a class called AbstractCollection. Most collections inherit from this class.



# Lists

---

- Java provides 3 concrete classes which implement the list interface
  - Vector
  - ArrayList
  - LinkedList
- Vectors try to optimize storage requirements by growing and shrinking as required
- Methods are synchronized (used for Multi threading)
- ArrayList is roughly equivalent to Vector except that its methods are not synchronized
- LinkedList implements a doubly linked list of elements
- Methods are not synchronized

# Sets

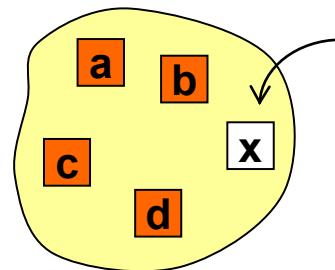
---

- Java provides 2 concrete classes which implement the Set interface
  - HashSet
  - TreeSet
- HashSet behaves like a HashMap except that the elements cannot be duplicated.
- TreeSet behaves like TreeMap except that the elements cannot be duplicated.
- Note: Sets are not as commonly used as Lists

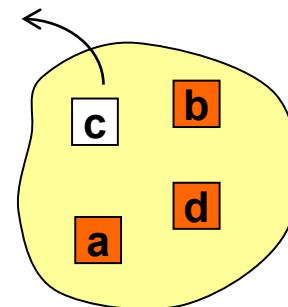
# Set<E>

Mathematical Set abstraction – contains **no duplicate** elements  
i.e. no two elements e1 and e2 such that e1.equals(e2)

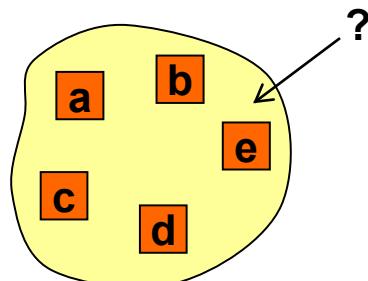
**add(x)**  
→true  
  
**add(b)**  
→false



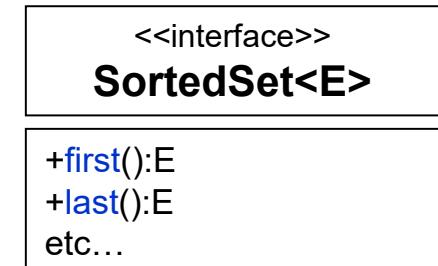
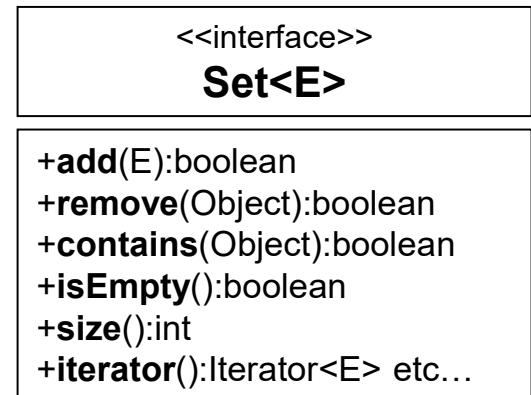
**remove(c)**  
→true  
  
**remove(x)**  
→false



**contains(e)**  
→true  
  
**contains(x)**  
→false



**isEmpty()**  
→false  
  
**size()**  
→5



# HashSet<E>

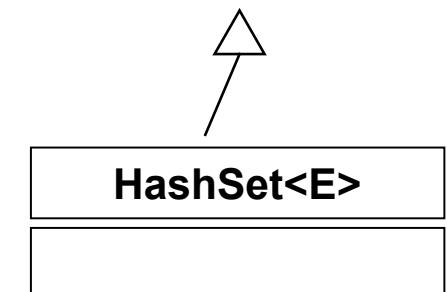
- .Typically used implementation of Set.
- .Parameterise Sets just as you parameterise Lists
- .Efficient (constant time) insert, removal and contains check
  - all done through hashing
- .x and y are duplicates if x.equals(y)
- .How are elements ordered? Quiz:

```
Set<String> words = new HashSet<String>();  
words.add("Bats");  
words.add("Ants");  
words.add("Crabs");  
words.add("Ants");  
System.out.println(words.size());  
for (String word : words) {  
    System.out.println(word);  
}
```

3  
? ←

<<interface>>  
**Set<E>**

+add(E):boolean  
+remove(Object):boolean  
+contains(Object):boolean  
+size():int  
+iterator():Iterator<E> etc...



- a) Bats, Ants, Crabs
- b) Ants, Bats, Crabs
- c) Crabs, Bats, Ants
- d) Nondeterministic

# TreeSet<E> (SortedSet<E>)

.If you want an ordered set, use an implementation of a SortedSet: TreeSet

.What's up with "Tree"? Red-black tree

.Guarantees that all elements are ordered (sorted) at all times

» `add()` and `remove()` preserve this condition

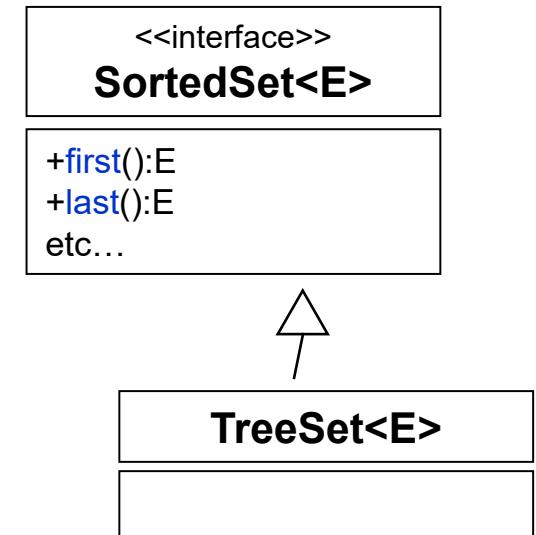
» `iterator()` always returns the elements in a specified order

.Two ways of specifying ordering

» Ensuring elements have natural ordering (**Comparable**)

» Giving a **Comparator<E>** to the constructor

**.Caution:** TreeSet considers x and y are duplicates if `x.compareTo(y) == 0` (or `compare(x,y) == 0`)



# TreeSet construction

```
Set<String> words = new TreeSet<String>();  
words.add("Bats");  
words.add("Ants");  
words.add("Crabs");  
for (String word : words) {  
    System.out.println(word);  
}
```

String has a **natural ordering**, so empty constructor

What's the output?  
**Ants; Bats; Crabs**

But CD doesn't, so you must pass in a Comparator to the constructor

```
Set<CD> albums = new TreeSet<CD>(new PriceComparator());  
albums.add(new CD("Street Signs", "O", new Money(3, 50)));  
albums.add(new CD("Jazzinho", "J", new Money(2, 80)));  
albums.add(new CD("Space Cowboy", "J", new Money(5, 00)));  
albums.add(new CD("Maiden Voyage", "HH", new Money(4, 00)));  
albums.add(new CD("Here's the Deal", "LS", new Money(2, 80)));  
System.out.println(albums.size());  
for (CD album : albums) {  
    System.out.println(album);  
}
```

What's the output?  
**4  
Jazzinho; Street; Maiden; Space**

# The Map Interface

---

- The Map interface provides the basis for dictionary or key-based collections in Java. The interface includes:

```
void clear()  
boolean containsKey(Object)  
boolean containsValue(Object)  
Set entrySet()  
boolean equals(Object)  
Object get(Object)  
boolean isEmpty()  
Set keySet()  
Object put(Object key, Object value)  
void putAll(Map)  
boolean remove(Object key)  
int size()  
Collection values()
```

# Maps

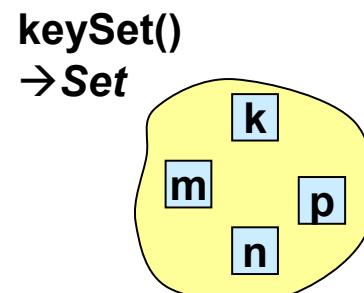
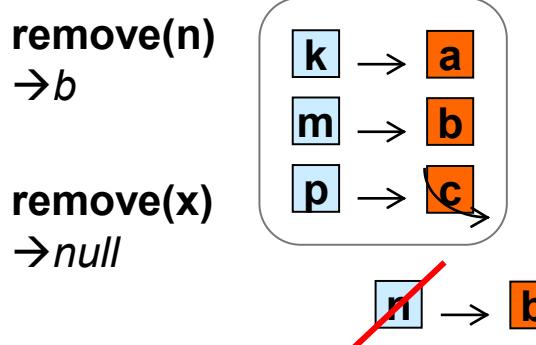
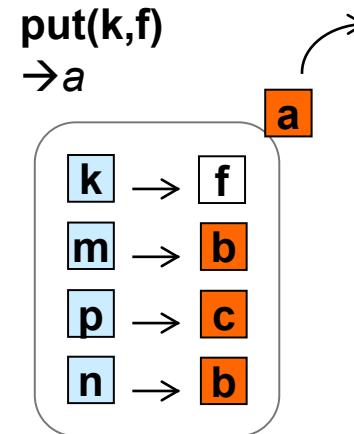
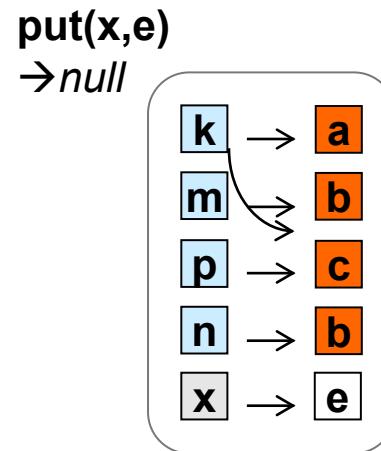
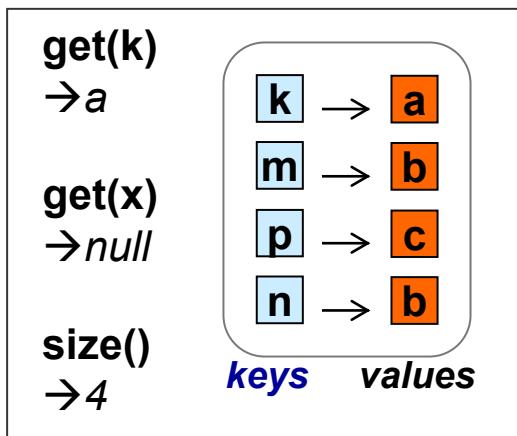
---

- Java provides 3 concrete classes which implement the map interface
  - HashMap
  - WeakHashMap
  - TreeMap
- HashMap is the most commonly used Map.
  - Provides access to elements through a key.
  - The keys can be iterated if they are not known.
- WeakHashMap provides the same functionality as Map except that if the key object is no longer used, the key and its value will be removed from the Map.
- A Red-Black implementation of the Map interface

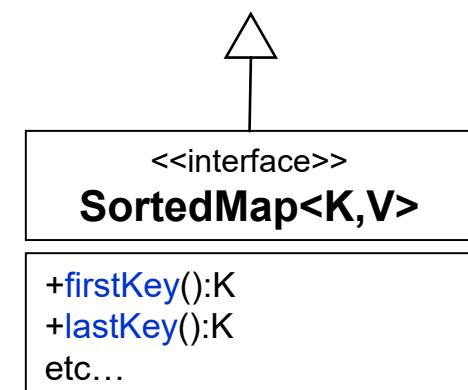
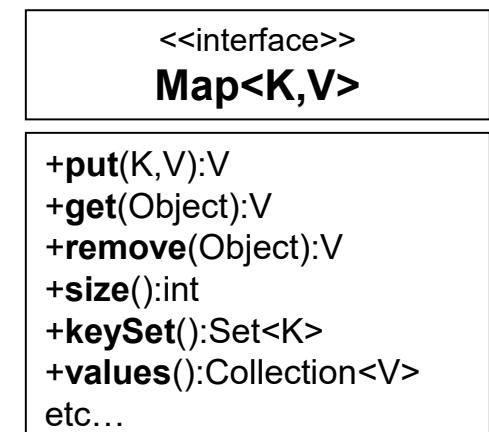
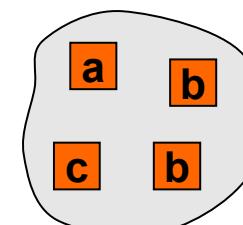
# Map<K , V>

- Stores mappings from (unique) keys (type K) to values (type V)
  - See, you can have more than one type parameters!

- Think of them as “arrays” but with objects (keys) as indexes
  - Or as “directories”: e.g. "Bob" → 021999887



**values()**  
→ Collection



# HashMap<K, V>

.keys are hashed using `Object.hashCode()`  
» i.e. no guaranteed ordering of keys

`.keySet()` returns a `HashSet`  
`.values()` returns a Collection

<<interface>>  
**Map<K,V>**

+`put(K,V):V`  
+`get(Object):V`  
+`remove(Object):V`  
+`size():int`  
+`keySet():Set<K>`  
+`values():Collection<V>`  
etc...

```
Map<String, Integer> directory
    = new HashMap<String, Integer>();
directory.put("Mum", new Integer(9998888));
directory.put("Dad", 9998888);
directory.put("Bob", 12345678);           ← "autoboxing"
directory.put("Edward", 5553535);
directory.put("Bob", 1000000);
System.out.println(directory.size());
for (String key : directory.keySet()) {
    System.out.print(key+"'s number: ");
    System.out.println(directory.get(key));
}
System.out.println(directory.values());
```

→      ← 4 or 5?  
→      ← Set<String>  
→      ← What's Bob's number?



**HashMap<K,V>**

# TreeMap<K, V>

.Guaranteed ordering of keys (like TreeSet)

» In fact, TreeSet is implemented using TreeMap ☺

» Hence `keySet()` returns a TreeSet

`.values()` returns a Collection – ordering depends on ordering of keys

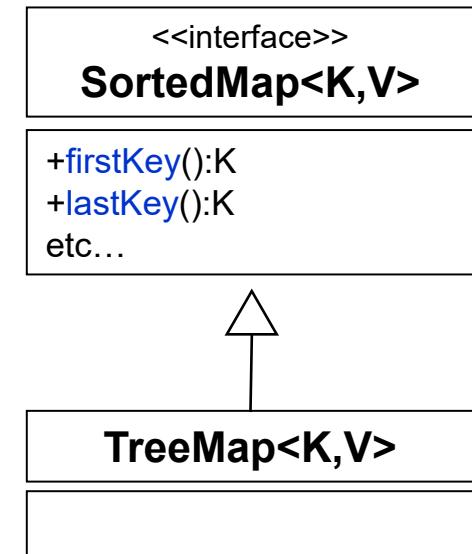
```
Map<String, Integer> directory
    = new TreeMap<String, Integer>();
directory.put("Mum", new Integer(9998888));
directory.put("Dad", 9998888);
directory.put("Bob", 12345678);
directory.put("Edward", 5553535);
directory.put("Bob", 1000000);
System.out.println(directory.size());
for (String key : directory.keySet()) {
    System.out.print(key+"'s #: ");
    System.out.println(directory.get(key));
}
System.out.println(directory.values());
```

Empty constructor  
→ natural ordering

4

Loop output?  
Bob's #: 1000000  
Dad's #: 9998888  
Edward's #: 5553535  
Mum's #: 9998888

?



# TreeMap with Comparator

As with TreeSet, another way of constructing TreeMap is to give a Comparator → necessary for non-Comparable keys

```
Map<CD, Double> ratings
    = new TreeMap<CD, Double>(new PriceComparator());
ratings.put(new CD("Street Signs", "O", new Money(3,50)), 8.5);
ratings.put(new CD("Jazzinho", "J", new Money(2,80)), 8.0);
ratings.put(new CD("Space Cowboy", "J", new Money(5,00)), 9.0);
ratings.put(new CD("Maiden Voyage", "H", new Money(4,00)), 9.5);
ratings.put(new CD("Here's the Deal", "LS", new Money(2,80)), 9.0);

System.out.println(ratings.size());
for (CD key : ratings.keySet()) {
    System.out.print("Rating for "+key+": ");
    System.out.println(ratings.get(key));
}
System.out.println("Ratings: "+ratings.values());
```

4

Ordered by key's price

Depends on key ordering

# Most Commonly Use Methods

---

- While it is a good idea to learn and understand all of the methods defined within this infrastructure, here are some of the most commonly used methods.
- For Lists:
  - add(Object), add(index, Object)
  - get(index)
  - set(index, Object)
  - remove(Object)
- For Maps:
  - put(Object key, Object value)
  - get(Object key)
  - remove(Object key)
  - keySet()

# Which class should I use?

---

- You'll notice that collection classes all provide the same or similar functionality. The difference between the different classes is how the structure is implemented.
- This generally has an impact on performance.

- **Use Vector**

- Fast access to elements using index
- Optimized for storage space
- Not optimized for inserts and deletes

- **Use ArrayList**

- Same as Vector except the methods are not synchronized. Better performance

- **Use linked list**

- Fast inserts and deletes
- Stacks and Queues (accessing elements near the beginning or end)
- Not optimized for random access

# Which class should I use?

---

- Use Sets
  - When you need a collection which does not allow duplicate entries
- Use Maps
  - Very Fast access to elements using keys
  - Fast addition and removal of elements
  - No duplicate keys allowed
- When choosing a class, it is worthwhile to read the class's documentation in the Java API specification. There you will find notes about the implementation of the Collection class and within which contexts it is best to use.

# Collections and Fundamental Data Types

---

- Note that collections can only hold Objects.
- One cannot put a fundamental data type into a Collection
- Java has defined "wrapper" classes which hold fundamental data type values within an Object
  - These classes are defined in `java.lang`
  - Each fundamental data type is represented by a wrapper class
- The wrapper classes are:

Boolean

Byte

Character

Double

Float

Short

Integer

Long

# Wrapper Classes

---

- The wrapper classes are usually used so that fundamental data values can be placed within a collection
- The wrapper classes have useful class variables.
  - Integer.MAX\_VALUE, Integer.MIN\_VALUE
  - Double.MAX\_VALUE, Double.MIN\_VALUE, Double.NaN, Double.NEGATIVE\_INFINITY, Double.POSITIVE\_INFINITY
- They also have useful class methods

Double.parseDouble(String) - converts a String to a double  
Integer.parseInt(String) - converts a String to an integer

# **Advanced Programming Methods**

**Lecture 4 and 5 - Java IO Operations**

# Overview

1. Java IO
2. Java NIO
3. Try-with-resources

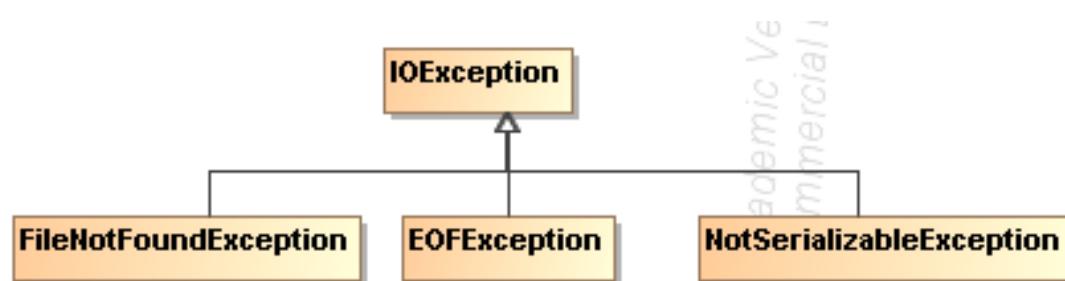
# **Java IO**

# Java.IO

## ■ Package java.io

- classes working on bytes (InputStream, OutputStream)
- Classes working on chars (Reader, Writer)
- Byte-char conversion (InputStreamReader, OutputStreamWriter)
- Random access (RandomAccessFile)
- Scanner

## ■ Exceptions:



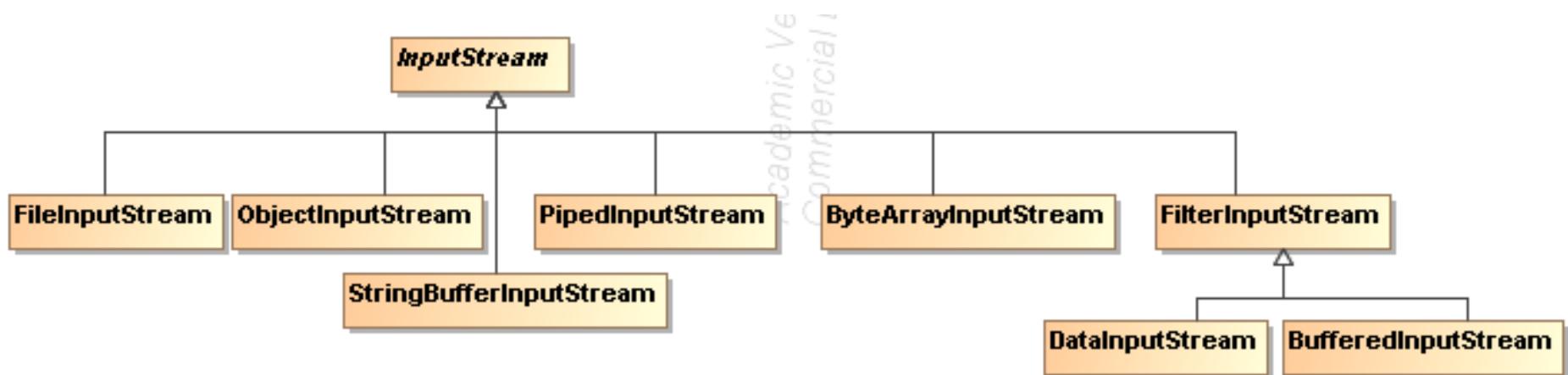
# IO Stream

- represents an input source or an output destination
- is a sequence of data
- supports many different kinds of data, including simple bytes, primitive data types, localized characters, and objects.
- simply passes on data; others manipulate and transform the data in useful ways.

# InputStream

- Abstract class that contains methods for reading bytes from a stream (file, memory, pipe, etc.)

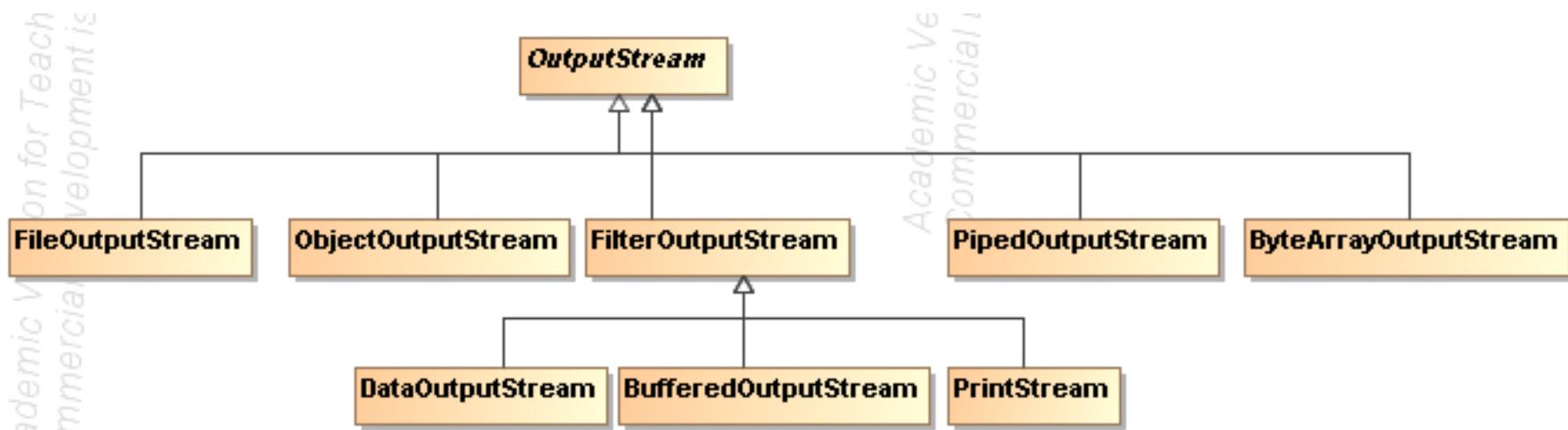
- `read():int` //read a byte, return -1 if no more bytes
- `read(cbuff:byte[]): int` //read max `cbuff.length bytes`, return the nr of bytes that has been read, or -1
- `read(buff:byte[], offset:int, length:int):int` //read max `length` bytes and write to `buff` starting with position `offset`, return the number of bytes that has been read, or -1
- `available(): int` //number of bytes available for reading
- `close()` //close the stream



# OutputStream

- Abstract class that contains methods for writing bytes into a stream (file, memory, pipe, etc.)

```
■ write(int)          //write a byte  
■ write(b:byte[])    //write b.length bytes from array b into the stream  
■ write(b:byte[], offset:int, len:int):int //write len bytes from array b starting with the position offset  
■ flush()            // force the effective writing into the stream  
■ close()            //close the stream
```



# Example

```
FileInputStream in = null;
FileOutputStream out = null;
try {
    in = new FileInputStream("fisier.txt");
    out = new FileOutputStream("fisier2.txt");
    int c;
    while ((c = in.read()) != -1) {
        out.write(c);
    }
} catch (IOException e) {
    System.err.println("Eroare "+e);
}finally {
    if (in != null)
        try {
            in.close();
        } catch (IOException e){ System.err.println("eroare "+e);}
    if (out != null)
        try {
            out.close();
        } catch (IOException e) { System.err.println("eroare "+e);}
}
```

# Reader

- Abstract class that contains methods for chars reading (1 char = 2 bytes) from a stream (file, memory, pipe, etc.)

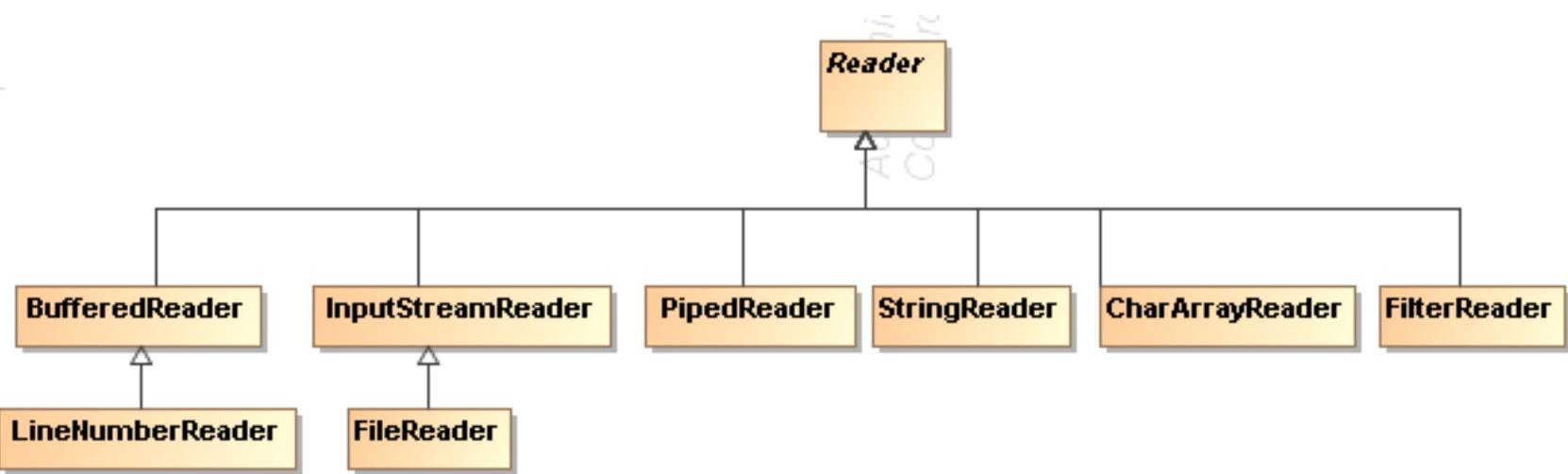
- `read():int` //read a char, return -1 for the end of the stream

- `read(cbuff:char[]): int` //read max `cbuff.length` chars, return nr of read chars or -1

- `read(buff:char[], offset:int, length:int):int` //read max `length` chars into array `buff` starting with offset `offset`, return the number of read chars or -1

- `close()` //close the stream

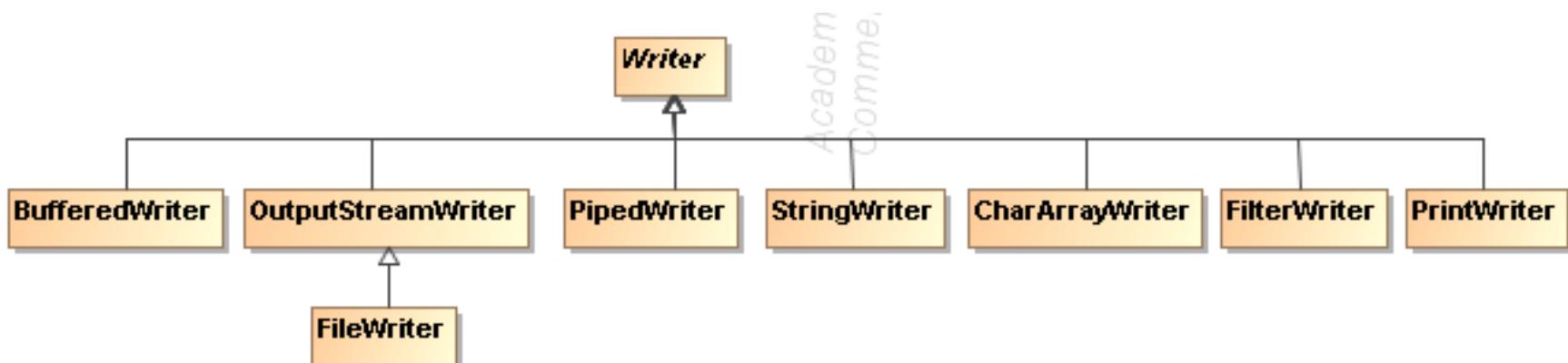
Java API Overview



# Writer

- Abstract class that contains the methods for writing chars into a stream

- `write(int)` //write a char
- `write(b:char[])` //write `b.length` chars from array `b` into the stream
- `write(b:char[], offset:int, len:int):int` //write `len` chars from array `b` starting with offset
- `write(s:String)` //write a `String`
- `write(s:String, off:int, len:int)` //write a part of a `String`
- `flush()` // force the effective writing
- `close()` //close the stream



# Example

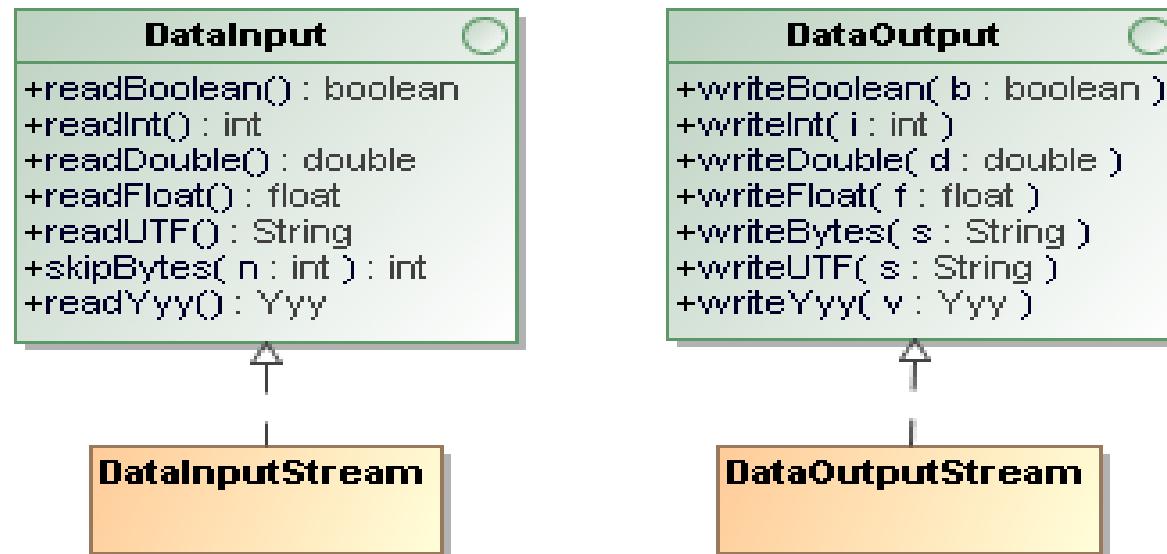
```
FileReader input = null;
FileWriter output = null;
try {
    input = new FileReader("Fisier.txt");
    output = new FileWriter("Fisier2out.txt");
    int c;
    while ((c = input.read()) != -1) output.write(c);
} catch (IOException e) {
    System.err.println("Eroare la citire scriere"+e);
} finally {
    if (input != null)
        try {
            input.close();
        } catch (IOException e) {System.err.println("eroare "+e);}
    if (output != null)
        try {
            output.close();
        } catch (IOException e) {System.err.println("Eroare "+e);}
}
```

# Classes

Operations	Byte	Char
Files	FileInputStream, FileOutputStream	FileReader, FileWriter
Memory	ByteArrayInputStream, ByteArrayOutputStream	CharArrayReader CharArrayWriter
Buffered Operations	BufferedInputStream BufferedOutputStream	BufferedReader BufferedWriter
Format	PrintStream	PrintWriter
Conversion Byte ↔ Char	InputStreamReader (byte -> char) OutputStreamWriter (char -> byte)	

# Writing and reading data of primitive types

## ■ Interfaces `DataInput` and `DataOutput`



■ `Yyy` can be primitive types (`byte`, `short`, `char`, ...): `readByte()`, `readShort()`, `readChar()`,  
`writeByte(byte)`, `writeShort(short)`, ...

Obs:

1. In order to read data of primitive types using methods `readYyy`, the data must be saved before using the methods `writeYyy`.
2. When there are no more data it throws the exception `EOFException`.

# Example

- Read a list of students (from a file, from keyboard)
- Saving the list of students in ascending order based on their average (into a file)

//Studenti.txt

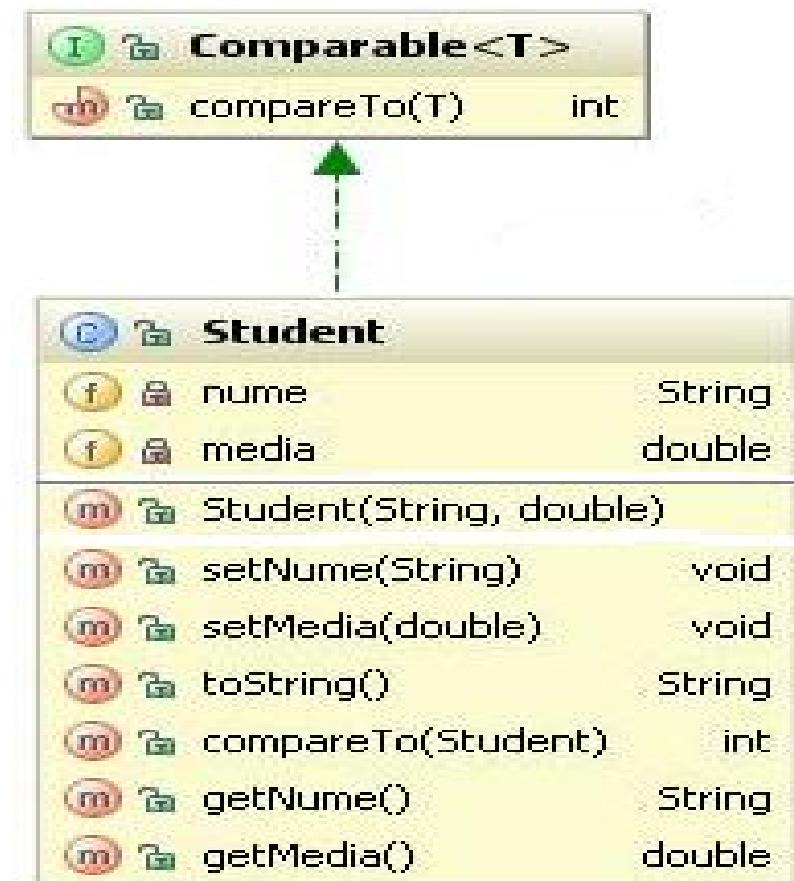
Vasilescu Maria|8.9

Popescu Ion|6.7

Marinescu Ana|9.6

Ionescu George|7.53

Pop Vasile|9.3



# Example DataOutput

```
void printStudentiDataOutput(List<Student> studs, String numefis) {  
    DataOutputStream output=null;  
    try{  
        output=new DataOutputStream(new FileOutputStream(numefis));  
        for(Student stud: studs){  
            output.writeUTF(stud.getNume());  
            output.writeDouble(stud.getMedia());  
        }  
    } catch (FileNotFoundException e) {  
        System.err.println("Eroare scriere DO "+e);  
    } catch (IOException e) {  
        System.err.println("Eroare scriere DO "+e);  
    }finally {  
        if (output!=null)  
            try {  
                output.close();  
            } catch (IOException e) {  
                System.err.println("Eroare scriere DO "+e);  
            }  
    }  
}
```

# Example DataInput

```
List<Student> citesteStudentiDataInput(String numefis) {  
    List<Student> studs=new ArrayList<Student>();  
    DataInputStream input=null;  
    try{  
        input=new DataInputStream(new FileInputStream(numefis));  
        while(true){  
            String nume=input.readUTF();  
            double media=input.readDouble();  
            studs.add(new Student(nume,media));  
        }  
    } catch(EOFException e) {}  
    catch (FileNotFoundException e) { System.err.println("Eroare citire"+e);}  
    catch (IOException e) { System.err.println("Eroare citire DI "+e);}  
    finally {  
        if (input!=null)  
            try { input.close();}  
            catch (IOException e) {  
                System.err.println("Eroare inchidere fisier "+e);  
            }  
    }  
    return studs;  
}
```

# Standard streams

- `System.in` of type `InputStream`
- `System.out` of type `PrintStream`
- `System.err` of type `PrintStream`

The associated streams can be modified using the methods:

`System.setIn()` , `System.setOut()` , `System.setErr()` ,

Example:

```
System.setOut(new PrintStream(new File("Output.txt")));
System.setErr(new PrintStream(new File("Erori.txt")));
```

# BufferedReader/BufferedWriter

- Use a buffer to keep the data which are going to be read/write from/to a stream.
- Read/Write operations are more efficient since the reading/writing is effectively done only when the buffer is empty/full.

## BufferedReader

```
+BufferedReader( reader : Reader )
+close()
+read() : int
+readLine() : String
+ready() : boolean
```

## BufferedWriter

```
+BufferedWriter( writer : Writer )
+newLine()
+flush()
+close()
+write( ... )
```

```
BufferedReader br=new BufferedReader(new FileReader(numefisier));
BufferedWriter bw=new BufferedWriter(new FileWriter(numefisier));
// rewrite the existing data in the file

//add at the end of the file
BufferedWriter bw=new BufferedWriter(new FileWriter(numefisier, true));
```

# Example BufferedReader

```
List<Student> citesteStudenti(String numefis) {
    List<Student> ls=new ArrayList<Student>();
    BufferedReader br=null;
    try{
        br=new BufferedReader(new FileReader(numefis));
        String linie;
        while((linie=br.readLine())!=null){
            String[] elems=linie.split("[|]");
            if (elems.length<2){
                System.err.println("Linie invalida "+linie);
                continue;
            }
            Student stud=new Student(elems[0], Double.parseDouble(elems[1]));
            ls.add(stud);
        }
    }catch (FileNotFoundException e) {System.err.println("Eroare citire "+e);}
    catch (IOException e) { System.err.println("Eroare citire "+e);}
    finally{
        if (br!=null)
            try { br.close(); }
            catch (IOException e) { System.err.println("Eroare inchidere fisier: "+e);
    }
}
return ls;
```

# Example BufferedWriter

```
void printStudentiBW(List<Student> studs, String numefis){  
    BufferedWriter bw=null;  
    try{  
        bw=new BufferedWriter(new FileWriter(numefis));  
        //bw=new BufferedWriter(new FileWriter(numefis,true));  
        for(Student stud: studs){  
            bw.write(stud.getNume()+'|'+stud.getMedia());  
            bw.newLine();      //scrie sfarsitul de linie  
        }  
    } catch (IOException e) {  
        System.err.println("Eroare scriere BW "+e);  
    } finally {  
        if (bw!=null)  
            try {  
                bw.close();  
            } catch (IOException e) {  
                System.err.println("Eroare inchidere fisier "+e);  
            }  
    }  
}
```

# PrintWriter

- Contains methods to save any type of data in text format.
- Contains methods to format the data .

PrintWriter
+PrintWriter( numefis : String )
+PrintWriter( numefisier : String, autoFlush : boolean )
+PrintWriter( writer : Writer )
+PrintWriter( ... )
+print( e : Yyy )
+println( e : Yyy )
+printf()
+format()
+flush()
+close()

- **y<sub>yy</sub>** is any primitive or reference type. If **y<sub>yy</sub>** is a reference type it is called the method **toString** corresponding to **e**.

# Example 1 PrintWriter

```
void printStudentiPrintWriter(List<Student> studs, String numefis) {  
    PrintWriter pw=null;  
    try{  
        pw=new PrintWriter(numefis);  
        for(Student stud: studs){  
            pw.println(stud.getNume()+' | '+stud.getMedia());  
        }  
  
    } catch (FileNotFoundException e) {  
        System.err.println("Eroare scriere PW "+e);  
    }finally {  
        if (pw!=null)  
            pw.close();  
    }  
}
```

# Example 2 PrintWriter

```
void printStudentiPWTabel(List<Student> studs, String numefis){  
    PrintWriter pw=null;  
    try{  
        pw=new PrintWriter(numefis);  
        String linie=getLinie('-',48);  
        int crt=0;  
        for(Student stud:studs){  
            pw.println(linie);  
//pw.printf(" | %3d | %-30s | %5.2f |%n", (++crt),stud.getNume(),stud.getMedia());  
            pw.format(" | %3d | %-30s | %5.2f |%n", (++crt),stud.getNume(),stud.getMedia());  
        }  
        if (crt>0)  
            pw.println(linie);  
    } catch (FileNotFoundException e) {  
        System.err.println("Eroare scriere PWTabel "+e);  
    } finally {  
        if (pw!=null)  
            pw.close();  
    }  
}
```

## Example 2 PrintWriter

```
String getLinie(char c, int length) {
    char[] tmp=new char[length];
    Arrays.fill(tmp,c);
    return String.valueOf(tmp);
}

//file result
-----
| 1 | Popescu Ion           | 6.70 |
-----
| 2 | Ionescu George        | 7.53 |
-----
| 3 | Vasilescu Maria       | 8.90 |
-----
| 4 | Pop Vasile             | 9.30 |
-----
| 5 | Marinescu Ana          | 9.60 |
```

# Reading from the keyboard

- Class `BufferedReader`

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

- Class `Scanner` (package `java.util`)

```
Scanner input=new Scanner(System.in);
```

- Class `Scanner` contains methods to read data of primitive types from the keyboard (or other stream):

- `nextInt():int`
- `nextDouble():double`
- `nextFloat():Float`
- `nextLine():String`
- ...
- `hasNextInt():boolean`
- `hasNextDouble():boolean`
- `hasNextFloat():boolean`
- ...

# Example BufferedReader (keyboard)

```
List<Student> citesteStudenti() {    //from the keyboard
    List<Student> ls=new ArrayList<Student>();
    BufferedReader br=null;
    try{
        br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("La terminare introduceti cuvantul \"gata\"");
        boolean gata=false;
        while(!gata){
            System.out.println("Introduceti numele: ");
            String snume=br.readLine();
            if ("gata".equalsIgnoreCase(snume)){gata=true; continue;}
            System.out.println("Introduceti media: ");
            String smedia=br.readLine();
            if ("gata".equalsIgnoreCase(smedia)){gata=true; continue;}
            try{
                double media=Double.parseDouble(smedia);
                lista.add(new Student(snume,media));
            }catch(NumberFormatException nfe){
                System.err.println("Eroare: "+nfe);
            }
        }
    } /*catch, finally, ...*/ } //citesteStudenti
```

# Example Scanner (keyboard)

```
List<Student> citesteStudentiScanner() {  
    List<Student> lista=new ArrayList<Student>();  
    Scanner scanner=null;  
    try{  
        scanner=new Scanner(System.in);  
        System.out.println("La terminare introduceti cuvantul \"gata\"");  
        boolean gata=false;  
        while(!gata){  
            System.out.println("Introduceti numele: ");  
            String snume=scanner.nextLine();  
            if ("gata".equalsIgnoreCase(snume)){gata=true; continue; }  
            System.out.println("Introduceti media: ");  
            if (scanner.hasNextDouble()) {  
                double media=scanner.nextDouble();  
                lista.add(new Student(snume,media));  
                scanner.nextLine(); //to read <Enter>  
                continue;  
            }else{  
                //next slide  
            }  
        }  
    }catch (Exception e){  
        e.printStackTrace();  
    }  
}
```

# Example Scanner (keyboard) cont.

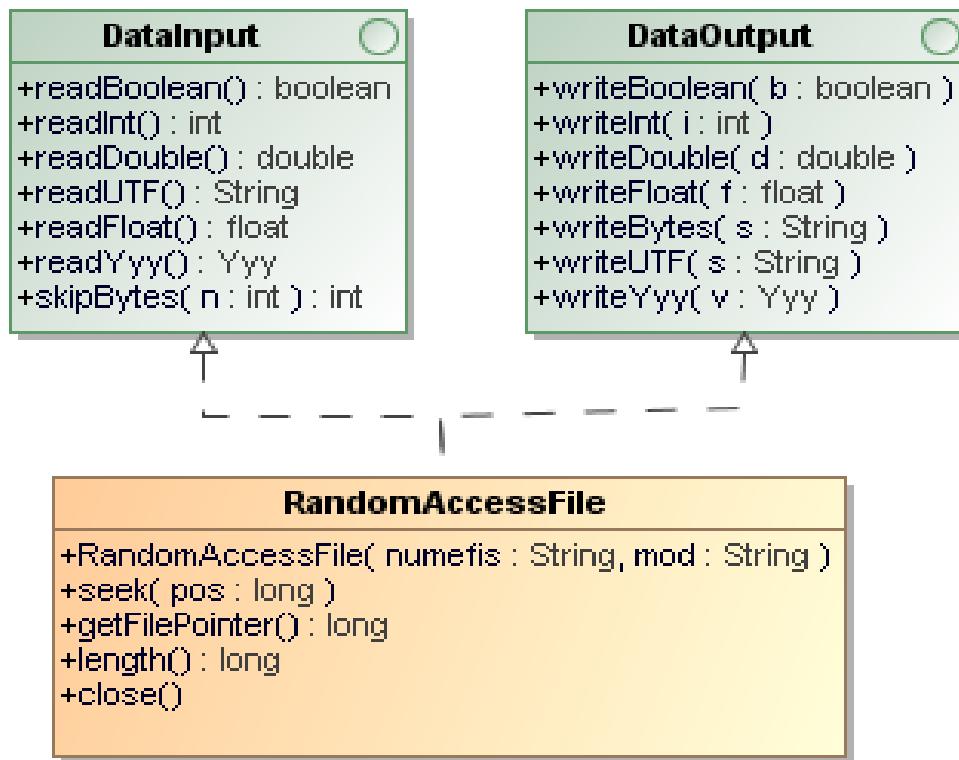
```
List<Student> citesteStudentiScanner() {  
    //...  
    if (scanner.hasNextDouble()) {  
        //...  
    } else {  
        String msj=scanner.nextLine();  
        if ("gata".equalsIgnoreCase(msj)) {  
            gata=true;  
            continue;  
        } else  
            System.out.println("Trebuie sa introduceti media studentului");  
    } //else  
} //while  
} finally {  
    if (scanner!=null)  
        scanner.close();  
}  
return lista;  
}
```

# Example Scanner file

```
Scanner inscan=null;  
try{  
    inscan=new Scanner(new BufferedReader(new FileReader("intregi.txt")));  
    //inscanf.useDelimiter(",");  
    while(inscanf.hasNextInt()){  
        int nr=inscanf.nextInt();  
        System.out.println("nr = " + nr);  
    }  
} catch (FileNotFoundException e) {  
    System.err.println("Eroare "+e);  
}finally {  
    if (inscanf!=null)  
        inscanf.close();  
}
```

# RandomAccessFile

- Allow random access to a file.
- Can be used for either reading or writing.
- Class uses the notion of cursor to denote the current position in the file. Initially the cursor is on the position 0 at the beginning of the file.
- Operations of reading/writing move the cursor according the number of bytes read/written.



# Example writing RandomAccessFile

```
void printStudentiRAF(List<Student> studs, String numefis){  
    RandomAccessFile out=null;  
    try{  
        out=new RandomAccessFile(numefis,"rw");  
        for(Student stud: studs){  
            out.writeUTF(stud.getNume());  
            out.writeDouble(stud.getMedia());  
        }  
    }catch (FileNotFoundException e){System.err.println("Eroare RAF "+e);}  
    catch (IOException e) { System.err.println("Eroare scriere RAF "+e);}  
    finally{  
        if (out!=null)  
            try {  
                out.close();  
            } catch (IOException e) {  
                System.err.println("Eroare inchidere fisier "+e);  
            }  
    }  
}
```

# Example reading RandomAccessFile

```
List<Student> citesteStudentiRAF(String numefis) {  
    List<Student> studs=new ArrayList<Student>();  
    RandomAccessFile in=null;  
    try{  
        in=new RandomAccessFile(numefis, "r");  
        while(true){  
            String nume=in.readUTF();  
            double media=in.readDouble();  
            studs.add(new Student(nume,media));  
        }  
    }catch(EOFException e){ }  
    catch (FileNotFoundException e){System.err.println("Eroare la citire: "+e);}  
    catch (IOException e) { System.err.println("Eroare la citire "+e);}  
    finally {  
        if (in!=null)  
            try {  
                in.close();  
            } catch (IOException e) {System.err.println("Eroare RAF "+e);}  
    }  
    return studs; }
```

# Example appending RandomAccessFile

```
void adaugaStudent(Student stud, String numefis){  
    RandomAccessFile out=null;  
    try{  
        out=new RandomAccessFile(numefis, "rw");  
        out.seek(out.length());  
        out.writeUTF(stud.getNume());  
        out.writeDouble(stud.getMedia());  
    } catch (FileNotFoundException e) {  
        System.err.println("Eroare RAF "+e);  
    } catch (IOException e) {  
        System.err.println("Eroare RAF "+e);  
    }finally {  
        if (out!=null)  
            try {  
                out.close();  
            } catch (IOException e) {  
                System.err.println("Eroare RAF "+e);  
            }  
    }  
}
```

# Class File

- Represent the name of a file (not its content).
- Allow platform-independent operations on the files (create, delete, rename, etc.) .

```
■File(name:String)      //name  is the path to a file or a directory  
■getName():String  
■getAbsolutePath():String  
■isFile():boolean  
■isDirectory():boolean  
■exists():boolean  
■delete():boolean  
■deleteOnExit()    //Directory/File is removed at the exit of JVM  
■mkdir()  
■list():String[]  
■list(filtru:FilenameFilter):String[]  
■...
```

# Class File: examples

## ■ Printing the current directory

```
File dirCurrent=new File(".");
System.out.println("Directory: " + dirCurrent.getAbsolutePath());
```

## ■ Creating an OS-independent path

```
String namefis=".." +File.separator+"data"+File.separator+"intregi.txt";
File f1=new File(namefis);
System.out.println("F1 "+f1.getName());
System.out.println("Exists f1? "+f1.exists());
```

## ■ Selecting the .txt files from a directory

```
File dir=new File(".");
String[] files=dir.list(new FilenameFilter(){
    public boolean accept(File dir, String name) {
        return name.toLowerCase().endsWith(".txt");
    }
});
System.out.println("Files "+ Arrays.toString(files));
```

## ■ Removing a file

```
File namef=new File("erori.txt");
if (namef.exists())
    boolean ok=namef.delete();
```

# Java NIO

# Java NIO (New IO)

- is an alternative IO API for Java (to the standard Java IO and Java Networking API's)
- consists of the following core components:
  - **Channels**
  - **Buffers**
  - **Selectors**

# Channels and Buffers

- In the standard IO API you work with byte streams and character streams.
- In NIO you work with channels and buffers. Data is always read from a channel into a buffer, or written from a buffer to a channel.
- all IO in NIO starts with a Channel.
-

# Channels and Buffers

- Channels are similar to streams with a few differences:
- You can both read and write to a Channels. Streams are typically one-way (read or write).
- Channels can be read and written asynchronously.
- Channels always read to, or write from, a Buffer.
-

# Channels and Buffers

the most important Channel implementations in Java NIO:

- FileChannel: reads data from and to files.
- DatagramChannel: can read and write data over the network via UDP.
- SocketChannel: can read and write data over the network via TCP.
- ServerSocketChannel: allows you to listen for incoming TCP connections, like a web server does.

# Channels and Buffers

- A buffer is essentially a block of memory into which you can write data, which you can then later read again.
- Buffer types let you work with the bytes in the buffer as char, short, int, long, float or double:
  - ByteBuffer
  - MappedByteBuffer
  - CharBuffer
  - DoubleBuffer
  - FloatBuffer
  - IntBuffer
  - LongBuffer
  - ShortBuffer

# Channels and Buffers

- Using a Buffer to read and write data typically follows this little 4-step process:
- **Write data into the Buffer:** The buffer keeps track of how much data you have written.
- **Call `buffer.flip()`:** in order to switch the buffer from writing mode into reading mode
- **Read data out of the Buffer:** In reading mode the buffer lets you read all the data written into the buffer.
- **Call `buffer.clear()` or `buffer.compact()`:** to make buffer ready for writing again

# Channels and Buffers

- The `clear()` method clears the whole buffer.
- The `compact()` method only clears the data which you have already read. Any unread data is moved to the beginning of the buffer, and data will now be written into the buffer after the unread data.

# Channels and Buffers

```
RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "rw");

FileChannel inChannel = aFile.getChannel();

//create buffer with capacity of 48 bytes

ByteBuffer buf = ByteBuffer.allocate(48);

int bytesRead = inChannel.read(buf); //read into buffer.

while (bytesRead != -1) {

    buf.flip(); //make buffer ready for read

    while(buf.hasRemaining()){

        System.out.print((char) buf.get()); // read 1 byte at a time

    }

    buf.clear(); //make buffer ready for writing

    bytesRead = inChannel.read(buf);

}

aFile.close();
```

# Channels and Buffers

- A Buffer has three properties:
  1. **Capacity**: a certain fixed size. Once the Buffer is full, you need to empty it (read the data, or clear it) before you can write more data into it.
  2. **Position**:
- **Write mode**: Initially the position is 0. When a byte, long etc. has been written into the Buffer the position is advanced to point to the next cell in the buffer to insert data into. Position can maximally become capacity – 1
- **Read mode**: When you flip a Buffer from writing mode to reading mode, the position is reset back to 0. As you read data from the Buffer you do so from position, and position is advanced to next position to read.

# Channels and Buffers

## 3. Limit:

- Write mode: is the limit of how much data you can write into the buffer and it is equal to the capacity of the Buffer
- Read mode: is the limit of how much data you can read from the data. Therefore, when flipping a Buffer into read mode, limit is set to write position of the write mode. In other words, you can read as many bytes as were written (limit is set to the number of bytes written, which is marked by position).

# Scattering Reads

- reads data from a single channel into multiple buffers

```
ByteBuffer buf1 = ByteBuffer.allocate(128);
ByteBuffer buf2 = ByteBuffer.allocate(1024);
ByteBuffer[] bufferArray = { buf1,buf2 };
channel.read(bufferArray);
```

# Gathering Writes

- writes data from multiple buffers into a single channel

```
ByteBuffer buf1 = ByteBuffer.allocate(128);
ByteBuffer buf2  = ByteBuffer.allocate(1024);
ByteBuffer[] bufferArray = { buf1,buf2  };
channel.write(bufferArray);
```

# Java NIO FileChannel

- is a channel that is connected to a file.
- you can read data from a file, and write data to a file.
- is an alternative to reading files with the standard Java IO API.

# Channel to Channel Transfer

- you can transfer data directly from one channel to another

```
RandomAccessFile fromFile = new RandomAccessFile("fromFile.txt", "rw");
```

```
FileChannel    fromChannel = fromFile.getChannel();
```

```
RandomAccessFile toFile = new RandomAccessFile("toFile.txt", "rw");
```

```
FileChannel    toChannel = toFile.getChannel();
```

```
long position = 0;
```

```
long count    = fromChannel.size();
```

```
toChannel.transferFrom(fromChannel, position, count);
```

# Java NIO Path

- an interface that is similar to the `java.io.File` class
- An instance represents a path in the file system. A path can point to either a file or a directory. A path can be absolute or relative.

# Java NIO Files

- provides several methods for manipulating files in the file system

# Java NIO AsynchronousFileChannel

- makes it possible to read data from, and write data to files asynchronously
- Read and write operations can be done either via a Future or via a CompletionHandler

# Reading via a Future

```
AsynchronousFileChannel fileChannel =
```

```
    AsynchronousFileChannel.open(path, StandardOpenOption.READ);
```

```
ByteBuffer buffer = ByteBuffer.allocate(1024);
```

```
long position = 0;
```

```
Future<Integer> operation = fileChannel.read(buffer, position);
```

```
while(!operation.isDone());
```

```
buffer.flip();
```

```
byte[] data = new byte[buffer.limit()];
```

```
buffer.get(data);
```

```
System.out.println(new String(data));
```

```
buffer.clear();
```

# Writing via a CompletionHandler

```
Path path = Paths.get("data/test-write.txt");
if(!Files.exists(path)){
    Files.createFile(path);
}
AsynchronousFileChannel fileChannel = AsynchronousFileChannel.open(path, StandardOpenOption.WRITE);
ByteBuffer buffer = ByteBuffer.allocate(1024);
long position = 0;
buffer.put("test data".getBytes());
buffer.flip();
fileChannel.write(buffer, position, buffer, new CompletionHandler<Integer, ByteBuffer>() {
    @Override
    public void completed(Integer result, ByteBuffer attachment) {
        System.out.println("bytes written: " + result);
    }
    @Override
    public void failed(Throwable exc, ByteBuffer attachment) {
        System.out.println("Write failed");
        exc.printStackTrace();
    }
});
```

# Selectors

- A Selector allows a single thread to handle multiple Channel's.
- is handy if your application has many Channels open
- To use a Selector you register the Channel's with it. Then you call it's select() method. This method will block until there is an event ready for one of the registered channels. Once the method returns, the thread can then process these events.  
Examples of events are incoming connection, data received etc.

# **Try-with-resources**

# Old Style

```
private static void printFile() throws IOException {  
    InputStream input = null;  
  
    try {  
        input = new FileInputStream("file.txt");  
  
        int data = input.read();  
  
        while(data != -1){  
            System.out.print((char) data);  
  
            data = input.read();  
        }  
  
    } finally {  
        if(input != null){  
            input.close();  
        }  
    }  
}
```

- The red marked code may throw exceptions

# Try-with-resources

- From Java 7 the previous code can be rewritten as follows:

```
private static void printFileJava7() throws IOException {
```

```
    try(FileInputStream input = new FileInputStream("file.txt")) {
```

```
        int data = input.read();
```

```
        while(data != -1){
```

```
            System.out.print((char) data);
```

```
            data = input.read();
```

```
}
```

```
}
```

```
}
```

# Try-with-resources

- When the try block finishes the FileInputStream will be closed automatically. This is possible because FileInputStream implements the Java interface java.lang.AutoCloseable.

```
public interface AutoClosable {
```

```
    public void close() throws Exception;
```

```
}
```

- All classes implementing this interface can be used inside the try-with-resources construct.

# **Advanced Programming Methods**

## **Lecture 6 - Java Serialization and Functional Programming**

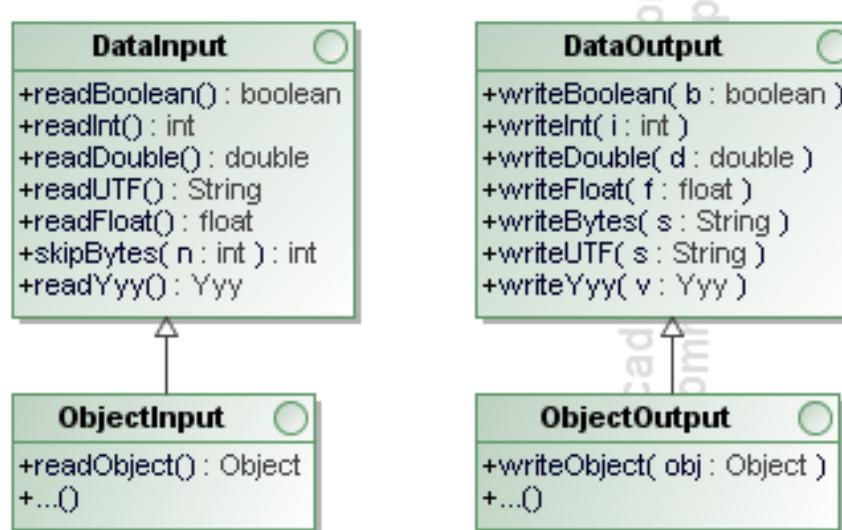
# Java Serialization

# Java Serialization

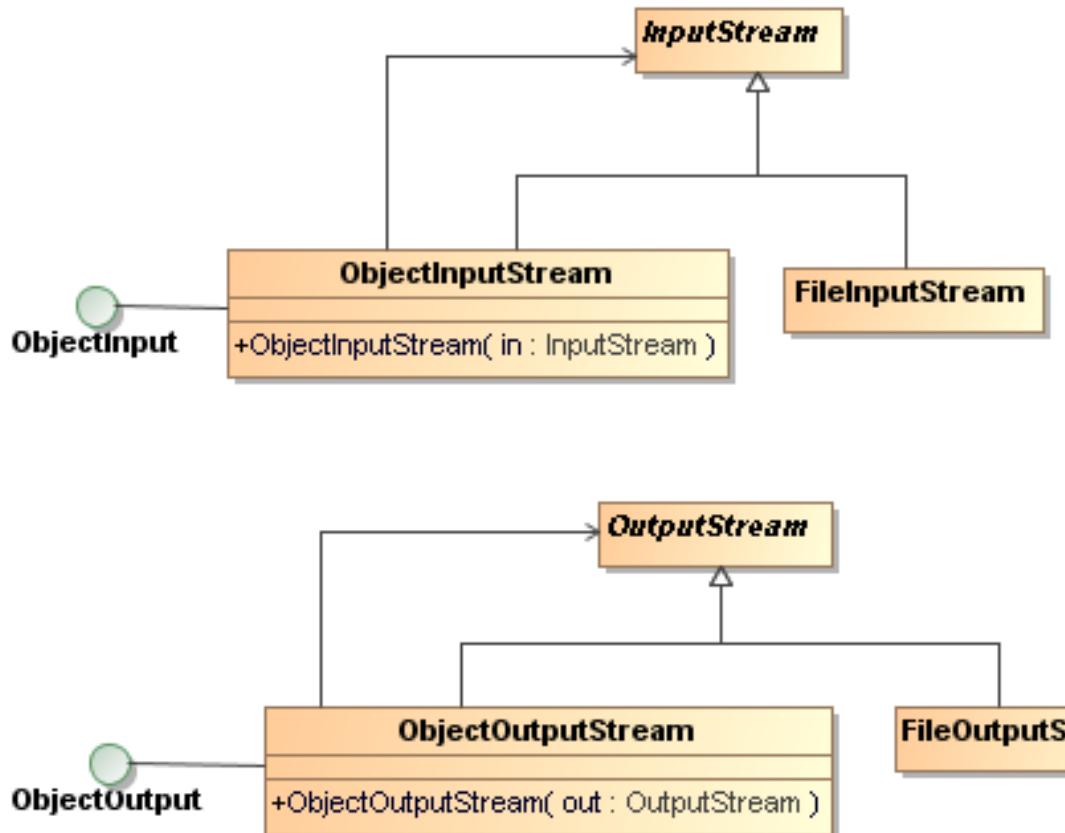
- serialization allows us to convert the state of an object into a byte stream, which then can be saved into a file on the local disk or sent over the network to any other machine.
- deserialization allows us to reverse the process, which means reconverting the serialized byte stream to an object again.

# Java Objects Serialization

- The process of writing/reading objects from/to a file/external support.
- An object is persistent (serializable ) if it can be written into a file/external support and can be read from a file/external support



# Objects Serialization



# Objects Serialization

```
void serializareObj(String numefis) {  
    ObjectOutputStream out=null;  
    try{  
        out=new ObjectOutputStream(new FileOutputStream(numefis));  
        out.writeObject(23);  
        out.writeObject("Vasilescu Ana");  
        out.writeObject(23.45f);  
    } catch (IOException e) {  
        System.err.println("Eroare "+e);  
    } finally {  
        if (out!=null)  
            try {  
                out.close();  
            } catch (IOException e) {  
                System.err.println("Eroare "+e);  
            }  
    }  
}
```

# Objects Serialization

```
void deserializareObj(String numefis) {  
    ObjectInputStream in=null;  
    try{  
        in=new ObjectInputStream(new FileInputStream(numefis));  
        Integer intreg=(Integer)in.readObject();  
        String text=(String)in.readObject();  
        Float nr=(Float)in.readObject();  
        System.out.println("Intreg: "+intreg+ " String: "+text+ " Float: "+nr);  
    } catch (IOException e) {System.err.println("Eroare "+e);}  
    catch (ClassNotFoundException e) {  
        System.err.println("Eroare deserializare "+e);  
    }finally {  
        if (in!=null){  
            try {  
                in.close();  
            } catch (IOException e) {System.err.println("Eroare "+e);}  
        }  
    }  
}
```

# Serializable Objects

- The classes whose objects are serializable must be declared to implement the interface `Serializable` (package `java.io`).
- Interface `Serializable` does not contain any method.

```
class Student implements Comparable<Student>, Serializable{  
    //...  
}  
  
class Test{  
    public static void main(String[] args) {  
        ObjectOutputStream out=  
            //... initialization  
        Student stud=new Student("Popescu Ioan", 7.9);  
        out.writeObject(stud);  
        //...  
    }  
}
```

- The state of `stud` (the values of its fields) is saved into the file.

# Serializable objects

- All the reachable objects (the objects that can be reach using the references) are saved into the file only once.

```
class CircularList implements Serializable{  
    private class Node implements Serializable{  
        Node urm;  
        //...  
    }  
    private Node head; //last node of the list refers to the head of the list  
    //...  
}
```

- The objects which are referred by a serializable object must be also serializable.

Obs:

Static attributes of a serializable class are not saved into the file/external support.

# Example serializable objects

```
void printSerializabil(List<Student> studs, String numefis) {  
    ObjectOutputStream out=null;  
    try{  
        out=new ObjectOutputStream(new FileOutputStream(numefis));  
        out.writeObject(studs);  
    } catch (IOException e) {  
        System.err.println("Eroare serializare "+e );  
    } finally {  
        if (out!=null)  
            try {  
                out.close();  
            } catch (IOException e) {  
                System.err.println("Eroare "+e );  
            }  
    }  
}
```

# Example serializable objects

```
@SuppressWarnings ("unchecked")
List<Student> citesteSerializabil(String numefis){
    List<Student> rez=null;
    ObjectInputStream in=null;
    try{
        in=new ObjectInputStream(new FileInputStream(numefis));
        rez=(List<Student>) in.readObject();
    } catch (IOException e) {
        System.err.println("Eroare deserializare"+e);
    } catch (ClassNotFoundException e) {
        System.err.println("Eroare deserializare "+e);
    }finally{
        if (in!=null)
            try {
                in.close();
            }catch (IOException e) {System.err.println("Eroare "+e); }
    }
    return rez;
}
```

# Objects Serialization

## ■ Method `in.readObject() : Object`

1. Read the object from the stream
2. Identify the object type
3. Initialize the non-static members of the object byte by byte (without a constructor call) and then return the new created object

## ■ Method `out.writeObject (Object)`

- Save the non-static members and the information required by JVM to rebuild the object
- an object (from a given reference ) is saved only once on a stream:

```
ObjectOutputStream out=...  
out.writeObject(new Produs ("A")) ;  
Produs produs2=new Produs ("B") ;  
out.writeObject(produs2) ;  
produs2.setNume ("BB") ;  
out.writeObject(produs2) ;  
//...  
out.close();
```

```
ObjectInputStream in=...  
Produs p1=(Produs)in.readObject () ;  
Produs p2=(Produs)in.readObject () ;  
Produs p3=(Produs)in.readObject () ;  
//...
```

# Objects Serialization - serialVersionUID

```
public class Student implements Serializable{
    private String nume;
    private double media;
    //...
}
```

Scenario :

- 1.The objects of class Student are serialized.
- 2.The class Student is changed (add/remove fields/methods).
- 3.We want to de-serialize the saved objects.

```
public class Student implements Serializable{
    [any modif access] static final long serialVersionUID = 1L;
    private String nume;
    private double media;
    private int grupa;
    //...
}
```

New added fields are initialized with the default values corresponding to their types.

# Objects Serialization - transient

- There are situations when we do not want to save the values of some fields (e.g. passwords, file descriptors, etc.)
- Those fields are declared using the keyword `transient`:

```
public class Student implements Serializable{  
    private String nume;  
    private double media;  
    private transient String parola;  
    //...  
}
```

At reading, the transient fields are initialized with the default values corresponding to their types.

# Serializable data structures

```
public class Stack implements Serializable{  
    private class Node implements Serializable{  
        //...  
    }  
    private Node top;  
    //...  
}  
//...  
Stack s=new Stack();  
s.push("ana");  
s.push(new Produs("Paine", 2.3));  
    //class Produs must be serializable  
//...  
ObjectOutputStream out=...  
out.writeObject(s);
```

# Java Functional Programming

# Overview

1. Anonymous inner classes in Java
2. Lambda expressions
3. Processing Data with Java Streams

Note: Lecture notes are based on Oracle tutorials.

# Anonymous Inner classes

- provide a way to implement classes that may occur only once in an application.

```
JButton testButton = new JButton("Test Button");
testButton.addActionListener(new ActionListener(){
    @Override public void actionPerformed(ActionEvent ae){
        System.out.println("Click Detected by Anon Class");
    }
});
```

# Functional Interfaces

- are interfaces with only one method
- Using functional interfaces with anonymous inner classes are a common pattern in Java

```
public interface ActionListener extends EventListener {  
    public void actionPerformed(ActionEvent e);  
}
```

# Lambda Expressions

- are Java's first step into functional programming
- can be created without belonging to any class
- can be passed around as if they were objects and executed on demand.

(int x, int y) -> x + y

() -> 42

(String s) -> { System.out.println(s); }

```
testButton.addActionListener(e -> System.out.println("Click Detected by  
Lambda Listener"));
```

# Lambda Expressions

- Lambda function body

```
(oldState, newState) -> System.out.println("State changed")
```

```
(oldState, newState) -> {  
    System.out.println("Old state: " + oldState);  
    System.out.println("New state: " + newState);  
}
```

- Returning a value

```
(param) -> {System.out.println("param: " + param); return "return value";} 
```

```
(a1, a2) -> { return a1 > a2; }
```

```
(a1, a2) -> a1 > a2;
```

# Lambdas as Objects

- A Java lambda expression is essentially an object.
- You can assign a lambda expression to a variable and pass it around, like you do with any other object.

```
public interface MyComparator {  
    public boolean compare(int a1, int a2);  
}
```

```
MyComparator myComparator = (a1, a2) -> return a1 > a2;  
boolean result = myComparator.compare(2, 5);
```

# Runnable Lambda

```
// Anonymous Runnable  
  
Runnable r1 = new Runnable(){  
    @Override  
    public void run(){ System.out.println("Hello world one!"); } };  
  
  
// Lambda Runnable  
  
Runnable r2 = () -> System.out.println("Hello world two!");  
  
  
// Run them!  
  
r1.run();  
  
r2.run();
```

# Comparator Lambda

```
List<Person> personList = Person.createShortList();

// Sort with Inner Class

Collections.sort(personList, new Comparator<Person>(){

    public int compare(Person p1, Person p2){

        return p1.getSurName().compareTo(p2.getSurName());

    }});
}

// Use Lambda instead

Collections.sort(personList, (Person p1, Person p2) →
    p1.getSurName().compareTo(p2.getSurName()));

Collections.sort(personList, (p1, p2) ->
    p2.getSurName().compareTo(p1.getSurName()));
```

# Lambda Expressions

- can improve your code
- provide a means to better support the Don't Repeat Yourself (DRY) principle
- make your code simpler and more readable.
- **Motivational example:** Given a list of people, various criteria are used to send messages to matching persons:
  - Drivers(persons over the age of 16) get phone calls
  - Draftees(male persons between the ages of 18 and 25) get emails
  - Pilots(persons between the ages of 23 and 65) get mails

# First Attempt

```
public class RoboContactMethods {  
    public void callDrivers(List<Person> pl){  
        for(Person p:pl){  
            if (p.getAge() >= 16){ roboCall(p);}  
        } }  
  
    public void emailDraftees(List<Person> pl){  
        for(Person p:pl){  
            if (p.getAge() >= 18 && p.getAge() <= 25 && p.getGender() == Gender.MALE){  
                roboEmail(p);  
            } }}  
  
    public void mailPilots(List<Person> pl){  
        for(Person p:pl){  
            if (p.getAge() >= 23 && p.getAge() <= 65){      roboMail(p); }  
        } }  
.....}
```

# First Attempt

- The DRY principle is not followed.
- Each method repeats a looping mechanism.
- The selection criteria must be rewritten for each method
- A large number of methods are required to implement each use case.
- The code is inflexible. If the search criteria changed, it would require a number of code changes for an update. Thus, the code is not very maintainable.

# Second Attempt

```
public class RoboContactMethods2 {  
  
    public void callDrivers(List<Person> pl){  
  
        for(Person p:pl){  
  
            if (isDriver(p)){ roboCall(p);} } }  
  
    public void emailDraftees(List<Person> pl){  
  
        for(Person p:pl){  
  
            if (isDraftee(p)){      roboEmail(p);} } }  
  
    public void mailPilots(List<Person> pl){  
  
        for(Person p:pl){  
  
            if (isPilot(p)){ roboMail(p);} } }  
  
    public boolean isDriver(Person p){ return p.getAge() >= 16; }  
  
    public boolean isDraftee(Person p){  
  
        return p.getAge() >= 18 && p.getAge() <= 25 && p.getGender() == Gender.MALE; }  
  
    public boolean isPilot(Person p){ return p.getAge() >= 23 && p.getAge() <= 65; }  
}
```

# Third Attempt

- Using a functional interface and anonymous inner classes

```
public interface Predicate<T> {  
    public boolean test(T t);  
}
```

```
public void phoneContacts(List<Person> pl, Predicate<Person> aTest){  
    for(Person p:pl){  
        if (aTest.test(p)){    roboCall(p); }  
    } }
```

```
robo.phoneContacts(pl, new Predicate<Person>(){  
    @Override  
    public boolean test(Person p){  
        return p.getAge() >=16; } } );
```

# Fourth Attempt

- Using lambda expressions

```
public void phoneContacts(List<Person> pl, Predicate<Person> pred){  
    for(Person p:pl){  
        if (pred.test(p)){    roboCall(p); }  
    } }
```

```
Predicate<Person> allDrivers = p -> p.getAge() >= 16;
```

```
Predicate<Person> allDraftees = p -> p.getAge() >= 18 && p.getAge() <= 25 &&  
    p.getGender() == Gender.MALE;
```

```
Predicate<Person> allPilots = p -> p.getAge() >= 23 && p.getAge() <= 65;
```

```
robo.phoneContacts(pl, allDrivers);
```

# java.util.function

- standard interfaces are designed as a starter set for developers:
- **Predicate**: A property of the object passed as argument
- **Consumer**: An action to be performed with the object passed as argument
- **Function**: Transform a T to a R
- **Supplier**: Provide an instance of a T (such as a factory)
- **UnaryOperator**: A unary operator from T -> T
- **BinaryOperator**: A binary operator from (T, T) -> T

# Function Interface

- It has only one method **apply** with the following signature:

**public R apply(T t)**

- Example for class Person:

```
public String printCustom(Function <Person, String> f){  
    return f.apply(this);}
```

```
Function<Person, String> westernStyle = p -> {return "\nName: " +  
    p.getGivenName() + " " + p.getSurName() + "\n"};
```

```
Function<Person, String> easternStyle = p -> "\nName: " + p.getSurName() + " " +  
    p.getGivenName() + "\n";
```

```
person.printCustom(westernStyle);
```

```
person.printCustom(easternStyle);
```

```
person.printCustom(p -> "Name: " + p.getGivenName() + " EMail: " + p.getEmail());
```

# Java Functional Streams

- is a new addition to the Java Collections API, which brings a new way to process collections of objects.
- declarative way
- Stream: a sequence of elements from a source that supports aggregate operations.

```
List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");  
myList.stream()  
.filter(s -> s.startsWith("c"))  
.map(String::toUpperCase)  
.sorted()  
.forEach(System.out::println);
```

- Output:

C1

C2

# Java Functional Streams

- **Sequence of elements:** A stream provides an interface to a sequenced set of values of a specific element type. However, streams don't actually store elements; they are computed on demand.
- **Source:** Streams consume from a data-providing source such as collections, arrays, or I/O resources.
- **Aggregate operations:** Streams support SQL-like operations and common operations from functional programming languages, such as filter, map, reduce, find, match, sorted, and so on.

# Streams vs Collections

- collections are about data
- **streams are about computations.**
- A collection is an in-memory data structure, which holds all the values that the data structure currently has—every element in the collection has to be computed before it can be added to the collection.
- In contrast, a stream is a conceptually fixed data structure in which elements are computed on demand.

# Streams vs Collections

Two fundamental characteristics that make stream operations very different from collection operations:

- **Pipelining:** Many stream operations return a stream themselves. This allows operations to be chained to form a larger pipeline. This enables certain optimizations, such as laziness and short-circuiting
- **Internal iteration:** In contrast to collections, which are iterated explicitly (external iteration), stream operations do the iteration behind the scenes for you.

# Obtaining a Stream From a Collection

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

```
items.add("one");
```

```
items.add("two");
```

```
items.add("three");
```

```
Stream<String> stream = items.stream();
```

- is similar to how you obtain an Iterator by calling the items.iterator() method, but a Stream is different than an Iterator.

# Stream Processing Phases

## 1. Configuration-- intermediate operations:

- filters, mappings
- can be connected together to form a pipeline
- return a stream
- Are lazy: do not perform any processing

## 2. Processing—terminal operations:

- operations that close a stream pipeline
- produce a result from a pipeline such as a List, an Integer, or even void (any non-Stream type).

# Filtering

- **stream.filter( item -> item.startsWith("o") );**
- **filter(Predicate):** Takes a predicate (java.util.function.Predicate) as an argument and returns a stream including all elements that match the given predicate
- **distinct:** Returns a stream with unique elements (according to the implementation of equals for a stream element)
- **limit(n):** Returns a stream that is no longer than the given size n
- **skip(n):** Returns a stream with the first n number of elements discarded

# Filtering

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
```

```
List<Integer> twoEvenSquares =
```

```
    numbers.stream()
```

```
        .filter(n -> {System.out.println("filtering " + n); return n % 2 == 0;})
```

```
        .map(n -> { System.out.println("mapping " + n); return n * n;})
```

```
        .limit(2)
```

```
        .collect(toList());
```

filtering 1

filtering 2

mapping 2

filtering 3

filtering 4

mapping 4

- `limit(2)` uses short-circuiting; we need to process only part of the stream, not all of it, to return a result.

# Mapping

- Streams support the method map, which takes a function (java.util.function.Function) as an argument to project the elements of a stream into another form. The function is applied to each element, “mapping” it into a new element.

`items.stream()`

`.map( item -> item.toUpperCase() )`

- maps all strings in the items collection to their uppercase equivalents.

**NOTE:** it doesn't actually perform the mapping. It only configures the stream for mapping. Once one of the stream processing methods are invoked, the mapping (and filtering) will be performed.

# Mapping

```
List<String> words = Arrays.asList("Oracle", "Java",  
"Magazine");
```

```
List<Integer> wordLengths =  
words.stream()  
.map(String::length)  
.collect(toList());
```

# Stream.collect()

- is an extremely useful terminal operation to transform the elements of the stream into a different kind of result, e.g. a List, Set or Map .
- Collect accepts a Collector which consists of four different operations: a *supplier*, an *accumulator*, a *combiner* and a *finisher*.
- Java supports various builtin collectors via the Collectors class. So for the most common operations you don't have to implement a collector yourself.

# `Stream.collect()`

```
List<Person> filtered =  
persons  
.stream()  
.filter(p -> p.name.startsWith("P"))  
.collect(Collectors.toList());
```

```
Double averageAge =  
persons  
.stream()  
.collect(Collectors.averagingInt(p -> p.age));
```

# Stream.collect()

```
String phrase =
```

```
persons
```

```
.stream()
```

```
.filter(p -> p.age >= 18)
```

```
.map(p -> p.name)
```

```
.collect(Collectors.joining(" and ", "In Germany ", " are of legal  
age."));
```

- The join collector accepts a delimiter as well as an optional prefix and suffix.

# Stream.collect()

- In order to transform the stream elements into a map, we have to specify how both the keys and the values should be mapped.
- the mapped keys must be unique, otherwise an IllegalStateException is thrown.
- You can optionally pass a merge function as an additional parameter to bypass the exception:

```
Map<Integer, String> map = persons
```

```
.stream()
```

```
.collect(Collectors.toMap(
```

```
    p -> p.age,
```

```
    p -> p.name,
```

```
    (name1, name2) -> name1 + ";" + name2));
```

# Stream.min() and Stream.max()

- Are terminal operations
- return an Optional instance which has a get() method on, which you use to obtain the value. In case the stream has no elements the get() method will return null
- take a Comparator as parameter. The Comparator.comparing() method creates a Comparator based on the lambda expression passed to it. In fact, the comparing() method takes a Function which is a functional interface suited for lambda expressions

**String shortest = items.stream()**

```
.min(Comparator.comparing(item -> item.length()))  
.get();
```

# Stream.min() and Stream.max()

- The Optional<T> class (java.util .Optional) is a container class to represent the existence or absence of a value
- we can choose to apply an operation on the optional object by using the ifPresent method

**Stream.of("a1", "a2", "a3")**

**.map(s -> s.substring(1))**

**.mapToInt(Integer::parseInt)**

**.max()**

**.ifPresent(System.out::println);**

- Stream.of() creates a stream from a bunch of object references

# Stream.count()

- Returns the number of elements in the stream

```
long count = items.stream()  
    .filter( item -> item.startsWith("t"))  
    .count();
```

# **Stream.reduce()**

- can reduce the elements of a stream to a single value
- takes a BinaryOperator as parameter, which can easily be implemented using a lambda expression.
- Returns an Optional
- The BinaryOperator.apply() method:
- takes two parameters. The acc which is the accumulated value, and item which is an element from the stream.

**String reduced2 = items.stream()**

```
.reduce((acc, item) -> acc + " " + item)  
.get();
```

# Stream.reduce()

- There is another reduce() method which takes two parameters: an initial value for the accumulated value, and then a BinaryOperator.

```
String reduced = items.stream()
```

```
.filter( item -> item.startsWith("o"))
```

```
.reduce("", (acc, item) -> acc + " " + item);
```

# Stream.reduce()

```
int sum = 0;
```

```
for (int x : numbers) {
```

```
    sum += x;
```

```
}
```

```
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```

```
int product = numbers.stream().reduce(1, (a, b) -> a * b);
```

```
int max = numbers.stream().reduce(1, Integer::max);
```

# Numerical Streams

- `IntStream`, `DoubleStream`, and `LongStream`—that respectively specialize the elements of a stream to be `int`, `double`, and `long`.
- to convert a stream to a specialized version: `mapToInt`, `mapToDouble`, and `mapToLong`.
- to help generate ranges: `range` and `rangeClosed`.

```
IntStream oddNumbers =
```

```
    IntStream.rangeClosed(10, 30)
```

```
        .filter(n -> n % 2 == 1);
```

# Building Streams

- `InStream<Integer> numbersFromValues = Stream.of(1, 2, 3, 4);`
- `int[] numbers = {1, 2, 3, 4};`
- `IntStream numbersFromArray = Arrays.stream(numbers);`
- Converting a file into a stream of lines:
  - `long numberOfLines =`
  - `Files.lines(Paths.get("yourFile.txt"), Charset.defaultCharset())`
  - `.count();`

# Infinite Streams

- There are two static methods—Stream.iterate and Stream.generate—that let you create a stream from a function.
- because elements are calculated on demand, these two operations can produce elements “forever.”
- **Stream<Integer> numbers = Stream.iterate(0, n -> n + 10);**
- The iterate method takes an initial value (here, 0) and a lambda (of type UnaryOperator<T>) to apply successively on each new value produced.

# Infinite Streams

- We can turn an infinite stream into a fixed-size stream using the limit operation:
- **numbers.limit(5).forEach(System.out::println);**  
// 0, 10, 20, 30, 40.

# Finding and Matching

- A common data processing pattern is determining whether some elements match a given property. You can use the **anyMatch**, **allMatch**, and **noneMatch** operations to help you do this. They all take a predicate as an argument and return a boolean as the result (they are, therefore, terminal operations)
- Stream interface provides the operations **findFirst** and **findAny** for retrieving arbitrary elements from a stream. Both `findFirst` and `findAny` return an `Optional` object

# Processing Order

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .map(s -> {System.out.println("map: " + s);return s.toUpperCase();})
    .filter(s -> {System.out.println("filter: " + s);return s.startsWith("A");})
    .forEach(s -> System.out.println("forEach: " + s));
```

```
// map: d2
// filter: D2
// map: a2
// filter: A2
// forEach: A2
// map: b1
// filter: B1
// map: b3
// filter: B3
// map: c
// filter: C
```

# Processing Order

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {System.out.println("filter: " + s);return s.startsWith("a");})
    .map(s -> {System.out.println("map: " + s);return s.toUpperCase();})
    .forEach(s -> System.out.println("forEach: " + s));
```

// filter: d2

// filter: a2

// map: a2

// forEach: A2

// filter: b1

// filter: b3

// filter: c

# Reusing Streams

- Java functional streams cannot be reused. As soon as you call any terminal operation the stream is closed

```
Stream<String> stream =
```

```
Stream.of("d2", "a2", "b1", "b3", "c")
```

```
.filter(s -> s.startsWith("a"));
```

```
stream.anyMatch(s -> true); // ok
```

```
stream.noneMatch(s -> true);
```

*// exception since stream has been consumed*

# Reusing Streams

```
Supplier<Stream<String>> streamSupplier =  
() -> Stream.of("d2", "a2", "b1", "b3", "c")  
.filter(s -> s.startsWith("a"));
```

```
streamSupplier.get().anyMatch(s -> true); // ok  
streamSupplier.get().noneMatch(s -> true); // ok
```

- *Each call to get() constructs a new stream on which we can call the desired terminal operation.*

# **Advanced Programming Methods**

**Lecture 7 - Java Concurrency(1)**

# Overview

1. Introduction into Concurrency. What is a thread
2. Define and launch a thread
3. The life-cycle of a thread
4. Interrupt a thread
5. Thread synchronization
6. Other issues

# References

**NOTE: The slides are based on the following free tutorials. You may want to consult them too.**

1. <http://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/>
2. <http://tutorials.jenkov.com/java-util-concurrent/index.html>
3. <http://www.javacodegeeks.com/2015/09/java-concurrency-essentials.html>
4. Oracle tutorials

# Concurrent Programming

- there are two basic units of execution: **processes and threads**
- a computer system normally has many active processes and threads –even for only one execution core
- processing time for a single core is shared among processes and threads through an OS feature called **time slicing**.
- is common for computer systems to have **multiple processors** or processors with **multiple execution cores**

# Processes

- a process has a self-contained execution environment.
- in general it has a complete, private set of basic run-time resources; for example, each process has its **own memory space**.
- most implementations of the Java virtual machine run as a single process

# Processes

- processes are fully isolated from each other
- to facilitate communication between processes, most operating systems support **Inter Process Communication (IPC) resources**, such as **pipes** and **sockets**. (for communication between processes either on the same system or on different systems)

# Threads

- are called lightweight processes
- creating a new thread requires fewer resources than creating a new process.
- threads exist within a process — every process has at least one.
- threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

# Threads

- Threads are the units that are scheduled by the system for executing on the processor
- On a single processor, each thread has its turn by multiplexing based on time
- On a multiple processor, each thread is running at the same time with each processor/core running a particular thread.

# Threads

- a thread is a particular execution path of a process.
- one allows multiple threads to read and write the same memory (no process can directly access the memory of another process).
- when one thread modifies a process resource, the change is immediately visible to sibling threads.

# Advantages of Multi-threading

- faster on a multi-CPU system
- even in a single CPU system, application can remain responsive by using worker thread runs concurrently with the main thread

# Cost of Multi-threading

- program overhead and additional complexity
- there are time and resource costs in both creating and destroying threads
- the time required for scheduling threads, loading them onto the process, and storing their states after each time slice is pure overhead.

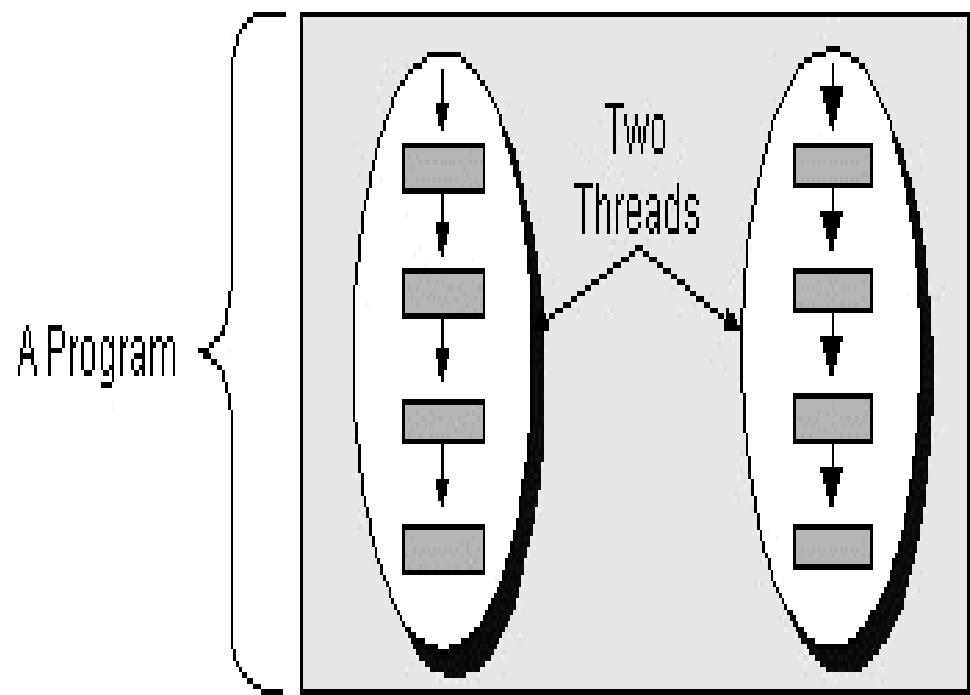
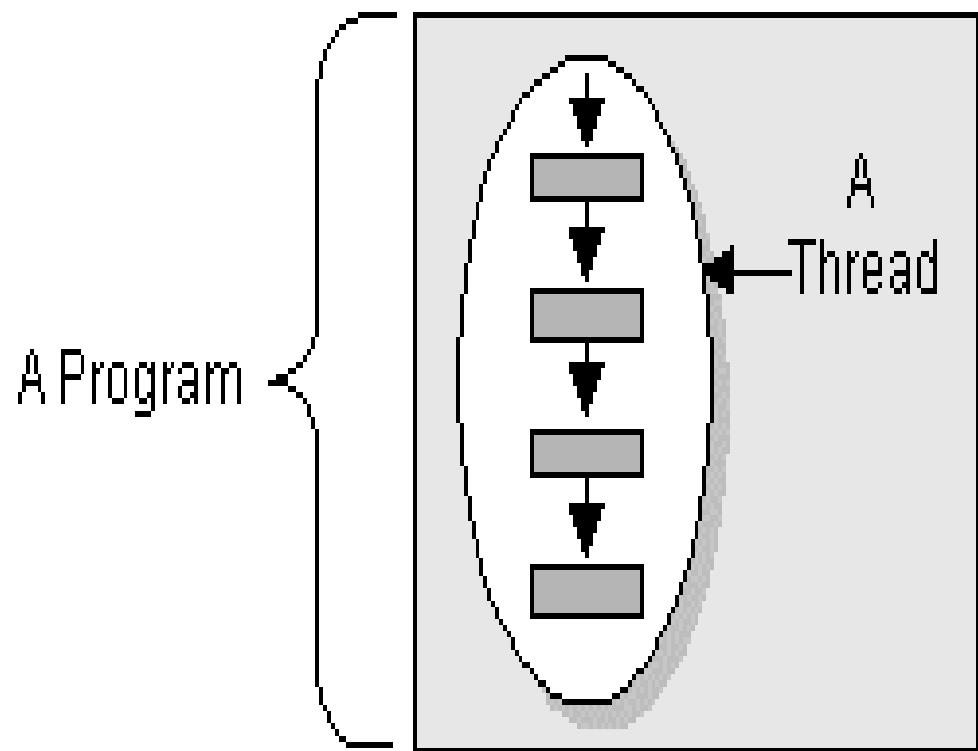
# Cost of Multi-threading

- **biggest cost:** Since the threads in a process all share the same resources and heap, it adds additional programming complexity to ensure that they are not ruining each other's work.
- debugging multithreaded programs can be quite difficult: the timing on each run of the program can be different; reproducing the same scheduling results is difficult

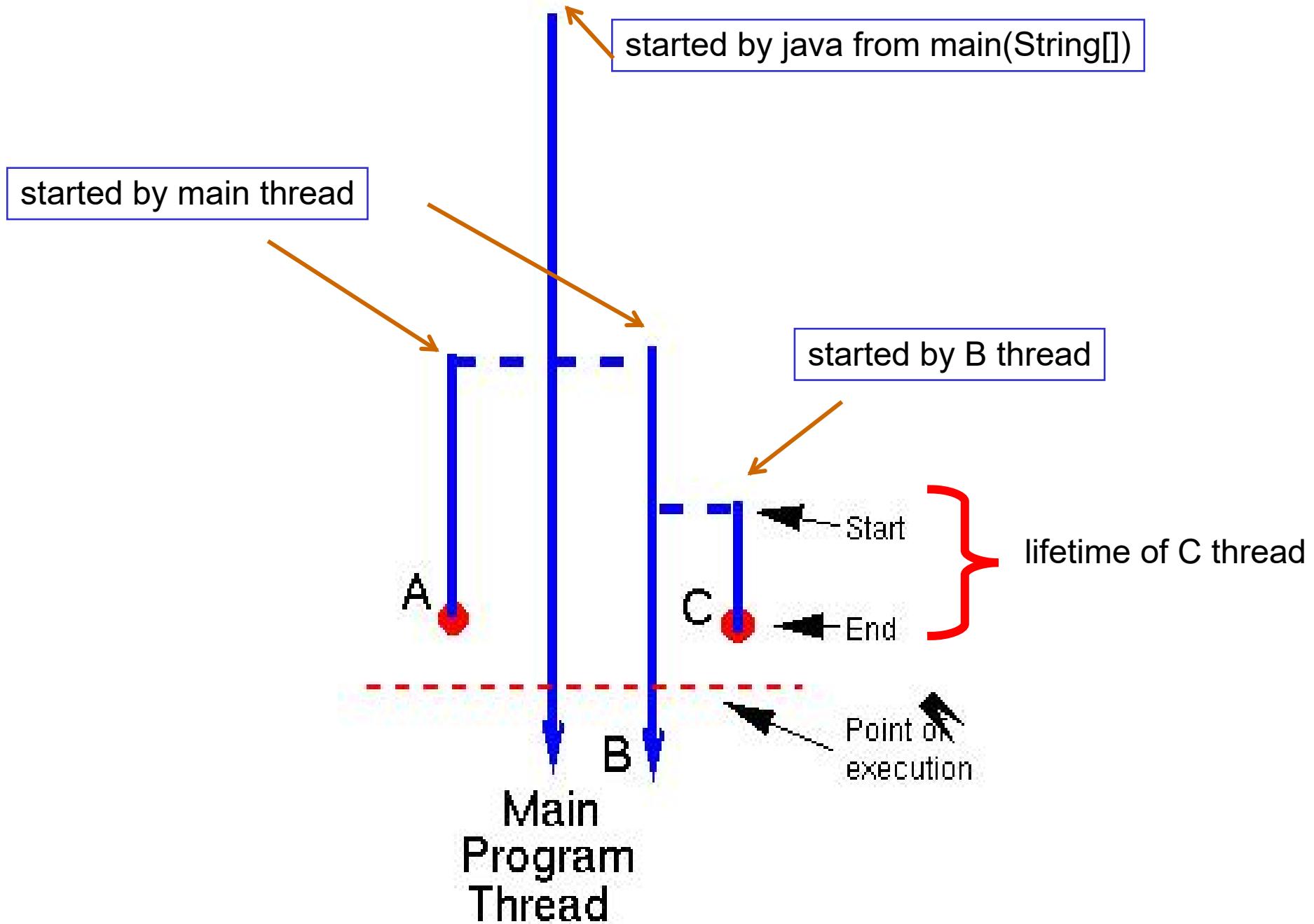
## What is a thread ?

- A sequential (or single-threaded) program is one that, when executed, **has only one single flow of control**.
  - i.e., at any time instant, there is at most only one instruction (or statement or execution point) that is being executed in the program.
- A **multi-threaded program** is one that can have **multiple flows of control** when executed.
  - At some time instance, there may exist **multiple instructions (or execution points) that are being executed** in the program
  - Ex: in a Web browser we may do the following tasks at the same time:
    - 1. scroll a page,
    - 2. download an applet or image,
    - 3. play sound,
    - 4. print a page.
- A thread is **a single sequential flow of control** within a program.

## single-threaded vs multithreaded programs



## Thread ecology in a java program



## 2. Define and launch a java thread

- Each Java Run time thread is encapsulated in a `java.lang.Thread` instance.
- Two ways to define a thread:
  - 1. Extend the Thread class
  - 2. Implement the Runnable interface :
    - `package java.lang;`
    - `public interface Runnable { public void run() ; }`
- Steps for extending the Thread class:
  - Subclass the Thread class;
  - Override the default Thread method `run()`, which is the entry point of the thread, like the `main(String[])` method in a java program.

## Define a thread

// Example:

```
public class Print2Console extends Thread {  
    public void run() { // run() is to a thread what main() is to a java program  
        for (int b = -128; b < 128; b++) out.println(b); }  
    ... // additional methods, fields ...  
}
```

Implement the Runnable interface if you need a parent class:

```
// by extending JTextArea we can reuse all existing code of JTextArea  
public class Print2GUI extend JTextArea implement Runnable {  
    public void run() {  
        for (int b = -128; b < 128; b++) append( Integer.toString(b) + "\n" ); }  
}
```

## How to launch a thread

1.create an instance of [ a subclass of ] Thread, say **thread**.

–**Thread thread = new Print2Console();**

–**Thread thread = new Thread( new Print2GUI( .. ) );**

2.2. call its **start()** method, **thread.start();**. // note: not call **run()** !!

3.Ex:

–**Printer2Console t1 = new Print2Console(); // t1 is a thread instance !**

–**t1.start() ; // this will start a new thread, which begins its execution by calling t1.run()**

–... // parent thread continue immediately here without waiting for the child thread to complete its execution. cf: **t1.run();**

–**Print2GUI jtext = new Print2GUI();**

–**Thread t2 = new Thread( jtext);**

–**t2.start();**

–...

## The java.lang.Thread constructors

// Public Constructors

**Thread([ ThreadGroup group,] [ Runnable target, ]  
[ String name ] );**

Instances :

**Thread();**

**Thread(Runnable target);**

**Thread(Runnable target, String name);**

**Thread(String name);**

**Thread(ThreadGroup group, Runnable target);**

**Thread(ThreadGroup group, Runnable target, String name);**

**Thread(ThreadGroup group, String name);**

**// name is a string used to identify the thread instance**

**// group is the thread group to which this thread belongs.**

## Some thread property access methods

- **int getId() // every thread has a unique ID**
- **String getName(); setName(String)**
  - // get/set the name of the thread
- **ThreadGroup getThreadGroup();**
- **int getPriority() ; setPriority(int) // thread has priority in [0, 31]**
- **Thread.State getState() // return current state of this thread**
  
- **boolean isAlive()**
  - Tests if this thread has been started and has not yet died. .
- **boolean isDaemon()**
  - Tests if this thread is a daemon thread.
- **boolean isInterrupted()**
  - Tests whether this thread has been interrupted.

## State methods for current thread accesses

- **static Thread currentThread()**

- Returns a reference to the currently executing thread object.

- **static boolean holdsLock(Object obj)**

- Returns true if and only if the current thread holds the monitor lock on the specified object.

- **static boolean interrupted()**

- Tests whether the current thread has been interrupted.

- **static void sleep( [ long millis [, int nanos ] ] )**

- Causes the currently executing thread to sleep (cease execution) for the specified time.

- **static void yield()**

- Causes the currently executing thread object to temporarily pause and allow other threads to execute.

## An example

```
public class SimpleThread extends Thread {  
    public SimpleThread(String str) { super(str); }  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println(i + " " + getName());  
            try { // at this point, current thread is 'this'.  
                Thread.sleep((long)(Math.random() * 1000));  
            } catch (InterruptedException e) {}  
        }  
        System.out.println("DONE! " + getName());  
    }  
}
```

## main program

```
public class TwoThreadsTest {  
    public static void main (String[] args) {  
        new SimpleThread("Thread1").start();  
        new SimpleThread("Thread2").start();    } }
```

**possible output:**

0 Thread1	5 Thread1	DONE! Thread2
0 Thread2	5 Thread2	9 Thread1
1 Thread2	6 Thread2	DONE! Thread1
1 Thread1	6 Thread1	
2 Thread1	7 Thread1	
2 Thread2	7 Thread2	
3 Thread2	8 Thread2	
3 Thread1	9 Thread2	
4 Thread1	8 Thread1	
4 Thread2		

### 3. The states(life cycle) of a thread

```
public class Thread { ..  
    public static enum State { //use Thread.State for referring to this nested class  
        NEW, // after new Thread(), but before start().  
        RUNNABLE, // after start(), when running or ready  
        BLOCKED, // blocked by monitor lock  
                  // blocked by a synchronized method/block  
        WAITING, // waiting for to be notified; no time out set  
                  // wait(), join()  
        TIMED_WAITING, // waiting for to be notified; time out set  
                      // sleep(time), wait(time), join(time)  
        TERMINATED // complete execution or after stop()  
    } ...  
}
```

## State transition methods for Thread

- **public synchronized native void start() {**
  - start a thread by calling its run() method ...
  - It is illegal to start a thread more than once }
- **public final void join( [long ms [, int ns]]);**
  - Let current thread wait for receiver thread to die for at most ms+ns time
- **static void yield() // callable by current thread only**
  - Causes the currently executing thread object to temporarily pause and allow other threads to execute.
- **public final void resume(); // deprecated**
- **public final void suspend();// deprecated → may lead to deadlock**
- **public final void stop(); // deprecated → lead to inconsistency**
- **// state checking**
- **public boolean isAlive() ; // true if runnable or blocked**

Note: When we call t.join(), we in fact use current thread's time to execute code of t thread

## 4. interrupting threads

- A blocking/waiting call (`sleep()`,`wait()` or `join()`) to a thread `t` can be terminated by an `InterruptedException` thrown by invoking `t.interrupt()`.
  - this provides an alternative way to leave the blocked state.
  - however, the control flow is different from the normal case.
- Ex: `public void run() {`
- `try { ... while (more work to do) {` `// Normal sleep() exit continue here`
- `• do some work;`
- `• sleep( ... ),` `// give another thread a chance to work`
- `• }`
- `• }`
- `• catch (InterruptedException e) {` `// if waked-up by interrupt() then continue here`
- `• ...` `// thread interrupted during sleep or wait` `}`
- `}`

- Note: the `interrupt()` method will not throw an `InterruptedException` if the thread is not blocked/waiting. In such case the thread needs to call the static `interrupted()` method to find out if it was recently interrupted. So we should rewrite the while loop by
- `while ( ! interrupted() && moreWorkToDo() ) { ... }`

## interrupt-related methods

### • **void interrupt()**

- send an Interrupt request to a thread.
- the “interrupted” status of the thread is set to true.
- if the thread is blocked by `sleep()`, `wait()` or `join()`, the The *interrupted status* of the thread is cleared and an `InterruptedException` is thrown.
- conclusion: runnable ==> “interrupted” bit set but no Exception thrown.
  - not runnable ==> Exception thrown but “interrupted” bit not set

### • **static boolean interrupted() // destructive query**

- Tests whether the current thread (self) has been interrupted.
- reset the “interrupted” status to false.

### • **boolean isInterrupted() // non-destructive query**

- Tests whether this thread has been interrupted without changing the “interrupted” status.
- may be used to query current executing thread or another non-executing thread. e.g. `if( t1.isInterrupted() | Thread.currentThread()...)`

# A complete example

- it consists of two threads.
- **the first thread (main):**
  - it is the main thread that every Java application has.
  - it creates a new thread from the Runnable object, MessageLoop, and waits for it to finish.
  - if the MessageLoop thread takes too long to finish, the main thread interrupts it.
- **the second thread (MessageLoop):**
  - it prints out a series of messages.
  - if interrupted before it has printed all its messages, it prints a message and exits.

```
public class SimpleThreads {  
    public static void main(String args[]) throws InterruptedException {  
        // Delay, in milliseconds before we interrupt MessageLoop thread  
        long patience = 1000 * 60 * 60;  
        threadMessage("Starting MessageLoop thread");  
        long startTime = System.currentTimeMillis();  
        Thread t = new Thread(new MessageLoop());  
        t.start();  
        threadMessage("Waiting for MessageLoop thread to finish");  
        // loop until MessageLoop thread exits  
        while (t.isAlive()) {  
            threadMessage("Still waiting...");  
            // Wait maximum of 1 second for MessageLoop thread to finish.  
            t.join(1000);  
        }  
    }  
}
```

```
if (((System.currentTimeMillis() - startTime) > patience) && t.isAlive()) {  
    threadMessage("Tired of waiting!");  
  
    t.interrupt();  
  
    // Shouldn't be long now -- wait indefinitely  
  
    t.join();  
  
}  
  
}  
  
threadMessage("Finally!");  
  
}  
  
// Display a message, preceded by the name of the current thread  
  
static void threadMessage(String message) {  
  
    String threadName = Thread.currentThread().getName();  
  
    System.out.format("%s: %s%n", threadName, message);  
  
}
```

```
private static class MessageLoop implements Runnable {  
    public void run() {  
        String importantInfo[] = { "A","B","C","D"};  
        try {  
            for (int i = 0; i < importantInfo.length; i++) {  
                // Pause for 4 seconds  
                Thread.sleep(4000);  
                // Print a message  
                threadMessage(importantInfo[i]);  
            }  
        } catch (InterruptedException e) {  
            threadMessage("I wasn't done!");  
        }  
    }  
}
```

## 5. Thread synchronization

- Problem with any multithreaded Java program :
  - Two or more Thread objects access the same pieces of data.
  - too little or no synchronization ==> there is inconsistency, loss or corruption of data.
  - too much synchronization ==> deadlock or system frozen.
  - In between there is unfair processing where several threads can starve another one hogging all resources between themselves.

## Multithreading may incur inconsistency : an Example

Two concurrent deposits of 50 into an account with 0 initial balance.:

```
void deposit(int amount) {  
    int x = account.getBalance();  
    x += amount;  
    account.setBalance(x); }
```

```
deposit(50) : // deposit 1  
x = account.getBalance() //1  
x += 50; //2  
account.setBalance(x) //3
```

- deposit(50) : // deposit 2
- x = account.getBalance() //4
- x += 50; //5
- account.setBalance(x) //6

The execution sequence: 1,4,2,5,3,6 will result in unwanted result !!  
**Final balance is 50 instead of 100!!**

## Synchronized methods and statements

- multithreading can lead to racing hazards where different orders of interleaving produce different results of computation.
  - Order of interleaving is generally unpredictable and is not determined by the programmer.
- Java's synchronized method (as well as synchronized statement) can prevent its body from being interleaved
  - synchronized( obj ) { ... } // synchronized statement with obj as lock
  - synchronized ... m(... ) {... } //synchronized method with this as lock
  - When one thread executes (the body of) a synchronized method/statement, all other threads are excluded from executing any synchronized method with the same object as lock.

## Synchronizing threads

- Java uses the **monitor** concept to achieve mutual exclusion and synchronization between threads.
- Synchronized methods /statements **guarantee mutual exclusion**.
  - Mutual exclusion may cause a thread to be unable to complete its task. So monitor allows a thread to wait until state change and then continue its work.
- **wait(), notify() and notifyAll()** control the synchronization of threads.
  - Allow one thread to wait for a condition (logical state) and another to set it and then notify waiting threads.
    - **condition variables => instance boolean variables**
    - **wait => wait();**
    - **notifying => notify(); notifyAll();**

## Typical usage

```
synchronized void doWhenCondition() {  
    while ( !condition )  
        wait(); // wait until someone notifies us of changes in condition  
    ... // do what needs to be done when condition is true  
}  
  
synchronized void changeCondition {  
    // change some values used in condition test  
    notify(); // Let waiting threads know something changed  
}
```

Note: A method may serve both roles; it may need some condition to occur to do something and its action may cause condition to change.

## Java's Monitor Model

**A monitor is a collection of code (called the critical section) associated with an object (called the lock)**

**At any time instant only one thread at most can has its execution point located in the critical section associated with the lock(mutual exclusion).**

**Java allows any object to be the lock of a monitor.**

## Java's Monitor Model(cont.)

The critical section of a monitor controlled by an object e [of class C ] comprises the following sections of code:

The body of all synchronized methods m() callable by e, that is, all synchronized methods m(...) defined in C or super classes of C.

The body of all synchronized statements with e as target:

**synchronized(e) { ... }.** // critical section is determined by the lock object e

## Java's Monitor Model(cont.)

A thread enters the critical section of a monitor by invoking e.m() or executing a synchronized statement.

Before it can run the method/statement, it must first own the lock and will need to wait until the lock is free if it cannot get the lock.

A thread owing a lock will release the lock automatically once it exits the critical section.

## Java's Monitor model (continued)

A thread executing in a monitor may encounter condition in which it cannot continue but still does not want to exit.

In such case, it can call the method `e.wait()` to enter the waiting list of the monitor.

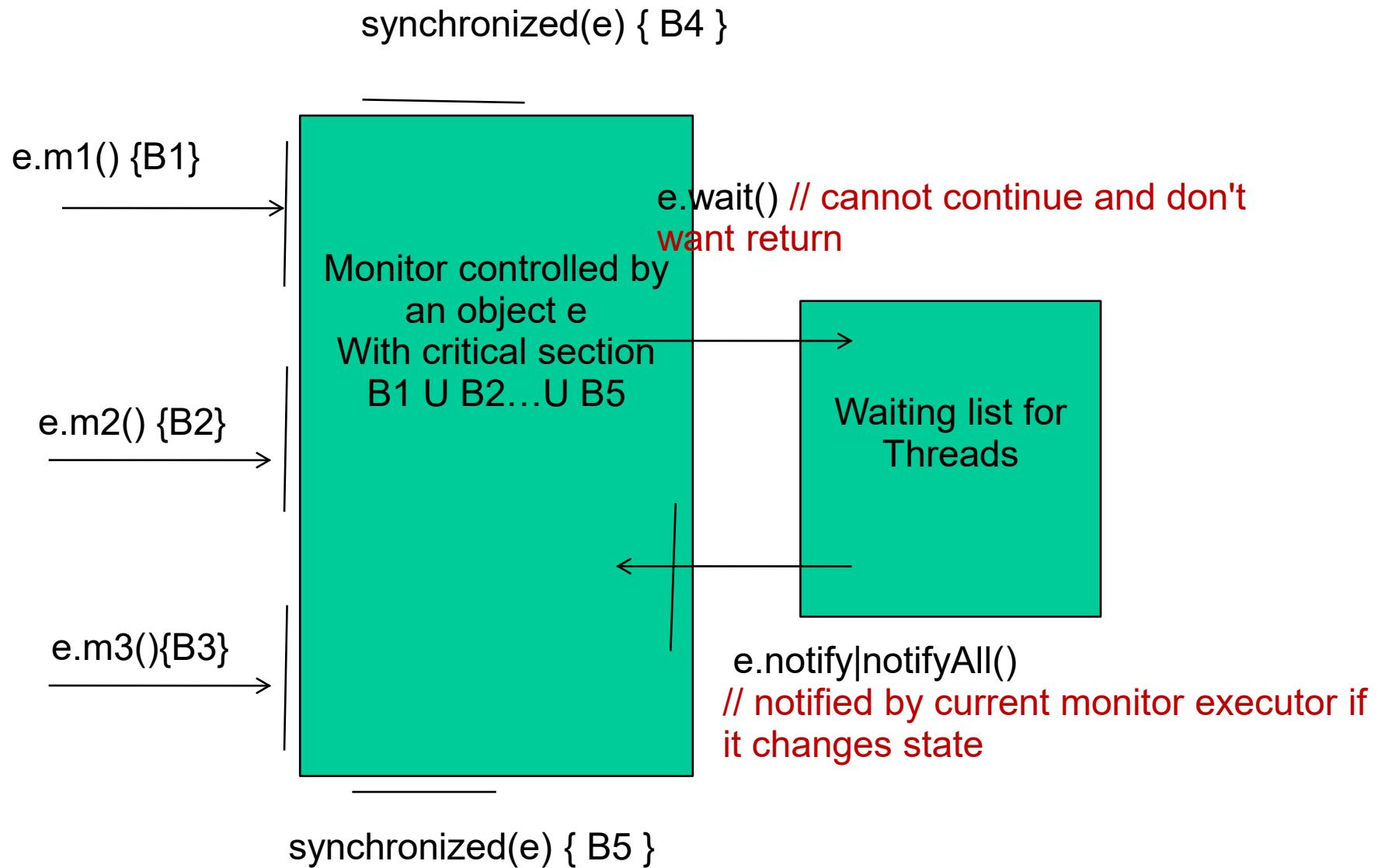
A thread entering waiting list will release the lock so that other outside threads have chance to get the lock.

## Java's Monitor model (continued)

A thread changing the monitor state should call `e.notify()` or `e.notifyAll()` to have one or all threads in the waiting list to compete with other outside threads for getting the lock to continue execution.

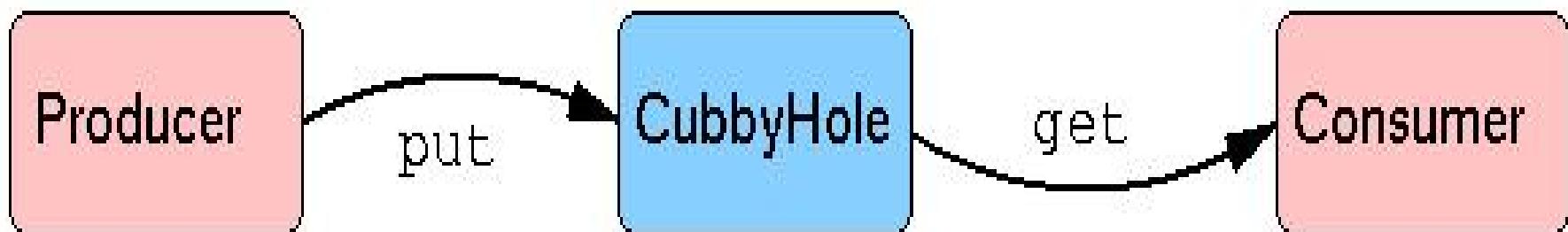
Note: A static method `m()` in class C can also be synchronized. In such case it belongs to the monitor whose lock object is `C.class`.

## Java's monitor model (continued)



## Producer/Consumer Problem

- Two threads: producer and consumer
- One monitor: CubbyHole
- The Producer :
  - generates a pair of integers between 0 and 9 (inclusive), stores it in a CubbyHole object, and prints the sum of each generated pair.
  - sleeps for a random amount of time between 0 and 100 milliseconds before repeating the number generating cycle.
- The Consumer:
  - consumes all pairs of integers from the CubbyHole as quickly as they become available.



## Producer.java

```
public class Producer extends Thread {  
    private CubbyHole cubbyhole;          private int id;  
    public Producer(CubbyHole c, int id) {  
        cubbyhole = c;      this.id = id;      }  
    public void run() {  
        for (int i = 0; i < 10; i++)  
            for(int j =0; j < 10; j++ ) {  
                cubbyhole.put(i, j);  
                System.out.println("Producer #" + this.id + " put: (" + i + "," + j + ");");  
                try { sleep((int)(Math.random() * 100)); }  
                catch (InterruptedException e) { }  
            }  
    }  
}
```

## Consumer.java

```
public class Consumer extends Thread {  
    private CubbyHole cubbyhole;  
    private int id;  
  
    public Consumer(CubbyHole c, int id) {  
        cubbyhole = c;      this.id = id;  }  
  
    public void run() {  
        int value = 0;  
        for (int i = 0; i < 10; i++) {  
            value = cubbyhole.get();  
            System.out.println("Consumer #" + this.id + " got: " + value);  
        }  
    }  
}
```

## CubbyHole without mutual exclusion

```
public class CubbyHole { private int x,y;  
    public synchronized int get() { return x+y; }  
    public synchronized void put(int i, int j) {x= i; y = j } }
```

Problem : data inconsistency for some possible execution sequence

Suppose after  $\text{put}(1,9)$  the data is correct , i.e.,  $(x,y) = (1,9)$

And then two method calls  $\text{get}()$  and  $\text{put}(2,0)$  try to access CubbyHole  
concurrently => **possible inconsistent result:**

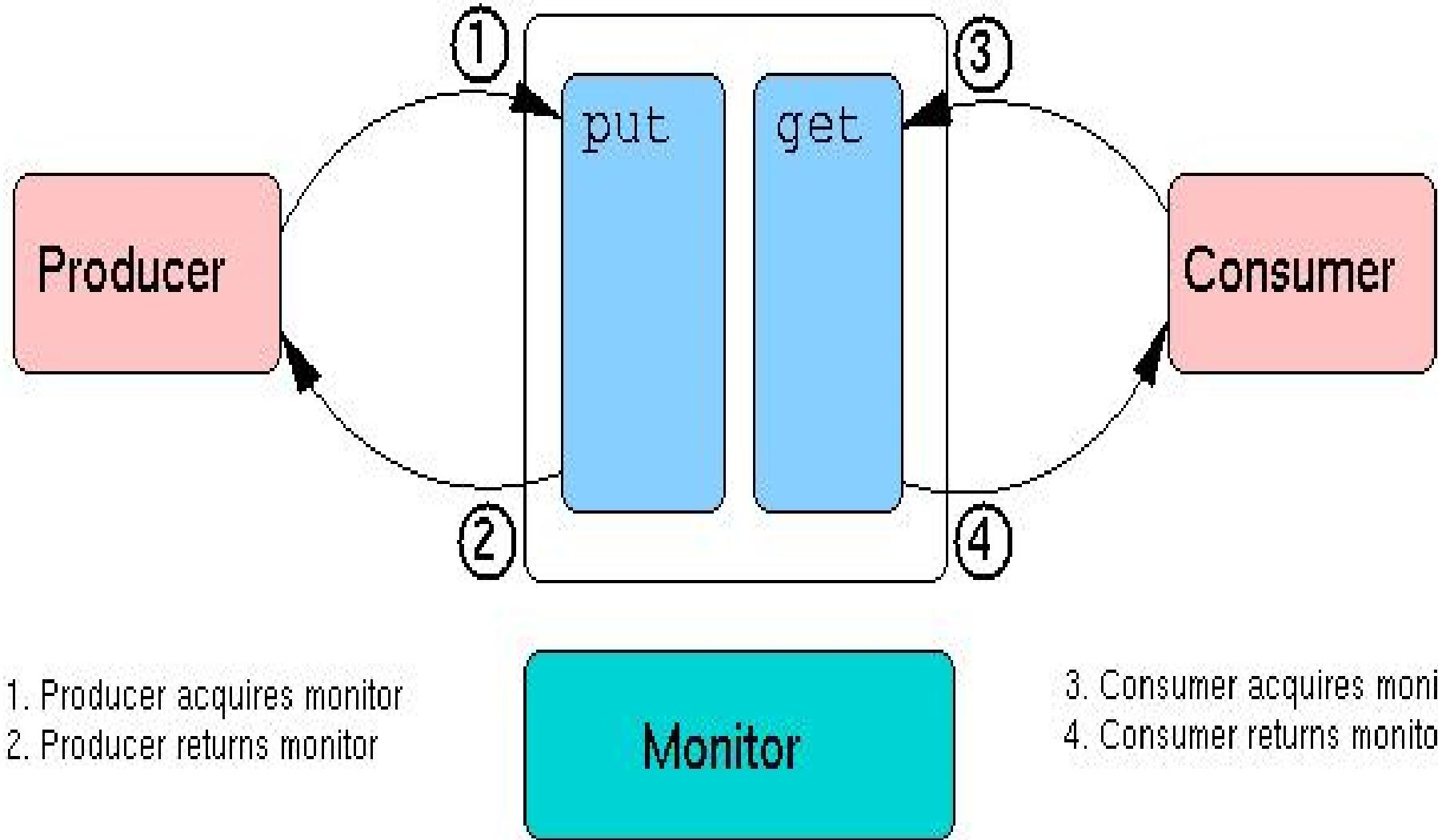
$(1,9) \rightarrow \text{get}() \{ \text{return } \underline{x} + y ; \} \rightarrow \{ \text{return } 1 + y ; \}$

$(1,9) \rightarrow \text{put}(2,0) \{x = 2; y = 0;\} \rightarrow (x,y) = (2,0)$

$(2,0) \rightarrow \text{get}() \{ \text{return } 1 + \underline{y} ; \} \rightarrow \text{return } \underline{1 + 0} = \text{return } 1 \text{ (instead of 10!)}$

By marking  $\text{get}()$  and  $\text{put}()$  as synchronized method, the inconsistent result  
cannot occur since, by definition, when either method is in execution by  
one thread, no other thread can execute any synchronized method with  
this CubbyHole object as lock.

## The CubbyHole



## CubbyHole without synchronization

```
public class CubbyHole {  
    private int x,y;  
    public synchronized int get() { return x+y; }  
    public synchronized void put(int i, int j) { x= i ; y = j; }  
}
```

### Problems:

**Consumer quicker than Producer : some data got more than once.**

**Producer quicker than Consumer: some put data not used by consumer.**

ex: Producer #1 put: (0,4)

Consumer #1 got: 4

Consumer #1 got: 4

Producer #1 put: (0,5)

Consumer #1 got: 3

Producer #1 put: (0,4)

Producer #1 put: (0,5)

Consumer #1 got: 5

## Another CubbyHole implementation (still incorrect!)

```
public class CubbyHole { int x,y; boolean available = false;  
public synchronized int get() { // won't work!  
    if (available == true) {  
        available = false; return x+y;  
    } } // compilation error!! must return a value in any case!!  
public synchronized void put(int a, int b) { // won't work!  
    if (available == false) {  
        available = true; x=a;y=b;  
    } } // but how about the case that available == true ?  
put(..); get(); get(); // 2nd get() must return something!  
put(..);put(..); // 2nd put() has no effect!
```

## CubbyHole.java

```
public class CubbyHole {  
    private int x,y;    private boolean available = false; // condition var  
    public synchronized int get() {  
        while (available == false) {  
            try { this.wait(); } catch (InterruptedException e) {}  
            available = false; // enforce consumers to wait again.  
            notifyAll(); // notify all producer/consumer to compete for execution!  
                         // use notify() if just wanting to wakeup one waiting thread!  
        return x+y;    }  
    public synchronized void put(int a, int b) {  
        while (available == true) {  
            try { wait(); } catch (InterruptedException e) {}  
        x= a; y = b;  
        available = true; // wake up waiting consumer to continue  
        notifyAll(); // or notify();    }  
    }  
}
```

## The main class

```
public class ProducerConsumerTest {  
    public static void main(String[] args) {  
        CubbyHole c = new CubbyHole();  
        Producer p1 = new Producer(c, 1);  
        Consumer c1 = new Consumer(c, 1);  
        p1.start();  
        c1.start();  
    }    }  
}
```

## Other issues

- **Thread priorities**

- **public final int getPriority();**
- **public final void setPriority();**
- get/set priority between MIN\_PRIORITY and MAX\_PRIORITY
- default priority : NORMAL\_PRIORITY

- **Daemon threads:**

- **isDaemon(), setDaemon(boolean)**
  - A Daemon thread is one that exists for service of other threads.
  - The JVM exits if all threads in it are Daemon threads.
  - **setDaemon(.)** must be called before the thread is started.

- **public static boolean holdsLock(Object obj)**

- check if this thread holds the lock on obj.
- ex: **synchronized( e ) { Thread.holdLock(e) ? true:false // is true ... }**

## Thread Groups

- Every Java thread is a member of a thread group.
- Thread groups provide a mechanism for **collecting multiple threads into a single object and manipulating those threads all at once, rather than individually.**
- When creating a thread,
  - let the runtime system put the new thread in some reasonable default group (the current thread group) or
  - explicitly set the new thread's group.
- you cannot move a thread to a new group after the thread has been created.
- when launched, main program thread belongs to main thread group.

## Creating a Thread Explicitly in a Group

```
public Thread(ThreadGroup group, Runnable runnable)
public Thread(ThreadGroup group, String name)
public Thread(ThreadGroup group, Runnable runnable, String
name)
```

```
ThreadGroup myThreadGroup = new ThreadGroup(
```

```
    "My Group of Threads");
```

```
Thread myThread = new Thread(myThreadGroup,
```

```
    "a thread for my group");
```

## Getting a Thread's Group

```
theGroup = myThread.getThreadGroup();
```

## The ThreadGroup Class

### Collection Management Methods:

```
public class EnumerateTest {  
    public void listCurrentThreads() {  
        ThreadGroup currentGroup =  
            Thread.currentThread().getThreadGroup();  
        int numThreads = currentGroup.activeCount();  
        Thread[] listOfThreads = new Thread[numThreads];  
  
        currentGroup.enumerate(listOfThreads);  
        for (int i = 0; i < numThreads; i++)  
            System.out.println("Thread #" + i + " = " +  
                listOfThreads[i].getName());  
    }  
}
```

# **Advanced Programming Methods**

**Lecture 8 - Concurrency in Java (2)**

# **Content(java.util.concurrent)**

**1)Executor Service**

**2)ForkJoinPool**

**3)Blocking Queue**

**4)Concurrent Collections**

**5)Semaphore**

**6)CountDownLatch**

**7)CyclicBarrier**

**8)Lock**

**9)Atomic Variables**

# Java.util.concurrency

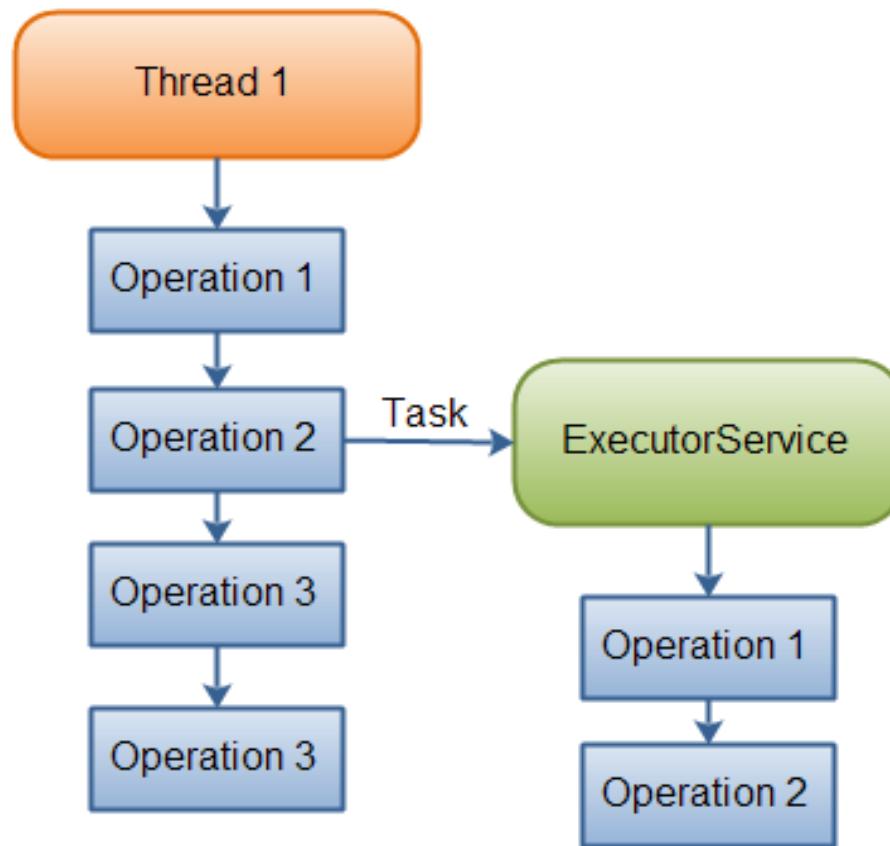
- A set of ready-to-use data structures and functionality for writing safe multithreaded applications
- it is **not always trivial** to write robust code that executes well in a multi-threaded environment

# Executor Service

- The `java.util.concurrent.ExecutorService` interface represents an asynchronous execution mechanism which is capable of executing tasks in the background.
- It is very similar to a thread pool. In fact, the implementation of `ExecutorService` present in the `java.util.concurrent` package is a thread pool implementation.

# ExecutorService

```
ExecutorService executorService  
=Executors.newFixedThreadPool(10);  
//10 threads for executing tasks are created  
  
executorService.execute(new Runnable() {  
    public void run() {  
        System.out.println("Asynchronous task");  
    }  
});  
  
executorService.shutdown();
```



**Thread 1 delegates a task to an Executor Service for asynchronous execution**

# Creating executor service

Using Executors factory class:

```
ExecutorService executorService1 =  
    Executors.newSingleThreadExecutor();
```

```
ExecutorService executorService2 =  
    Executors.newFixedThreadPool(10);
```

```
ExecutorService executorService3 =  
    Executors.newScheduledThreadPool(10);
```

# ExecutorService usage

There are a few different ways to delegate tasks for execution to an ExecutorService:

- **.execute(Runnable)**
- **.submit(Runnable)**
- **.submit(Callable)**
- **.invokeAny(...)**
- **.invokeAll(...)**

# **execute(Runnable)**

```
ExecutorService executorService =
    Executors.newSingleThreadExecutor();

executorService.execute(new Runnable() {
        public void run() {
                System.out.println("Asynchronous task");
        }
});

executorService.shutdown();
```

- There is no way of obtaining the result of the executed Runnable, if necessary. We have to use a Callable for that

# submit(Callable)

```
Future future = executorService.submit(new Callable(){  
    public Object call() throws Exception {  
        System.out.println("Asynchronous Callable");  
        return "Callable Result";  
    }  
});  
  
System.out.println("future.get() = " + future.get());
```

- The Callable's result can be obtained via the Future object returned

# invokeAll()

```
ExecutorService executorService =
    Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();
callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 1";
    }
});

List<Future<String>> futures = executorService.invokeAll(callables);
for(Future<String> future : futures){
    System.out.println("future.get = " + future.get());
}

executorService.shutdown();
```

```
ExecutorService executor = Executors.newWorkStealingPool();

List<Callable<String>> callables = Arrays.asList(
    () -> "task1",
    () -> "task2",
    () -> "task3");

executor.invokeAll(callables)

.stream()

.map(future -> {
    try {
        return future.get();
    }
    catch (Exception e) {
        throw new IllegalStateException(e);
    }
})

.forEach(System.out::println);
```

# Callables and Futures

- Callables are functional interfaces just like runnables but instead of being void they return a value.

```
Callable<Integer> task = () -> {
    try {
        TimeUnit.SECONDS.sleep(1);
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
    }
};
```

# Callables and Futures

- Callables can be submitted to executor services just like runnables.
- the executor returns a special result of type Future which can be used to retrieve the actual result at a later point in time.
- After submitting the callable to the executor we can check if the future has already been finished execution via isDone()

```
ExecutorService executor = Executors.newFixedThreadPool(1);
```

```
Future<Integer> future = executor.submit(task);
```

```
System.out.println("future done? " + future.isDone());
```

```
Integer result = future.get();
```

```
System.out.println("future done? " + future.isDone());
```

```
System.out.print("result: " + result);
```

```
//future done? false
```

```
//future done? true
```

```
//result: 123
```

# ExecutorService Shutdown

- To terminate the threads inside the ExecutorService you call its shutdown() method. It will not shut down immediately, but it will no longer accept new tasks, and once all threads have finished current tasks, the ExecutorService shuts down
- to shut down the ExecutorService immediately, you can call the shutdownNow() method. This will attempt to stop all executing tasks right away, and skips all submitted but non-processed tasks.

# `java.util.concurrent.ThreadPoolExecutor`

- is an implementation of the `ExecutorService` interface.
- executes the given task (`Callable` or `Runnable`) using one of its internally pooled threads.
- The number of threads in the pool is determined by these variables:
  - `corePoolSize`
  - `maximumPoolSize`

# ScheduledExecutorService

- It is an interface
- can schedule tasks to run after a delay, or to execute repeatedly with a fixed interval of time in between each execution.
- Tasks are executed asynchronously by a worker thread, and not by the thread handing the task to the ScheduledExecutorService.

# ScheduledExecutorService

```
ScheduledExecutorService scheduledExecutorService =  
    Executors.newScheduledThreadPool(5);
```

```
ScheduledFuture scheduledFuture =  
    scheduledExecutorService.schedule(new Callable() {  
        public Object call() throws Exception {  
            System.out.println("Executed!");  
            return "Called!";  
        }  
    }, 5, TimeUnit.SECONDS);
```

- the Callable should be executed after 5 seconds

# ScheduledExecutorService Usage

Once you have created a ScheduledExecutorService you use it by calling one of its methods:

- schedule (Callable task, long delay, TimeUnit timeunit)
- schedule (Runnable task, long delay, TimeUnit timeunit)
- scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)
- scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)

# ForkJoinPool

- is similar to the ExecutorService but with one difference
- implements the work-stealing strategy, i.e. every time a running thread has to wait for some result; the thread removes the current task from the work queue and executes some other task ready to run. This way the current thread is not blocked and can be used to execute other tasks. Once the result for the originally suspended task has been computed the task gets executed again and the join() method returns the result.

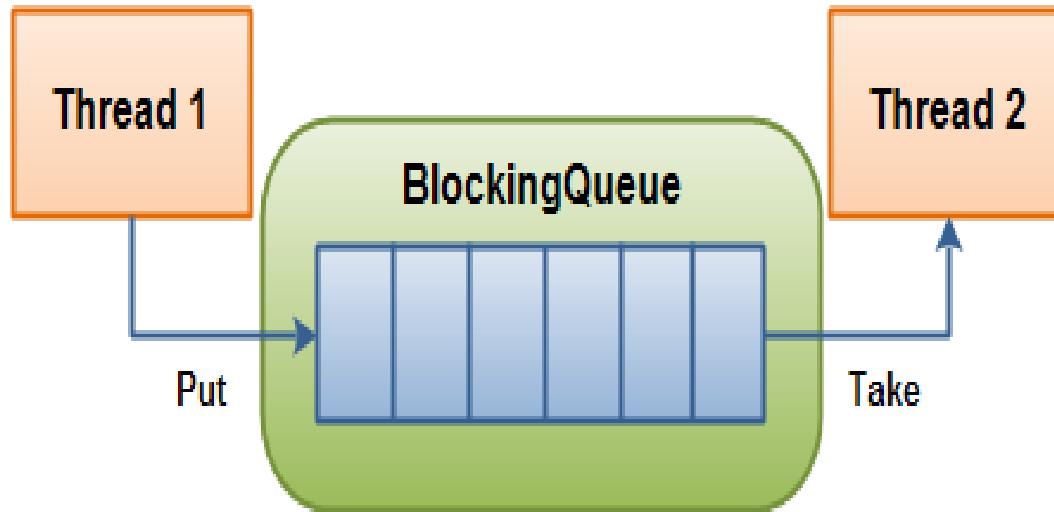
# ForkJoinPool

- a call of fork() will start an asynchronous execution of the task,
- a call of join() will wait until the task has finished and retrieve its result.
- makes it easy for tasks to split their work up into smaller tasks(divide and conquer approach) which are then submitted to the ForkJoinPool too.

```
01 public class FindMin extends RecursiveTask<Integer> {  
02     private static final long serialVersionUID = 1L;  
03     private int[] numbers;  
04     private int startIndex;  
05     private int endIndex;  
06  
07     public FindMin(int[] numbers, int startIndex, int endIndex) {  
08         this.numbers = numbers;  
09         this.startIndex = startIndex;  
10         this.endIndex = endIndex;  
11     }  
12  
13     @Override  
14     protected Integer compute() {  
15         int sliceLength = (endIndex - startIndex) + 1;  
16         if (sliceLength > 2) {  
17             FindMin lowerFindMin = new FindMin(numbers, startIndex, startIndex + (sliceLength / 2) - 1);  
18             lowerFindMin.fork();  
19         }  
20         return numbers[startIndex];  
21     }  
22 }
```

```
19 FindMin upperFindMin = new FindMin(numbers, startIndex + (sliceLength / 2), endIndex);
20 upperFindMin.fork();
21 return Math.min(lowerFindMin.join(), upperFindMin.join());
22 } else {
23 return Math.min(numbers[startIndex], numbers[endIndex]);
24 }
25 }
26
27 public static void main(String[] args) {
28 int[] numbers = new int[100];
29 Random random = new Random(System.currentTimeMillis());
30 for (int i = 0; i < numbers.length; i++) {
31 numbers[i] = random.nextInt(100);
32 }
33 ForkJoinPool pool = new ForkJoinPool(Runtime.getRuntime().availableProcessors());
34 Integer min = pool.invoke(new FindMin(numbers, 0, numbers.length - 1));
35 System.out.println(min);
36 }
37 }
```

# Interface BlockingQueue



# Interface BlockingQueue

- methods come in four forms, with different ways of handling operations that cannot be satisfied immediately, but may be satisfied at some point in the future:
  - one throws an exception: **add(e)**, **remove()**, **element()**
  - the second returns a special value (either null or false, depending on the operation): **offer(e)**, **poll()**, **peek()**
  - the third blocks the current thread indefinitely until the operation can succeed: **put(e)**, **take()**
  - the fourth blocks for only a given maximum time limit before giving up: **offer(e, time, unit)**, **poll(time, unit)**

```
public class BlockingQueueExample {  
  
    public static void main(String[] args) throws Exception {  
  
        BlockingQueue queue = new ArrayBlockingQueue(1024);  
  
        Producer producer = new Producer(queue);  
        Consumer consumer = new Consumer(queue);  
  
        new Thread(producer).start();  
        new Thread(consumer).start();  
  
        Thread.sleep(4000);  
    }  
}
```

```
public class Producer implements Runnable{  
    protected BlockingQueue queue = null;  
    public Producer(BlockingQueue queue) {  
        this.queue = queue;}  
  
    public void run() {  
        try {  
            queue.put("1");  
            Thread.sleep(1000);  
            queue.put("2");  
            Thread.sleep(1000);  
            queue.put("3");  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
public class Consumer implements Runnable{  
    protected BlockingQueue queue = null;  
  
    public Consumer(BlockingQueue queue) {  
        this.queue = queue; }  
  
    public void run() {  
        try {  
            System.out.println(queue.take());  
            System.out.println(queue.take());  
            System.out.println(queue.take());  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

# ConcurrentHashMap

- is very similar to the `java.util.HashTable` class, except that `ConcurrentHashMap` offers better concurrency than `HashTable` does.
- does not lock the Map while you are reading from it.
- does not lock the entire Map when writing to it. It only locks the part of the Map that is being written to, internally.

# Semaphore

- the `java.util.concurrent.Semaphore` class is a counting semaphore.
- The counting semaphore is initialized with a given number of "permits".
- For each call to `acquire()` a permit is taken by the calling thread.
- For each call to `release()` a permit is returned to the semaphore.
- Thus, at most N threads can pass the `acquire()` method without any `release()` calls, where N is the number of permits the semaphore was initialized with.

# Semaphore

As semaphore typically has two uses:

- To guard a critical section against entry by more than N threads at a time.
- To send signals between two threads.

```
ExecutorService executor = Executors.newFixedThreadPool(10);
```

```
Semaphore semaphore = new Semaphore(5);
```

```
Runnable longRunningTask = () -> {
```

```
    boolean permit = false;
```

```
    try {
```

```
        permit = semaphore.tryAcquire(1, TimeUnit.SECONDS);
```

```
        if (permit) {
```

```
            System.out.println("Semaphore acquired");
```

```
            sleep(5);
```

```
        } else {System.out.println("Could not acquire semaphore");}
```

```
    } catch (InterruptedException e) {
```

```
        throw new IllegalStateException(e);
```

```
    } finally {
```

```
        if (permit) {semaphore.release();}
```

```
    }}
```

```
IntStream.range(0, 10).forEach(i -> executor.submit(longRunningTask));
```

```
stop(executor);
```

# CountDownLatch

- is initialized with a given count.
- This count is decremented by calls to the countDown() method.
- Threads waiting for this count to reach zero can call one of the await() methods. Calling await() blocks the thread until the count reaches zero.

```
CountDownLatch latch = new CountDownLatch(3);
```

```
Waiter waiter = new Waiter(latch);
```

```
Decrementer decrementer = new Decrementer(latch);
```

```
new Thread(waiter).start();
```

```
new Thread(decrementer).start();
```

```
Thread.sleep(4000);
```

```
public class Waiter implements Runnable{  
    CountDownLatch latch = null;  
  
    public Waiter(CountDownLatch latch) {  
        this.latch = latch;}  
  
    public void run() {  
        try {  
            latch.await();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        System.out.println("Waiter Released");  
    }  
}
```

```
public class Decrementer implements Runnable {  
    CountDownLatch latch = null;  
  
    public Decrementer(CountDownLatch latch) {  
        this.latch = latch;  
    }  
  
    public void run() {  
        try {  
            Thread.sleep(1000);  
            this.latch.countDown();  
            Thread.sleep(1000);  
            this.latch.countDown();  
            Thread.sleep(1000);  
            this.latch.countDown();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

# Cyclic Barrier

- is a synchronization mechanism that can synchronize threads progressing through some algorithm.
- it is a barrier that all threads must wait at, until all threads reach it, before any of the threads can continue.
- The threads wait for each other by calling the await() method on the CyclicBarrier.
- Once N threads are waiting at the CyclicBarrier, all threads are released and can continue running.

```
Runnable barrier1Action = new Runnable() {  
    public void run() {  
        System.out.println("BarrierAction 1 executed ");}};  
  
Runnable barrier2Action = new Runnable() {  
    public void run() {  
        System.out.println("BarrierAction 2 executed ");}}
```

```
CyclicBarrier barrier1 = new CyclicBarrier(2, barrier1Action);  
CyclicBarrier barrier2 = new CyclicBarrier(2, barrier2Action);  
  
CyclicBarrierRunnable barrierRunnable1 =  
    new CyclicBarrierRunnable(barrier1, barrier2);  
  
CyclicBarrierRunnable barrierRunnable2 =  
    new CyclicBarrierRunnable(barrier1, barrier2);  
  
new Thread(barrierRunnable1).start();  
new Thread(barrierRunnable2).start();
```

```
public class CyclicBarrierRunnable implements Runnable{  
  
CyclicBarrier barrier1 = null;  
CyclicBarrier barrier2 = null;  
  
public CyclicBarrierRunnable(  
    CyclicBarrier barrier1,  
    CyclicBarrier barrier2) {  
  
this.barrier1 = barrier1;  
this.barrier2 = barrier2;  
}
```

```
public void run() {  
    try {  
        Thread.sleep(1000);  
  
        System.out.println(Thread.currentThread().getName() +  
            " waiting at barrier 1");  
  
        this.barrier1.await();  
  
        Thread.sleep(1000);  
  
        System.out.println(Thread.currentThread().getName() +  
            " waiting at barrier 2");  
  
        this.barrier2.await();  
  
        System.out.println(Thread.currentThread().getName() + " done!");  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    } catch (BrokenBarrierException e) {  
        e.printStackTrace();  
    }  
}
```

# Lock

- is a thread synchronization mechanism just like synchronized blocks. A Lock is, however, more flexible and more sophisticated than a synchronized block.

```
Lock lock = new ReentrantLock();
```

```
lock.lock();
```

```
//critical section
```

```
lock.unlock();
```

# ReadWriteLock

- allows multiple threads to read a certain resource, but only one to write it, at a time.
- Read Lock: If no threads have locked the ReadWriteLock for writing, and no thread have requested a write lock. Thus, multiple threads can lock the lock for reading.
- Write Lock: If no threads are reading or writing. Thus, only one thread at a time can lock the lock for writing.

```
ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
```

```
readWriteLock.readLock().lock();
```

```
// multiple readers can enter this section
```

```
// if not locked for writing, and not writers waiting
```

```
// to lock for writing.
```

```
readWriteLock.readLock().unlock();
```

```
readWriteLock.writeLock().lock();
```

```
// only one writer can enter this section,
```

```
// and only if no threads are currently reading.
```

```
readWriteLock.writeLock().unlock();
```

# Atomic Variables

- the atomic classes make heavy use of compare-and-swap (CAS), an atomic instruction directly supported by most modern CPUs.
- Those instructions usually are much faster than synchronizing via locks.
- The advice is to prefer atomic classes over locks in case you just have to change a single mutable variable concurrently.
- Many atomic classes: AtomicBoolean, AtomicInteger, AtomicReference, AtomicIntegerArray, etc

# AtomicBoolean

- provides a boolean variable which can be read and written atomically, and which also contains advanced atomic operations like compareAndSet()

- Example:

```
AtomicBoolean atomicBoolean = new AtomicBoolean(true);  
boolean expectedValue = true;  
boolean newValue      = false;  
boolean wasNewValueSet =  
    atomicBoolean.compareAndSet(expectedValue, newValue);
```

# AtomicInteger

```
AtomicInteger atomicInt = new AtomicInteger(0);
ExecutorService executor = Executors.newFixedThreadPool(2);
IntStream.range(0, 1000)
    .forEach(i -> {
        Runnable task = () ->
            atomicInt.updateAndGet(n -> n + 2); //thread-safe without synchronization
        executor.submit(task);
});
stop(executor);
System.out.println(atomicInt.get()); // => 2000
```

# **Advanced Programming Methods**

## **Lecture 9 - JavaFX**

# OUR SLIDES USE EXAMPLES FROM THE FOLLOWINGS:

## JavaFX tutorials

- <https://docs.oracle.com/javase/8/javase-clienttechnologies.htm>
- <https://wiki.eclipse.org/Efxclipse/Tutorials>
- <http://o7planning.org/en/11009/javafx>
- <http://code.makery.ch/library/javafx-8-tutorial/>
- <https://o7planning.org/10623/javafx-tutorial-for-beginners>

## JavaFX API

- <https://docs.oracle.com/javase/8/javafx/api/toc.htm>

# CONTENT

.What is JavaFX

.JavaFx Architecture

.Steps to install and set JavaFx (on Eclipse)

.Scene graph

.Simple JavaFx Application

.Layout management

.Event Driven Programming – Event Handling

.A Simple Application without SceneBuilder

# WHAT IS JAVAFX ?

- Classes and interfaces that provide support for creating Java applications that can be designed, implemented, tested on different platforms.
- Provides support for the use of Web components such as HTML5 code or JavaScript scripts
- Contains graphic UI components for creating graphical interfaces and manage their appearance through CSS files
- Provides support for interactive 3D graphics
- Provides support for handling multimedia content
- Supports RIAs (Rich Internet Application)
- Portability: desktop, browser, mobile, TV, game consoles, Blu-ray, etc.
- Ensures interoperability to Swing

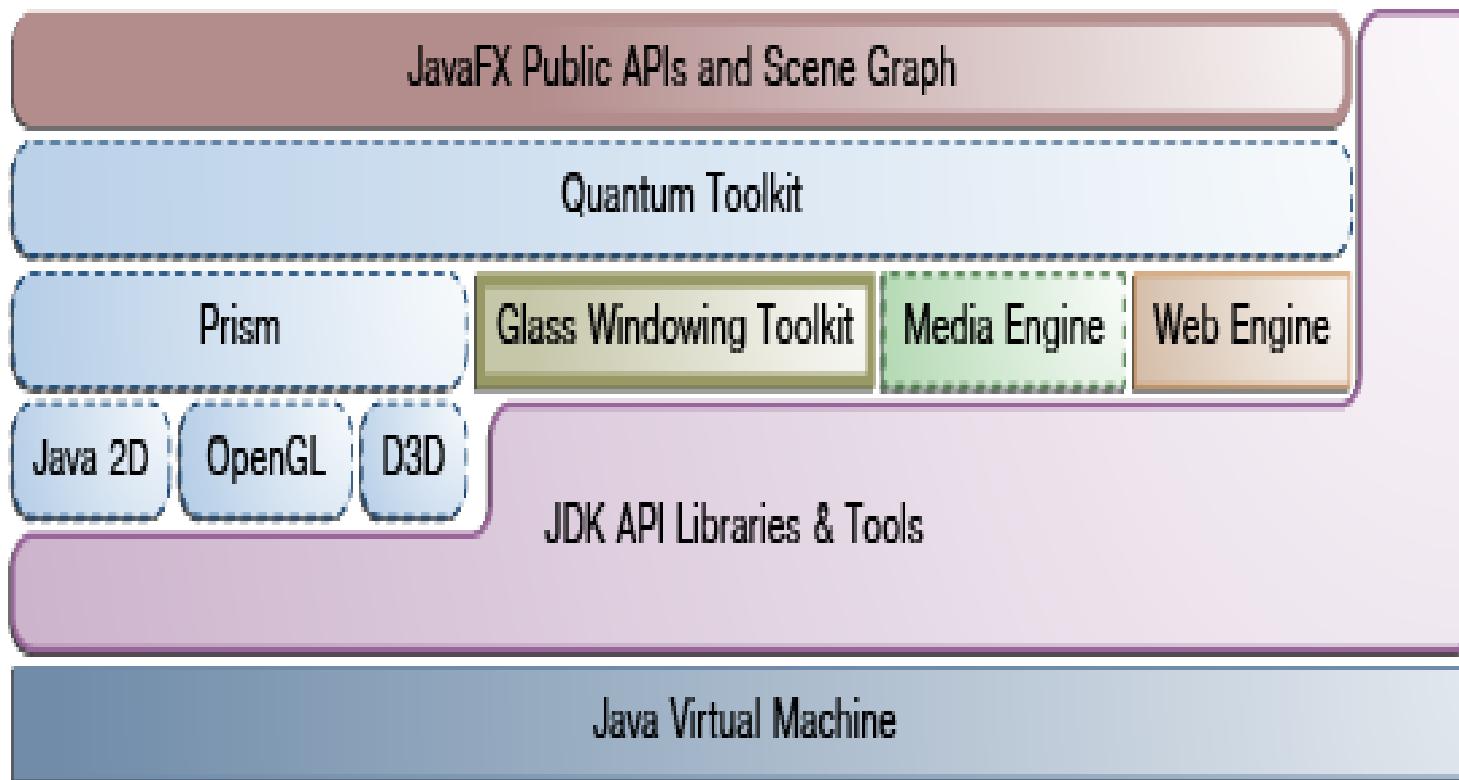
# WHAT IS JAVAFX ?

- Java 8 JavaFX is bundled with the Java platform, so JavaFX is available everywhere Java is
- From Java 8 you can also create standalone install packages for Windows, Mac and Linux with Java, which includes the JRE needed to run them.
  - This means that you can distribute JavaFX applications to these platforms even if the platform does not have Java installed already
- Starting from Java 11, JavaFx is not longer part of the JVM therefore you require to do more settings

# JAVAFX VS. SWING

- JavaFX is intended to replace Swing as the default GUI toolkit in Java.
- JavaFX is more consistent in its design than Swing, and has more features:
  - It is more modern too, enabling you to design GUI using layout files (XML) and style them with CSS, just like we are used to with web applications.
  - JavaFX also integrates 2D + 3D graphics, charts, audio, video, and embedded web applications into one coherent GUI toolkit.

# JAVAFX ARCHITECTURE



# JAVAFX ARCHITECTURE

- Scene Graph:

- is the starting point for constructing a JavaFX application.
  - is a hierarchical tree of nodes that represents all of the visual elements of the application's user interface.
  - can handle input and can be rendered.

- Java Public API:

- provides a complete set of Java public APIs that support rich client application development.

# JAVAFX ARCHITECTURE

- Graphics System:

- **Prism:**

- processes render jobs.
    - can run on both hardware and software renderers, including 3-D.
    - is responsible for rasterization and rendering of JavaFX scenes.

- Quantum Toolkit**

- ties Prism and Glass Windowing Toolkit together and makes them available to the JavaFX layer above them in the stack.
    - also manages the threading rules related to rendering versus events handling.

# JAVAFX ARCHITECTURE

- Glass Windowing Toolkit

- provides native operating services, such as managing the windows, timers, and surfaces.
  - serves as the platform-dependent layer that connects the JavaFX platform to the native operating system.
    - is also responsible for managing the event queue.
    - runs on the same thread as the JavaFX application

# JAVAFX ARCHITECTURE

- Running Threads

- JavaFX application thread:**

- primary thread used by JavaFX application developers
    - Any "live" scene, which is a scene that is part of a window, must be accessed from this thread
    - A scene graph can be created and manipulated in a background thread, but when its root node is attached to any live object in the scene, that scene graph must be accessed from the JavaFX application thread

- Prism render thread:**

- handles the rendering separately from the event dispatcher.

- Media thread:**

- runs in the background and synchronizes the latest frames through the scene graph by using the JavaFX application thread.

# JAVAFX ARCHITECTURE

- Pulse:

- is an event that indicates to the JavaFX scene graph that it is time to synchronize the state of the elements on the scene graph with Prism.
  - The Glass Windowing Toolkit is responsible for executing the pulse events

- Media and Images:

- JavaFX supports both visual and audio media

- Web Component:

- is a JavaFX UI control, based on Webkit, that provides a Web viewer and full browsing functionality through its API.

# JAVAFX ARCHITECTURE

- JavaFX Cascading Style Sheets (CSS)

- provides the ability to apply customized styling to the user interface of a JavaFX application without changing any of that application's source code.
- can be applied to any node in the JavaFX scene graph and are applied to the nodes asynchronously.
- can also be easily assigned to the scene at runtime, allowing an application's appearance to dynamically change.

# JAVAFX ARCHITECTURE

## JavaFX UI Controls

- available through the JavaFX API
- are built by using nodes in the scene graph.
- are portable across different platforms



# JAVAFX ARCHITECTURE

- Layout:

- Layout containers (panes) can be used to allow for flexible and dynamic arrangements of the UI controls within a scene graph

- Transformations 2-D and 3-D

- Each node in the JavaFX scene graph can be transformed in the x-y coordinate

- Visual Effects:

- to enhance the look of JavaFX applications in real time.
  - are primarily image pixel-based

# STEPS TO INSTALL AND SET JAVAFX (ON ECLIPSE) (OLD VERSIONS OF ECLIPSE AND JDK LESS THAN JAVA11)

## .Step 1: install e(fx)clipse into Eclipse (JavaFx tooling)

–You can follow the steps from one of the followings:

–For the latest Eclipse please use

<https://marketplace.eclipse.org/content/efxclipse> and then follow

- <http://o7planning.org/en/10619/install-efxclipse-into-eclipse>
- [https://wiki.eclipse.org/Efxclipse/Tutorials/AddingE\(fx\)clipse\\_to\\_eclipse](https://wiki.eclipse.org/Efxclipse/Tutorials/AddingE(fx)clipse_to_eclipse)

## .Step 2: download and set JavaFx SceneBuilder

–You can follow the steps from <http://o7planning.org/en/10621/install-javafx-scene-builder-into-eclipse>

–You can download the SceneBuilder

- Either from Oracle <http://www.oracle.com/technetwork/java/javase/downloads/javafxscenebuilder-1x-archive-2199384.html>
- Or from <http://gluonhq.com/products/scene-builder/>

# STEPS TO INSTALL AND SET JAVAFX (ON ECLIPSE)

.For the latest versions of ECLIPSE please follow the steps from

<https://coderslegacy.com/java/introduction-to-javafx/>

.To install Java FX Scene Builder please follow the steps from

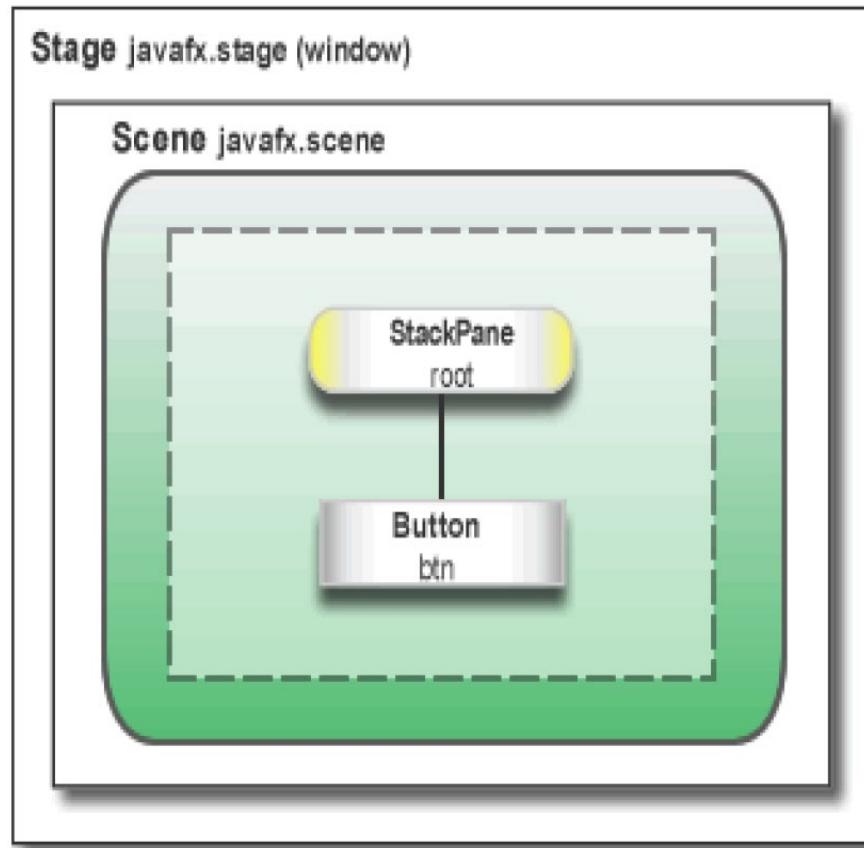
<https://coderslegacy.com/java/introduction-to-javafx/>

# SCENE GRAPH

scene-graph-based programming model

A JavaFX application contains:

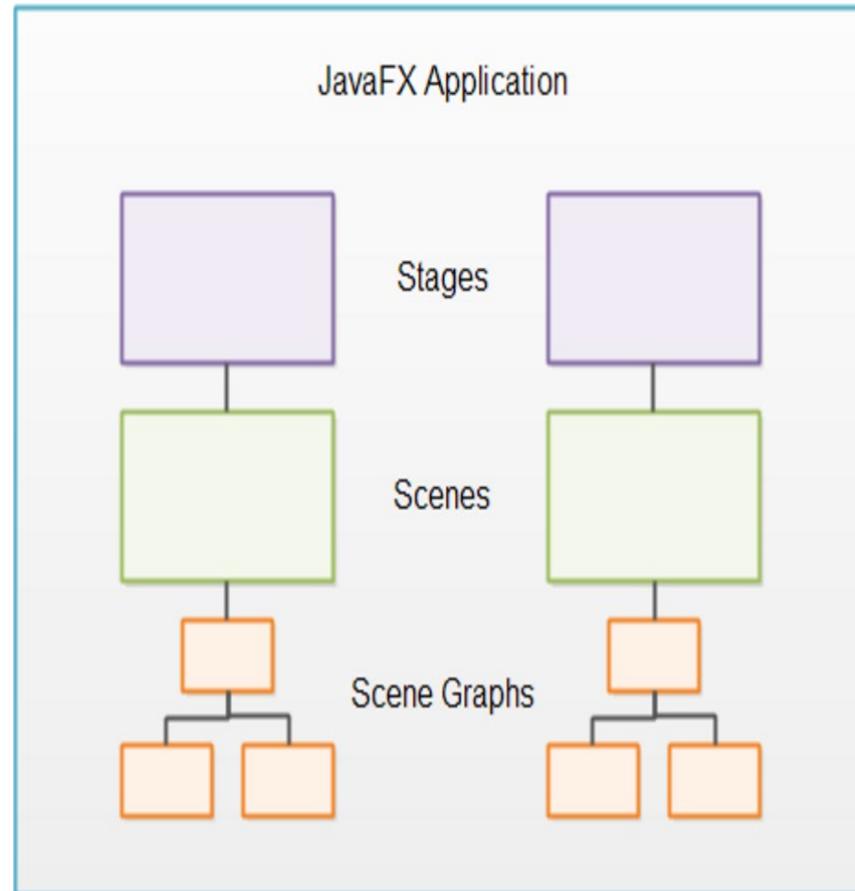
- .An object **Stage (window)**
- .One or more objects **Scene**



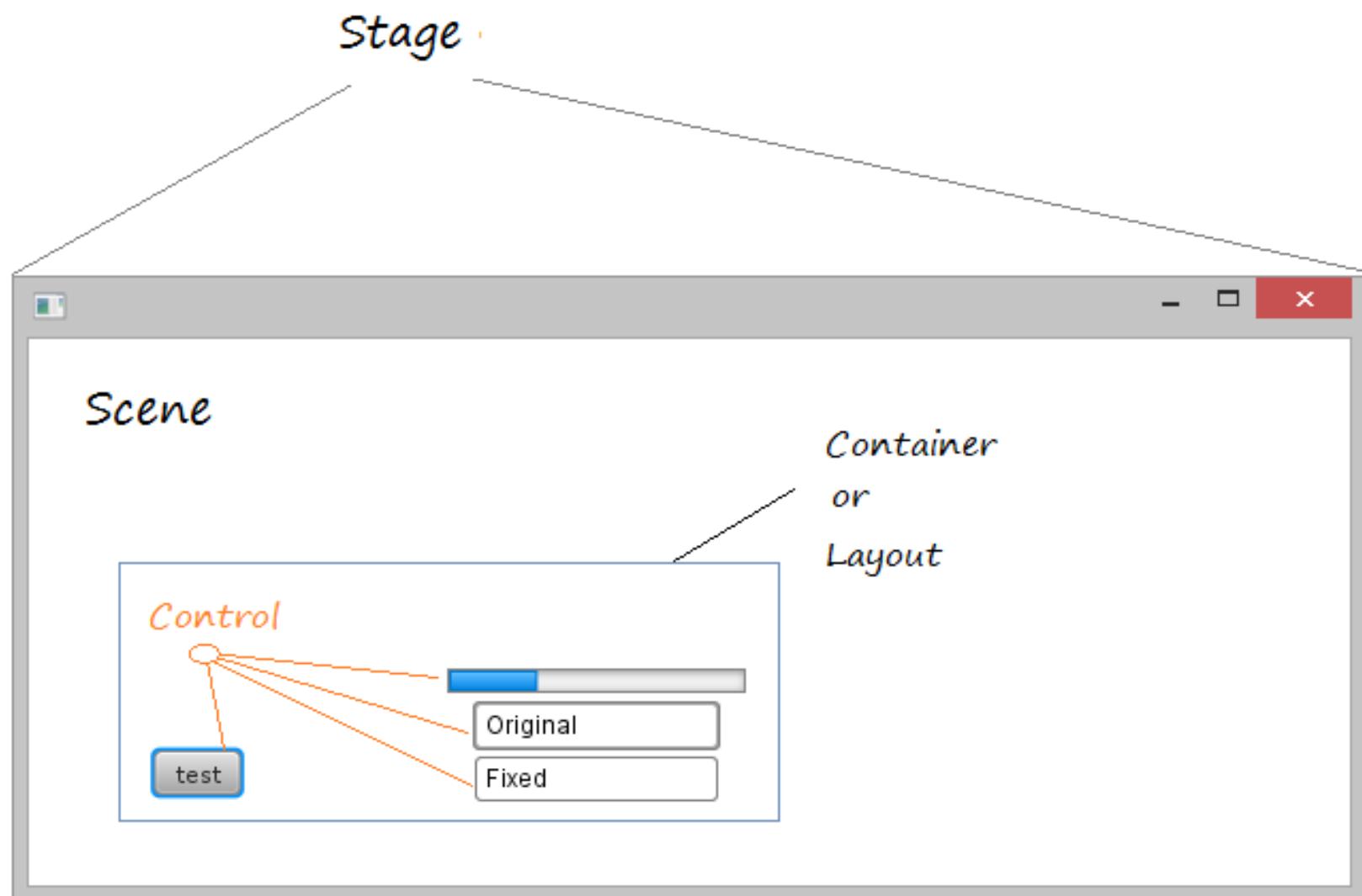
# SCENE GRAPH

- .A Scene Graph is a tree of graphical user interface components.
- .A scene graph element is a node.
- .Each node has an ID, a class style, a bounding volume, etc.
- .Except the root node, each node has a single parent and 0 or more children.
- .Nodes can be internal (Parent) or leaf
- .A node can be associated with various properties (effects (blur, shadow), opacity, transformations) and events (event handlers (Mouse, Keyboard))

# SCENE GRAPH



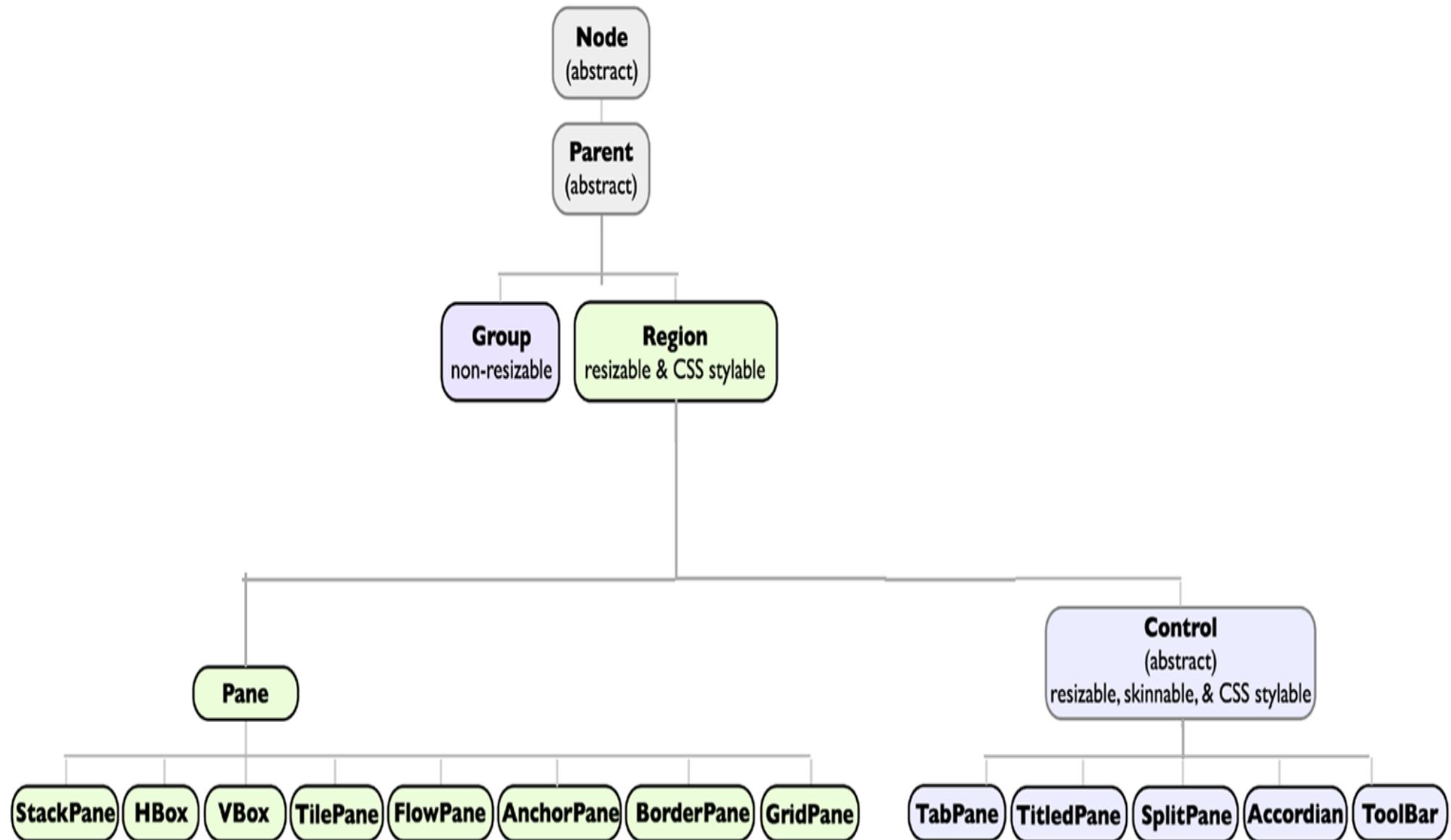
# SCENE GRAPH



# SCENE GRAPH

- **Node:** The abstract base class for all scene graph nodes.
- **Parent:** The abstract base class for all branch nodes.  
(This class directly extends Node).
- **Scene:** The base container class for all content in the scene graph.

# SCENE GRAPH



# JAVAFX APPLICATION

- A JavaFX application is an instance of class **Application**

```
public abstract class Application extends Object;
```

- Creation of a new object of class Application is done by executing the static method **launch()** from the class **Application**:

```
public static void Launch(String... args);
```

- **args application parameters**(parameters of the method *main*).

- JavaFX runtime executes the following steps:

1. Create a new object Application
2. Call the **method init** of the new object Application
3. Call the **method start** of the new object Application
4. Wait for the application to terminate

# JAVAFX APPLICATION

- Note that the start method is abstract and must be overridden.
- The init and stop methods have concrete implementations that do nothing.
- Application parameters can be obtained by **calling *getParameters()* method** from the init() method, or any time after the init method has been called.

# JAVAFX APPLICATION THREAD

- JavaFX creates an **application thread** for running the application start method, processing input events, and running animation timelines.
- Creation of JavaFX Scene and Stage objects as well as modification of scene graph operations to live objects (those objects already attached to a scene) must be done on the **JavaFX application thread**.
- The Java launcher loads and initializes the specified Application class on the **JavaFX application thread**.
  - If there is no main method in the Application class, or if the main method calls Application.launch(), then an instance of the Application is then constructed on the **JavaFX application thread**.
- The init method is called on the launcher thread, not on the **JavaFX application thread**.
  - This means that an application must not construct a Scene or a Stage in the init method. An application may construct other JavaFX objects in the init method.

# JAVAFX APPLICATION

```
public class Main extends Application {  
  
    @Override  
    public void start(Stage stage) {  
        Group root = new Group();  
        Scene scene = new Scene(root, 500, 500, Color.PINK);  
        stage.setTitle("Welcome to JavaFX!");  
  
        stage.setScene(scene);  
        stage.show();  
    }  
  
    public static void main(String[] args) {  
        Launch(args); //an object Application is created  
    }  
}
```

# ADDING NODES

*// A node of type Group is created*

```
Group group = new Group();
```

*// A node of type Rectangle is created*

```
Rectangle r = new Rectangle(25,25,50,50);
r.setFill(Color.BLUE);
group.getChildren().add(r);
```

*// A node of type Circle is created*

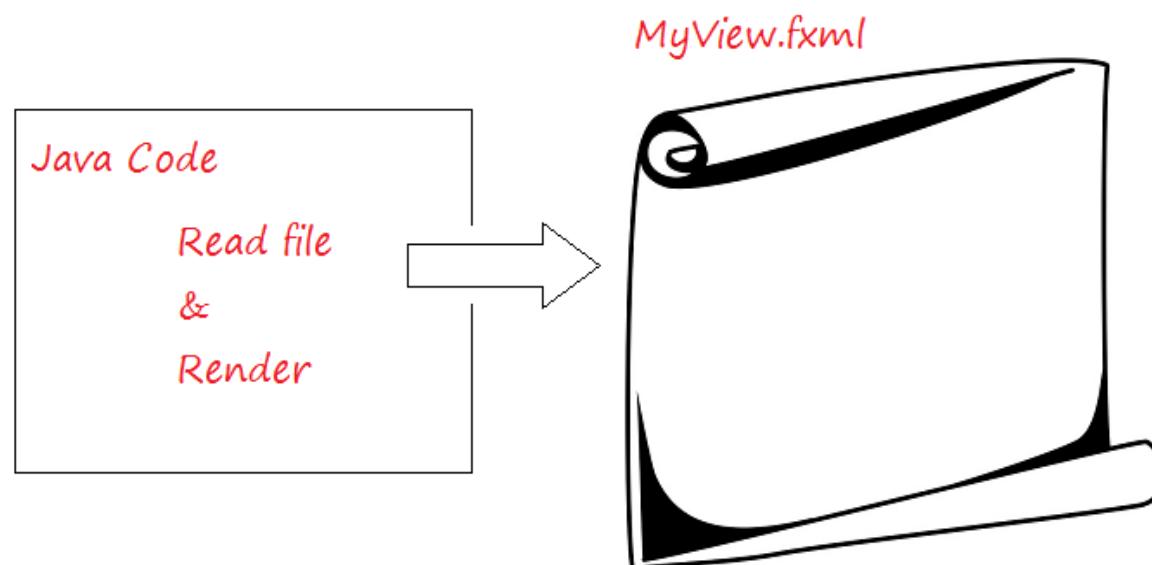
```
Circle c = new Circle(200,200,50, Color.web("blue", 0.5f));
group.getChildren().add(c);
```

# JAVAFX APPLICATION

**DEMO**

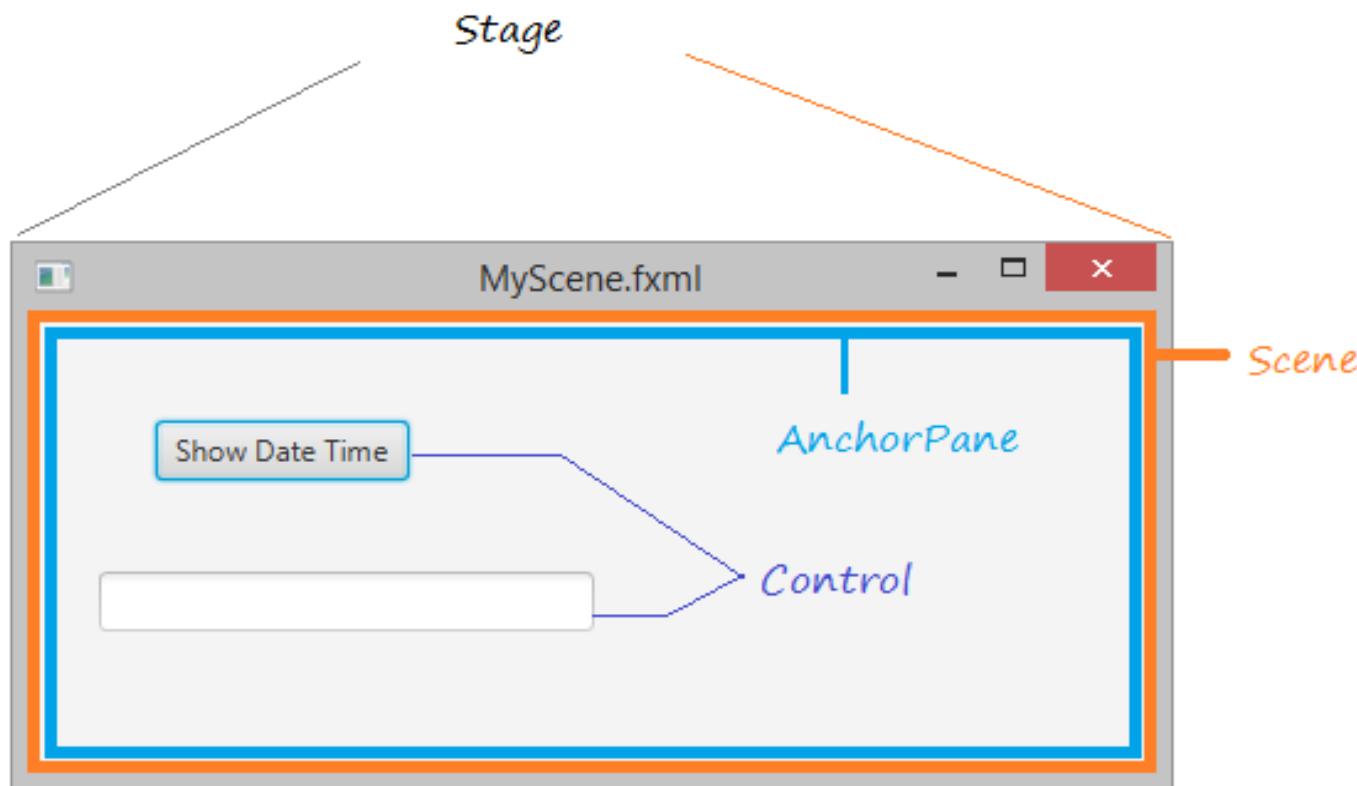
# JAVAFX SCENE BUILDER

- .to create a JavaFX application interface, you can write code in Java entirely.
- .it takes so much time to do this.
  
- .JavaFX Scene Builder is a visual tool allowing you to design the interface of Scene.
- .The code which is generated, is XML code saved on \*.fxml file.



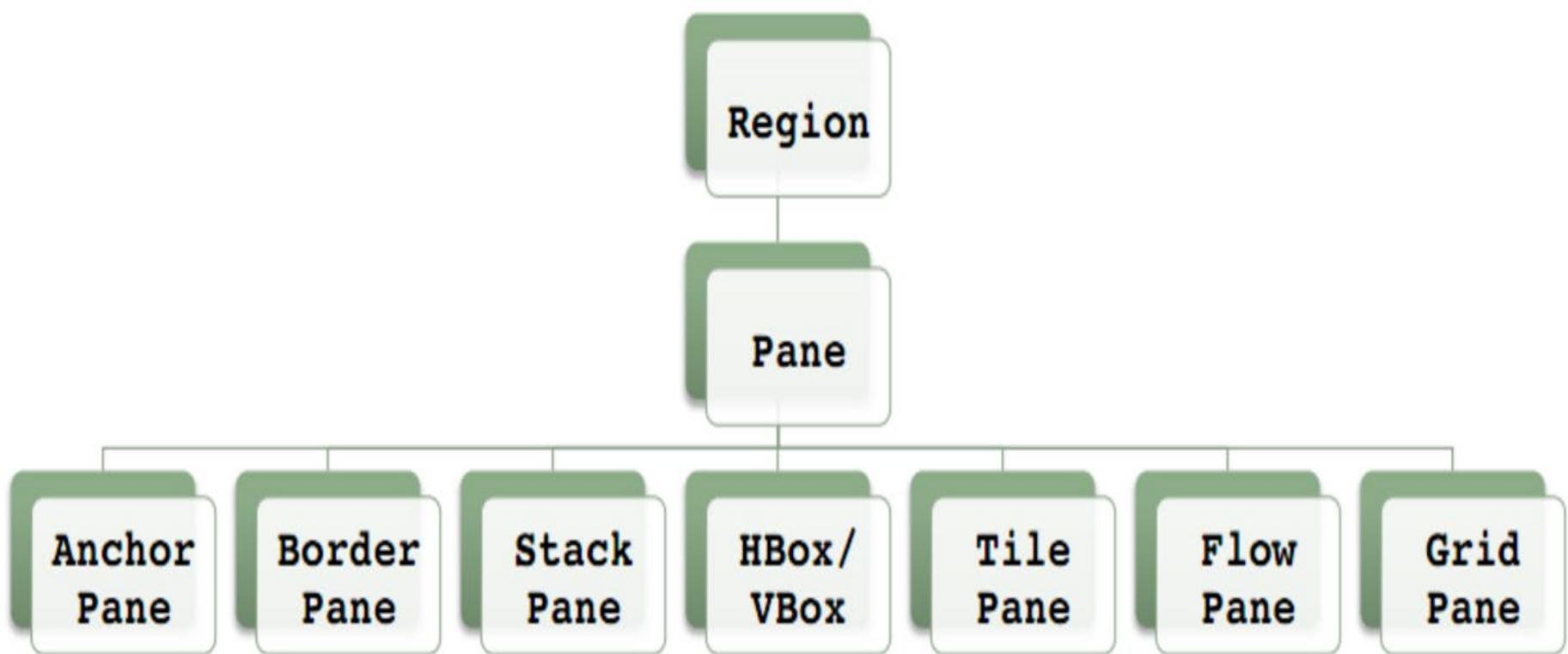
# JAVAFX SCENE BUILDER

## • DEMO

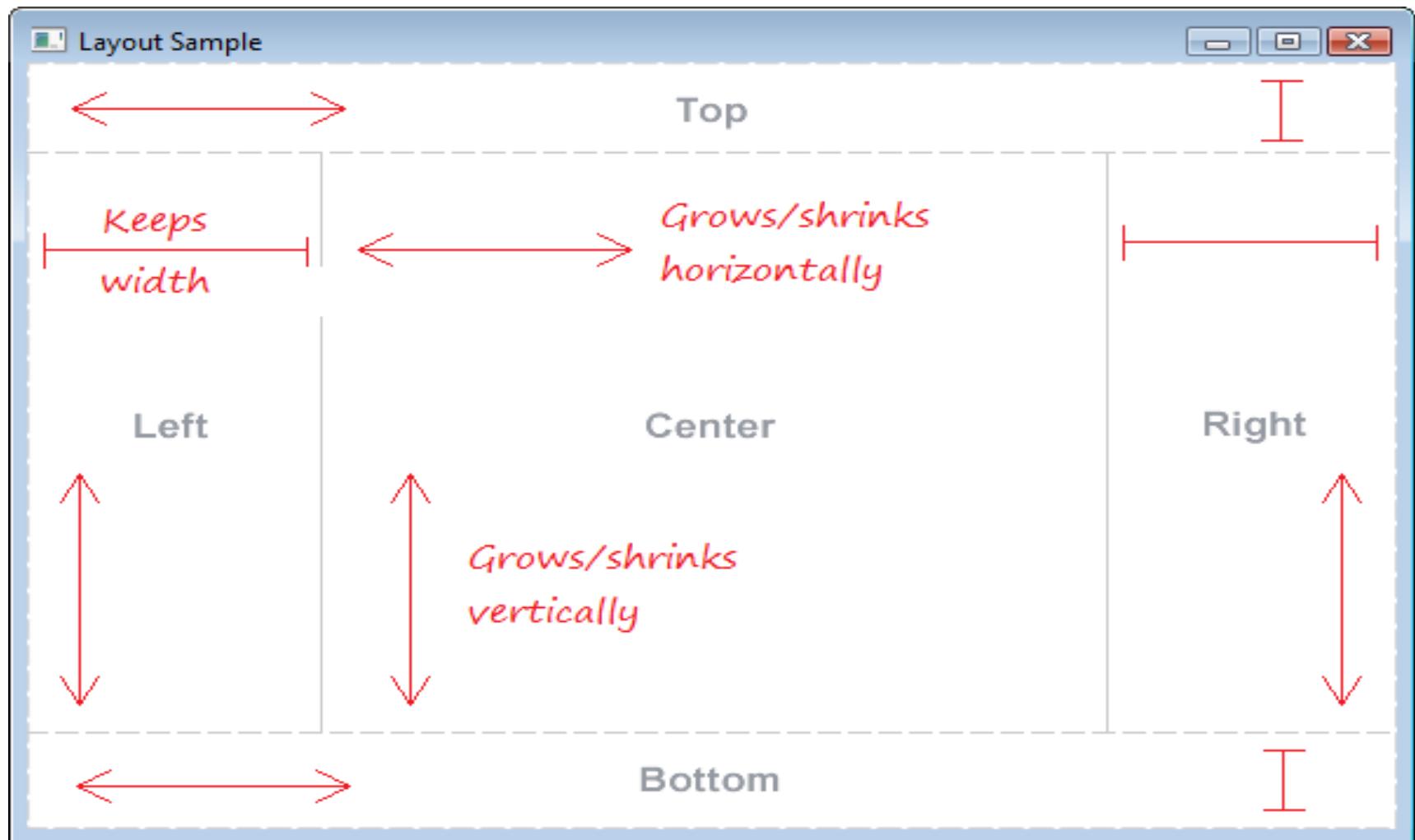


# LAYOUT MANAGEMENT

.layouts are components which contains other components inside them and manage the nested components



# LAYOUT MANAGEMENT - BORDERPANE

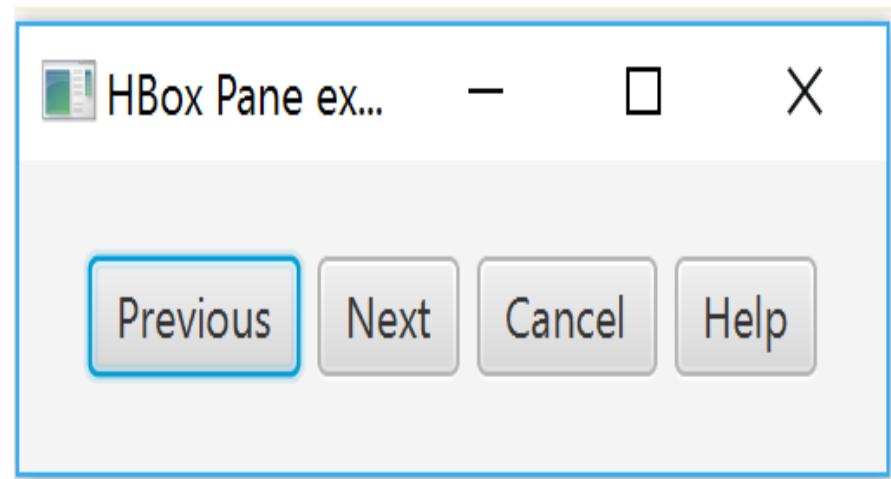


# LAYOUT MANAGEMENT - HBOX

```
HBox root = new HBox(5);
root.setPadding(new Insets(100));
root.setAlignment(Pos.BASELINE_RIGHT);

Button prevBtn = new Button("Previous");
Button nextBtn = new Button("Next");
Button cancBtn = new Button("Cancel");
Button helpBtn = new Button("Help");

root.getChildren().addAll(prevBtn,
nextBtn, cancBtn, helpBtn);
```

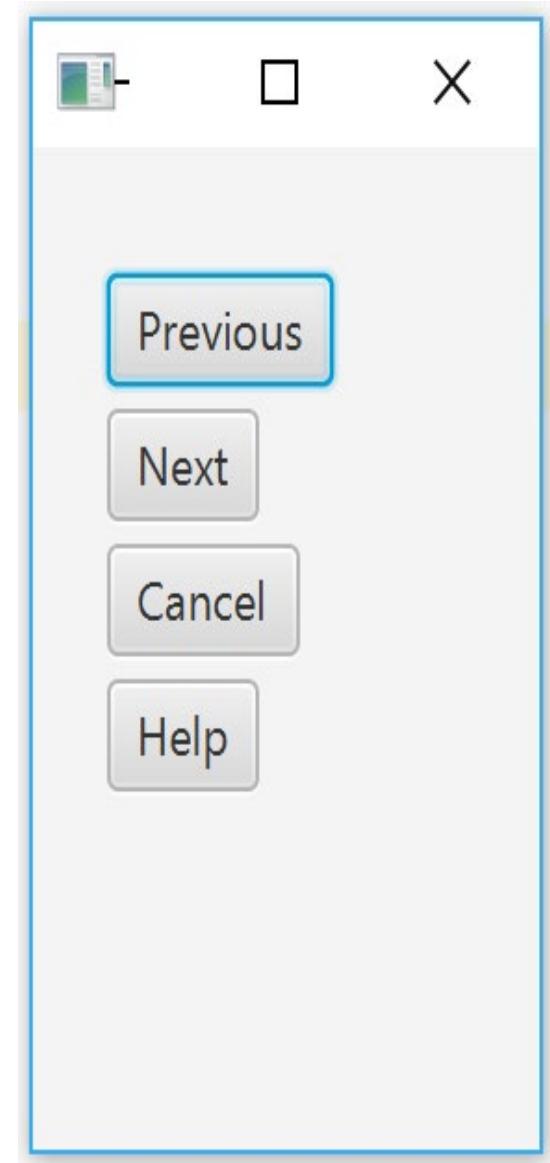


# LAYOUT MANAGEMENT - VBOX

```
VBox root = new VBox(5);
root.setPadding(new Insets(20));
root.setAlignment(Pos.BASELINE_LEFT);

Button prevBtn = new Button("Previous");
Button nextBtn = new Button("Next");
Button cancBtn = new Button("Cancel");
Button helpBtn = new Button("Help");

root.getChildren().addAll(prevBtn,
nextBtn, cancBtn, helpBtn);
Scene scene = new Scene(root, 150, 200);
```



# LAYOUT MANAGEMENT – ANCHORPANE

```
AnchorPane root = new AnchorPane();

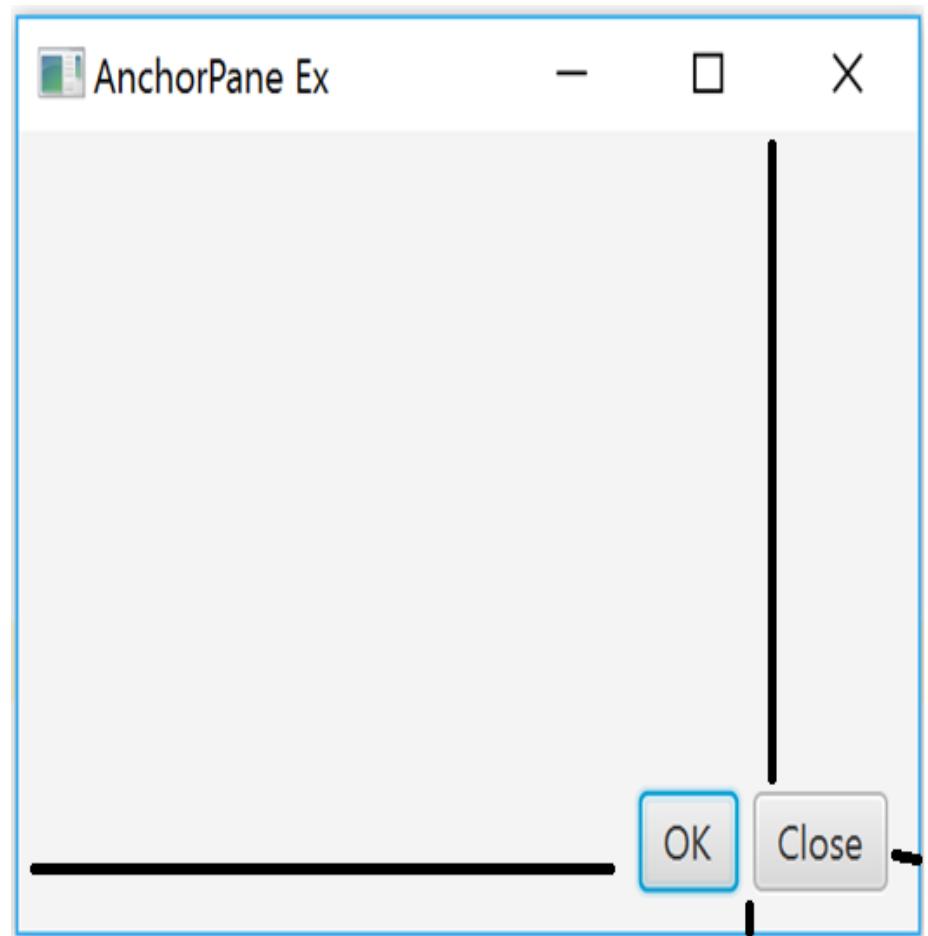
Button okBtn = new Button("OK");
Button closeBtn = new Button("Close");
HBox hbox = new HBox(5, okBtn, closeBtn);

root.getChildren().addAll(hbox);

AnchorPane.setRightAnchor(hbox, 10d);
AnchorPane.setBottomAnchor(hbox, 10d);

Scene scene = new Scene(root, 300, 200);

stage.setTitle("AnchorPane Ex");
stage.setScene(scene);
stage.show();
```



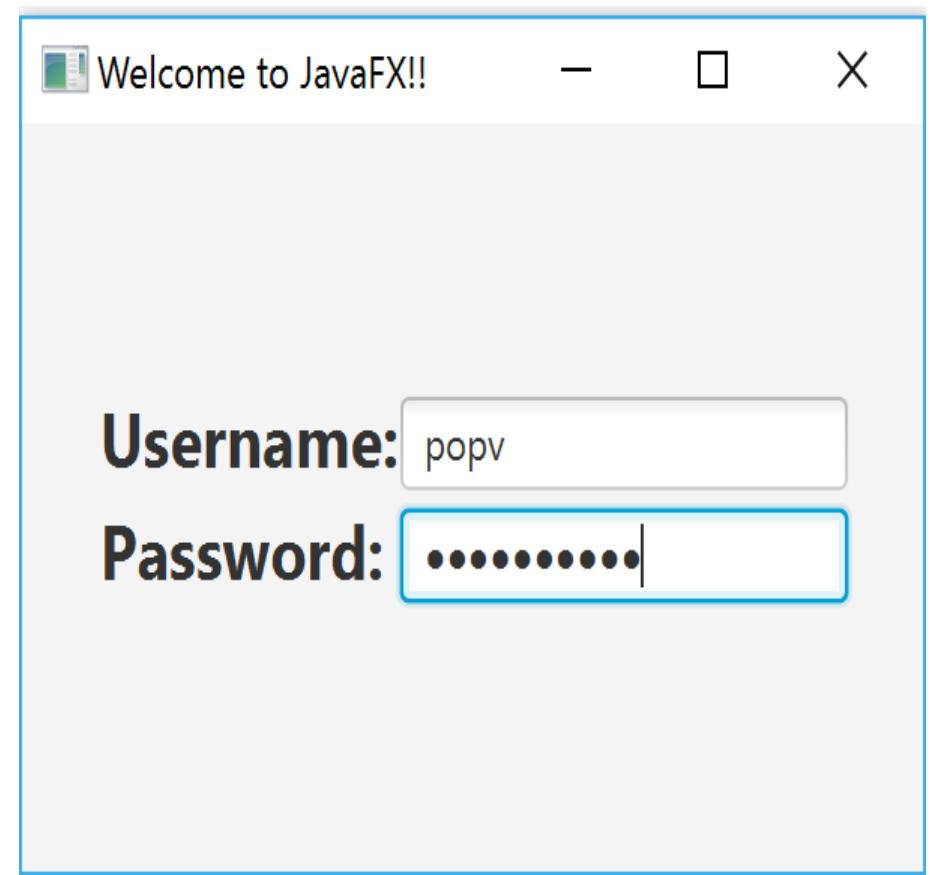
# LAYOUT MANAGEMENT - GRIDPANE

```
GridPane gr=new GridPane();
gr.setPadding(new Insets(20));
gr.setAlignment(Pos.CENTER);

gr.add(createLabel("Username:"),0,0);
gr.add(createLabel("Password:"),0,1);

gr.add(new TextField(),1,0);
gr.add(new PasswordField(),1,1);

Scene scene = new Scene(gr, 300, 200);
stage.setTitle("Welcome to JavaFX!!");
stage.setScene(scene);
stage.show();
```



# JAVAFX CONTROLS

.Are the main components of the GUI

.A control is a node in the scene graph

.Can be manipulated by the user

.Java FX Controls reference:

[https://docs.oracle.com/javase/8/javafx/user-interface-tutorial/ui\\_controls.htm#JFXUI336](https://docs.oracle.com/javase/8/javafx/user-interface-tutorial/ui_controls.htm#JFXUI336)

[Button](#)

[Radio Button](#)

[Toggle Button](#)

[Checkbox](#)

[Choice Box](#)

[Text FieldLabel](#)

[Password Field](#)

[ScrollBar](#)

[ScrollPane](#)

[List View](#)

[Table View](#)

[Tree View](#)

[Combo Box](#)

[Separator](#)

[Slider](#)

[Progress Bar and Progress](#)

[Indicator](#)

[Hyperlink](#)

[Tooltip](#)

[HTML Editor](#)

[Titled Pane and Accordion](#)

[Menu](#)

[Color Picker](#)

[Pagination Control](#)

[File Chooser](#)

[Customization of UI Controls](#)

# EVENT DRIVEN PROGRAMMING

**Event:** Any user action generates an event:

- pressing or releasing the keyboard,
- moving the mouse,
- pressing or releasing a button mouse,
- opening or closing a window,
- performing a mouse click on a component of the interface,
- entering / leaving the mouse cursor in/out of a component area
- ...

- There are also events that are not generated by the application user.
- An event can be treated by executing a program module.

# EVENTS HANDLING

## .Delegation Event Model

We can distinguish three categories of objects used to handle events:

- .Events Sources** - those objects that generate the events;
- .Events** -which are objects (generated by sources and received by consumers)
- .Events consumers or listeners** - those objects that receive and treat the events.

# EVENTS HANDLING

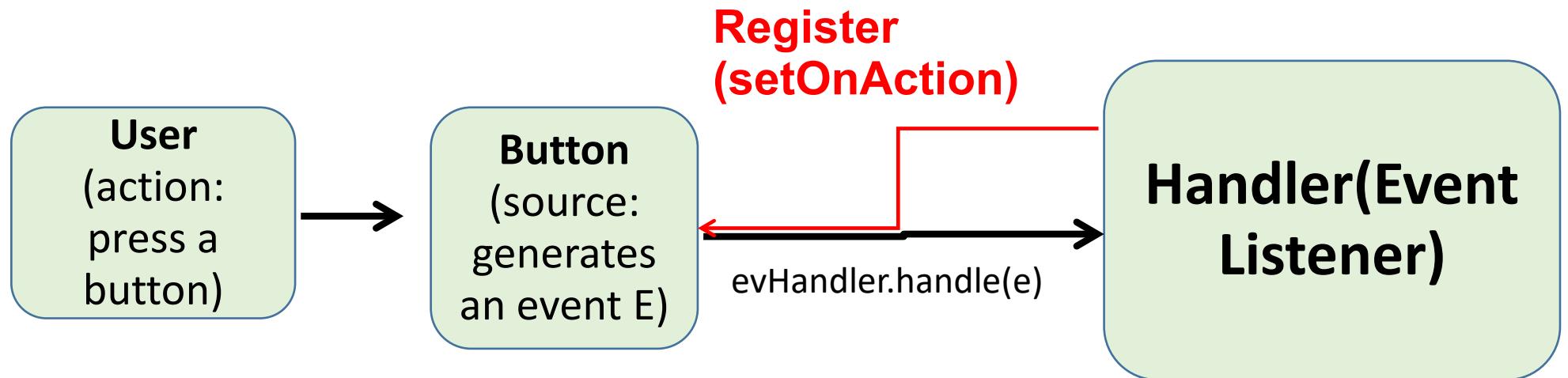


Each consumer must be registered with the event source. This process ensures that the source knows all the consumers to which must submit its generated events.

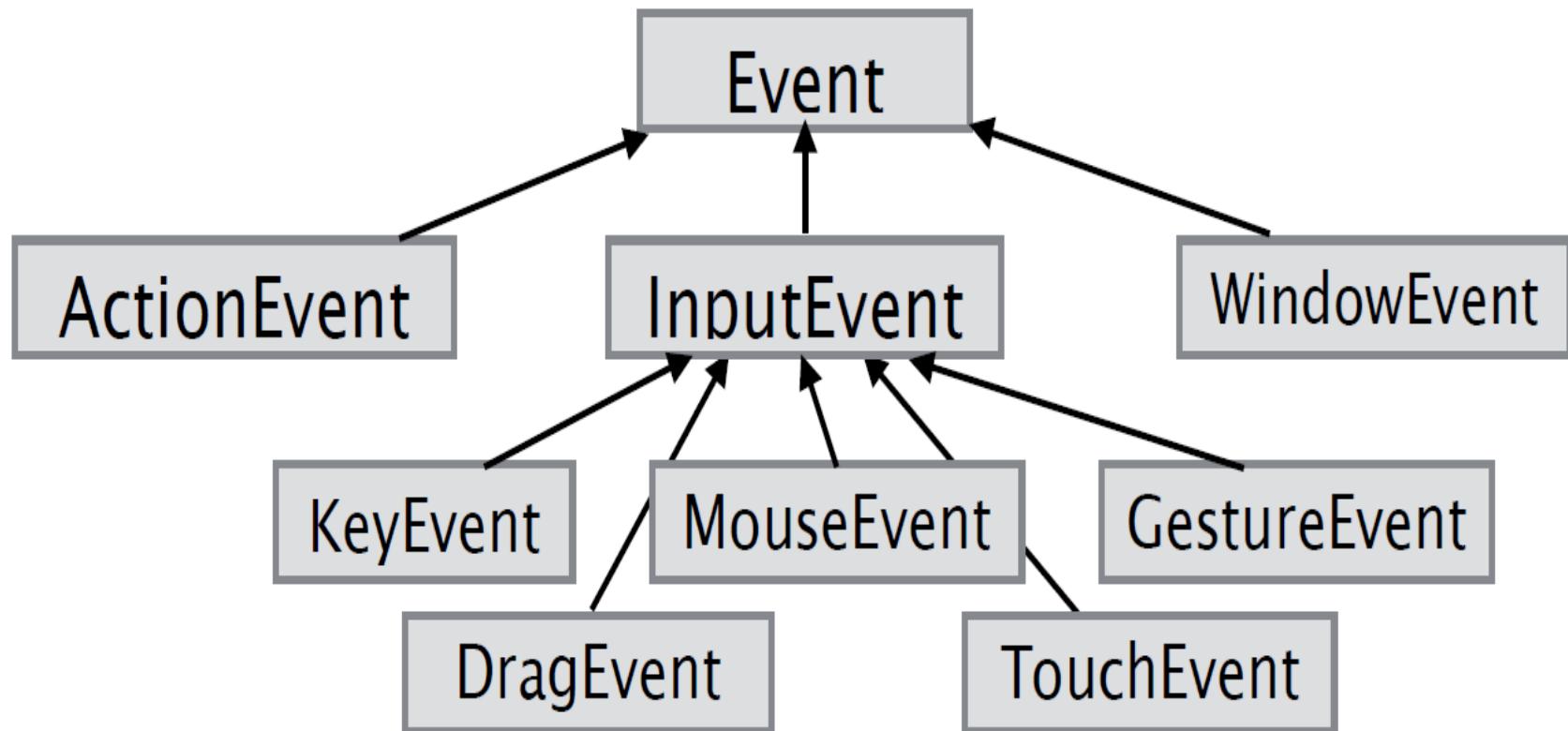
The "delegation" assumes that an event source (an object) transmits all its generated events to those consumers which were recorded at it.

A consumer receives events only from those sources to which it have been registered !!!

# EVENTS HANDLING



# TYPES OF EVENTS



# EVENT HANDLER

```
@FunctionalInterface  
Interface EventHandler<T extends Event> extends  
EventListener{  
    void handle(T event);  
}
```

# EVENT HANDLER

## Button Events

Only one handler can be associated to the button clicked event!!!

```
Button btn = new Button("Ding!");  
btn.setStyle("-fx-font: 42 arial; -fx-base: #f8e7f9;");  
// handle the button clicked event  
btn.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent e) {  
        System.out.println("Hello World!");  
    }  
});
```

Or using lambda expressions:

```
btn.setOnAction(e->System.out.println("Hello World!"));
```

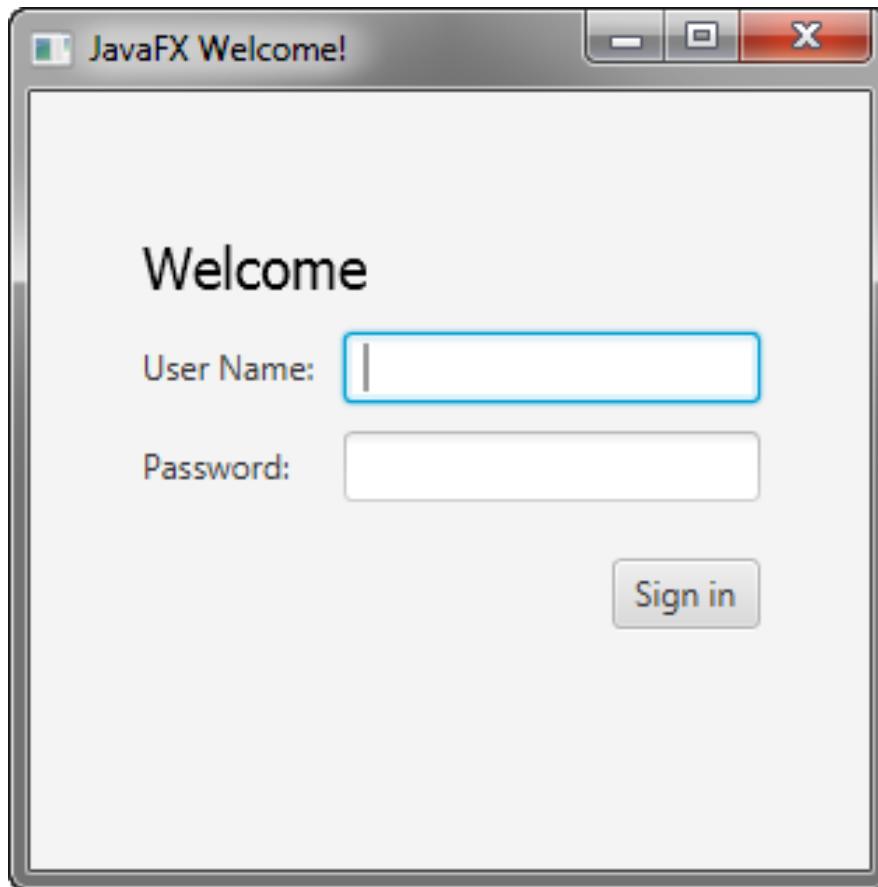
**See HelloWord.zip**

# A SIMPLE APPLICATION

We follow the Oracle tutorial to create manually (without SceneBuilder) a simple form JavaFx application

**See Ex1-NoSceneBuilder.zip**

.



## CREATE A GRIDPANE WITH GAP AND PADDING PROPERTIES

```
public void start(Stage primaryStage) {  
  
    Try {  
  
        primaryStage.setTitle("JavaFX Welcome");  
  
        GridPane grid = new GridPane();  
  
        grid.setAlignment(Pos.CENTER);  
  
        grid.setHgap(10);  
  
        grid.setVgap(10);  
  
        grid.setPadding(new Insets(25, 25, 25, 25));  
  
        .....  
  
        Scene scene = new Scene(grid,400,400);  
  
        scene.getStylesheets().add(getClass().getResource("application.cs  
s").toExternalForm());  
  
        primaryStage.setScene(scene);  
  
        primaryStage.show();  
  
    } catch(Exception e) {e.printStackTrace();}  
}
```

## ADD TEXT, LABELS, AND TEXT FIELDS

```
Text scenetitle = new Text("Welcome");
scenetitle.setFont(Font.font("Tahoma", FontWeight.NORMAL, 20));
grid.add(scenetitle, 0, 0, 2, 1);
```

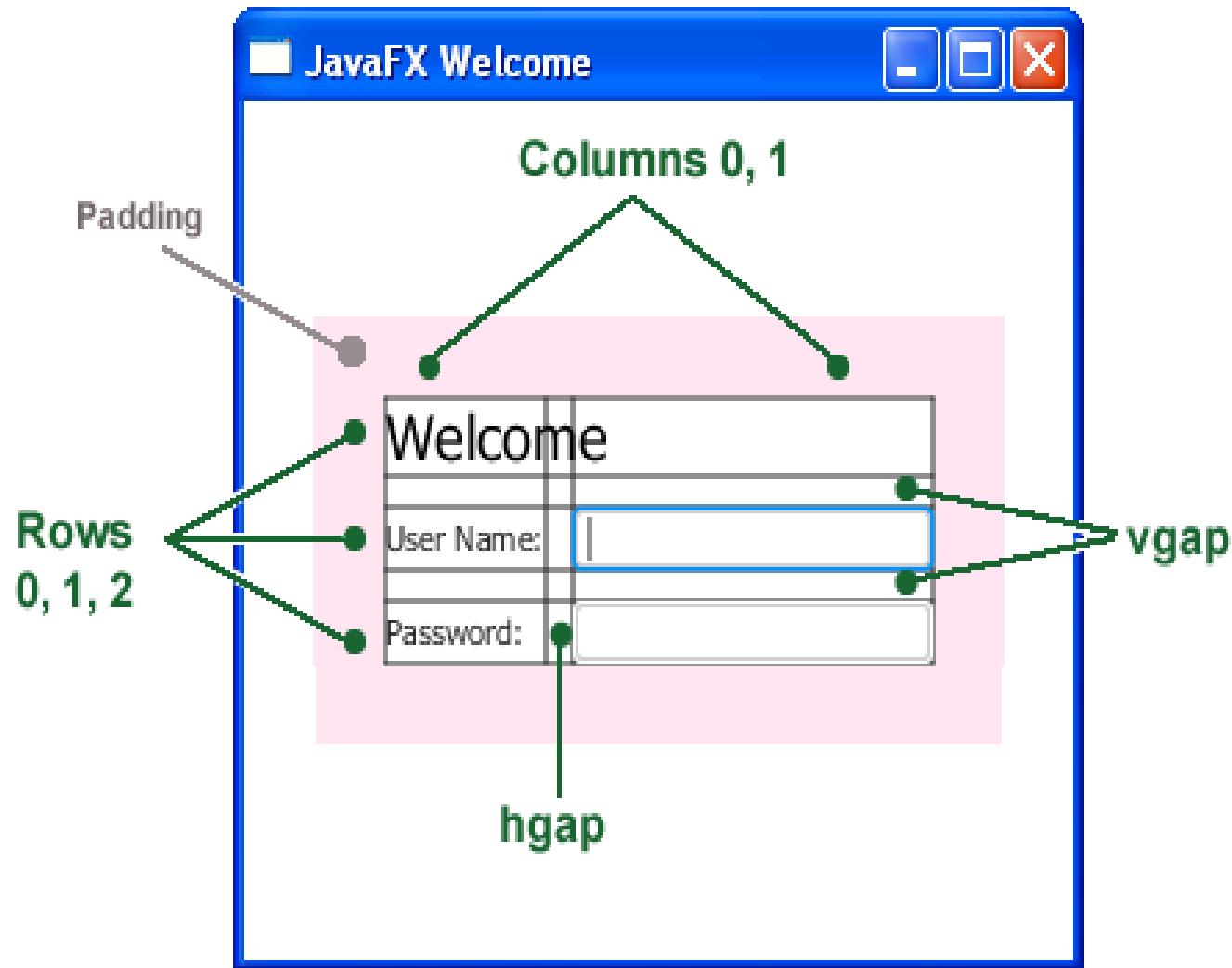
```
Label userName = new Label("User Name:");
grid.add(userName, 0, 1);
```

```
TextField userTextField = new TextField();
grid.add(userTextField, 1, 1);
```

```
Label pw = new Label("Password:");
grid.add(pw, 0, 2);
```

```
PasswordField pwBox = new PasswordField();
grid.add(pwBox, 1, 2);
```

# ADD TEXT, LABELS, AND TEXT FIELDS



# ADD A BUTTON AND TEXT

```
Button btn = new Button("Sign in");

HBox hbBtn = new HBox(10);

hbBtn.setAlignment(Pos.BOTTOM_RIGHT);

hbBtn.getChildren().add(btn);

grid.add(hbBtn, 1, 4);

final Text actiontarget = new Text();

grid.add(actiontarget, 1, 6);
```



# STYLLING A BUTTON

```
btn.setStyle("-fx-font: 22 arial; -fx-base: #b6e7c9;");
```

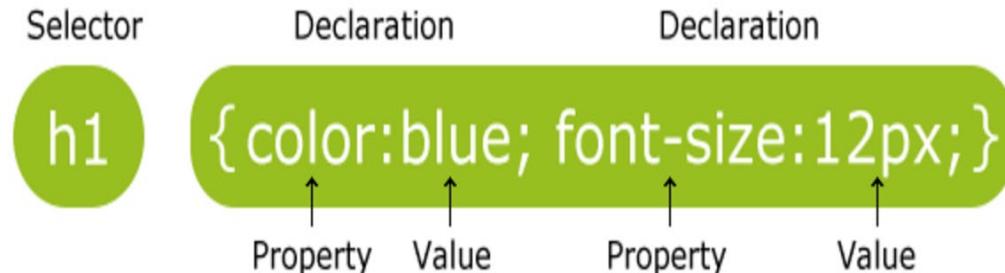


# ADD A CASCADING STYLE SHEET (CSS)

- Create a .css file and apply the new styles on the previous example
- By switching to CSS over inline styles, we separate the design from the content.
- This approach makes it easier for a designer to have control over the style without having to modify content.
- JavaFx CSS Reference Guide: <https://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html>
- First: Initialize the stylesheets Variable

```
scene.getStylesheets().add(getClass().getResource("application.css")
).toExternalForm());
```

# CSS



<http://www.w3schools.com/css/>

```
.button {  
    -fx-padding: 5 22 5 22;  
    -fx-border-color: #e2e2e2;  
    -fx-border-width: 2;  
    -fx-background-radius: 0;  
    -fx-background-color: #1e2e2e2;  
    -fx-font-family: "Segoe UI", Helvetica, Arial,  
    sans-serif;  
    -fx-font-size: 11pt;  
    -fx-text-fill: black;  
    -fx-background-insets: 0 0 0 0, 0, 1, 2;  
}
```

```
.button:hover {  
    -fx-background-color: #3a3a3a;  
}
```

```
.background {  
    -fx-background-color:  
    #1d1d1d;  
}  
  
.label {  
    -fx-font-size: 11pt;  
    -fx-font-family: "Segoe UI  
    Semibold";  
    -fx-text-fill: white;  
    -fx-opacity: 0.6;  
}
```

# ADD A BACKGROUND IMAGE

- The background image is applied to the .root style, which means it is applied to the root node of the Scene instance.
- The style definition consists of the name of the property (-fx-background-image) and the value for the property (url("backgr.jpg")).

```
.root {  
    -fx-background-image: url("backgr.jpg");  
}
```



## STYLE THE LABELS

.the .label style class will affect all labels in the form

```
.label {  
    -fx-font-size: 12px;  
    -fx-font-weight: bold;  
    -fx-text-fill: #333333;  
    -fx-effect: dropshadow( gaussian , rgba(255,255,255,0.5) ,  
0,0,0,1 );  
}
```

# STYLE TEXT

- .Remove code that define the inline styles

```
// scenetitle.setFont(Font.font("Tahoma",  
FontWeight.NORMAL, 20));  
  
// btn.setStyle("-fx-font: 22 arial; -fx-base:  
#b6e7c9;");  
  
// actiontarget.setFill(Color.FIREBRICK);
```

- .Create an ID for each text node by using the setId() method of the Node class

```
scenetitle.setId("welcome-text");  
  
actiontarget.setId("actiontarget");
```

# STYLE TEXT

• Add to CSS file

```
#welcome-text {  
    -fx-font-size: 32px;  
    -fx-font-family: "Arial Black";  
    -fx-fill: #818181;  
    -fx-effect: innershadow( three-pass-box , rgba(0,0,0,0.7) , 6,  
    0.0 , 0 , 2 );  
}  
  
#actiontarget {  
    -fx-fill: FIREBRICK;  
    -fx-font-weight: bold;  
    -fx-effect: dropshadow( gaussian , rgba(255,255,255,0.5) ,  
    0,0,0,1 );  
}
```

# STYLE TEXT

- Text with shadow effects



# STYLE THE BUTTON

.Initial state

```
.button {  
    -fx-text-fill: white;  
    -fx-font-family: "Arial Narrow";  
    -fx-font-weight: bold;  
    -fx-background-color: linear-gradient(#61a2b1, #2A5058);  
    -fx-effect: dropshadow( three-pass-box , rgba(0,0,0,0.6) , 5,  
0.0 , 0 , 1 );  
}
```

.Hover state

```
.button:hover {  
    -fx-background-color: linear-gradient(#2A5058, #61a2b1);  
}
```



# **Advanced Programming Methods**

**Lecture 10 - JavaFX(continuation)**

## AN EXAMPLE BASED ON FXML

- we use FXML to create the same login user interface,
  - We separate the application design from the application logic,
  - it makes the code easier to maintain.
- .See Ex2-FXML.zip**

# AN EXAMPLE BASED ON FXML

- FXMLLoader class is responsible for loading the FXML source file and returning the resulting object graph.

```
public class Main extends Application {  
  
    @Override  
  
    public void start(Stage primaryStage) {  
  
        try {  
  
            GridPane root =  
                (GridPane)FXMLLoader.load(getClass().getResource("Ex2.fxml"));  
  
            Scene scene = new Scene(root,400,400);  
  
            scene.getStylesheets().add(getClass().getResource("application.css").toExternalForm());  
  
            primaryStage.setTitle("FXML Welcome");  
  
            primaryStage.setScene(scene);  
  
            primaryStage.show();  
  
        } catch(Exception e) { e.printStackTrace();}  
  
    }  
  
    public static void main(String[] args) { launch(args);}  
  
}
```

# FXML FILE

- the GridPane layout is the root element of the FXML document and has two attributes:
  - The fx:controller attribute is required when you specify controller-based event handlers in your markup.
  - The xmlns:fx attribute is always required and specifies the fx namespace.
- The remainder of the code controls the alignment and spacing of the grid pane.

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.Insets?>

<?import javafx.scene.layout.GridPane?>

<GridPane alignment="CENTER" gridLinesVisible="true" hgap="10.0"
prefHeight="292.0" prefWidth="364.0" vgap="10.0"
xmlns:fx="http://javafx.com/fxml/1" xmlns="http://javafx.com/javafx/8.0.141"
fx:controller="application.Ex2Controller">

    <padding>
        <Insets bottom="10.0" left="25.0" right="25.0" top="25.0" />
    </padding>
</GridPane>
```

# TEXT, LABEL, TEXTFIELD, AND PASSWORD FIELD CONTROLS

```
<children>
```

```
    <Text strokeWidth="0.0" text="Welcome"  
wrappingWidth="63.576171875" />  
  
    <Label text="User Name" GridPane.rowIndex="1" />  
  
    <Label text="Password" GridPane.rowIndex="2" />  
  
    <TextField GridPane.columnIndex="1" GridPane.rowIndex="1" />  
  
    <PasswordField fx:id="passwordField" GridPane.columnIndex="1"  
GridPane.rowIndex="2" />  
  
</children>
```

# ADD A BUTTON AND TEXT

```
<HBox alignment="BOTTOM_RIGHT" prefHeight="100.0" prefWidth="200.0" spacing="10.0"
GridPane.columnIndex="1" GridPane.rowIndex="4">

    <children>
        <Button mnemonicParsing="false" onAction="#handleSubmitButtonAction"
prefHeight="28.0" prefWidth="81.0" text="Sign In" />
    </children>

</HBox>

<Text fx:id="actionTarget" strokeType="OUTSIDE" strokeWidth="0.0"
GridPane.columnSpan="2" GridPane.rowIndex="6" />
```

# ADD CODE TO HANDLE AN EVENT

.The controller has to be created (either a skeleton from SceneBuilder or from fxml file using the option source/generate controller) to handle the events

```
package application;

import javafx.fxml.FXML;

import javafx.scene.text.Text;

import javafx.event.ActionEvent;

import javafx.scene.control.PasswordField;

public class Ex2Controller {

    @FXML

    private PasswordField passwordField;

    @FXML

    private Text actionTarget;

    // Event Listener on Button.onAction

    @FXML

    public void handleSubmitButtonAction(ActionEvent event) {

        actionTarget.setText("Sign in button pressed");

    }

}
```

# STYLE THE APPLICATION WITH CSS

- Inside fxml file we can set the previous css file to obtain the same style

```
<GridPane alignment="CENTER" hgap="10.0" prefHeight="217.0"
prefWidth="300.0" styleClass="root" stylesheets="@application.css"
vgap="10.0" xmlns:fx="http://javafx.com/fxml/1"
xmlns="http://javafx.com/javafx/8.0.141"
fx:controller="application.Ex2Controller">
```

# **FXML**

**FXML** = is a scriptable, XML-based markup language for constructing Java object graphs

- .declarative approach (versus programmatic)
- .Tree of components
- .Role separation
- .Language independent (Java, Scala, Clojure, etc.)
- .Support for internationalization

# XML

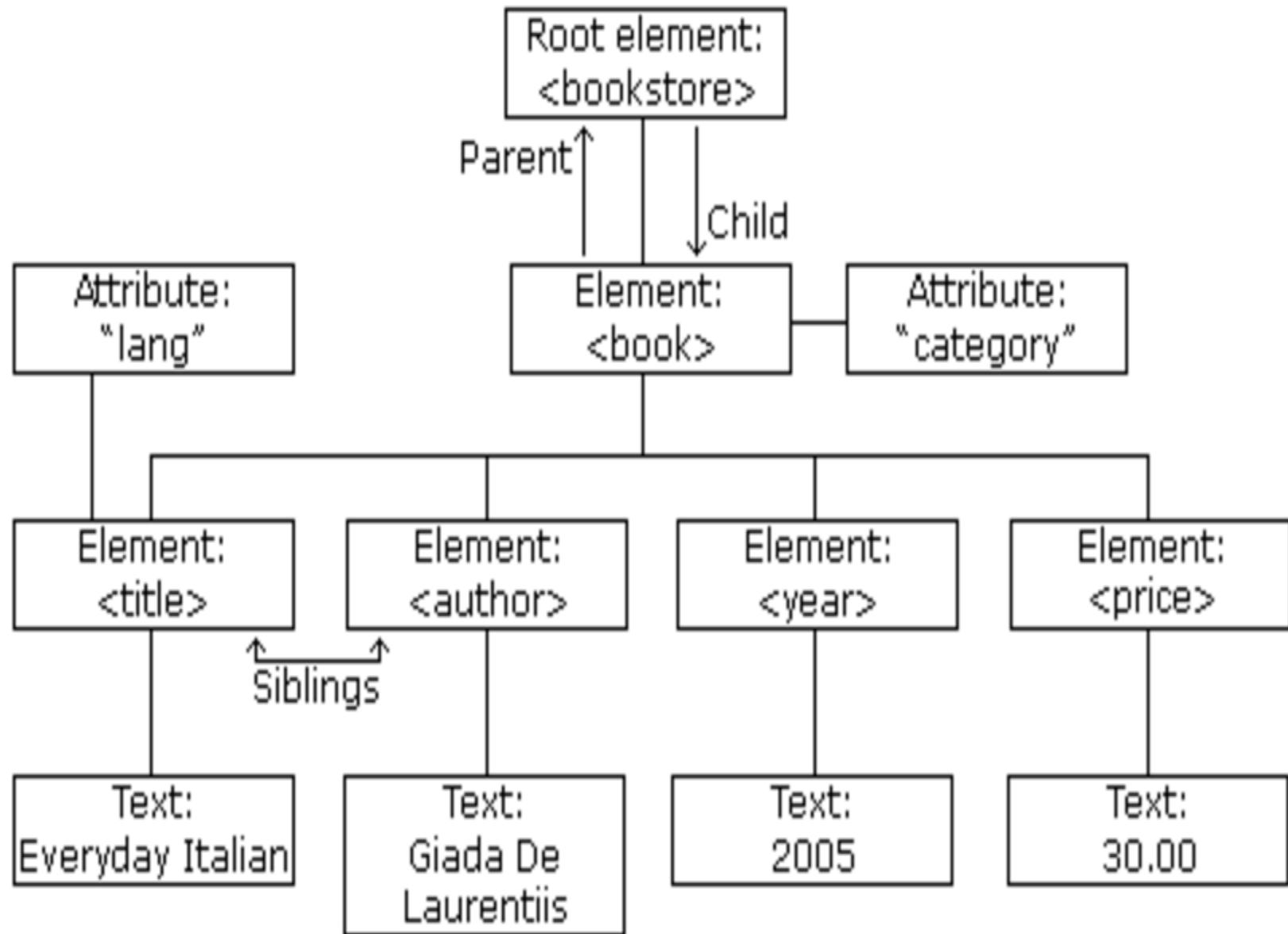
XML stands for EXtensible Markup Language.

XML was designed to store and transport data.

XML was designed to be self-descriptive- human- and machine-readable.

```
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget!</body>
</note>
```

# XML TREE STRUCTURE



# EXAMPLE

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J. K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

# FXML

From a Model View Controller (MVC) perspective:

- . the FXML file that contains the description of the user interface is the view.
- .The controller is a Java class (optionally implementing the Initializable class), which is declared as the controller for the FXML file.
- .The model consists of domain objects, defined on the Java side, that you connect to the view through the controller.

# FXML

- .does not have a schema, but it does have a basic predefined structure.
- .maps directly to Java
- .most JavaFX classes can be used as elements
- .most properties can be used as attributes

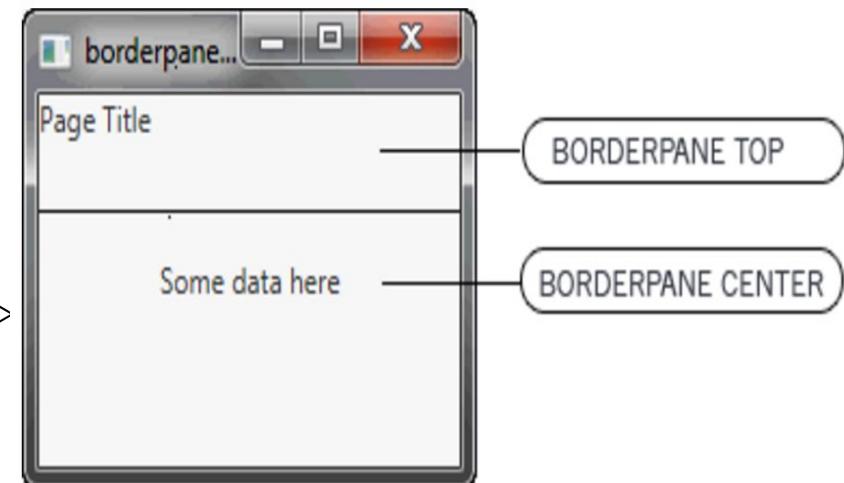
# PROGRAMMATIC VS. DECLARATIVE

## Programmatic

```
BorderPane border = new BorderPane();  
  
Label top = new Label("Page Title");  
  
border.setTop(top);  
  
Label center = new Label ("Some data here");  
  
border.setCenter(center);
```

## Declarative

```
<BorderPane>  
  <top>  
    <Label text="Page Title"/>  
  </top>  
  
  <center>  
    <Label text="Some data here"/>  
  </center>  
</BorderPane>
```



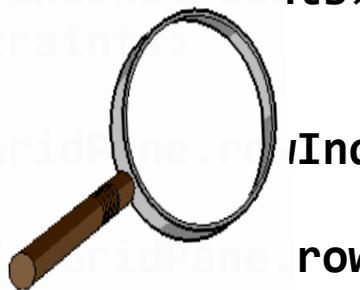
# FXML STRUCTURE

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.geometry.Insets?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.AnchorPane?>
<?import javafx.scene.layout.ColumnConstraints?>
<?import javafx.scene.layout.GridPane?>
<?import javafx.scene.layout.RowConstraints?>

<AnchorPane xmlns="http://javafx.com/javafx/8.0.60" xmlns:fx="http://javafx.com/fxml/1">
    <children>
        <GridPane hgap="5.0" vgap="5.0">
            <columnConstraints>
                <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
                <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
            </columnConstraints>
            <rowConstraints>
                <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
                <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
                <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
            </rowConstraints>
            <children>
                <Button mnemonicParsing="false" prefHeight="25.0" prefWidth="99.0" text="Login"
                    GridPane.rowIndex="2" />
                <Label prefHeight="30.0" prefWidth="98.0" text="Username" />
                <Label prefHeight="40.0" prefWidth="100.0" text="Password" GridPane.rowIndex="1" />
                <TextField GridPane.columnIndex="1" />
                <TextField prefHeight="31.0" prefWidth="100.0" GridPane.columnIndex="1" GridPane.rowIndex="1" />
            </children>
            <padding>
                <Insets bottom="5.0" left="5.0" right="5.0" top="5.0" />
            </padding>
        </GridPane>
    </children>
</AnchorPane>
```

# FXML STRUCTURE

```
<?xml version="1.0" encoding="UTF-8"?>
<AnchorPane>
    <children>
        <GridPane hgap="5.0" vgap="5.0">
            <columnConstraints> </columnConstraints>
            <rowConstraints> </rowConstraints>
            <children>
                <Button text="Login" GridPane.columnIndex="0" GridPane.rowIndex="0" />
                <Label text="Username" GridPane.columnIndex="0" GridPane.rowIndex="1" />
                <Label text="Password" GridPane.columnIndex="1" GridPane.rowIndex="0" />
                <TextField GridPane.columnIndex="1" GridPane.rowIndex="1" />
                <TextField GridPane.columnIndex="1" GridPane.rowIndex="2" />
            </children>
            <padding>
                <Insets bottom="5.0" left="5.0" right="5.0" top="5.0" />
            </padding>
        </GridPane>
    </children>
</AnchorPane>
```



# FXML LOADER

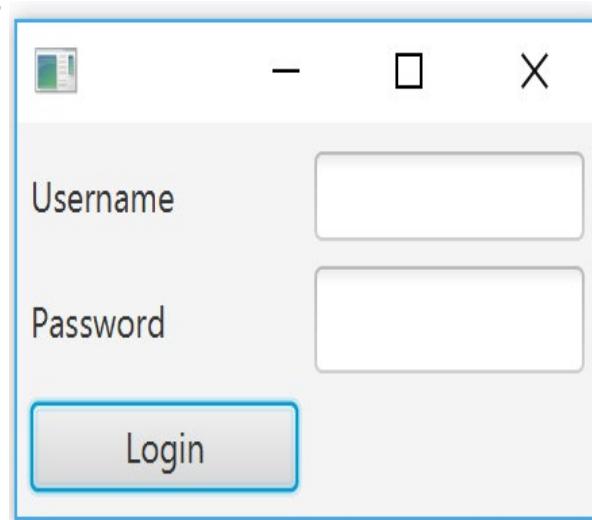
```
public class Main extends Application {  
    public static void main(String[] args) {  
        launch(args);  
    }  
    @Override  
    public void start(Stage primaryStage) {  
        try {  
            //Load root layout from fxml file.  
            FXMLLoader loader=new FXMLLoader();  
            // Loads an object hierarchy from an XML document.  
            loader.setLocation(Main.class.getResource("View.fxml"));  
            //Finds a resource with a given name -> URL.  
            AnchorPane rootLayout= (AnchorPane) loader.load();  
            // Show the scene containing the root layout.  
            Scene scene = new Scene(rootLayout);  
            primaryStage.setScene(scene);  
            primaryStage.show();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

# EXAMPLE

View.fxml

```
public class Main extends Application {  
    public static void main(String[] args) {  
        launch(args);  
    }  
    @Override  
    public void start(Stage primaryStage) {  
        try {  
            //Load root Layout from fxml file.  
            FXMLLoader loader=new FXMLLoader();  
  
loader.setLocation(Main.class.getResource("View.fxml"));  
//URL  
        AnchorPane rootLayout= (AnchorPane)  
loader.load();  
  
        // Show the scene containing the root  
layout.  
        Scene scene = new Scene(rootLayout);  
primaryStage.setScene(scene);  
primaryStage.show();  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
    }  
}
```

```
<?xml version="1.0" encoding="UTF-8"?>  
<AnchorPane>  
    <children>  
        <GridPane hgap="5.0" vgap="5.0">  
            <columnConstraints> </columnConstraints>  
            <rowConstraints></rowConstraints>  
            <children>  
                <Button text="Login" GridPane.rowIndex="2"  
GridPane.columnIndex= "0" />  
                <Label text="Username" GridPane.rowIndex="0"  
GridPane.columnIndex= "0" />  
                <Label text="Password"  
GridPane.rowIndex="1" GridPane.columnIndex= "0" />  
                <TextField GridPane.columnIndex="1" GridPane.rowIndex="0"  
/>  
                <TextField GridPane.columnIndex="1" GridPane.rowIndex="1"  
/>  
            </children>  
            <padding>  
                <Insets bottom="5.0" left="5.0" right="5.0" top="5.0" />  
            </padding>  
        </GridPane>  
    </children>  
</AnchorPane>
```



# FXML – CONTROLLER

- Define the GUI in the fxml file
- Events are treated in the Controller file. How?
  - Define a file with the name XXXController.java
  - The link to controller in XXX.fxml file is specified as:  
`<AnchorPane fx:controller="XXXController.java">`
  - Define handlers in XXXController.java to treat the events

# FXML – CONTROLLER EXAMPLE

```
<?xml version="1.0" encoding="UTF-8"?>
<AnchorPane
    fx:controller="View2Controller"
>
    <children>
        <GridPane hgap="5.0" vgap="5.0">
            <columnConstraints> </columnConstraints>
            <rowConstraints> </rowConstraints>
            <children>
                <Button text="Login" GridPane.rowIndex="2" GridPane.columnIndex= "0" />
                <Label text="Username" GridPane.rowIndex="0" GridPane.columnIndex= "0" />
                <Label text="Password" GridPane.rowIndex="1" GridPane.columnIndex= "0" />
                <TextField GridPane.columnIndex="1" GridPane.rowIndex="0" />
                <TextField GridPane.columnIndex="1" GridPane.rowIndex="1" />
            </children>
            <padding>
                <Insets bottom="5.0" left="5.0" right="5.0" top="5.0" />
            </padding>
        </GridPane>
    </children>
</AnchorPane>
```

```
public class View2Controller {

    /**
     * Initializes the controller class. This method is
     * automatically called
     * after the fxml file has been loaded.
     */
    @FXML
    private void initialize() {

    }
}
```

# FXML – CONTROLLER- EVENTS TREATMENT

```
<?xml version="1.0" encoding="UTF-8"?>
<AnchorPane fx:controller="View2Controller">
    <children>
        <GridPane hgap="5.0" vgap="5.0">
            <rowConstraints></rowConstraints>
            <children>
                <Button fx:id="buttonLogin" onAction="#handleLogin" text="Login" GridPane.rowIndex="2"
GridPane.columnIndex= "0" />
                <Label text="Username" GridPane.rowIndex="0" GridPane.columnIndex= "0" />
                <Label text="Password" GridPane.rowIndex="1" GridPane.columnIndex= "0" />
                <TextField fx:id="textFieldUsername" GridPane.columnIndex="1" GridPane.rowIndex="0" />
                <TextField fx:id="textFieldPasword" GridPane.columnIndex="1" GridPane.rowIndex="1" />
            </children>
            <padding>
                <Insets bottom="5.0" left="5.0" right="5.0" top="5.0" />
            </padding>
        </GridPane>
    </children>
</AnchorPane>
```

```
public class View2Controller {
    @FXML
    private TextField textFieldUsername;
    @FXML
    private TextField textFieldPasword;

    @FXML
    public void handleLogin() {
        User u=new User(textFieldUsername.getText(),
textFieldPasword.getText());
        //...
    }
}
```

# GETTING THE CONTROLLER OBJECT

```
@Override  
public void start(Stage primaryStage) {  
    try {  
        //Load root Layout from fxml file.  
        FXMLLoader loader=new FXMLLoader();  
        loader.setLocation(Main2.class.getResource("View2.fxml")); //URL  
        AnchorPane rootLayout= (AnchorPane) loader.load();  
  
        View2Controller controller=loader.getController();  
  
        // Show the scene containing the root Layout.  
        Scene scene = new Scene(rootLayout);  
        primaryStage.setScene(scene);  
        primaryStage.show();  
  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

# **ObservableValue<T>**

.Generic interface **ObservableValue<T>** is used to wrap different values type and to provide a mechanism to observe the values changes through notifications

```
public interface ObservableValue<T>extends Observable;
```

.Methods:

```
T getValue(); //get the wrapped value
```

```
// adding/removing a ChangeListener (which is notified when the value changes)
```

```
void addListener(ChangeListener<? super T> listener);
```

```
void removeListener(ChangeListener<? super T> listener);
```

# Property<T>

- .A generic interface that defines the methods common to all properties independent of their type.
- .It implements ObservableValue<T>

- .Examples of implementations:

```
public class SimpleStringProperty extends StringPropertyBase;  
public class SimpleObjectProperty<T> extends ObjectPropertyBase<T>;  
public class SimpleDoubleProperty extends DoublePropertyBase;
```

# Property binding

.JavaFX property binding allows you to synchronize the value of two properties so that whenever one of the properties changes, the value of the other property is updated automatically.

- Unidirectional binding

- `void bind(ObservableValue<? extends T> observable)`

- Bidirectional binding

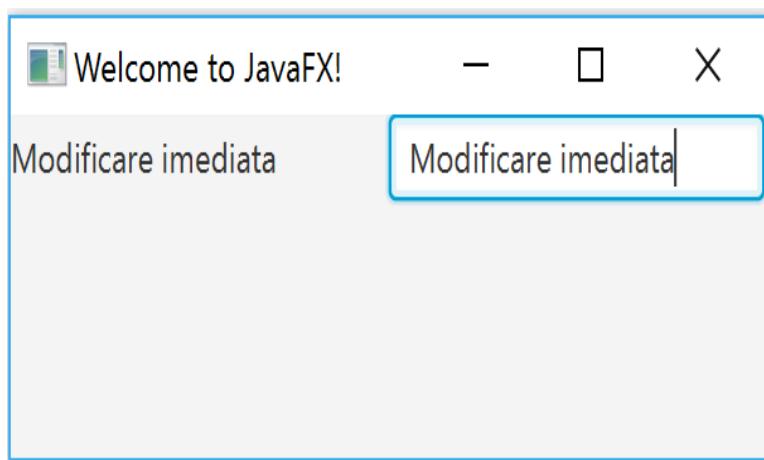
- `void bindBidirectional(Property<T> other)`

# PROPERTY - OBSERVABLE -LISTENER

```
BooleanProperty booleanProperty = new SimpleBooleanProperty(true);
// Add change listener
booleanProperty.addListener(new ChangeListener<Boolean>() {
    @Override
    public void changed(ObservableValue<? extends Boolean> observable,
                        Boolean oldValue, Boolean newValue) {
        System.out.println("changed " + oldValue + "->" + newValue);
        //myFunc();
    }
});
Button btn = new Button();
btn.setText("Switch boolean flag");
btn.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        booleanProperty.set(!booleanProperty.get()); //switch
        System.out.println("Switch to " + booleanProperty.get());
    }
});
// Bind to another property variable
btn.underlineProperty().bind(booleanProperty); //button text is underlined
according to the booleanProperty value
```

# TextField- ChangeListener

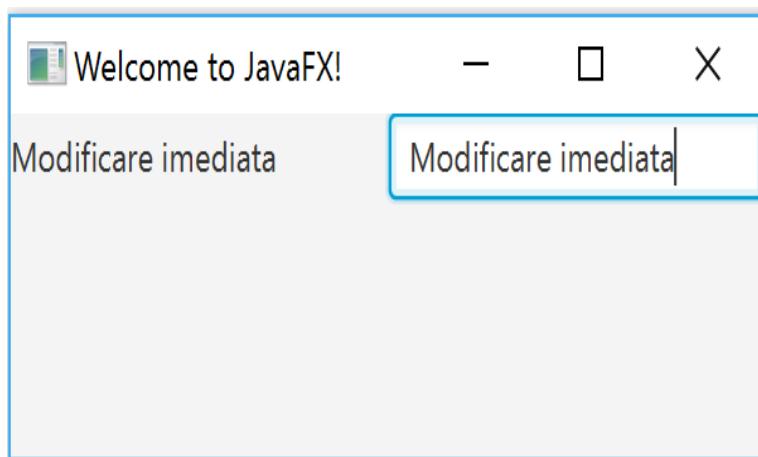
```
Label l=new Label("search item ...");  
l.setPrefWidth(150);  
TextField txt=new TextField();  
gr.add(l,0,0);  
gr.add(txt,1,0);  
  
txt.textProperty().addListener(new ChangeListener<String>() {  
    @Override  
    public void changed(ObservableValue<? extends String> observable,  
String oldValue, String newValue) { l.setText(newValue);  
    }  
});
```



```
//the same effect using key event handler  
txt.setOnKeyPressed(new EventHandler<KeyEvent>() {  
    @Override  
    public void handle(KeyEvent event) {  
        l.setText(txt.getText());  
    }  
}) ;
```

Or

```
//the same effect using the properties binding  
l.textProperty().bindBidirectional(txt.textProperty());  
l.setText("third approach");
```



# OTHER EXAMPLES

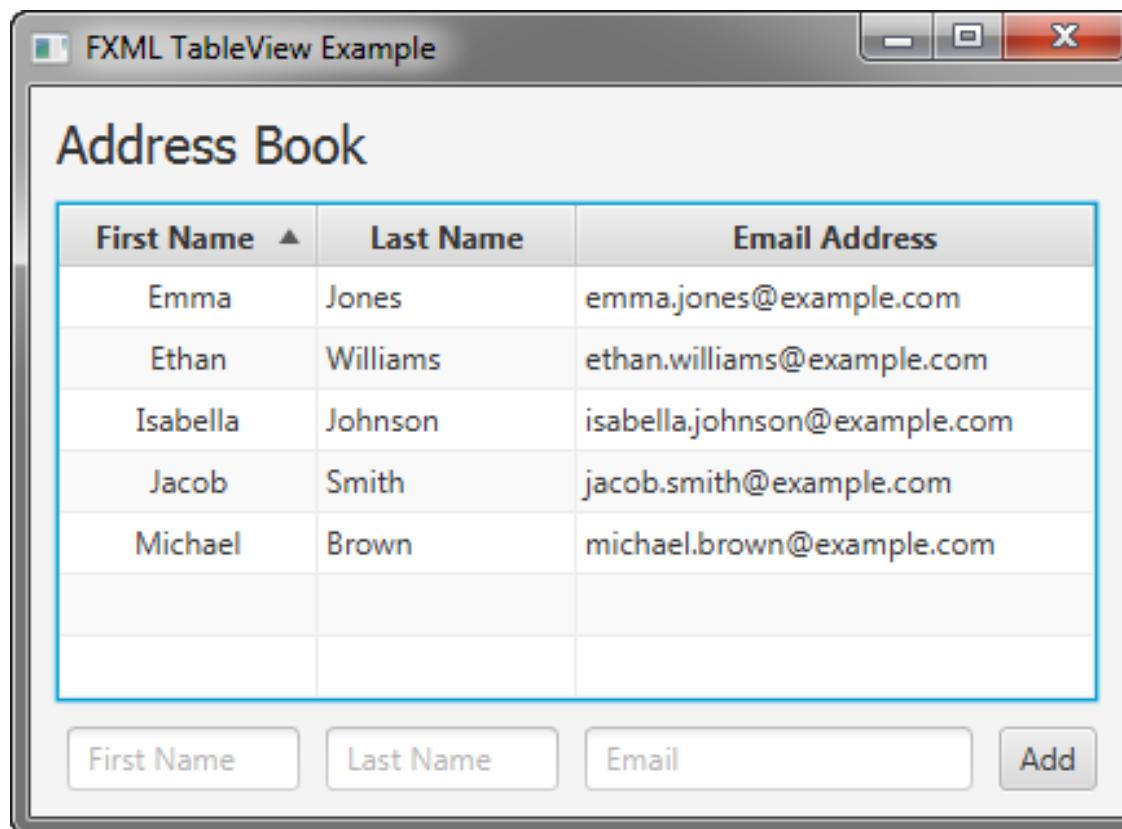
- TableView Examples
  - Sorting data
  - Editing data
  - Adding a combo box column
- ListView Example
- Menu Example
- ContextMenu Example
- Opening a new window
- Switching to different screens Example

# TABLEVIEW EXAMPLE

This example is based on the Oracle tutorial

**See Ex3-TableView.zip**

•



# DEFINE THE DATA MODEL

- .When you create a table in a JavaFX application, it is a best practice to implement a class that defines the data model and provides methods and fields to further work with the table.
- .Create a Person class to define the data for the address book.
- .

# CONTROLLER

- Method initialize

- Factory methods for each column: see the correspondence between Person fields and table columns
    - Fills the default values

# TABLEVIEW EXAMPLE CONT.

**.See Ex4-TableView.zip**

## **.Sorting Data in Columns**

- The TableView class provides built-in capabilities to sort data in columns.
- Users can alter the order of data by clicking column headers.
  - you can set sorting preferences for each column in your application by applying the `setSortType` method
  - You can also specify which columns to sort by adding and removing TableColumn instances from the `TableView.sortOrder` observable list
  - See method `changeSorting` from `TableController` class

# TABLEVIEW EXAMPLE CONT.

## **.Editing data in the table**

- Use the `setCellFactory` method to reimplement the table cell as a text field with the help of the `TextFieldTableCell` class.
- The `setOnEditCommit` method processes editing and assigns the updated value to the corresponding table cell (requires that users press the Enter key to commit the edit)

# TABLEVIEW EXAMPLE CONT.

## .Adding a combo box column

–See Gender.java as an enum example

–Please note that Enum in Java are reference types like class or interface and you can define constructor, methods and variables inside java Enum which makes it more powerful than Enum in C and C++.

–See the part about combobox in initialize method from TableController class

- `genderColumn.setCellValueFactory`

- `genderColumn.setCellFactory`

- `genderColumn.setOnEditCommit` – for editing

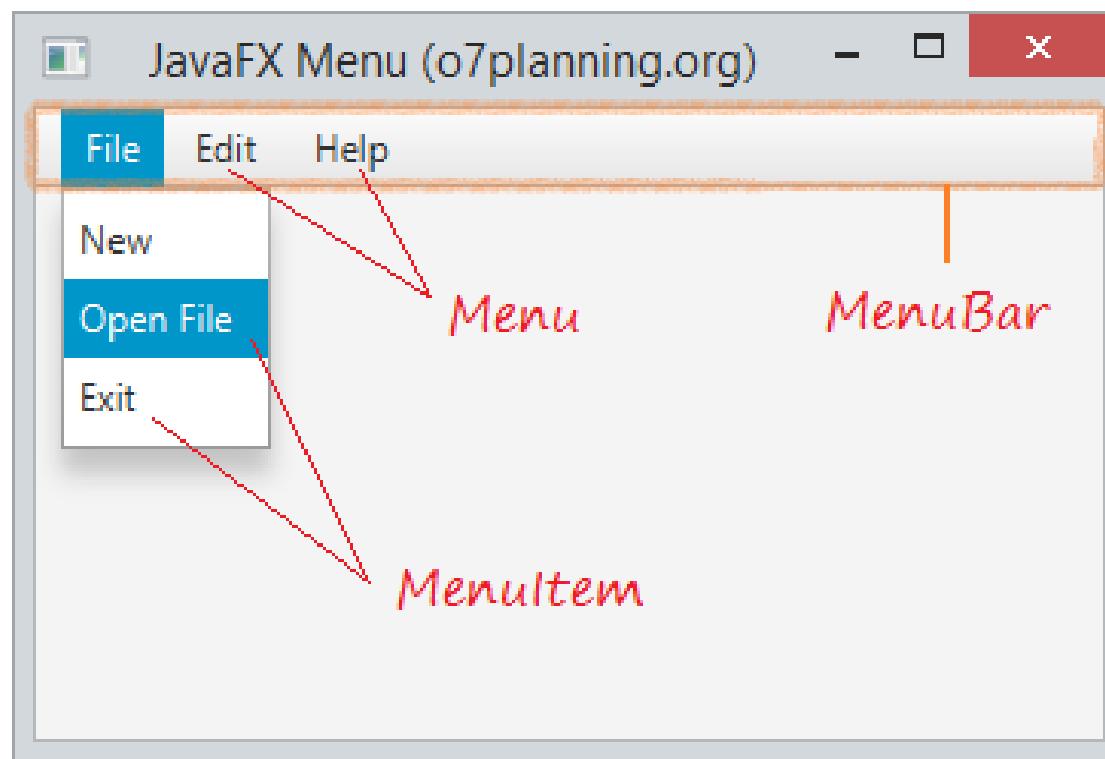
# LISTVIEW EXAMPLE

**.See Ex5-ListView.zip**

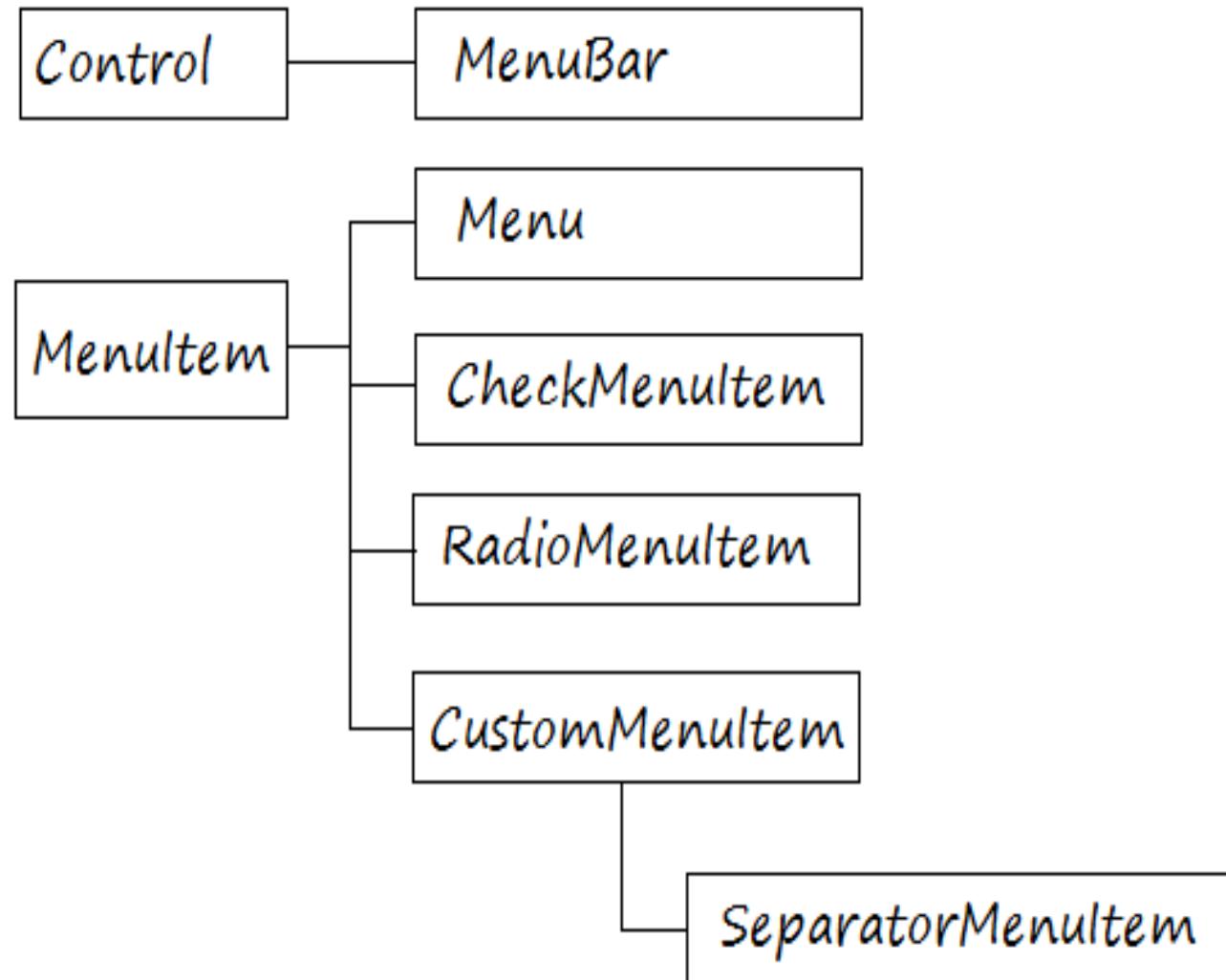
.In ListViewController class

- the list initialization
- Setting the focus model
- Setting the selection model
- Handling the selection

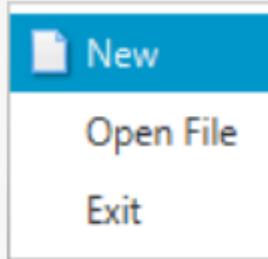
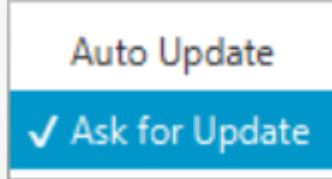
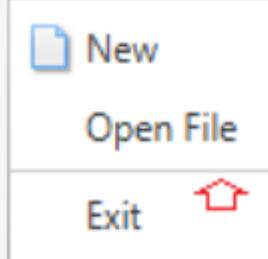
# MENU EXAMPLE



# MENU EXAMPLE



# MENU EXAMPLE

MenuItem	
CheckMenuItem	
RadioMenuItem	
SeparatorMenuItem	

# MENU EXAMPLE

**.See Ex6-Menu.zip**

- .Setting the menu structure in SceneBuilder:CheckMenuItem, RadioMenuItem, etc.
- .Additional settings for RadioMenuItem in initialize method of MenuController class
- .Set accelerator for Exit MenuItem
- .Set the handler when click on Exit MenuItem

# CONTEXTMENU EXAMPLE

**.See Ex7-CtxMenu.zip**

.Create a context Menu

.Setting when the context menu is showed

# OPENING A NEW WINDOW EXAMPLE

- When creating a new Stage, you can set up a parent window for it (also called the window owning it), via the stage.initOwner(parentStage) method
- There are three modalities that you can apply to the Stage through the stage.initModality(Modality) method.
  - Modality.NONE: When you open a new window with this modality, the new window will be independent from the parent window. You can interact with the parent window, or close it without affecting the new window.
  - Modality.WINDOW\_MODAL: When you open a new window with this modality, it will lock the parent window. You can not interact with the parent window until this window is closed.
  - Modality.APPLICATION\_MODAL: When you open a new window with this modality, it will lock any other windows of the application. You can not interact with any other windows until this window is closed.

# OPENING A NEW WINDOW EXAMPLE

**.See Ex8-NewWindow.zip**

- .See how new stages are created in controller
- .See how a reference to the primary stage is transmitted to the controller

# SWITCHING TO DIFFERENT SCREENS

- See **Ex9-SwitchScenes.zip**
- See how a new scene is changed when the button is pressed

# **Advanced Programming Methods**

## **Lecture 11 – More on C#**

# C# GENERICS

# C#'s genericity mechanism, available since C# 2.0

Most common use:

- Use (and implement) generic yet type-safe containers

```
List<String> safeBox = new List<String>();
```

Compile-time type-checking is enforced

- Custom generic classes (and methods)

Syntax: different position of the generic parameter w.r.t. Java

Java: **public <T> T foo (T x);**

C#: **public T foo <T> (T x);**

- Bounded genericity

```
public T test <T> (T x) where T:Interface1, Interface2
```

# C# vs Java

- Unlike Java, genericity is supported natively by .NET bytecode
- Hence, basically all limitations of Java generics disappear:
  - Can instantiate generic parameter with value types
  - At runtime you can tell the difference between `List<Integer>` and `List<String>`
  - Exception classes can be generic classes
  - Can instantiate a generic type parameter provided a clause `where G : new()` constrains the parameter to have a default constructor

# C# vs. Java

- Can get the default value of a generic type parameter  
`T t = default (T);`
- Arrays can have elements of a generic type parameter
- A static member can reference a generic type parameter

Another consequence is that **raw types** (unchecked generic types without any type argument) don't exist in C#

# Generics and Inheritance

- Let S be a subtype of T
- There is no inheritance relation between:  
`SomeGenericClass<S>` and `SomeGenericClass<T>`  
In particular: the former is not a subtype of the latter
- However, let AClass be a non-generic type:  
`S<AClass>` is a subtype of `T<AClass>`

# Replacing Wildcards in C#

- There's no C# equivalent of Java's wildcards
  - But most of Java's wildcard code can be ported to C# (not necessarily resulting in cleaner code)

Consider the following hierarchy of classes:

```
class Circle:Shapes{...}
```

```
class Rectangle:Shapes{...}
```

What should be the signature of a method **drawShapes** that takes a list of **Shape** objects and draws all of them?

**DrawShapes( List<Shapes> shapes )**

- **this doesn't work on a List<Circle>, which is not a subtype of List<Shape>**

# Replacing Wildcards in C#

Solution: use a helper class with bounded genericity

```
class DrawHelper <T> where T: Shape {  
    public static void DrawShapes( List<T> shapes);  
}
```

The use of the method:

```
DrawHelper<Shape>.DrawShapes(listOfShapes);
```

```
DrawHelper<Circle>.DrawShapes(listOfCircles);
```

# Generics can be used with:

- Types
  - Struct
  - Interface
  - Class
  - Delegate
- Methods
- Generic Constraints

# Generics can be used:

- to easily create non-generic derived types:

```
public class IntStack : Stack<int> {...}
```

- in internal fields, properties and methods of a class:

```
public class Customer<T>{  
    private static List<T> customerList;  
    private T customerInfo;  
    public T CustomerInfo { get; set; }  
    public int CompareCustomers( T customerInfo );}
```

- A base class or interface can be used as a constraint. For instance:

```
public interface IDrawable { public void Draw(); }
```

- Need a constraint that our type T implements the IDrawable interface.

```
public class SceneGraph<T> where T : IDrawable {  
    public void Render() { ... T node; ...  
        node.Draw();  
    }  
}
```

- No need to cast
  - Compiler uses type information to decide

- Can also specify a class constraint. That is, require a reference type:

```
public class CarFactory<T> where T : class {  
private T currentCar = null;
```

- Forbids `CarFactory<int>` and other value types.
  - Useful since I can not set an int to null.

- Alternatively, require a value (struct) type.

```
public struct Nullable<T> where T : struct {  
private T value;
```

- The ***default*** keyword

```
public class GraphNode<T> {  
    private T nodeLabel;  
    private void ClearLabel() { nodeLabel = default(T); }
```

- If T is a reference type `default(T)` will be null.
- For value types all bits are set to zero.

- Special constraint using the *new* keyword:

```
public class Stack<T> where T : new() {  
    public T PopEmpty() {  
        return new T();  
    }  
}
```

- Parameter-less ***constructor constraint***
- Type T must provide a public parameter-less constructor
- No support for other constructors or other method syntaxes.
- The new() constraint must be the last constraint.

- A generic type parameter, like a regular type, can have zero or more interface constraints

```
public class GraphNode<T> {  
    where T : ICloneable, IComparable  
    ...}
```

- A type parameter can only have one where clause, so all constraints must be specified within a single where clause.
- You can also have one type parameter be dependent on another.

```
public class SubSet<U,V> where U : V  
public class Group<U,V>
```

```
where V : IEnumerable<U> { ... }
```

- C# also allows you to parameterize a method with generic types:

```
public static void Swap<T>( ref T a, ref T b ){  
    T temp = a;  
    a = b;  
    b = temp; }
```

- The method does not need to be static.

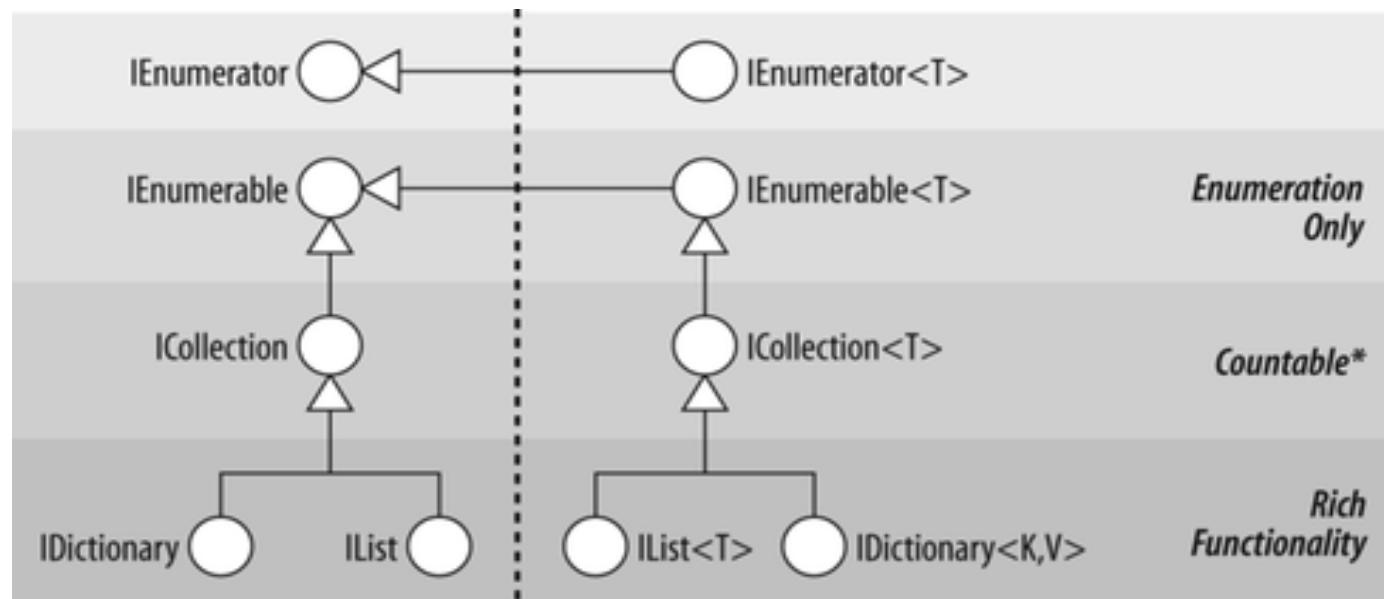
```
public class Report<T> : where T: Iformatter{...}  
  
public class Insurance {  
    public Report<T> ProduceReport<T>() where T : Iformatter { ... }  
}
```

- In C#, generic types can be compiled into a class library or dll and used by many applications.
- Differs from C++ templates, which use the source code to create a new type at compile time.
- Hence, when compiling a generic type, the compiler needs to ensure that the code will work for any type.

# C# COLLECTIONS

# Data Structures in C#

- # `System.Collections` for nongeneric collection classes and interfaces.
- # `System.Collections.Specialized` for strongly typed nongeneric collection classes.
- # `System.Collections.Generic` for generic collection classes and interfaces.
- # `System.Collections.ObjectModel` contains proxies and bases for custom collections.



# IEnumerator - IEnumerable

```
//System.Collections
public interface IEnumerator {
    bool MoveNext();
    object Current { get; }
    void Reset();
}
```

```
//System.Collections.Generic
public interface IEnumerator<T> :
    IEnumerator, IDisposable{
    T Current { get; }
```

```
//System.Collections
public interface IEnumerable{
    IEnumerator GetGetEnumerator();
}
```

```
//System.Collections.Generic
public interface IEnumerable<T> :
    IEnumerable{
    IEnumerator<T> GetGetEnumerator();
}
```

If a class implements the `IEnumerable` interface, then the `foreach` statement can be used on that class.

# IEnumerable -foreach

```
class Set : ISet, IEnumerable{
    object[] elems;
    public IEnumerator GetEnumerator() { ... }
    //...
}
...
Set s=new Set();
s.add("ana");
s.add("are");
s.add("mere");
foreach(Object o in s){
    Console.WriteLine("{0} ",o);
}
```

# ICollection/ ICollection<T>

```
public interface ICollection : IEnumerable{
    void CopyTo (Array array, int index);
    int Count {get;}
    bool IsSynchronized {get;}
    object SyncRoot {get;}
}

public interface ICollection<T> : IEnumerable<T>, IEnumerable{
    void Add(T item);
    void Clear( );
    bool Contains (T item);
    void CopyTo (T[] array, int arrayIndex);
    int Count { get; }
    bool IsReadOnly { get; }
    bool Remove (T item);
}
```

# IList / IList<T>

```
public interface IList : ICollection, IEnumerable{
    object this [int index] { get; set }
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    int Add (object value);
    void Clear();
    bool Contains (object value);
    int IndexOf (object value);
    void Insert (int index, object value);
    void Remove (object value);
    void RemoveAt (int index);
}

public interface IList<T> : ICollection<T>, IEnumerable<T>,
    IEnumerable{
    T this [int index] { get; set; }
    int IndexOf (T item);
    void Insert (int index, T item);
    void RemoveAt (int index);
}
```

# IDictionary

```
public interface IDictionary : ICollection, IEnumerable{
    IDictionaryEnumerator GetEnumerator( );
    bool Contains (object key);
    void Add      (object key, object value);
    void Remove   (object key);
    void Clear( );
    object this [object key] { get; set; }
    bool IsFixedSize      { get; }
    bool IsReadOnly       { get; }
    ICollection Keys     { get; }
    ICollection Values   { get; }
}

public interface IDictionaryEnumerator : IEnumerator
{
    DictionaryEntry Entry { get; }
    object Key { get; }
    object Value { get; }
}
```

# The IComparable Interface

This require one method `CompareTo` which returns  
-1 if the first value is less than the second  
0 if the values are equal  
1 if the first value is greater than the second

```
public interface IComparable {  
    int CompareTo(object obj)  
}
```

# The IComparer Interface

This is similar to **IComparable** but is designed to be implemented in a class outside the class whose instances are being compared

**Compare()** works just like **CompareTo()**

```
public interface IComparer {  
    int Compare(object o1, object o2);  
}
```

# Sorting an ArrayList

To use CompareTo() of IComparable

`ArrayList.Sort()`

To use a custom comparer object

`ArrayList.Sort(IComparer cmp)`

To sort a range

`ArrayList.Sort(int start, int len, IComparer cmp)`

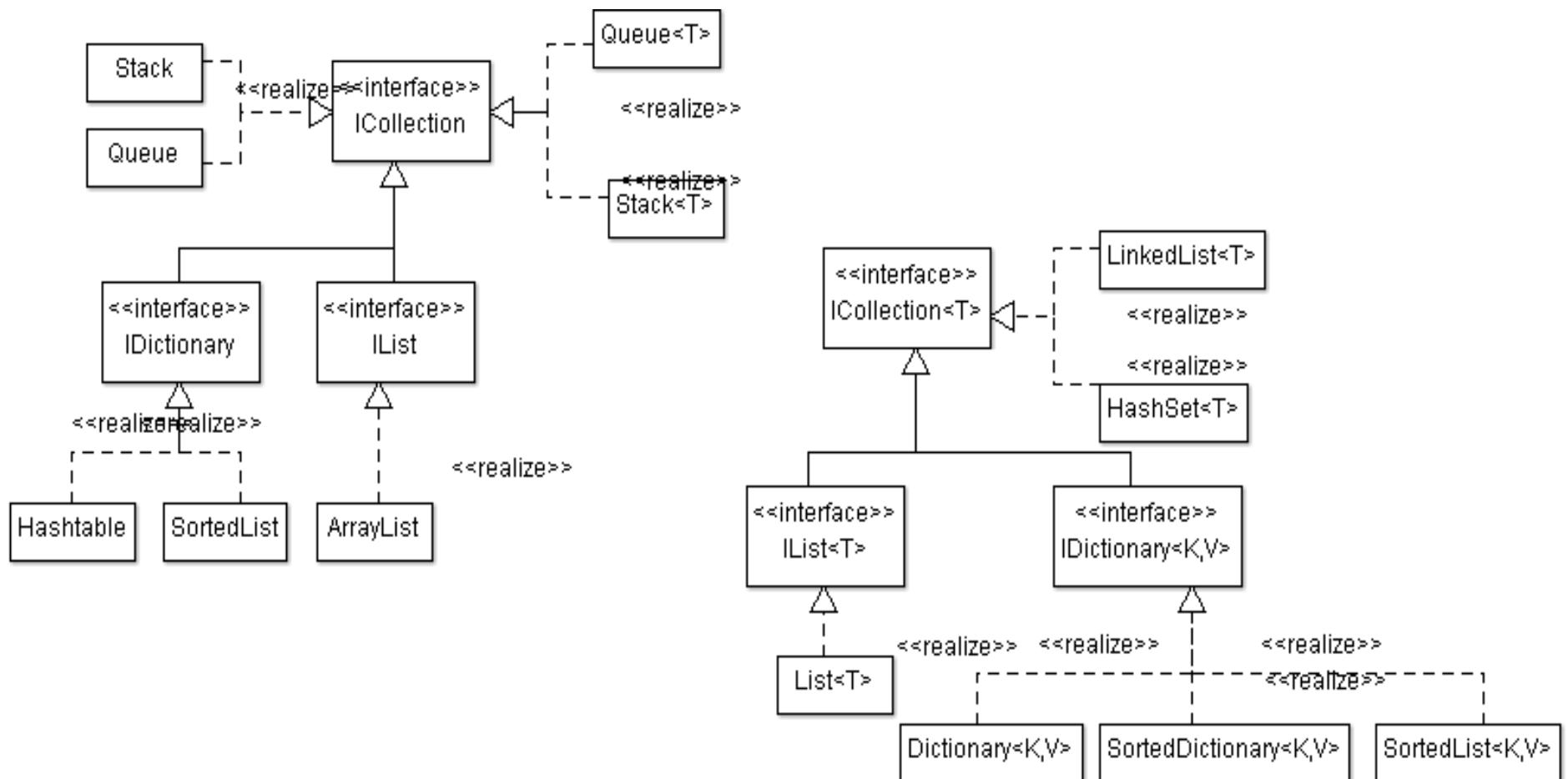
# ICloneable Interface

This guarantees that a class can be cloned

The Clone method can be implemented to make a shallow or deep clone

```
public interface ICloneable {  
    object Clone();  
}
```

# Data structures hierarchy



Generic Class	Description
Dictionary<K, V>	Generic unordered dictionary
LinkedList<E>	Generic doubly linked list
List<E>	Generic ArrayList
Queue<E>	Generic queue
SortedDictionary<K, V>	Generic dictionary implemented as a tree so that elements are stored in order of the keys
SortedList<K, E>	Generic binary tree implementation of a list. Can have any type of subscript. More efficient than SortedDictionary in some cases.
Stack<E>	Generic stack

# Array Class

The **Array** class is the implicit base class for all single and multidimensional arrays.

It provides a common set of methods to all arrays, regardless of their declaration or underlying element type.

Length and Rank:

```
public int GetLength      (int dimension);  
public int Length        { get; }  
public int Rank { get; }   // Returns number of dimensions in array
```

Searching in a one-dimensional array: **BinarySearch**, **IndexOf**, **LastIndexOf**, etc (static methods, overloaded).

Sorting: **Sort** (overloaded static method).

Reversing elements: **Reverse** (overloaded static method).

Copying: **Copy** (overloaded static method).

# IDisposable

Some objects require explicit code to release resources such as open files, locks, operating system handles, and unmanaged objects. This is called dispose, and it is supported through the `IDisposable` interface.

```
public interface IDisposable{  
    void Dispose( );  
}
```

# IDisposable

⌘ C#'s `using` statement provides a syntactic shortcut for calling `Dispose` on objects that implement `IDisposable`, using a `try / finally` block.

```
using (Font font = new Font("Courier", 12.0f)) {  
    //code that uses the object font  
}
```

The compiler converts this to:

```
Font font = new Font("Courier", 12.0f);  
try{  
    // ... Use font  
}  
finally{  
    if (font != null) font.Dispose();  
}
```

The `finally` block ensures that the `Dispose` method is called even when an exception is thrown, or the code exits the block early.

# IDisposable

- Multiple objects can be used with a `using` statement, but they must be declared inside the `using` statement, or nested:

```
using (Font f3 = new Font("Arial", 10.0f), f4 = new Font("Arial", 9.0f)) {
    // Use fonts f3 and f4.
}

using (Font f3 = new Font("Arial", 10.0f))
    using (Font f4 = new Font("Arial", 9.0f)){
        // Use fonts f3 and f4.
    }
}
```

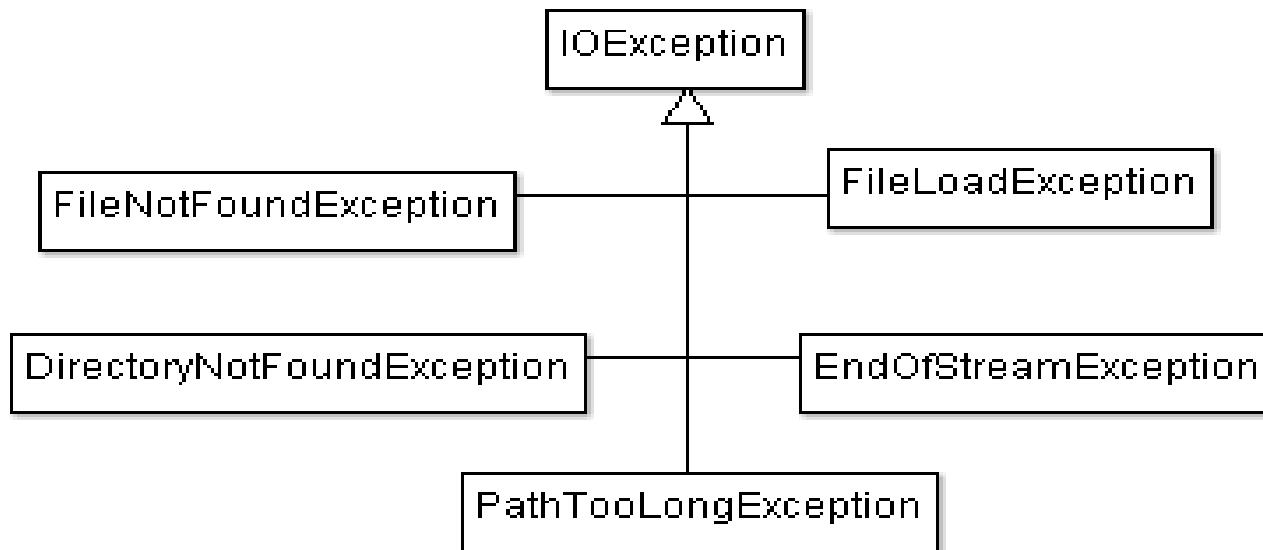
# IDisposable

- The Framework follows a set of rules in its disposal logic.
- The rules purpose is to define a consistent protocol to users.
  - » Once disposed, an object cannot be reactivated, and calling its methods or properties may throw exceptions or give incorrect results.
  - » Calling an object's **Dispose** method repeatedly causes no error.
  - » If disposable object **x** contains disposable object **y**, the **Dispose** method of **x** automatically calls the **Dispose** method of **y** - unless instructed otherwise.

# C# I/O

# I/O

Namespace `System.IO`  
Exceptions



# Utility I/O Classes

The `System.IO` namespace provides a set of types for performing utility file and directory operations, such as copying, moving, creating directories, and setting file attributes and permissions.

Static classes: `File` and `Directory`

The static methods on File and Directory are convenient for executing a single file or directory operation.

Instance method classes (constructed with a file or directory name): `FileInfo` and  `DirectoryInfo`

Static class, `Path`, that provides string manipulation methods for filenames and directory paths.

# File / FileInfo

```
public static class File{
    bool Exists (string path);
    void Delete (string path);
    void Copy   (string sourceFileName, string destFileName);
    void Move   (string sourceFileName, string destFileName);
    void Replace (string source, string destination, String backup);
    FileAttributes GetAttributes (string path);
    void SetAttributes(string path, FileAttributes fileAttributes);
    DateTime GetCreationTime (string path);
    FileSecurity GetAccessControl (string path);
    void SetAccessControl (string path, FileSecurity fileSecurity);
    //...
}
```

- # **FileInfo** offers most of the **File**'s static methods in instance form—with some additional properties such as **Extension**, **Length**, **IsReadOnly**

# Directory / DirectoryInfo

```
public static class Directory{  
    static static bool Exists(string path);  
    static void Move(string sourceDirName, string destDirName)  
    static string GetCurrentDirectory();  
    static void SetCurrentDirectory(string path);  
    static DirectoryInfo CreateDirectory(string path);  
    static DirectoryInfo GetParent(string path);  
    static string[] GetLogicalDrives();  
    static string[] GetFiles(string path);  
    static string[] GetDirectories(string path);  
    static void Delete(string path)  
    //...  
}
```

# **DirectoryInfo** exposes instance methods for creating, moving, and enumerating through directories and subdirectories.

# Stream Architecture

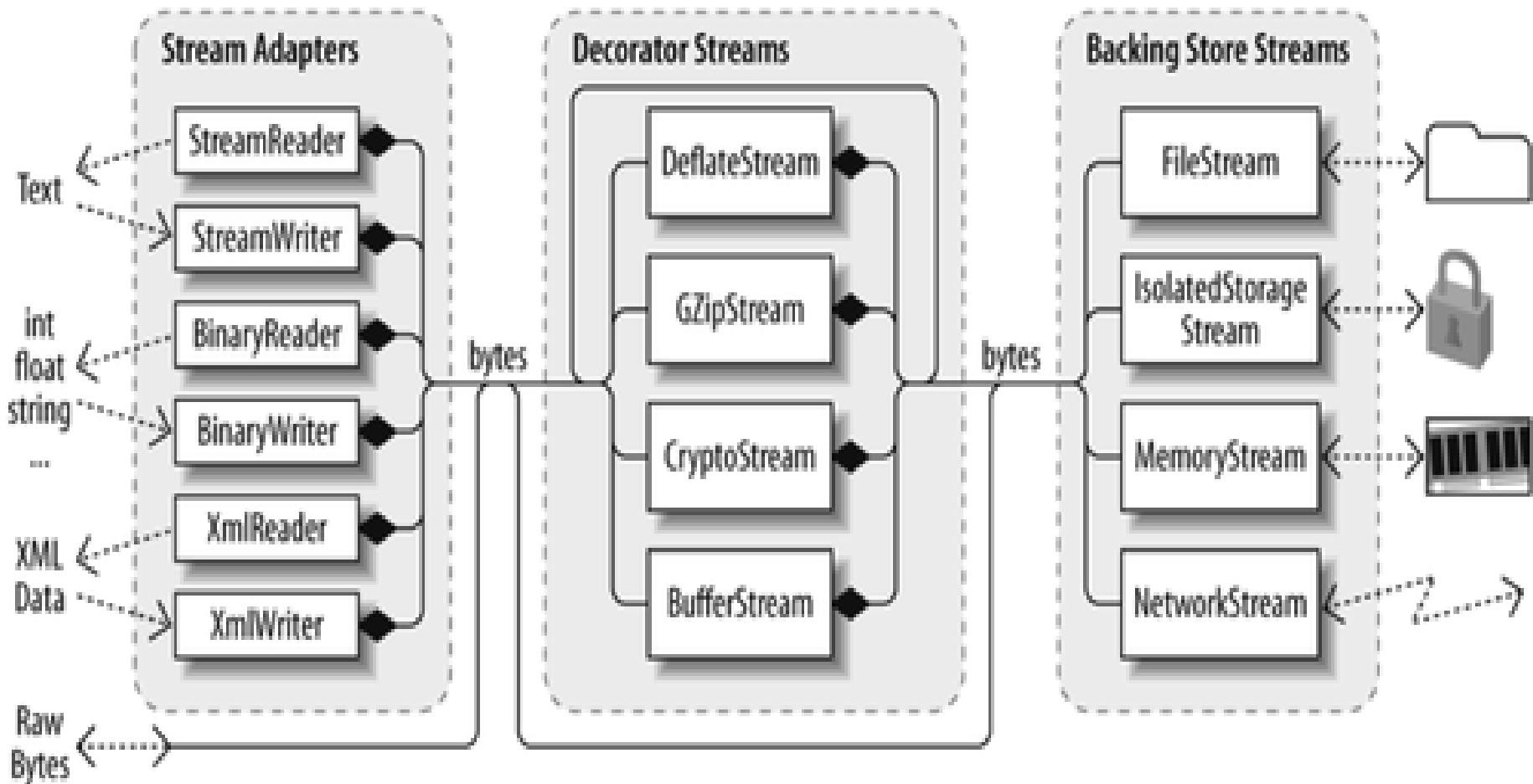
The .NET stream architecture centers on three concepts: backing stores, decorators, and adapters:

- A *backing store* is the endpoint that makes input and output useful, such as a file or network connection. It is one or both of the following:
  - A source from which bytes can be sequentially read.
  - A *destination* to which bytes can be sequentially written.
- *Decorator streams* feed off another stream, transforming the data in some way.
- *Adapter* wraps a stream in a class with specialized methods typed to a particular format.

Remark:

An adapter wraps a stream, just as a decorator. Unlike a decorator, however, an adapter is not itself a stream; it typically hides the byte-oriented methods completely.

# Stream Architecture



# **Stream** class

The abstract **Stream** class is the base for all streams.

It defines methods and properties for three fundamental operations:

- Reading
- Writing
- Seeking
- as well as for administrative tasks such as closing, flushing, and configuring timeouts.

# Stream class

Reading

```
public abstract bool CanRead { get; }
public abstract int Read (byte[] buffer, int offset, int count)
public virtual int ReadByte( );
```

Writing

```
public abstract bool CanWrite { get; }
public abstract void Write (byte[] buffer, int offset, int count);
public virtual void WriteByte (byte value);
```

Seeking

```
public abstract bool CanSeek { get; }
public abstract long Position { get; set; }
public abstract void SetLength (long value);
public abstract long Length { get; }
public abstract long Seek (long offset, SeekOrigin origin);
```

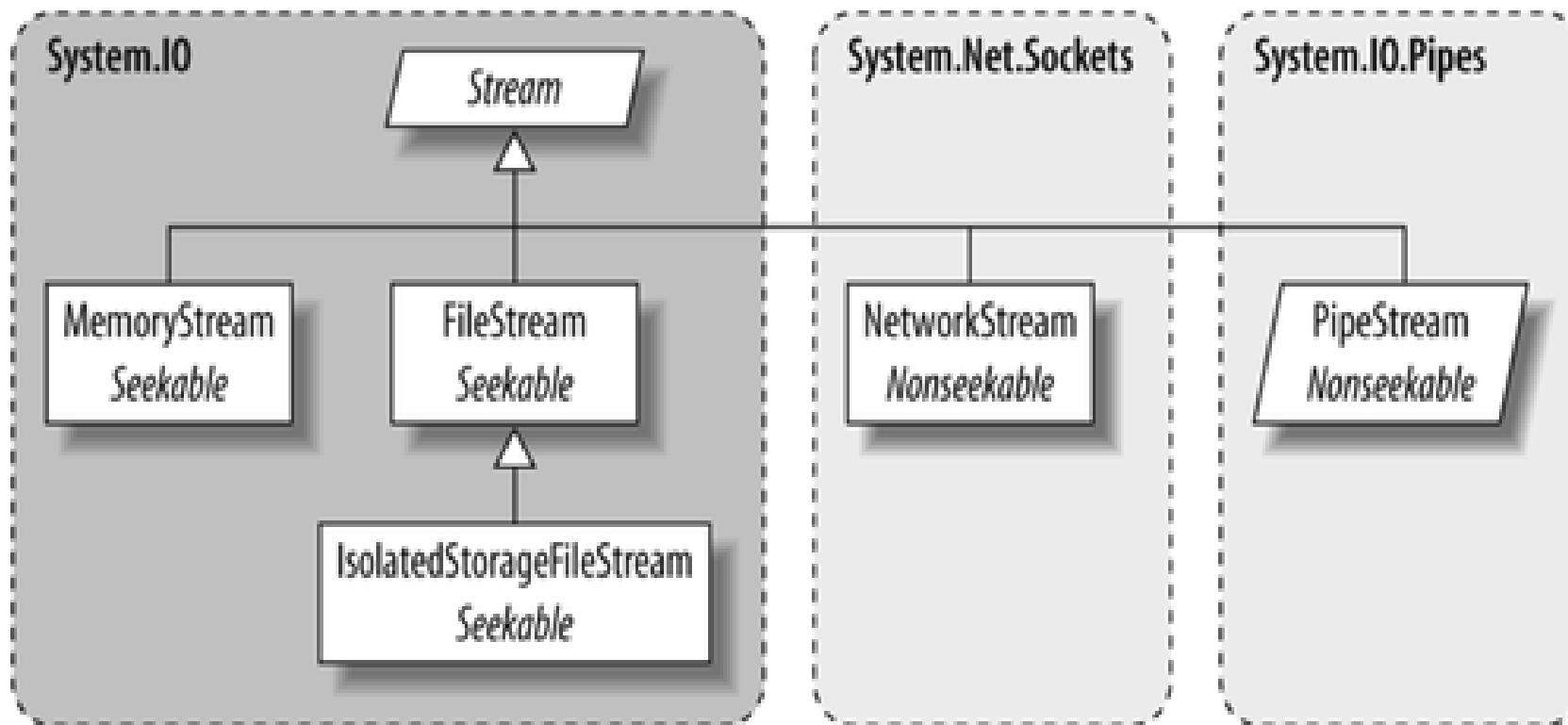
Closing/  
flushing

```
public virtual void Close( );
public void Dispose( );
public abstract void Flush( );
```

Timeouts

```
public abstract bool CanTimeout { get; }
public override int ReadTimeout { get; set; }
public override int WriteTimeout { get; set; }
```

# Backing Store Streams



# FileStream class

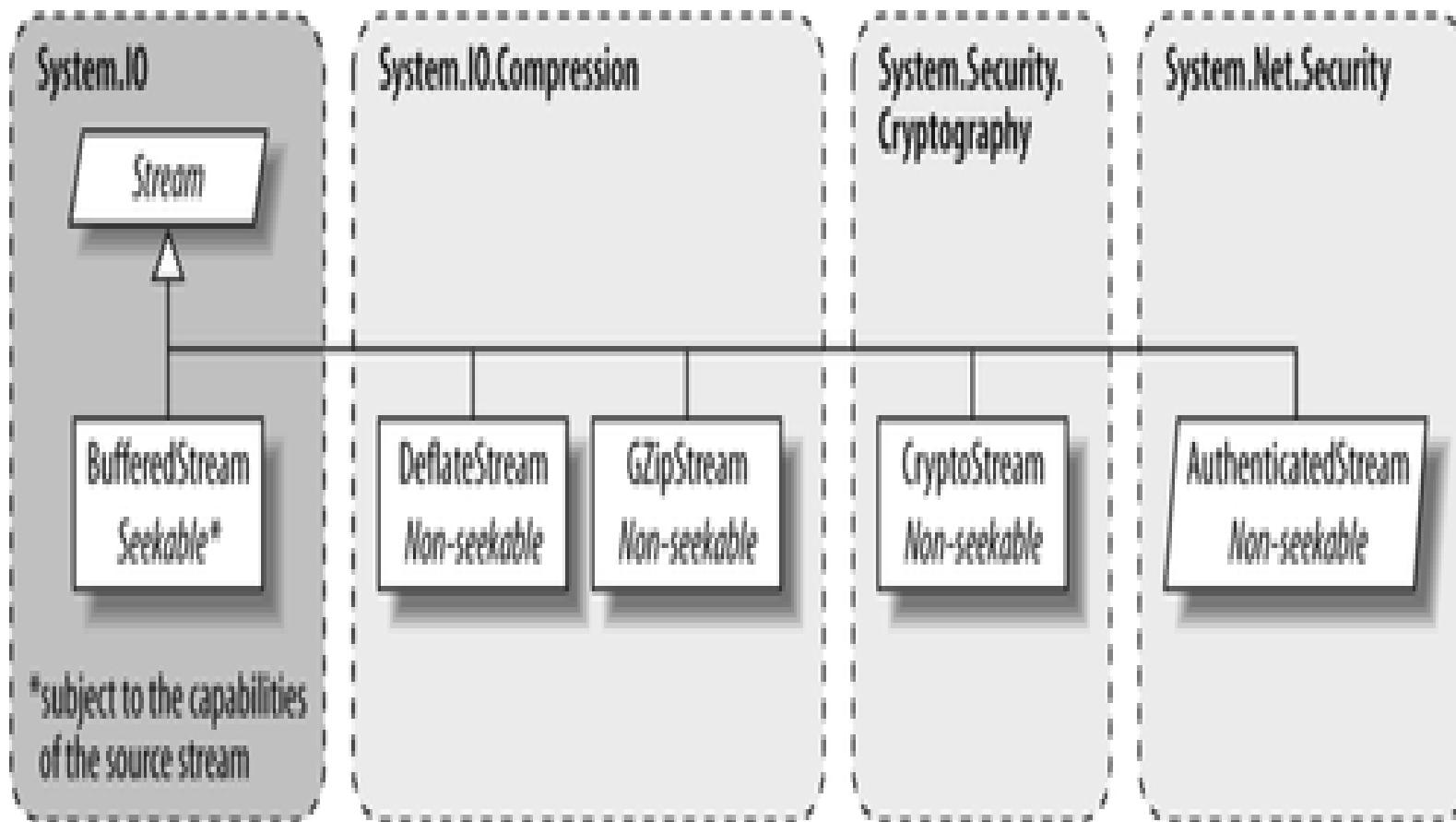
```
public class FileStream : Stream{
    public FileStream(string path, FileMode mode); //overloaded
    public override int Read (byte[] buffer, int offset, int count);
    public override int ReadByte( );
    public override long Seek (long offset, SeekOrigin origin);
    //...
}
public enum SeekOrigin{Begin, Current, End}

public enum FileMode{CreateNew, Create, Open, OpenOrCreate, Truncate, Append }
```

# FileStream class - example

```
using(FileStream fs=new FileStream("testFileStream.txt", FileMode.Create)) {  
    String message = "Ana are mere.";  
    //helper class to transform a string in an array of bytes, System.Text  
    UnicodeEncoding unienc=new UnicodeEncoding();  
    //writing to the file  
    fs.Write(unienc.GetBytes(message), 0, unienc.GetByteCount(message));  
  
    //creates an array of bytes for reading the data from the file  
    byte[] rbytes=new byte[fs.Length];  
    //position the file pointer to the beginning  
    fs.Seek(0, SeekOrigin.Begin);  
    //reads the data from the file  
    fs.Read(rbytes, 0, (int) fs.Length);  
    //transforms the bytes in a string  
    String newMess=new string(unienc.GetChars(rbytes,0,rbytes.Length));  
    Console.WriteLine("Written text {0}, read text {1}", message, newMess);  
}
```

# Decorator Streams



# Stream Adapters

A `Stream` deals only with bytes.

In order to read or write data types such as `strings`, `integers`, or XML elements a stream adapter should be used:

Text adapters (for string and character data):

- `TextReader`, `TextWriter`,
- `StreamReader`, `StreamWriter`,
- `StringReader`, `StringWriter`

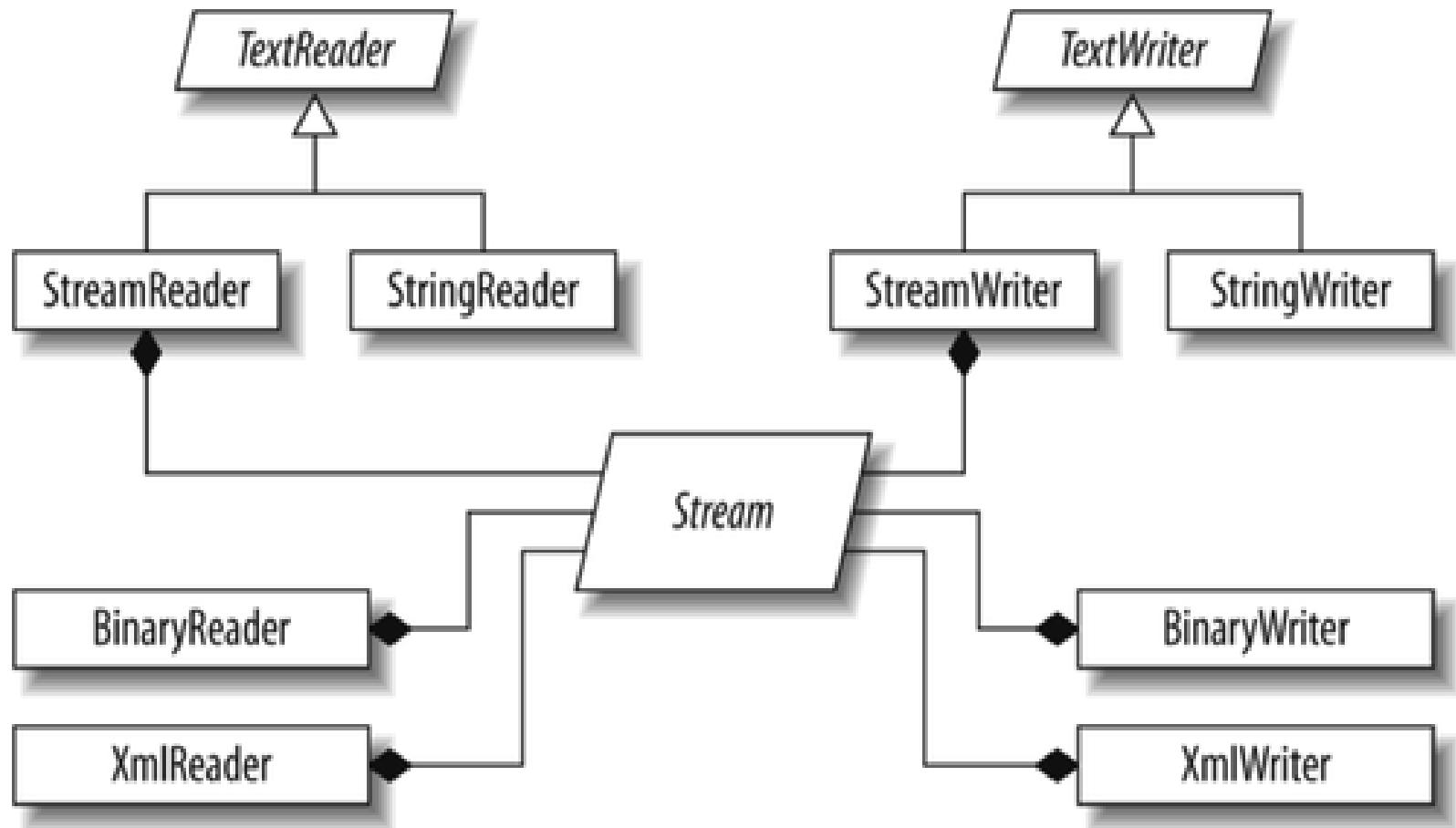
Binary adapters (for primitive types such as `int`, `bool`, `string`, and `float`)

- `BinaryReader`, `BinaryWriter`

XML adapters

- `XmlReader`, `XmlWriter`

# Stream Adapters



# Text Adapters

- # **TextReader** and **TextWriter** are the abstract base classes for adapters that deal exclusively with characters and strings. Each has two general-purpose implementations in the framework:

**StreamReader/StreamWriter**: uses a **Stream** for its data store, and translates the stream's bytes into characters or strings

**StringReader/StringWriter**: implements **TextReader/TextWriter** using in-memory strings

Because text adapters are often associated with files, the **File** class provides the static methods **CreateText**, **AppendText**, and **OpenText** to ease the process of creating file-based text adapters.

# Text Adapters

```
//Writing to a text file
using (FileStream fs = File.Create ("test.txt"))
using (TextWriter writer = new StreamWriter (fs))
//or TextWriter writer=File.CreateText("test.txt")
{
    writer.WriteLine ("Ana are mere.");
    writer.WriteLine ("mere");
}
//Reading from a text file
using (FileStream fs = File.OpenRead ("test.txt"))
using (TextReader reader = new StreamReader (fs))
//or TextReader reader=File.OpenText("test.txt")
{
    Console.WriteLine (reader.ReadLine());           // reads line 1
    Console.WriteLine (reader.ReadLine());           // reads line 2
}
```

# Binary Adapters

- # **BinaryReader** and **BinaryWriter** read and write native data types: **bool**, **byte**, **char**, **decimal**, **float**, **double**, **short**, **int**, **long**, **sbyte**, **ushort**, **uint**, and **ulong**, as well as **strings** and **arrays** of the primitive data types.

```
using (BinaryWriter w = new BinaryWriter(File.Create("testBinary.txt")))
{
    w.Write("Ana");
    w.Write(23);
    w.Write(12.4);
    w.Flush();
}
using(BinaryReader r = new BinaryReader(File.Open("testBinary.txt", FileMode.Open)))
{
    String name = r.ReadString();
    int age = r.ReadInt32();
    double d= r.ReadDouble();
    Console.WriteLine("name= {0}, age={1}, d={2}", name, age, d);
}
```

LINQ

# Before LINQ

```
1.int[] numbers = { 3, 6, 7, 9, 2, 5, 3, 7 };
```

```
2.int i = 0;
```

```
3.
```

```
4.// Display numbers larger than 5
```

```
5.while (i < numbers.GetLength(0))
```

```
6. {
```

```
7.if (numbers[i] > 5)
```

```
8.Console.WriteLine(numbers[i]);
```

```
9.++i;
```

```
10.}
```

# LINQ

```
int[] numbers = { 3, 6, 7, 9, 2, 5, 3, 7 };
```

```
var res = from n in numbers  
          where n > 5  
          select n;
```

```
foreach (int n in res)  
    Console.WriteLine(n);
```

# LINQ

- # querying data from various sources, such as arrays, dictionaries, xml and entities created from entity framework.
- # instead of having to use a different API for each data source, LINQ provides a consistent and uniform programming model to work with all supported data sources.

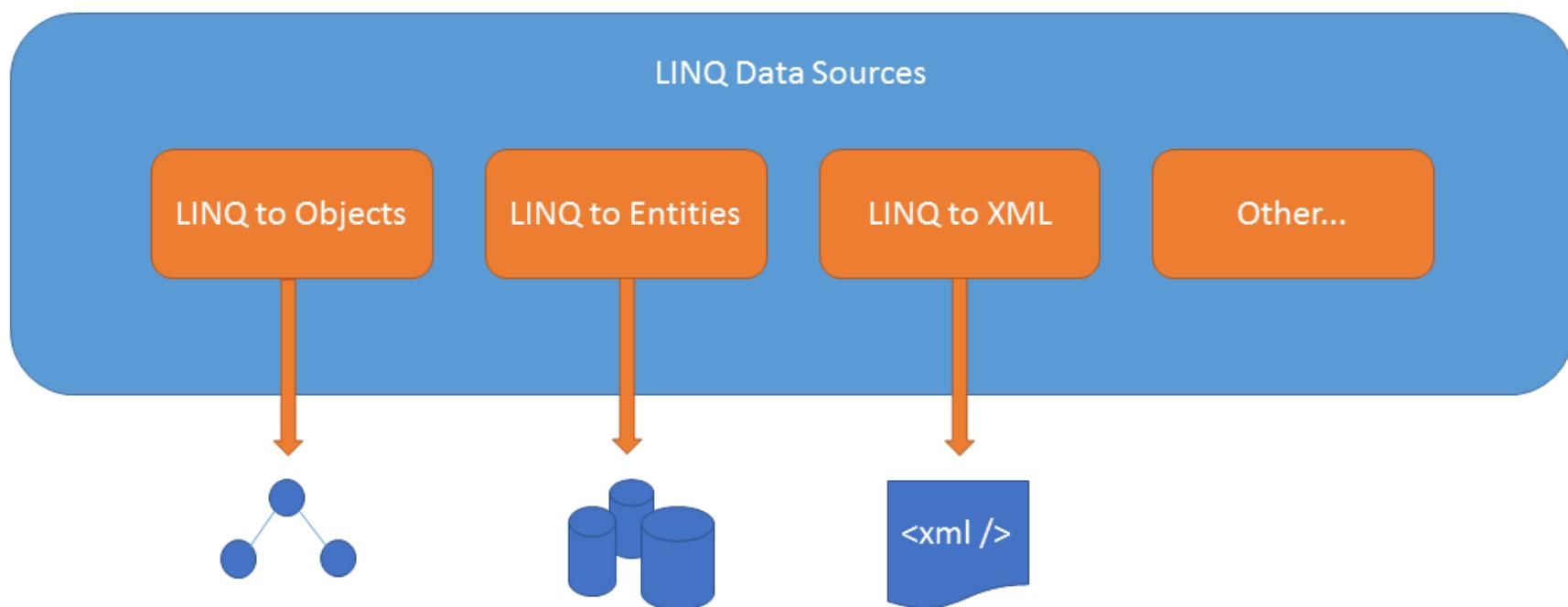
Some of the most used LINQ data sources, which are all part the .NET framework, are:

- **LINQ to Objects:** for in-memory collections based on *IEnumerable*, such as *Dictionary* and *List*.
- **LINQ to Entities:** for Entity Framework on object context.
- **LINQ to XML:** for in-memory XML documents.

# LINQ



.NET Language Integrated Query



# LINQ

- 1) with SQL-like syntax called *Query expressions*

```
int[] numbers = { 7, 53, 45, 99 };
```

```
var res = from n in numbers  
          where n > 50  
          orderby n  
          select n.ToString();
```

# LINQ

- 2) a method like approach called *Lambda expressions*

```
int[] numbers = { 7, 53, 45, 99 };
```

```
var res = numbers.Where(n => n > 50)
                  .OrderBy(n => n)
                  .Select(n => n.ToString());
```

# LINQ

LINQ queries can execute in two different ways: deferred and immediate.

With deferred execution, the resulting sequence of a LINQ query is not generated until it is required. The following query does not actually execute until `Max()` is called, and a final result is required:

```
int[] numbers = { 1, 2, 3, 4, 5 };
    var result = numbers.Where(n => n >= 2 && n <= 4);
    Console.WriteLine(result.Max()); // <- query executes at this point

// Output:
//4
```

# LINQ

Deferred execution makes it useful to combine or extend queries. Have a look at this example, which creates a base query and then extends it into two new separate queries:

```
int[] numbers = { 1, 5, 10, 18, 23};  
var baseQuery = from n in numbers select n;  
var oddQuery = from b in baseQuery where b % 2 == 1 select b;  
  
Debug.WriteLine("Sum of odd numbers: " + oddQuery.Sum()); // <- query executes at this point  
  
var evenQuery = from b in baseQuery where b % 2 == 0 select b;  
Debug.WriteLine("Sum of even numbers: " + evenQuery.Sum()); // <- query executes at this point  
  
// Output:  
// Sum of odd numbers: 29  
// Sum of even numbers: 28
```

# **Advanced Programming Methods**

## **Lecture12 –**

# **C# Concurrency**

# Content

- Basic Concurrency
- Thread pooling
  - Task Parallel Library
- Synchronization mechanisms
- Asynchronous programming

# References

**NOTE: The slides are based on the following free tutorials. You may want to consult them too.**

1. [https://msdn.microsoft.com/en-us/library/hh156548\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/hh156548(v=vs.110).aspx)
2. <https://msdn.microsoft.com/en-us/library/hh191443.aspx>
3. <http://www.albahari.com/threading/>

# Threads

- a C# program starts in a single thread (the “**main**” thread) created automatically by the CLR (Common Language Runtime) and operating system , and is made multithreaded by creating additional threads
- CLR assigns each thread its own memory stack so that local variables are kept separate.
- Threads share data if they have a common reference to the same object instance.

```
using System;  
using System.Threading;  
  
class ThreadTest{  
    static void Main(){  
        Thread t = new Thread (WriteY);          // Kick off a new thread  
        t.Start();                            // running WriteY()  
  
        // Simultaneously, do something on the main thread.  
        for (int i = 0; i < 1000; i++) Console.Write ("x");  
    }  
  
    static void WriteY() {  
        for (int i = 0; i < 1000; i++) Console.Write ("y");  
    }  
}
```

# Threads

- once started, a thread's **IsAlive** property returns true, until the point where the thread ends.
- a thread ends when the delegate passed to the Thread's constructor finishes executing.
- once ended, a thread cannot restart.

# Thread life cycle

- **The Unstarted State:** it is the situation when the instance of the thread is created but the Start method has not been called.
- **The Ready State:** it is the situation when the thread is ready to run and waiting CPU cycle.
- **The Not Runnable State:** a thread is not runnable, when:
  - Sleep method has been called
  - Wait method has been called
  - Blocked (e.g. by I/O operations)
- **The Dead State:** it is the situation when the thread has completed execution or has been aborted.

# Threads scheduler

- it manages multithreading
- it is a function that the CLR typically delegates to the operating system
- it ensures all active threads are allocated appropriate execution time, and that threads that are waiting or blocked (for instance, on an exclusive lock or on user input) do not consume CPU time

# Thread's unsafety

```
class ThreadTest {  
    static bool done; // Static fields are shared between all threads  
    static void Main(){  
        new Thread (Go).Start();  
        Go();  
    }  
  
    static void Go(){  
        if (!done) { Console.WriteLine ("Done");done = true; }  
    }  
}  
  
//How many times and which “Done” is printed first? 9
```

# Thread's Safety

```
class ThreadSafe {  
    static bool done;  
    static readonly object locker = new object();  
    static void Main() {  
        new Thread (Go).Start();  
        Go();  
    }  
    static void Go(){  
        lock (locker){ // only one thread can execute,  
            //other threads are blocked without consuming CPU  
            if (!done) { Console.WriteLine ("Done"); done = true; }  
        }  
    }  
}
```

# Join and Sleep

- a thread can wait for a second thread to end by calling the second thread **Join** method.
- **Thread.Sleep** pauses the current thread for a specified period
- While waiting on a Sleep or Join, a thread is blocked and so **does not consume CPU resources**.

# Join example

```
static void Main()
{
    Thread t = new Thread (Go);
    t.Start();
    t.Join();
    Console.WriteLine ("Thread t has ended!");
}
```

```
static void Go()
{
    for (int i = 0; i < 1000; i++) Console.Write ("y");
}
```

# Creating and Starting Threads

- threads are created using the Thread class's constructor, passing in a **ThreadStart delegate** which indicates where execution should begin
- ThreadStart delegate is defined:

**public delegate void ThreadStart();**

```
class ThreadTest  
{  
    static void Main()  
    {  
        Thread t = new Thread (new ThreadStart (Go));  
  
        t.Start(); // Run Go() on the new thread.  
        Go(); // Simultaneously run Go() in the main thread.  
    }  
  
    static void Go(){  
        Console.WriteLine ("hello!");  
    }  
}
```

# Passing data to a thread

```
static void Main()
{
    // use a lambda expression to pass data to the thread's target method
    Thread t = new Thread ( () => Print ("Hello from t!") );
    t.Start();
}

static void Print (string message)
{
    Console.WriteLine (message);
}
```

# Passing data to a thread

```
static void Main(){
    Thread t = new Thread (Print);
    //pass an argument into Thread's Start method
    t.Start ("Hello from t!");
}

static void Print (object messageObj){
    string message = (string) messageObj; // We need to cast here
    Console.WriteLine (message);
}
```

# Naming Threads

```
class ThreadNaming {  
    static void Main() {  
        Thread.CurrentThread.Name = "main";  
        Thread worker = new Thread (Go);  
        worker.Name = "worker";  
        worker.Start();  
        Go();  
    }  
    static void Go() {  
        Console.WriteLine ("Hello from " + Thread.CurrentThread.Name);  
    }  
}  
  
//thread's name can be set using Thread.CurrentThread property
```

# Foreground/Background threads

## Foreground Threads:

- have the ability to prevent the current application from terminating.
- CLR will not shut down an application until all foreground threads have ended.
- by default, every thread we create via the `Thread.Start()` method is automatically a foreground thread

# Foreground/Background threads

## Background Threads (also called daemon threads)

- are viewed by the CLR as expendable paths of execution that can be ignored at any point in time even if they are currently active doing work.
- if all foreground threads have terminated, all background threads are automatically terminated

We can query or change a thread's background status using its **IsBackground** property

```
using System;  
using System.Threading;  
namespace MyThread{  
    public class BackgroundThread{  
        public static void Main(string[] args){  
            Thread worker = new Thread(delegate() {  
                Console.ReadLine(); });  
            if (args.Length > 0) {  
                //the worker is assigned background status, and the  
                //program exits almost immediately as the main thread  
                //ends (terminating the ReadLine)  
                worker.IsBackground = true;  
            } else{  
                //the main thread exits, but the application keeps running  
                // because a foreground thread is still alive  
            } worker.Start();}}}
```

# Thread priority

- a thread's Priority property determines how much execution time it gets relative to other active threads in the operating system

```
enum ThreadPriority { Lowest, BelowNormal,  
    Normal, AboveNormal, Highest }
```

- it is relevant only when multiple threads are simultaneously active.
- elevating a thread's priority doesn't make it capable of performing real-time work, because it's still throttled by the application's process priority

# Exception handling

- Any try/catch/finally blocks in scope when a thread is created are of no relevance to the thread when it starts executing

```
public static void Main(){  
    try{  
        new Thread (Go).Start();  
    }catch (Exception ex){  
        // We'll never get here!  
        Console.WriteLine ("Exception!");  
    }  
}  
  
static void Go() { throw null; } // Throws a NullReferenceException
```

# Exception handling

```
public static void Main(){
    new Thread (Go).Start();
}

static void Go(){
    try{
        // ...
        throw null;  // The NullReferenceException will get caught below
        // ...
    }catch (Exception ex){
        // Typically log the exception, and/or signal another thread that we've
        // come unstuck
        // ...
    }
}
```

# Thread pooling

# Thread pool

- when a thread starts, a few hundred microseconds are spent organizing such things as a fresh private local variable stack.
- each thread also consumes (by default) around 1 MB of memory.
- the thread pool **cuts these overheads by sharing and recycling threads**, allowing multithreading to be applied at a very granular level without a performance penalty.

# Thread pool

Ways to enter the thread pool:

- By calling `ThreadPool.QueueUserWorkItem`
- Via asynchronous delegates
- Via `BackgroundWorker`
- Via the Task Parallel Library (from Framework 4.0 is the easiest way)

# ThreadPool.QueueUserWorkItem

- it is called with a delegate that you want to run on a pooled thread

```
static void Main(){
```

```
    ThreadPool.QueueUserWorkItem (Go);
```

```
    ThreadPool.QueueUserWorkItem (Go, 123);
```

```
    Console.ReadLine();}
```

```
//satisfies WaitCallback delegate
```

```
static void Go (object data){ // data will be null with the first call
```

```
    Console.WriteLine ("Hello from the thread pool! " + data);}
```

- doesn't return an object to help you subsequently manage execution

# Asynchronous delegates

1. Instantiate a delegate targeting the method you want to run in parallel (typically one of the predefined Func delegates).
2. Call BeginInvoke on the delegate, saving its IAsyncResult return value. BeginInvoke returns immediately to the caller. You can then perform other activities while the pooled thread is working.
3. When you need the results, call EndInvoke on the delegate, passing in the saved IAsyncResult object.

# Asynchronous delegates

```
static void Main() {  
    Func<string, int> method = Work;  
  
    IAsyncResult cookie = method.BeginInvoke ("test", null, null);  
  
    //  
  
    // ... here's where we can do other work in parallel...  
  
    //  
  
    int result = method.EndInvoke (cookie);  
  
    //EndInvoke waits for the asynchronous delegate to finish executing  
    //and it receives the return value  
  
    Console.WriteLine ("String length is: " + result);  
}
```

```
static int Work (string s) { return s.Length; }
```

# BackgroundWorker

1. Instantiate BackgroundWorker and handle the DoWork event.
2. Call RunWorkerAsync, optionally with an object argument. Any argument passed to RunWorkerAsync will be forwarded to DoWork's event handler, via the event argument's Argument property.

# BackgroundWorker

```
class Program {  
  
    static BackgroundWorker _bw = new BackgroundWorker();  
  
    static void Main() {  
  
        _bw.DoWork += bw_DoWork;  
  
        _bw.RunWorkerAsync ("Message to worker");  
  
        Console.ReadLine();  
  
    }  
  
    static void bw_DoWork (object sender, DoWorkEventArgs e){  
  
        // This is called on the worker thread  
  
        Console.WriteLine (e.Argument);      // writes "Message to worker"  
  
        // Perform time-consuming task...  
    }  
}
```

# **Task Parallel Library**

# Via Task Parallel Library

- enter the thread pool using the **Task class from System.Threading.Tasks**

```
static void Main() {  
    Task.Factory.StartNew (Go);  
  
/*Task.Factory.StartNew returns a Task object, which can  
then be used to monitor the task — for instance, you  
can wait for it to complete by calling its Wait method */  
}  
  
static void Go(){  
    Console.WriteLine ("Hello from the thread pool!");  
}
```

```
static void Main(){
    //Task<TResult> class lets you get a return value back from
    //the task after it finishes executing
    Task<string> task = Task.Factory.StartNew<string>
        ( () => DownloadString ("http://www.aaa.com") );
    // do other work here and it will execute in parallel
    RunSomeOtherMethod();

    // When we need the task's return value, we query its Result property:
    // If it's still executing, the current thread will now block (wait)
    // until the task finishes:
    string result = task.Result;

    static string DownloadString (string uri){
        using (var wc = new System.Net.WebClient())
            return wc.DownloadString (uri);}
```

# Waiting on Tasks

You can explicitly wait for a task to complete in two ways:

- Calling its `Wait` method (optionally with a timeout)
- Accessing its `Result` property (in the case of `Task<TResult>`)

You can also wait on multiple tasks at once via the static methods:

- `Task.WaitAll` (waits for all the specified tasks to finish)
- `Task.WaitAny` (waits for just one task to finish).

# Exception handling

```
int x = 0;  
  
Task<int> calc = Task.Factory.StartNew (() => 7 / x);  
  
try  
{  
    Console.WriteLine (calc.Result);  
}  
  
catch (AggregateException aex)  
{  
    Console.Write (aex.InnerException.Message);  
    // Attempted to divide by 0  
}
```

# Cancelling tasks

- You can optionally pass in a cancellation token when starting a task
- The cancel can be done by calling cancel on the CancellationSource

```
var cancelSource = new CancellationTokenSource();
CancellationToken token = cancelSource.Token;
Task task = Task.Factory.StartNew (() => {
    // Do some stuff...
    token.ThrowIfCancellationRequested(); // Check for cancellation request
    // Do some stuff...
}, token);
...
cancelSource.Cancel();
```

# Cancelling tasks

- To detect a canceled task:

```
try
{
    task.Wait();
}

catch (AggregateException ex)
{
    if (ex.InnerException is OperationCanceledException)
        Console.WriteLine("Task canceled!");
}
```

# Continuations

- it's useful to start a task right after another one completes (or fails).
- the ContinueWith method on the Task class does exactly this:

```
Task.Factory.StartNew<int> (() => 8)
```

```
.ContinueWith (ant => ant.Result * 2)
```

```
.ContinueWith (ant => Math.Sqrt (ant.Result))
```

```
.ContinueWith (ant => Console.WriteLine (ant.Result)); // 4
```

# Parallel class

- a basic form of structured parallelism via three static methods in the Parallel class:
  - `Parallel.Invoke`: executes an array of delegates in parallel
  - `Parallel.For`: performs the parallel equivalent of a C# for loop
  - `Parallel.ForEach`: performs the parallel equivalent of a C# foreach loop

# Synchronization

# Synchronization

- coordinating the actions of threads for a predictable outcome
- its constructs can be divided into four categories:
  1. **Simple blocking methods** (e.g. Sleep, Join): wait for another thread to finish or for a period of time to elapse

# Synchronization

2. **Locking constructs** (e.g. Lock): limit the number of threads that can perform some activity or execute a section of code at a time.
3. **Signaling constructs**: allow a thread to pause until receiving a notification from another, avoiding the need for inefficient polling.
4. **Nonblocking synchronization constructs** (e.g. Volatile, Interlocked): protect access to a common field by calling upon processor primitives

# Blocking

- thread execution is paused for some reason
- thread consumes no processor time until blocking condition is satisfied

Unblocking happens in one of four ways:

- by the blocking condition being satisfied
- by the operation timing out (if a timeout is specified)
- by being interrupted via `Thread.Interrupt`
- by being aborted via `Thread.Abort`

# Blocking Versus Spinning

A thread must pause until a certain condition is met:

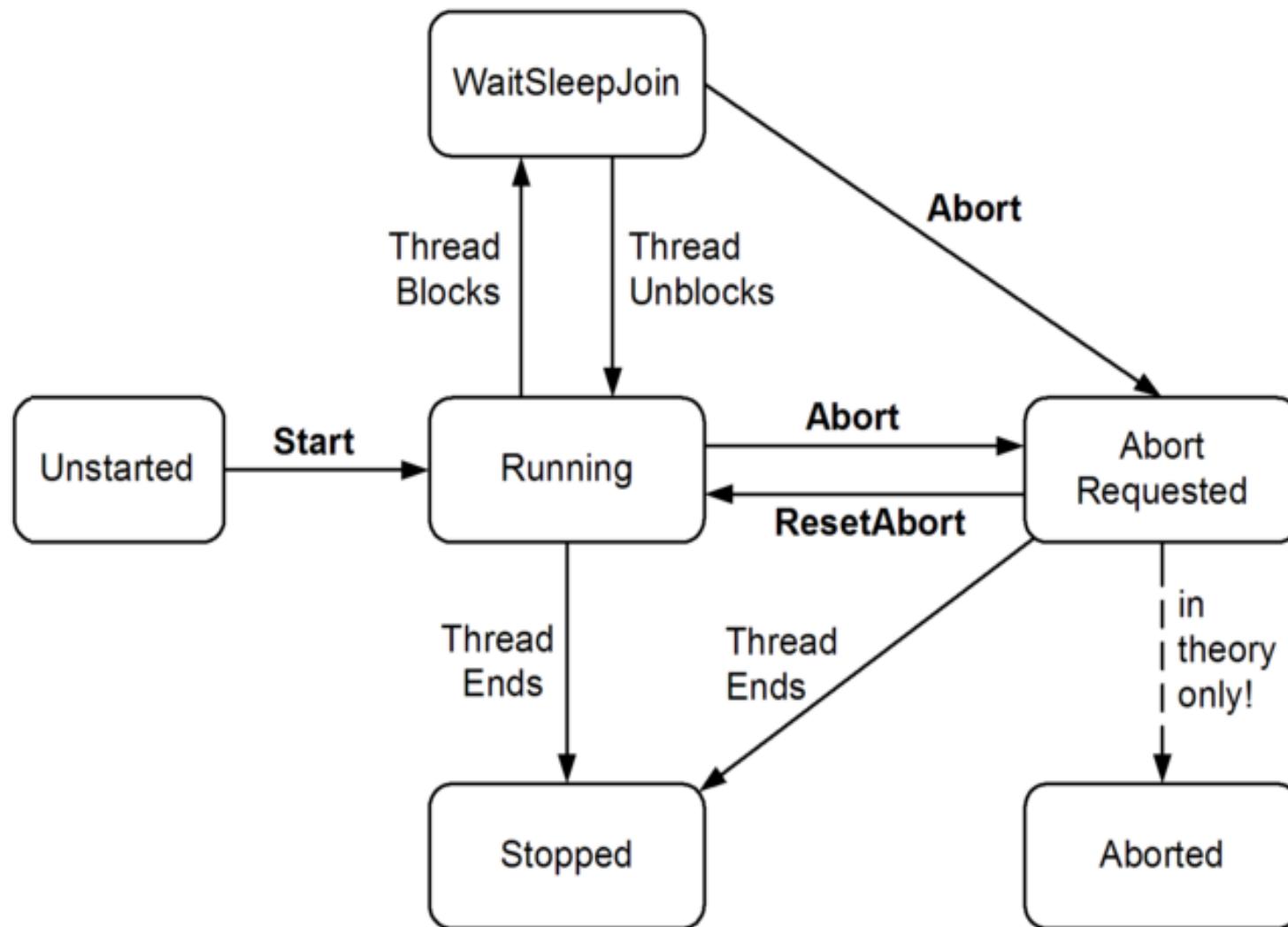
- efficiently: signaling and locking constructs achieve this by blocking until condition is satisfied
- simply but inefficiently: by spinning in a polling loop, e.g.:

```
while (!proceed);
```

or in a mixed way:

```
while (!proceed) Thread.Sleep (10);
```

# ThreadState property



# Locking

- Exclusive locking is used to ensure that only one thread can enter particular sections of code at a time

```
class ThreadSafe{  
    static readonly object _locker = new object();  
    static int _val1, _val2;  
  
    static void Go(){  
        lock (_locker){  
            if (_val2 != 0) Console.WriteLine (_val1 / _val2);  
            _val2 = 0;  
        }  
    }  
}
```

# Locking

- **lock** statement is in fact a syntactic shortcut for a call to the methods **Monitor.Enter** and **Monitor.Exit**

```
Monitor.Enter (_locker);  
try  
{  
    if (_val2 != 0) Console.WriteLine (_val1 / _val2);  
    _val2 = 0;  
}  
finally { Monitor.Exit (_locker); }
```

# Choosing the Synchronization Object

1. class ThreadSafe{

```
List <string> _list = new List <string>();
```

```
void Test(){
```

```
lock (_list) {
```

```
_list.Add ("Item 1");
```

```
...
```

2. lock the entire object: **lock (this) { ... }**

3. in case of static fields/methods: **lock (typeof (Widget)) { ... }**

# When to Lock

- need to lock around accessing any writable shared field

```
class ThreadSafe{  
    static readonly object _locker = new object();  
    static int _x;  
  
    static void Increment() { lock (_locker) _x++; }  
    static void Assign()  { lock (_locker) _x = 123; }  
}
```

# Locking and Atomicity

- if a group of variables are always read and written within the same lock, we can say the variables are **read and written atomically**
- for example x and y are accessed atomically:  
**lock (locker) { if (x != 0) y /= x; }**

# Nested Locking

```
static readonly object _locker = new object();

static void Main(){
    lock (_locker){
        AnotherMethod();
        // We still have the lock - because locks are reentrant.
    }
}

static void AnotherMethod(){
    lock (_locker) { Console.WriteLine ("Another method"); }
}
```

# Deadlocks

- when two threads each waits for a resource held by the other, so neither can proceed

```
object locker1 = new object();
object locker2 = new object();

new Thread () => {

    lock (locker1) {

        Thread.Sleep (1000);

        lock (locker2); // Deadlock

    }

}).Start();

lock (locker2){

    Thread.Sleep (1000);

    lock (locker1); // Deadlock

}
```

# Mutex

- is like a lock, but it can work across multiple processes
- can be computer-wide as well as application-wide
- Mutex class:
  - WaitOne method to lock
  - ReleaseMutex to unlock
- a Mutex can be released only from the same thread that obtained it.

Example: A common use for a cross-process Mutex is to ensure that only one instance of a program can run at a time

```
class OneAtATimePlease{

    static void Main() {

        // Naming a Mutex makes it available computer-wide.

        using (var mutex = new Mutex (false, "UniqueName")){
            // Wait a few seconds, in case another instance
            // of the program is still in the process of shutting down.

            if (!mutex.WaitOne (TimeSpan.FromSeconds (3), false)) {
                Console.WriteLine ("Another app instance is running. Bye!");
                return;
            }

            RunProgram();
        }
    }
}
```

```
static void RunProgram(){

    Console.WriteLine ("Running. Press Enter to exit");

    Console.ReadLine();
}}
```

# Semaphore

- preventing too many threads from executing a particular piece of code at once.
- is like a room, it has a certain capacity. Once it's full, no more people can enter, and a queue builds up outside. Then, for each person that leaves, one person enters from the head of the queue.
- it has no owner, any thread can call release on a semaphore

```
class TheRoom {  
    static SemaphoreSlim _sem = new SemaphoreSlim (3); // Capacity of 3  
  
    static void Main(){  
        for (int i = 1; i <= 5; i++) new Thread (Enter).Start (i);  
    }  
  
    static void Enter (object id){  
        Console.WriteLine (id + " wants to enter");  
        _sem.Wait();  
        Console.WriteLine (id + " is in!"); // Only three threads  
        Thread.Sleep (1000 * (int) id); // can be here at  
        Console.WriteLine (id + " is leaving"); // a time.  
        _sem.Release();  
    }  
}
```

# Signaling with Event Wait Handles

## Signaling:

- when one thread waits until it receives notification from another

## Event wait handles:

- are the simplest of the signaling constructs
- are unrelated to C# events
- come in three flavors:
  - AutoResetEvent,
  - ManualResetEvent,
  - CountdownEvent

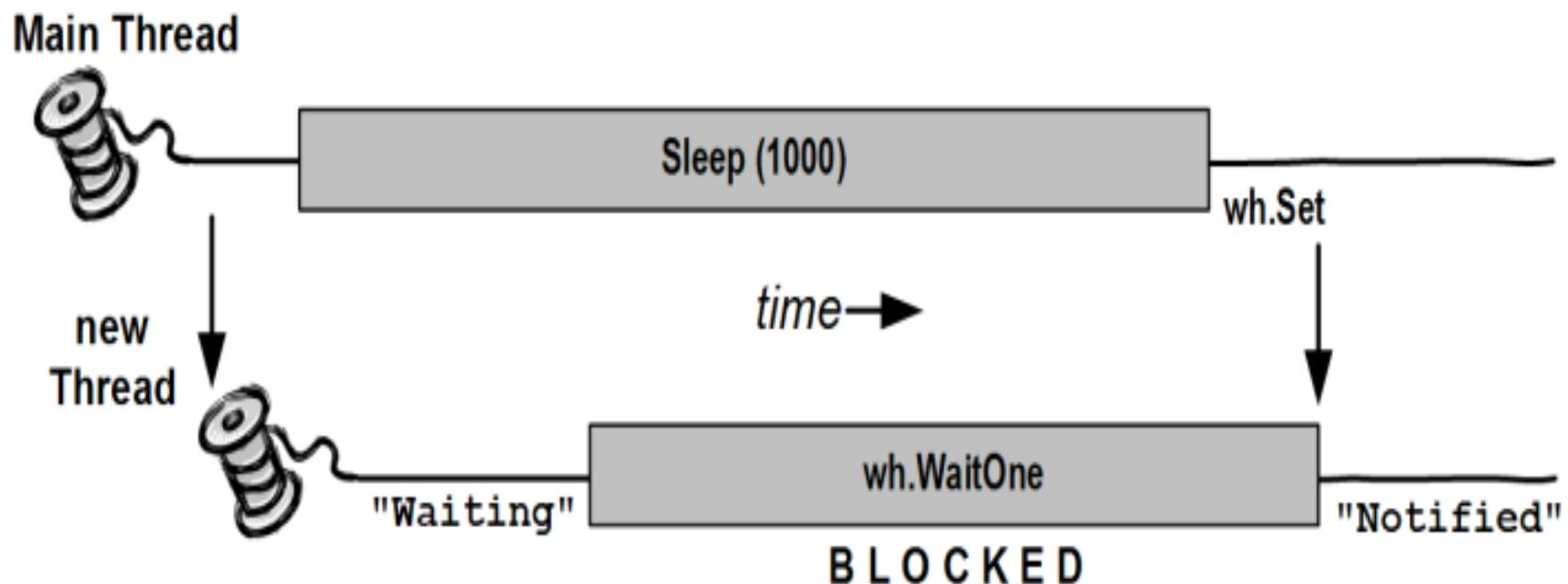
# AutoResetEvent

- is like a ticket turnstile: inserting a ticket lets exactly one person through
- “auto” in the class’s name refers to the fact that an open turnstile automatically closes or “resets” after someone steps through
- a thread waits, or blocks, at the turnstile by calling **WaitOne**
- a ticket is inserted by calling the **Set** method

# AutoResetEvent

- if a number of threads call WaitOne, a **queue** builds up behind the turnstile
- **any thread** with access to the AutoResetEvent object **can call Set** on it to release one blocked thread
- If **Set is called when no thread is waiting**, the handle stays open for as long as it takes until some thread calls WaitOne
- **calling Set repeatedly** on a turnstile at which no one is waiting: only the next single person is let through and the extra tickets are “wasted.”

Example: a thread is started whose job is simply to wait until signaled by another thread:



```
class BasicWaitHandle {  
    static EventWaitHandle _waitHandle = new AutoResetEvent (false);  
  
    static void Main(){  
        new Thread (Waiter).Start();  
  
        Thread.Sleep (1000);           // Pause for a second...  
  
        _waitHandle.Set();           // Wake up the Waiter.  
    }  
  
    static void Waiter(){  
        Console.WriteLine ("Waiting...");  
  
        _waitHandle.WaitOne();       // Wait for notification  
  
        Console.WriteLine ("Notified");  
    }  
}
```

# Two-way Signaling

Example:

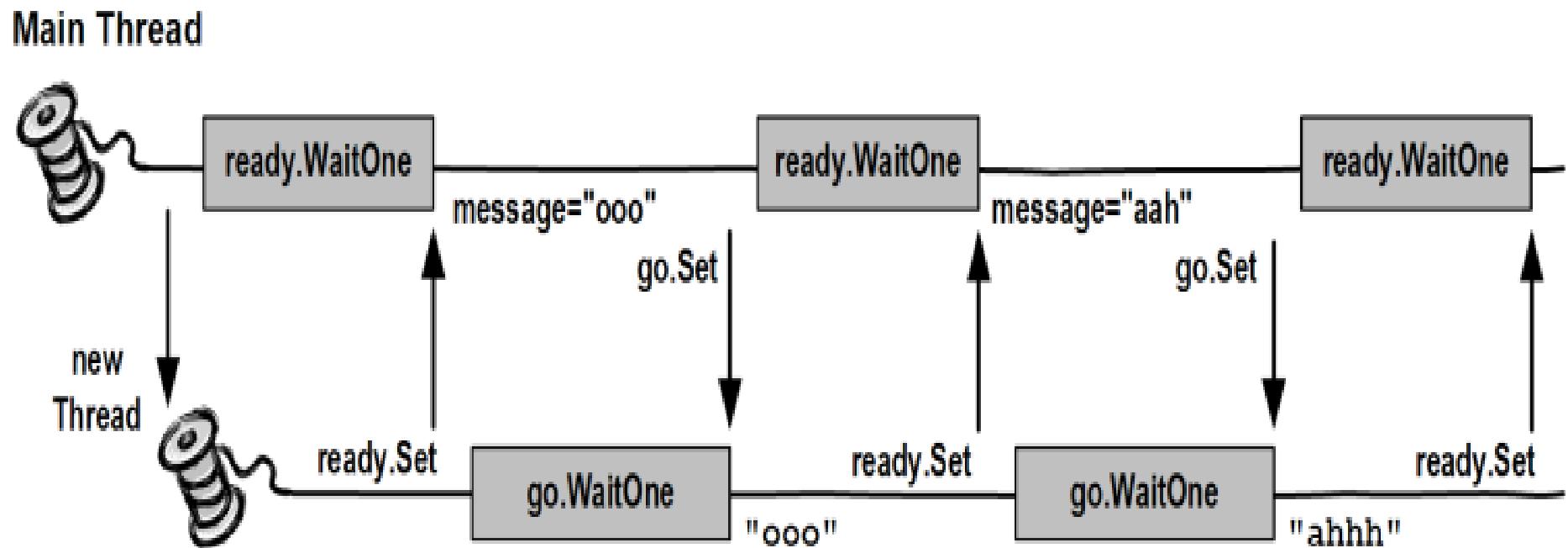
the main thread must signal a worker thread  
three times in a row

Bad solution:

- If the main thread simply calls Set on a wait handle several times in rapid succession, the second or third signal may get lost, since the worker may take time to process each signal.

# Correct solution:

- the main thread must wait until the worker is ready before signaling it (using 2 AutoResetEvent)



```
class TwoWaySignaling {  
    static EventWaitHandle _ready = new AutoResetEvent (false);  
    static EventWaitHandle _go = new AutoResetEvent (false);  
    static readonly object _locker = new object();  
    static string _message;  
  
    static void Main() {  
        new Thread (Work).Start();  
        _ready.WaitOne();           // First wait until worker is ready  
        lock (_locker) _message = "ooo";  
        _go.Set();                 // Tell worker to go  
  
        _ready.WaitOne();  
        lock (_locker) _message = "ahhh"; // Give the worker another message  
        _go.Set();
```

```
_ready.WaitOne();

lock (_locker) _message = null; // Signal the worker to exit

_go.Set();

}

static void Work(){

while (true){

_ready.Set(); // Indicate that we're ready

_go.WaitOne(); // Wait to be kicked off...

lock (_locker){

if (_message == null) return; // Gracefully exit

Console.WriteLine (_message);

}

}

}

}
```

# Producer/consumer queue

A producer/consumer queue works as follows:

- a queue is set up to describe work items — or data upon which work is performed.
- when a task needs executing, it's enqueued, allowing the caller to get on with other things.
- one or more worker threads plug away in the background, picking off and executing queued items.

```
using System;  
using System.Threading;  
using System.Collections.Generic;  
class ProducerConsumerQueue : IDisposable {  
    EventWaitHandle _wh = new AutoResetEvent (false);  
    Thread _worker;  
    readonly object _locker = new object();  
    Queue<string> _tasks = new Queue<string>();  
  
    public ProducerConsumerQueue() {  
        _worker = new Thread (Work);  
        _worker.Start();}  
  
    public void EnqueueTask (string task){  
        lock (_locker) _tasks.Enqueue (task);  
        _wh.Set(); }  
}
```

```
void Work() {  
    while (true){  
        string task = null;  
        lock (_locker)  
        if (_tasks.Count > 0) {  
            task = _tasks.Dequeue();  
            if (task == null) return;  
        }  
        if (task != null){  
            Console.WriteLine ("Performing task: " + task);  
            Thread.Sleep (1000); // simulate work...  
        } else  
            _wh.WaitOne(); // No more tasks - wait for a signal  
    }  
}
```

```
public void Dispose(){
    EnqueueTask (null);      // Signal the consumer to exit.
    _worker.Join();          // Wait for the consumer's thread to finish.
    _wh.Close();             // Release any OS resources.
}

static void Main(){
    using (ProducerConsumerQueue q = new ProducerConsumerQueue()){
        q.EnqueueTask ("Hello");
        for (int i = 0; i < 10; i++) q.EnqueueTask ("Say " + i);
        q.EnqueueTask ("Goodbye!");
    }
    // Exiting the using statement calls q's Dispose method, which
    // enqueues a null task and waits until the consumer finishes.
}
}
```

# ManualResetEvent

- is useful in allowing one thread to unblock many other threads
- functions like an ordinary gate
- calling Set opens the gate, allowing any number of threads calling WaitOne to be let through
- calling Reset closes the gate
- threads that call WaitOne on a closed gate will block; when the gate is next opened, they will be released all at once.

# CountdownEvent

- allows waiting on more than one thread
- the class is instantiated with the number of threads or “counts” that have to be waited on
- calling **Signal** decrements the “count”
- calling **Wait** blocks until the count goes down to zero

```
static CountdownEvent _countdown = new CountdownEvent (3);

static void Main(){
    new Thread (SaySomething).Start ("I am thread 1");
    new Thread (SaySomething).Start ("I am thread 2");
    new Thread (SaySomething).Start ("I am thread 3");
    _countdown.Wait(); // Blocks until Signal has been called 3 times
    Console.WriteLine ("All threads have finished speaking!");
}

static void SaySomething (object thing){
    Thread.Sleep (1000);
    Console.WriteLine (thing);
    _countdown.Signal();
}
```

# Barrier class

- A barrier is a user-defined synchronization primitive that enables multiple threads (known as participants) to work concurrently on an algorithm in phases.
- Each participant executes until it reaches the barrier point in the code. The barrier represents the end of one phase of work.
- When a participant reaches the barrier, it blocks until all participants have reached the same barrier.
- After all participants have reached the barrier, you can optionally invoke a post-phase action. This post-phase action can be used to perform actions by a single thread while all other threads are still blocked. After the action has been executed, the participants are all unblocked.

# Barrier class

```
static Barrier _barrier = new Barrier (3);
```

```
static void Main(){
```

```
    new Thread (Speak).Start();
```

```
    new Thread (Speak).Start();
```

```
    new Thread (Speak).Start();
```

```
}
```

```
static void Speak(){
```

```
    for (int i = 0; i < 5; i++){
```

```
        Console.Write (i + " ");
```

```
        _barrier.SignalAndWait();
```

```
    }}
```

```
//0 0 0 1 1 1 2 2 2 3 3 3 4 4 4
```

# Asynchronous programming patterns

# Asynchronous Programming Model (APM) pattern (also called the IAsyncResult pattern)

- asynchronous operations require Begin and End methods (for example, BeginWrite and EndWrite for asynchronous write operations).
- This pattern is no longer recommended for new development

# Event-based Asynchronous Pattern (EAP)

- requires a method that has the Async suffix, and also requires one or more events, event handler delegate types, and EventArg-derived types.
- was introduced in the .NET Framework 2.0.
- It is no longer recommended for new development.

# Task-based Asynchronous Pattern (TAP)

- uses a single method to represent the initiation and completion of an asynchronous operation.
- was introduced in the .NET Framework 4 and is the recommended approach to asynchronous programming in the .NET Framework.
- The `async` and `await` keywords add language support for TAP.

# Async methods

- The method signature includes an `async` modifier.
- The name of an `async` method, by convention, ends with an "Async" suffix.
- The return type is one of the following types:
  - `Task<TResult>` if your method has a return statement in which the operand has type `Tresult`.
  - `Task` if your method has no return statement or has a return statement with no operand.
  - `Void` if you're writing an `async` event handler.

# Async methods

- The method usually includes at least one await expression,
  - which marks a point where the method can't continue until the awaited asynchronous operation is complete.
  - In the meantime, the method is suspended, and control returns to the method's caller.

```
// Three things to note in the signature:  
// - The method has an async modifier.  
// - The return type is Task<int> because the return statement  
// returns an integer.  
// - The method name ends in "Async."  
async Task<int> AccessTheWebAsync() {  
    // You need to add a reference to System.Net.Http to declare client.  
    HttpClient client = new HttpClient();  
  
    // GetStringAsync returns a Task<string>.  
    // That means that when you await the  
    // task you'll get a string (urlContents).  
Task<string> getStringTask =  
    client.GetStringAsync("http://msdn.microsoft.com");  
    82
```

```
// You can do work here that doesn't rely on the string
```

```
// from GetStringAsync.
```

```
DoIndependentWork();
```

```
// The await operator suspends AccessTheWebAsync.
```

```
// - AccessTheWebAsync can't continue until
```

```
// getStringTask is complete.
```

```
// - Meanwhile, control returns to the caller of AccessTheWebAsync.
```

```
// - Control resumes here when getStringTask is complete.
```

```
// - The await operator then retrieves the string
```

```
// result from getStringTask.
```

```
string urlContents = await getStringTask;
```

```
// The return statement specifies an integer result.  
// Any methods that are awaiting AccessTheWebAsync  
//retrieve the length value.  
return urlContents.Length;  
}
```