



Explore new perspectives

ESTECO

Java for Android Applications Development

Carlos Kavka
ESTECO SpA
Area Science Park, Trieste, Italy

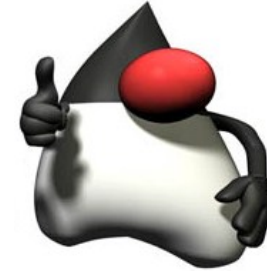
First Regional Workshop on Distributed Embedded Systems

Universiti Tunku Abdul Rahman (UTAR) - Kampar - Malaysia

30 July - 17 August 2012

Objectives

- ✓ provide a general **overview** of Java



- ✓ consider **Android** applications development requirements

- ✓ not a tutorial nor a complete reference, most concepts are introduced with **examples**

Java for Android Applications Development

Part I: Basic concepts

Explore new perspectives

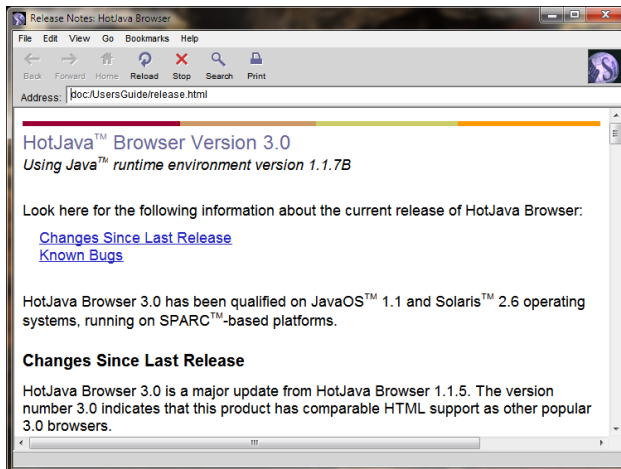
ESTECO

A bit of history



*7

HotJava browser



current installer



1992

Oak

1994

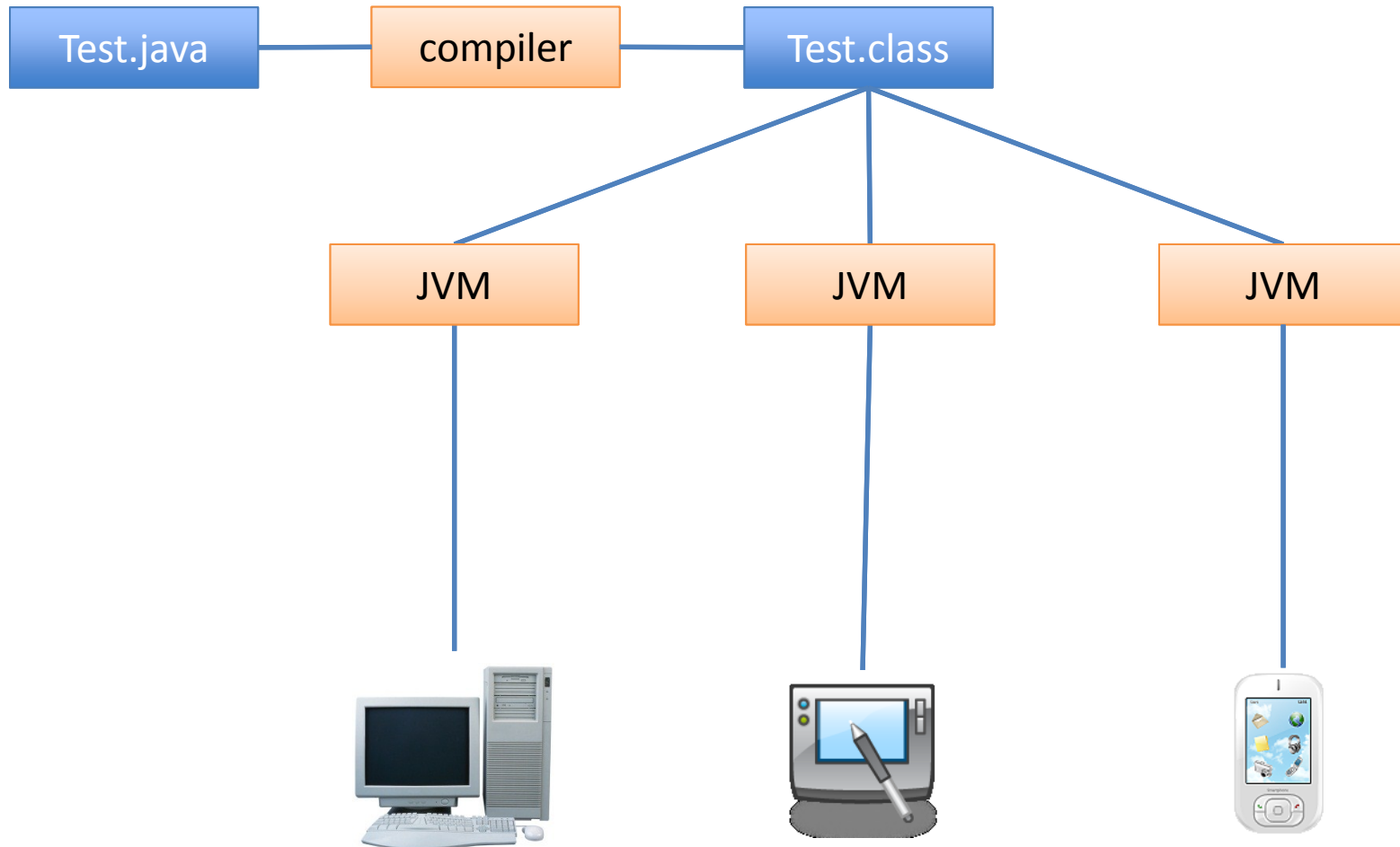
Java

2012

Java SE7

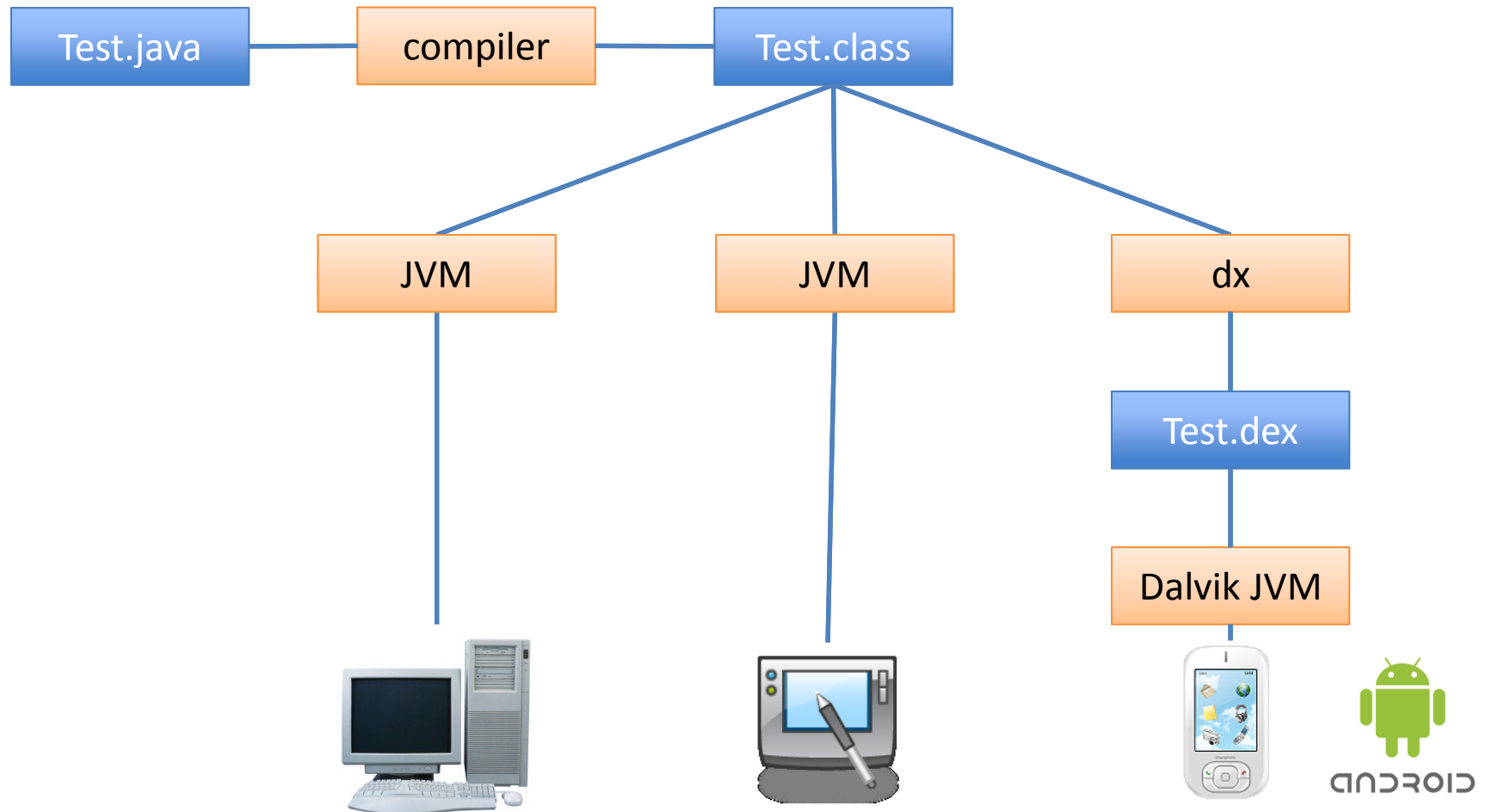
Source: Java Technology, an early history (<http://java.sun.com>)

Java platform



The compiled code is **independent** of the architecture of the computer

Java for Android



Android uses its own virtual machine called **Dalvik**

A first example

```
/**
 * * Hello World Application
 */
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // display output
    }
}
```

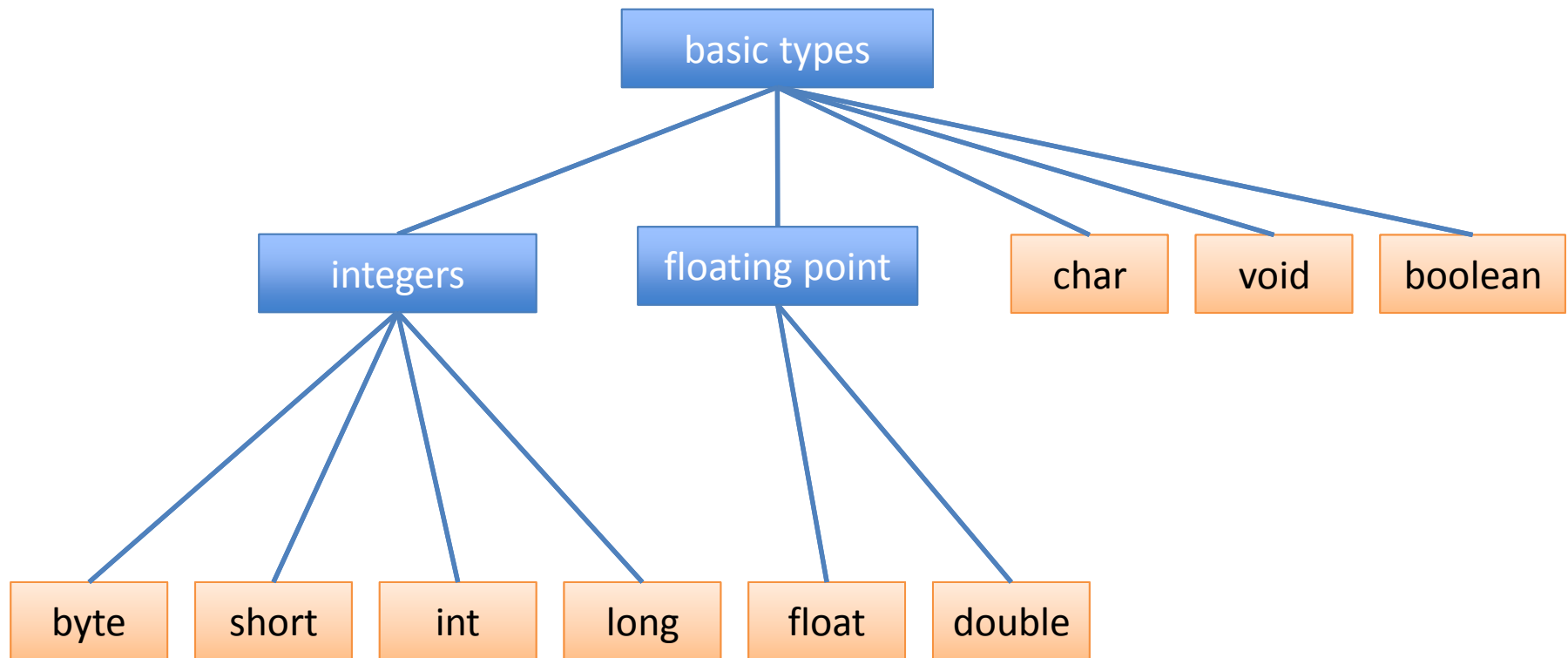
```
$ javac HelloWorld.java
```

```
$ ls
HelloWorld.class
HelloWorld.java
```

```
$ java HelloWorld
Hello World
```

Basic types

Java provides the following basic **types**



Variables and constants definition

```
int x;  
double d = 0.33;  
float f = 0.22f;  
char c = 'a';  
boolean ready = true;  
  
x = 55;
```

The **variables** are declared specifying its type and name, and initialized in the point of declaration, or later with the assignment expression

```
final double pi = 3.1415;  
final int maxSize = 100;  
final char lastLetter = 'z';
```

Constants are declared with the word final in front. The specification of the initial value is compulsory

Strings

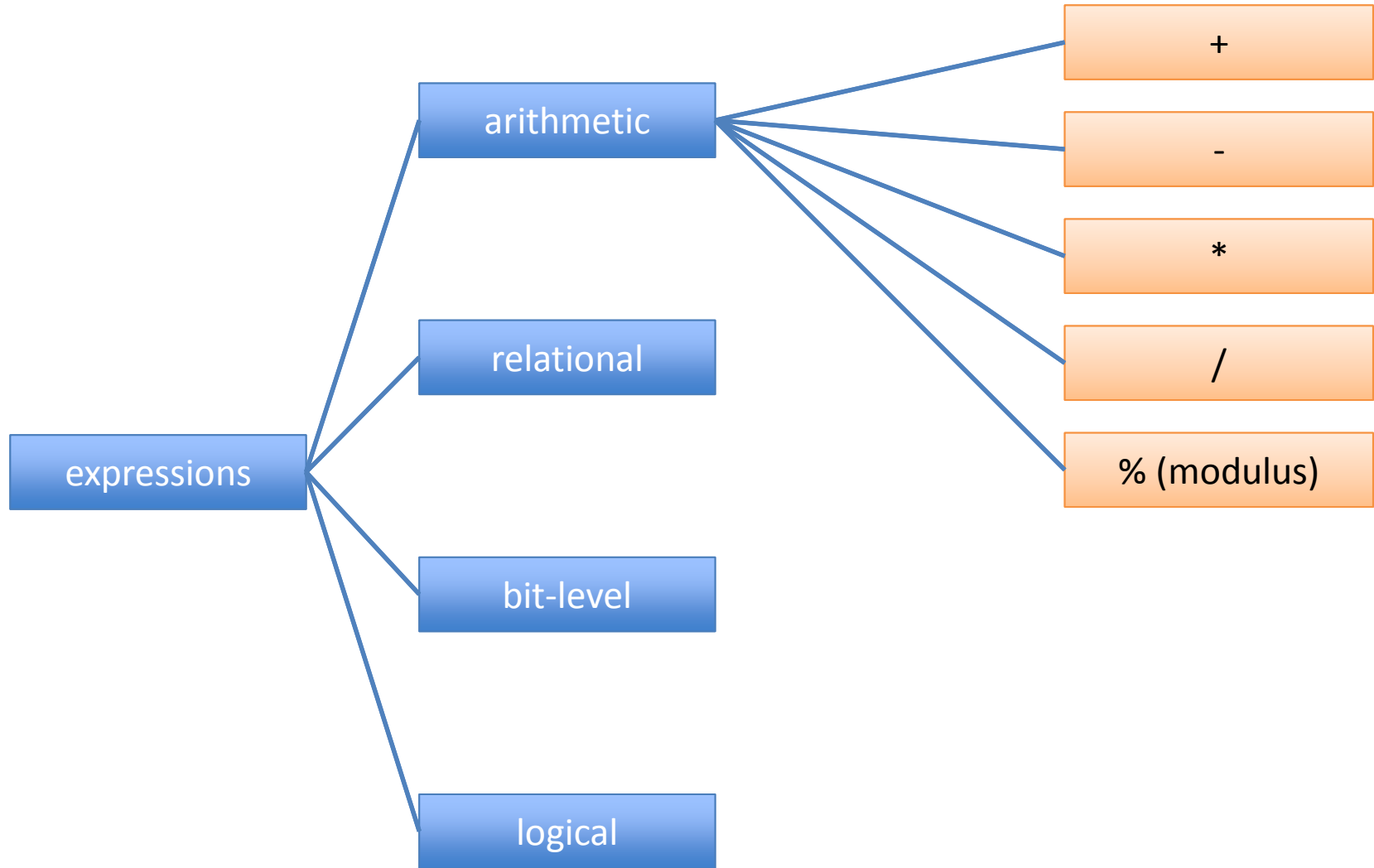
Strings are not a basic type, but defined as a class, more details later!

```
String a = "abc";
```

If the expression begins with a string and uses the + operator, then the next argument is **converted** to a string

```
int cost = 22;  
String b = "the cost is " + cost + " euro";
```

Arithmetic expressions



Example with arithmetic operators

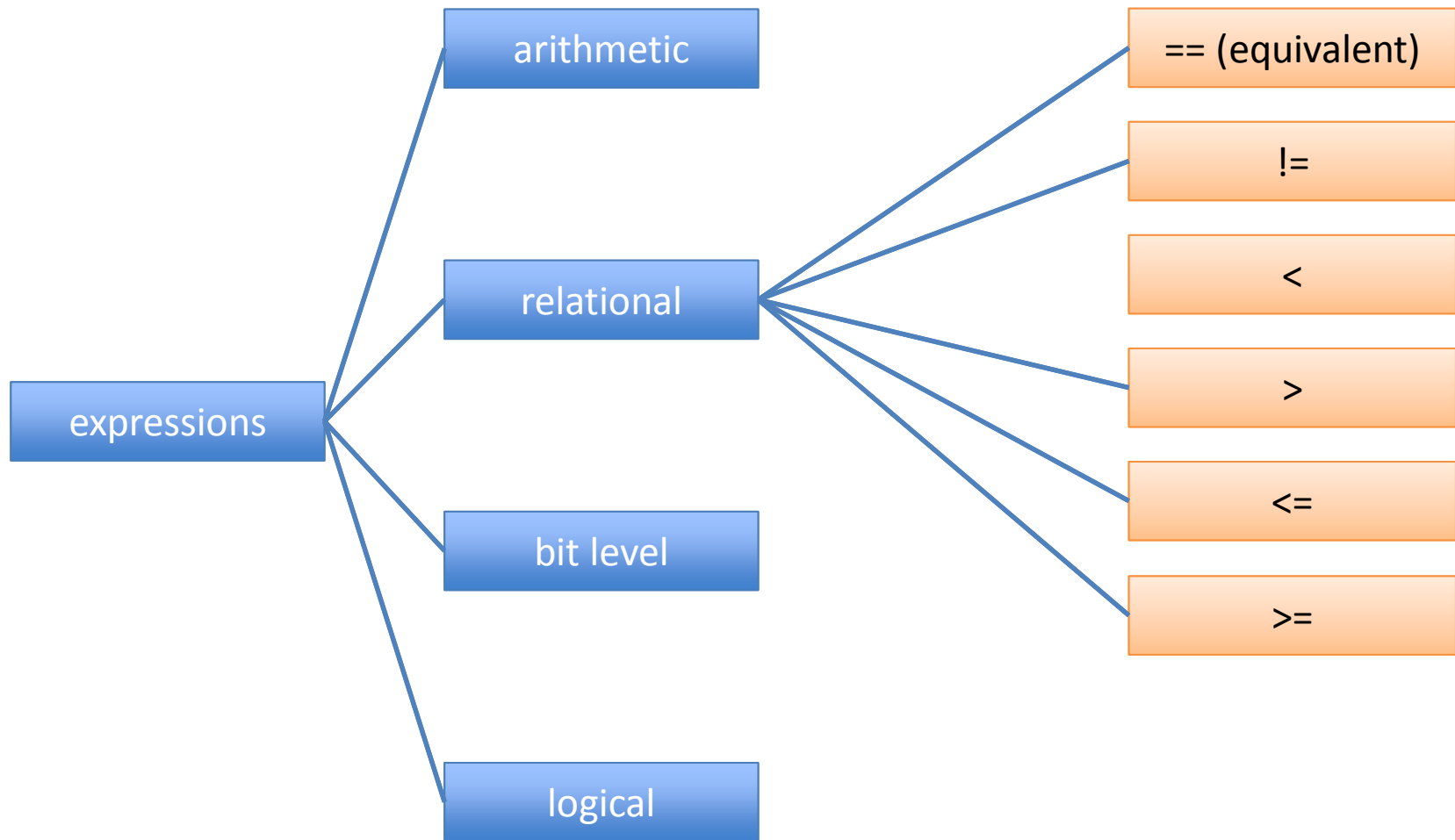
```
class Arithmetic{
    public static void main(String[] args) {
        int x = 12;
        x += 5;                // x = x + 5
        System.out.println(x);

        int a = 12,b = 12;
        System.out.print(a++);  // printed and then incremented
        System.out.print(a);

        System.out.print(++b);  // incremented and then printed
        System.out.println(b);
    }
}
```

```
$ java Arithmetic
17
12 13 13 13
```

Relational expressions

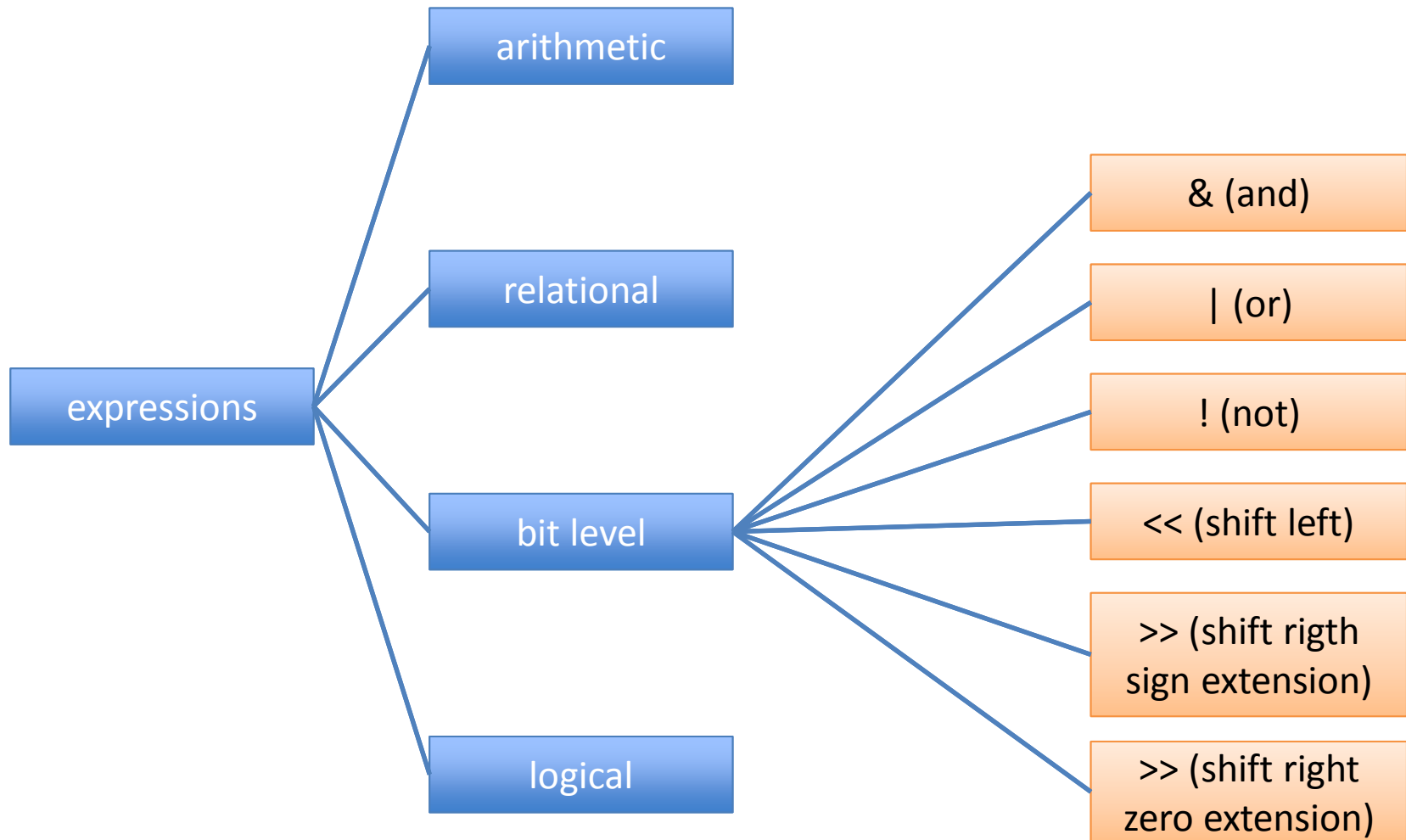


Example with relational operators

```
class Boolean {  
    public static void main(String[] args) {  
        int x = 12,y = 33;  
  
        System.out.println(x < y);  
        System.out.println(x != y - 21);  
  
        boolean test = x >= 10;  
        System.out.println(test);  
    }  
}
```

```
$ java Boolean  
true  
false  
true
```

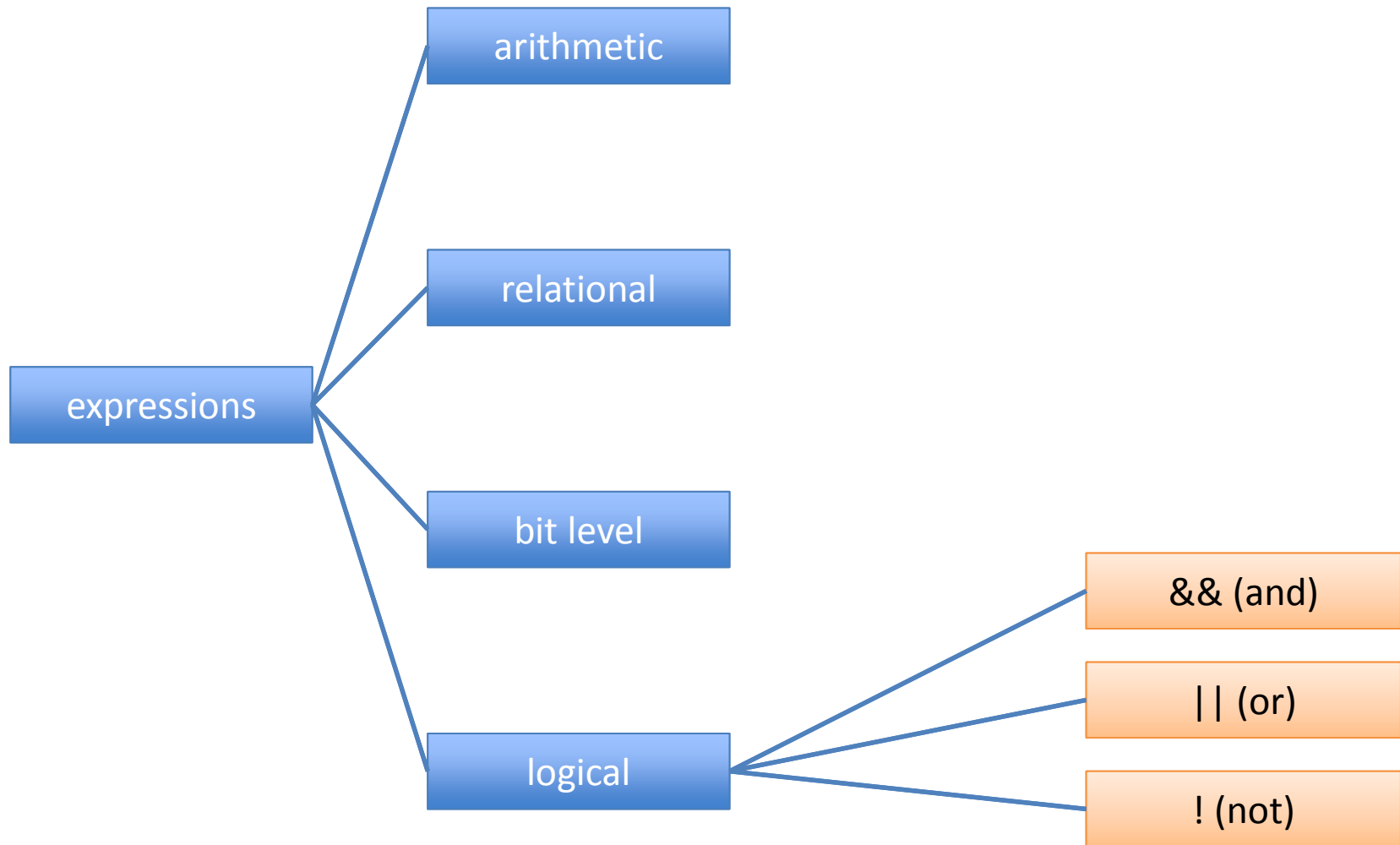
Bit level expressions



Example with bit-level operators

```
class Bits {  
    public static void main(String[] args) {  
        int x = 0x16;                // 00000000000000000000000000000010110  
        int y = 0x33;                // 000000000000000000000000000000110011  
  
        System.out.println(x & y);   // 00000000000000000000000000000010010  
        System.out.println(x | y);   // 000000000000000000000000000000110111  
        System.out.println(~x);       // 111111111111111111111111111111101001  
  
        x = 9;                       // 0000000000000000000000000000001001  
        System.out.println(x >> 3);   // 000000000000000000000000000000000001  
        System.out.println(x >>> 3);  // 000000000000000000000000000000000001  
  
        x = -9;                      // 11111111111111111111111111111110111  
        System.out.println(x >> 3);   // 11111111111111111111111111111111110  
        System.out.println(x >>> 3);  // 00011111111111111111111111111111110  
    }  
}
```


Logical expressions



Example with logical operators

```
class Logical {  
    public static void main(String[] args) {  
        int x = 12,y = 33;  
        double d = 2.45,e = 4.54;  
  
        System.out.println(x < y && d < e);  
        System.out.println(!(x < y));  
  
        boolean test = 'a' > 'z';  
        System.out.println(test || d - 2.1 > 0);  
    }  
}
```

```
$ java Logical  
true  
false  
true
```

Casting

Java performs a **automatic** type conversion when there is no risk for data to be lost.

In order to specify conversions where data can be lost it is necessary to use the **cast** operator.

```
class TestCast {  
    public static void main(String[] args) {  
  
        int a = 'x';    // 34 is an int  
        long b = 34;    // 34 is an int  
        float c = 1002; // 1002 is an int  
        double d = 3.45F; // 3.45F is a float  
  
        long e = 34;  
        int f = (int)e;    // a is a long  
        double g = 3.45;  
        float h = (float)g; // d is a double  
    }  
}
```

Control structures: if

```
class If {  
    public static void main(String[] args) {  
        char c = 'x';  
  
        if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'))  
            System.out.println("letter: " + c);  
        else  
            if (c >= '0' && c <= '9')  
                System.out.println("digit: " + c);  
            else {  
                System.out.println("the character is: " + c);  
                System.out.println("it is not a letter nor a digit");  
            }  
        }  
    }  
}
```

```
$ java If  
letter: x
```

Control structures: while

```
class While {  
    public static void main(String[] args) {  
        final float initialValue = 2.34F;  
        final float step = 0.11F;  
        final float limit = 4.69F;  
        float var = initialValue;  
  
        int counter = 0;  
        while (var < limit) {  
            var += step;  
            counter++;  
        }  
        System.out.println("Incremented " + counter + " times");  
    }  
}
```

```
$ java While  
Incremented 22 times
```

Control structures: for

```
class For {  
    public static void main(String[] args) {  
        final float initialValue = 2.34F;  
        final float step = 0.11F;  
        final float limit = 4.69F;  
        int counter = 0;  
  
        for (float var = initialValue; var < limit; var += step)  
            counter++;  
        System.out.println("Incremented " + counter + " times");  
    }  
}
```

```
$ java For  
Incremented 22 times
```

Control structures: break/continue

```
class BreakContinue {  
    public static void main(String[] args) {  
  
        for (int counter = 0; counter < 10; counter++) {  
  
            if (counter % 2 == 1) continue; // start a new iteration if the counter is odd  
            if (counter == 8) break; // abandon the loop if the counter is equal to 8  
  
            System.out.println(counter);  
        }  
        System.out.println("done.");  
    }  
}
```

```
$ java BreakContinue  
0 2 4 6 done.
```

Control structures: switch

```
class Switch {  
    public static void main(String[] args) {  
  
        boolean leapYear = true;  
        int days = 0;  
  
        for(int month = 1; month <= 12; month++) {  
            switch(month) {  
                case 1: // months with 31 days  
                case 3:  
                case 5:  
                case 7:  
                case 8:  
                case 10:  
                case 12:  
                    days += 31;  
                    break;  

```

```
                case 2: // February is a special case  
                    if (leapYear)  
                        days += 29;  
                    else  
                        days += 28;  
                    break;  
                default: // a month with 30 days  
                    days += 30;  
                    break;  
            }  
        }  
        System.out.println(days);  
    }  
}
```

```
$ java Switch  
366
```


Arrays

Arrays can be used to store a number of elements of the **same** type

```
int[] a;  
float[] b;  
String[] c
```

```
int[] a = {13,56,2034,4,55};  
float[] b = {1.23F,2.1F};  
String[] c = {"Java", "is", "great"};
```

Important: The declaration does not specify a **size**. However, it can be inferred when initialized

Other possibility to allocate space for arrays consists in the use of the operator **new**

```
int i = 3, j = 5;  
double[] d;  
  
d = new double[i+j];
```

Arrays

Components can be accessed with an integer **index** with values from 0 to length minus 1.

```
a[2] = 1000;
```

```
int len = a.length;
```

Every array has a member called **length** that can be used to get the length of the array

Components of the arrays are initialized with **default** values

```
int []a = new int[3];  
for(int i = 0;i < a.length;i++)  
    System.out.println(a[i]);  
}
```

0
0
0

Arrays

```
class Arrays {  
    public static void main(String[] args) {  
        int[] a = {2,4,3,1};  
  
        // compute the summation of the elements of a  
        int sum = 0;  
        for(int i = 0; i < a.length; i++) sum += a[i];  
  
        // create an array of the size computed before  
        float[] d = new float[sum];  
        for(int i = 0; i < d.length; i++) d[i] = 1.0F / (i+1);  
  
        // print values in odd positions  
        for(int i = 1; i < d.length; i += 2)  
            System.out.println("d[" + i + "]=" + d[i]);  
    }  
}
```

```
$ java Arrays  
d[1]=0.5  
d[3]=0.25  
d[5]=0.16666667  
d[7]=0.125  
d[9]=0.1
```

Java for Android Applications Development

Part II: Classes

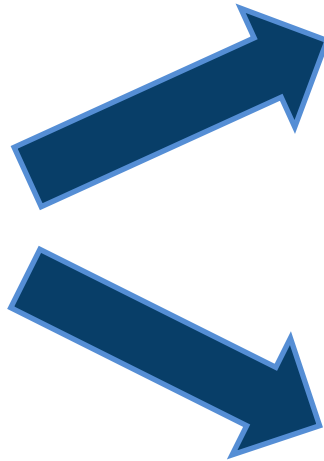
Explore new perspectives

ESTECO

Classes

A class is a **template** for data objects

Inside a class it is possible to define



data elements
(called **instance variables**)

functions
(called **methods**)

Classes

Class Book with three
instance variables

```
class Book {  
    String title;  
    String author;  
    int numberOfPages;  
}
```

```
Book b1 = new Book();  
Book b2 = new Book();  
  
b3 = new Book();
```

New **instances** of the class can be
created with new

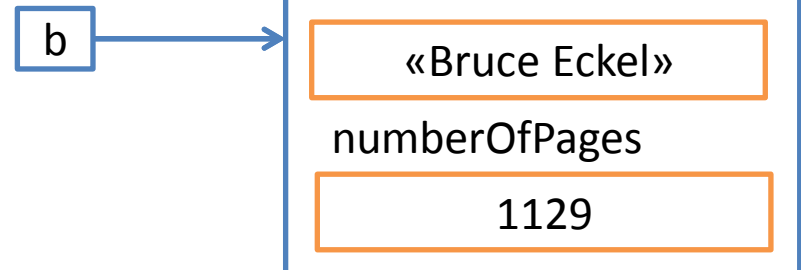
The **instance** variables can
be accessed with the dot
notation

```
b1.title = "Thinking in Java";
```

Classes

```
class ExampleBooks {  
    public static void main(String[] args) {  
  
        Book b = new Book();  
  
        b.title = "Thinking in Java";  
        b.author = "Bruce Eckel";  
        b.numberOfPages = 1129;  
        System.out.println(b.title + " : " + b.author +  
                             " : " + b.numberOfPages);  
    }  
}
```

```
class Book {  
    String title;  
    String author;  
    int numberOfPages;  
}
```



Constructors

- ✓ The constructors allow the **creation** of instances that are properly initialized
 - ✓ A constructor is a method that:
has the **same name** of class to which it belongs and has no specification for the return value.
- ✓ It is possible to define **more than one** constructor for a single class

Constructors

```
class Book {  
    String title;  
    String author;  
    int numberOfPages;  
  
    Book(String tit,String aut,int num) {  
        title = tit;  
        author = aut;  
        numberOfPages = num;  
    }  
}
```

Once a constructor has been defined, the **default** constructor `Book()` is not available any more.

```
class ExampleBooks2 {  
    public static void main(String[] args) {  
        Book b = new Book("Thinking in Java","Bruce Eckel",1129);  
        System.out.println(b.title + " : " + b.author + " : " + b.numberOfPages);  
    }  
}
```

Multiple constructors

It must be possibly
to **identify** them
through the
argument definition

```
class Book {  
    String title;  
    String author;  
    int numberOfPages;  
    String ISBN;
```

```
    Book(String tit,String aut,int num) {  
        title = tit; author = aut;  
        numberOfPages = num;  
        ISBN = "unknown";  
    }
```

```
    Book(String tit,String aut,int num,String isbn) {  
        title = tit; author = aut;  
        numberOfPages = num;  
        ISBN = isbn;  
    }  
}
```

```
a = new Book("Thinking in Java","Bruce Eckel",1129);
```

```
b = new Book("Thinking in Java","Bruce Eckel",1129,"0-13-027363");
```

Methods

- ✓ A method is used to implement the **messages** that an instance (or a class) can receive.
- ✓ It is called by using the **dot** notation.
- ✓ It is implemented as a **function**, specifying arguments and type of the return value.

Methods

```
class Book {  
    ...  
    public String getInitials() {  
        String initials = "";  
        for(int i = 0; i < author.length(); i++) {  
            char currentChar = author.charAt(i);  
            if (currentChar >= 'A' && currentChar <= 'Z')  
                initials = initials + currentChar + '.';  
        }  
        return initials;  
    }  
}
```

Initials: B.E.

```
b = new Book("Thinking in Java",  
            "Bruce Eckel", 1129);  
System.out.println(b.getInitials());
```

b

title

«Thinking in Java»

author

«Bruce Eckel»

numberOfPages

1129

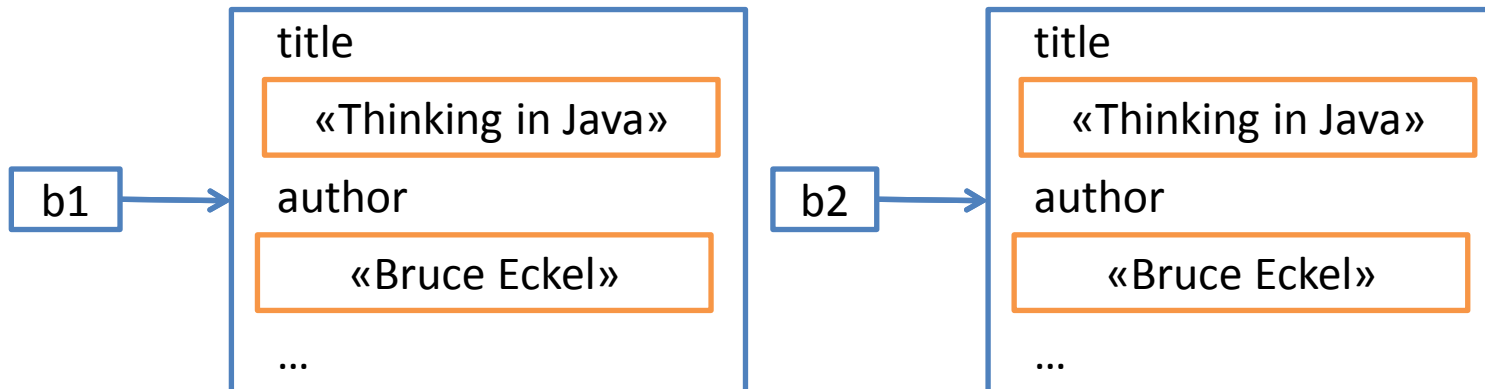
isbn

«unknown»

Equality and equivalence

```
class ExampleBooks6 {  
    public static void main(String[] args) {  
  
        Book b1 = new Book("Thinking in Java", "Bruce Eckel", 1129);  
        Book b2 = new Book("Thinking in Java", "Bruce Eckel", 1129);  
  
        if (b1 == b2)  
            System.out.println(«Same»);  
        else  
            System.out.println(«Different»);  
    }  
}
```

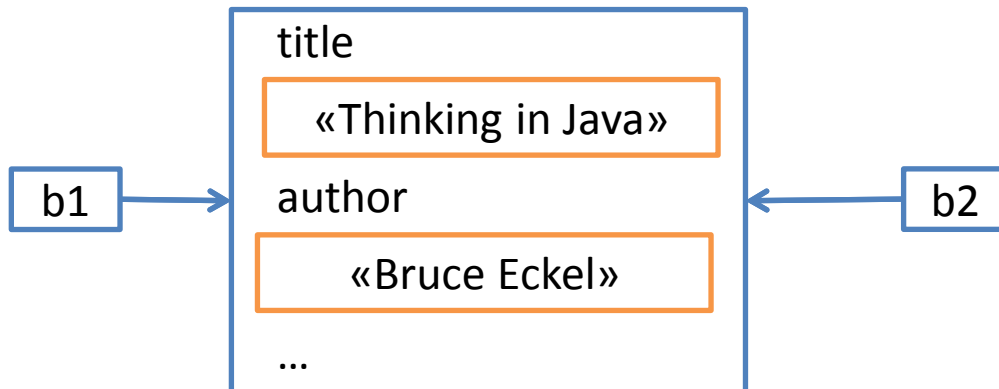
\$ java ExampleBooks6
Different



Equality and equivalence

```
class ExampleBooks6a {  
    public static void main(String[] args) {  
  
        Book b1 = new Book("Thinking in Java","Bruce Eckel",1129);  
        Book b2 = b1;  
  
        if (b1 == b2)  
            System.out.println("Same");  
        else  
            System.out.println("Different");  
    }  
}
```

\$ java ExampleBooks6a
Same



Static instance variables

- ✓ Class variables are fields that **belong to the class** and do not exist in each instance.
- ✓ There is always **only one copy** of this data field, independently of the number of the instances that were created.

Static instance variables

```
class Book {  
    ...  
    static String location;  
    ...  
    public void setLocation(String name) {  
        location = name;  
    }  
    public String getLocation() {  
        return location;  
    }  
}
```

Location of book b1: Kampar
Location of book b2: Kampar

```
Book b1,b2;  
b1 = new Book("Thinking in Java","Bruce Eckel",1129);  
b2 = new Book("Java in a nutshell","David Flanagan",353);  
b1.setLocation("Kampar");  
System.out.println("Location of book b1: " + b1.getLocation());  
System.out.println("location of book b2: " + b2.getLocation());
```


Static methods

- ✓ With the **same idea** of the static data members, it is possible to define class methods or static methods
 - ✓ These methods **do not work** directly with instances but with the class
- ✓ Can access **only** static instance variables

Static methods

The method `getLocation()` is a good candidate to be defined as a **static** method

```
class Book {  
    ...  
    static String location;  
    ...  
    public static String getLocation() {  
        return "Books are located in" + location;  
    }  
}
```

Book are located in: Kampar
Books are located in: Kampar

```
Book b1,b2;  
b1 = new Book("Thinking in Java","Bruce Eckel",1129);  
b2 = new Book("Java in a nutshell","David Flanagan",353);  
b1.setLocation("Kampar");  
System.out.println(b1.getLocation ());  
System.out.println(Book.getLocation());
```

Instance variables initialization

- ✓ All instance variables are **guaranteed** to have an initial value.
- ✓ The **value is 0** for basic types and null for references
- ✓ Instance variables can be **also** initialized by calling instance methods

Instance variables initialization

```
class Values {  
    int x = 2;  
    int y;  
    float f = inverse(x);  
    String s;  
    Book b;  
    Values(String str) { s = str; }  
    public float inverse(int value) { return 1.0F / value; }  
    public void dump() { System.out.println("" + x + "," + y + "," + f + "," + s + "," + b); }  
}
```

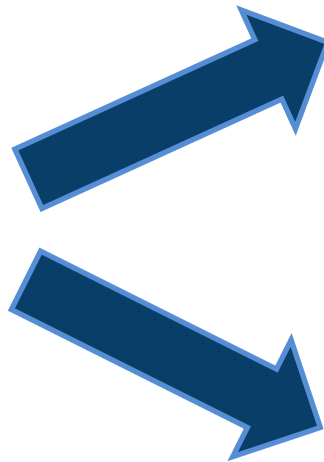
```
$ java InitialValues  
2, 0, 0.5, hello, null
```

```
class InitialValues {  
    public static void main(String[] args) {  
        Values v = new Values("hello");  
        v.dump()  
    }  
}
```

The “this” keyword

The keyword **this**, when used inside a method, refers to the receiver object

It has **two main uses**:



to **return a reference** to the receiver object from a method

to **call constructors** from other constructors.

The “this” keyword

The class Book has
two constructors

```
Book(String tit,String aut,int num) {  
    title = tit; author = aut; numberOfPages = num;  
    ISBN = "unknown";  
}  
Book(String tit,String aut,int num,String isbn) {  
    title = tit; author = aut; numberOfPages = num;  
    ISBN = isbn;  
}
```

```
Book(String tit,String aut,int num,String isbn) {  
    this(tit,aut,num); ISBN = isbn;  
}
```

The second can be better
defined in terms of the first
one

The “this” keyword

The method setLocation
in the previous Book
class could have been
defined as

```
class Book {  
    ...  
    static String location;  
    ...  
    public Book setLocation(String name) {  
        location = name;  
        return this;  
    }  
}
```

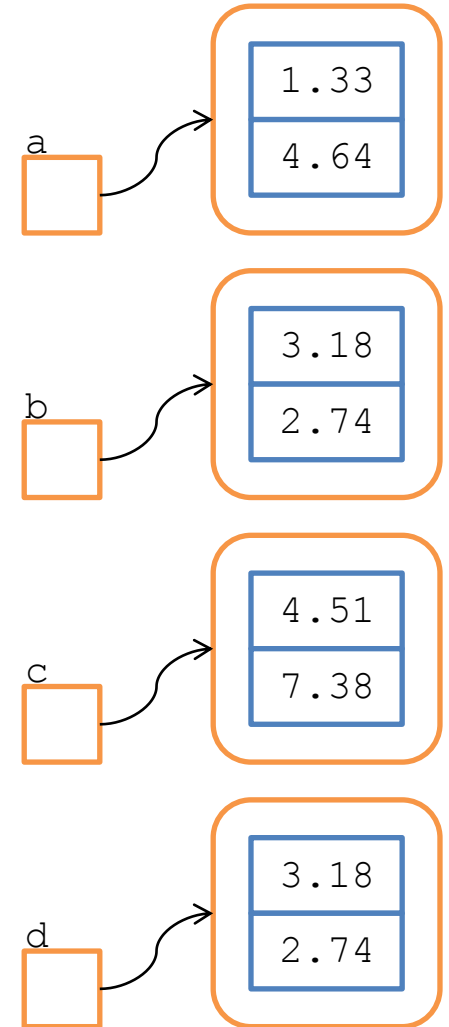
Operations can be
performed now in
“cascade” mode

```
Book b1,b2;  
b1 = new Book("Thinking in Java","Bruce Eckel",1129);  
  
System.out.println(«Initials: " + b1.setLocation("Kampar").getInitials());
```

A complete example

```
class TestComplex {  
  
    public static void main(String[] args) {  
        Complex a = new Complex(1.33,4.64);  
        Complex b = new Complex(3.18,2.74);  
        Complex c = a.add(b);  
  
        System.out.println("c=a+b=" + c.getReal() +  
                           " " + c.getImaginary());  
  
        Complex d = c.sub(a);  
        System.out.println("d=c-a=" + d.getReal() +  
                           " " + d.getImaginary());  
    }  
}
```

```
$ java TestComplex  
c=a+b= 4.51 7.38 d=c-a= 3.18 2.74
```



A complete example

```
class Complex {  
  
    double real;    // real part  
    double im;      // imaginary part  
  
    Complex(double r, double i) {  
        real = r;  
        im = i;  
    }  
  
    public double getReal() {  
        return real;  
    }  
  
    public double getImaginary() {  
        return im;  
    }  
}
```

```
a = new Complex(1.33, 4.64);
```

```
double realPart = a.getReal();
```

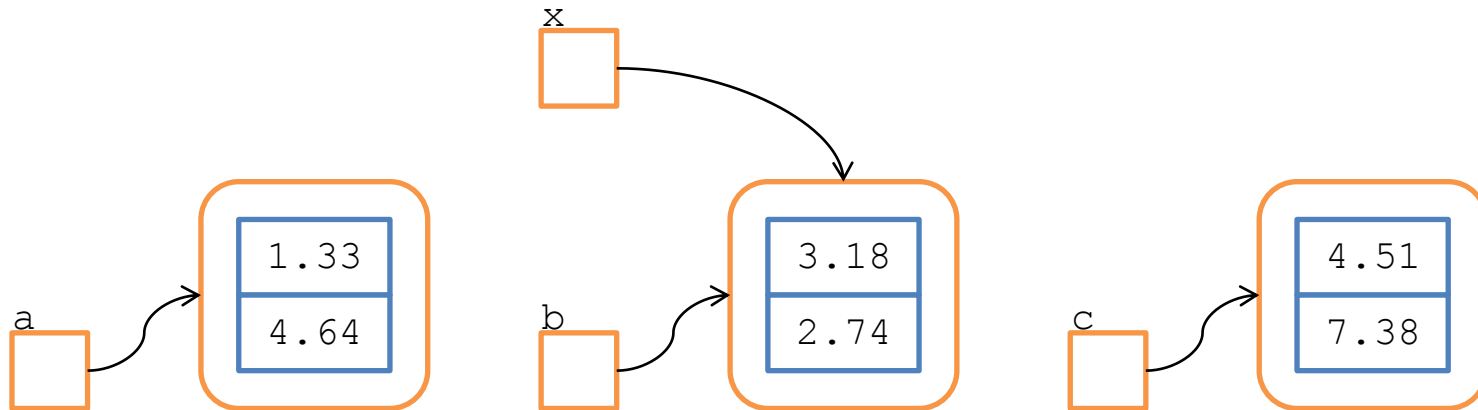
```
double imPart = a.getImaginary();
```

A complete example

```
public Complex add(Complex x) {  
    return new Complex(real + x.real, im + x.im);  
}
```

```
public Complex sub(Complex x) {  
    return new Complex(real - x.real, im - x.im);  
}
```

```
Complex c = a.add(b);
```



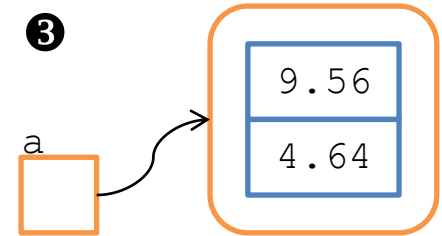
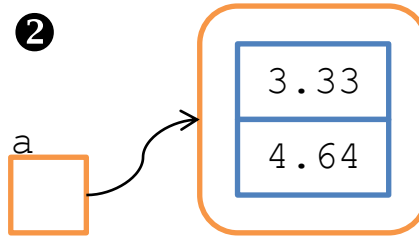
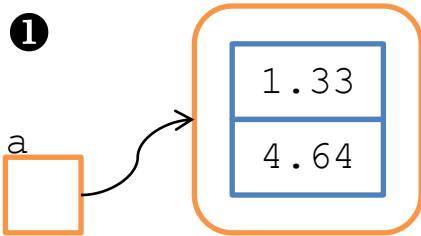
A complete example

```
public Complex addReal(double x) {  
    real += x;  
    return this;  
}
```

```
Complex(Complex x) {  
    this(x.real,x.im);  
}
```

The method `addReal` increments just the real part of the receptor of the message with the value passed as argument

- 1 `Complex a = new Complex(1.33, 4.64);`
- 2 `a.addReal(2.0);`
- 3 `a.addReal(3.0).addReal(3.23);`



Type wrappers

- ✓ Primitive types are used for **performance** reasons, however many situations require an object
- ✓ Type wrappers are classes that **encapsulate primitive types** within an object
- ✓ There exist one type wrapper class **for each primitive type**

Boxing and Unboxing operations

boxing and unboxing operations are provided to **encapsulate/extract** the values to/from an object.

```
Integer iObject = new Integer(21);  
int i = iObject.intValue();
```

```
Integer iObject = 21;  
int i = iObject;
```

Auto-boxing and auto-unboxing operations are provided to make **easier** to work with wrapped objects:

However... these operations add **overhead**, to be used only when required.

Methods with variable number of arguments

A variable length argument list is specified with three periods:

```
int add(int ... values) {  
    int summation = 0;  
    for(int i = 0; i < values.length; i++) {  
        summation += values[i];  
    }  
    return summation;  
}
```

```
int sum = add(1, 2, 3, 4, 5);
```

The argument is implicitly declared as an array, however, it can be called with a variable number of arguments

Inner classes

```
public class A {  
    int x;  
    B b;  
  
    class B {  
        int y;  
        B(int y) { this.y = y; }  
        public int add() { return x + y; }  
    }  
  
    A(int x, int y) {  
        this.x = x;  
        this.b = new B(y);  
    }  
  
    public int add() {  
        return b.add();  
    }  
}
```

An **inner** class is a class defined inside other class

An inner class can be even created without a name.

More details later....

```
A a = new A(3, 4);  
System.out.println(a.add());
```

Java for Android Applications Development

Part III: Inheritance

Explore new perspectives

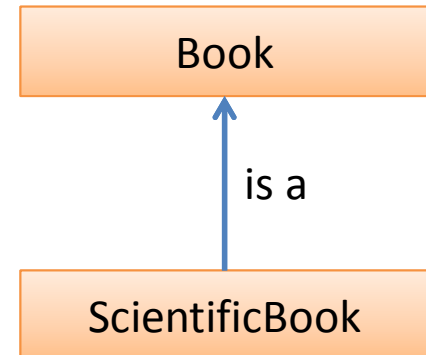
ESTECO

Inheritance

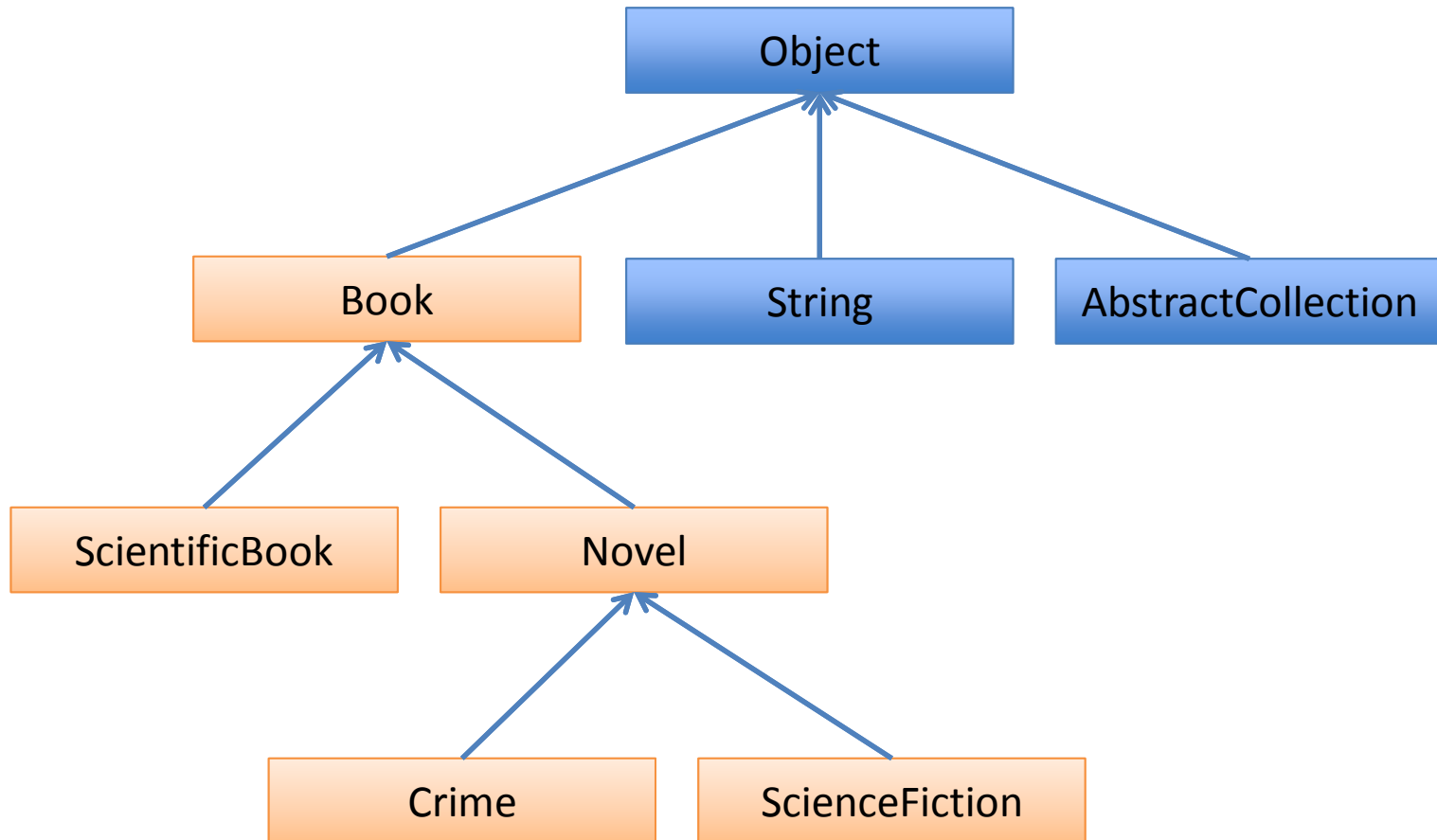
Inheritance allows to define new classes by **reusing** other classes, specifying just the differences.

It is possible to define a new class (subclass) by specifying that the class must be **like other class** (superclass)

```
class ScientificBook extends Book {  
    String area;  
    boolean proceeding = false;  
}
```

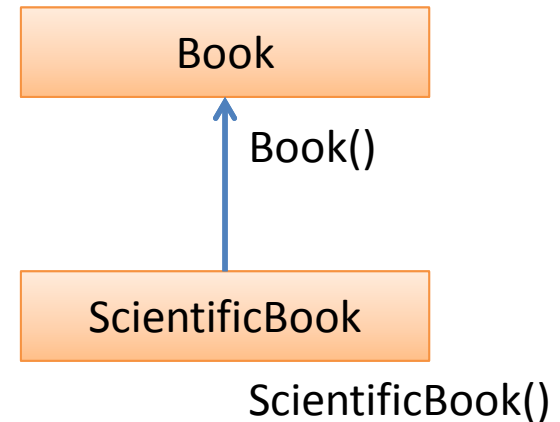


Inheritance



Constructors definition

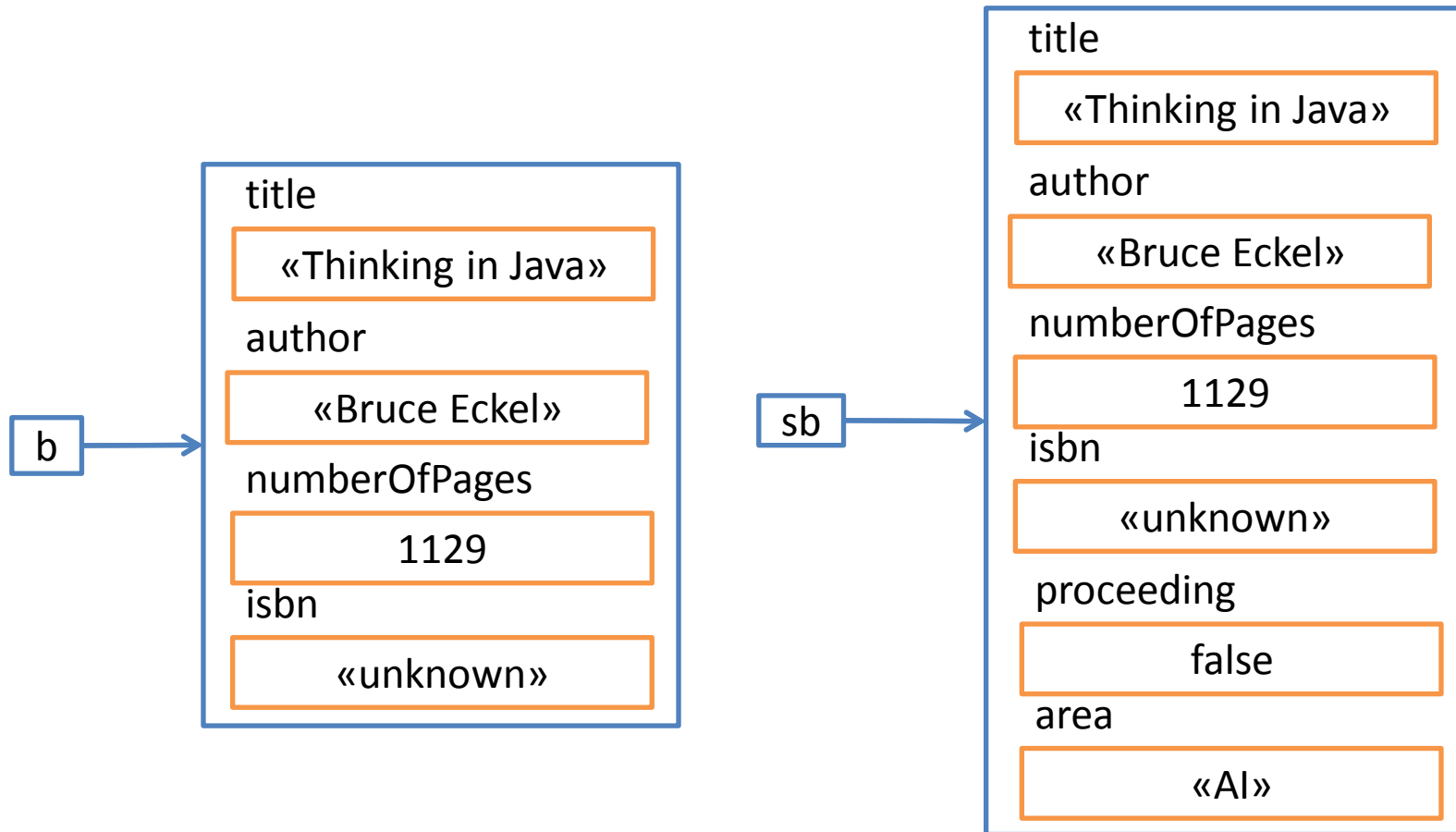
```
class ScientificBook extends Book {  
    String area;  
    boolean proceeding = false;  
  
    ScientificBook(String tit, String aut, int num,  
                   String isbn, String a) {  
        super(tit, aut, num, isbn);  
        area = a;  
    }  
}
```



If the superclass defines a constructor, the subclass has to define it and call the higher one by using **super**

```
ScientificBook sb;  
sb = new ScientificBook("Neural Networks", "Simon Haykin", 696, "0-02-352761-7", "AI");
```

Constructors definition



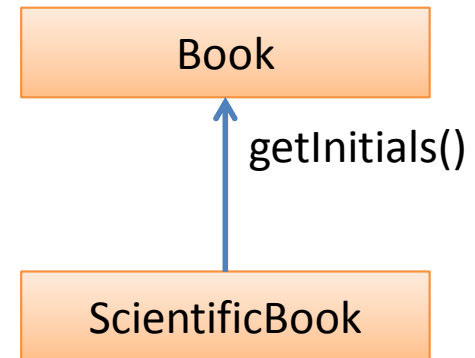
```
ScientificBook sb;  
sb = new ScientificBook("Neural Networks","Simon Haykin",696,"0-02-352761-7","AI");  
Book b = new Book("Thinking in Java","Bruce Eckel",1129);
```

Inheritance with methods

- ✓ **New methods** can be defined in the subclass to specify the behavior of the objects of this class
- ✓ When a message is sent to an object, the method is **searched for in the class of the receptor object**.
- ✓ If it is not found then it is **searched for higher up** in the hierarchy.

Inherited methods

Inherited method can
be used **directly** on
the instances of the
subclass



```
ScientificBook sb;  
sb = new ScientificBook("Neural Networks", "Simon Haykin", 696,  
    "0-02-352761-7", "AI");  
System.out.println(sb.getInitials());
```

S . H .

Overridden methods

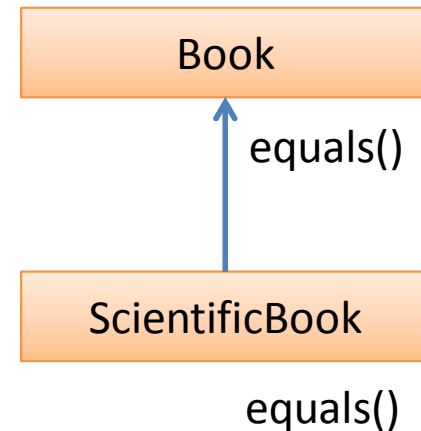
```
class ScientificBook extends Book {
    String area;
    boolean proceeding = false;

    ScientificBook(String tit, String aut,
        int num, String isbn, String a) {
        super(tit, aut, num, isbn);
        area = a;
    }

    @override
    public boolean equals(ScientificBook b){
        return super.equals(b) && area.equals(b.area) &&
            proceeding == b.proceeding;
    }

    @override
    public static String getLocation() {
        return "ScientificBooks are located in" + location;
    }
}
```

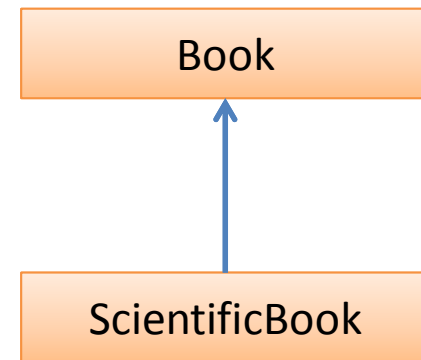
Methods in the subclass can **override** the methods in the superclass



New methods definition

```
class ScientificBook extends Book {  
    ...  
  
    Boolean proceeding = false;  
  
    ...  
  
    public void setProceeding() {  
        proceeding = true;  
    }  
  
    public boolean isProceeding() {  
        return proceeding;  
    }  
}
```

New methods can
also be defined



setproceeding()
isProceeding()

Methods: an example

```
class TestScientificBooks {  
    public static void main(String[] args) {  
        ScientificBook sb1,sb2;  
  
        sb1 = new ScientificBook("Neural Networks","Simon Haykin",  
                                696,"0-02-352761-7", "AI");  
        sb2 = new ScientificBook("Neural Networks","Simon Haykin",  
                                696,"0-02-352761-7", "AI");  
        sb2.setProceeding();  
        ScientificBook.setLocation("Kampar");  
  
        System.out.println(sb1.getInitials());  
        System.out.println(sb1.equals(sb2));  
        System.out.println(sb1.getLocation());  
    }  
}
```

```
$ java TestScientificBooks  
S.H.  
false  
ScientificBooks are located in Kampar
```

InstanceOf and getClass()

`getClass()` returns the runtime class of an object

```
Book b1 = new Book("Thinking in Java", "Bruce Eckel", 1129);  
System.out.println(b1.getClass().getName());
```

Book

`instanceof` is an operator that determines if an object is an instance of a specified class

```
Book b1 = new Book("Thinking in Java", "Bruce Eckel", 1129);  
System.out.println(b1 instanceof Book);
```

true

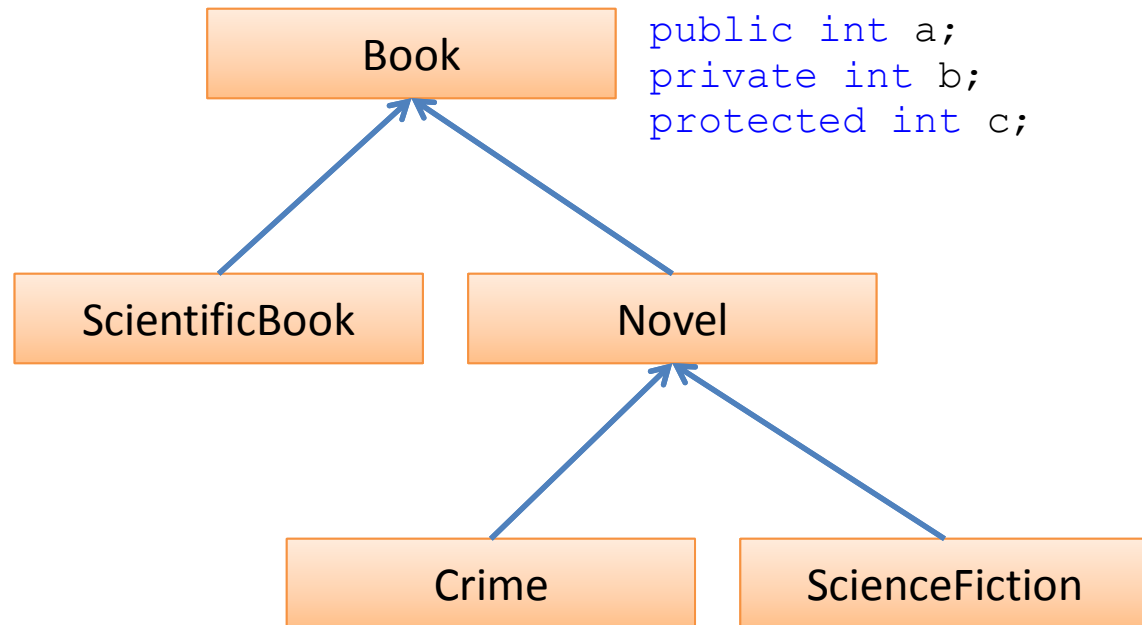
InstanceOf and getClass(): an example

```
class TestClass {  
    public static void main(String[] args) {  
        Book b1 = new Book("Thinking in Java", "Bruce Eckel", 1129);  
        ScientificBook sb1 = new ScientificBook("Neural Networks",  
                                                "Simon Haykin", 696, "0-02-352761-7",  
                                                "Artificial Intelligence");  
  
        System.out.println(b1.getClass().getName());  
        System.out.println(sb1.getClass().getName());  
        System.out.println(b1 instanceof Book);  
        System.out.println(sb1 instanceof Book);  
        System.out.println(b1 instanceof ScientificBook);  
        System.out.println(sb1 instanceof ScientificBook);  
    }  
}
```

```
$ java TestClass  
Book  
ScientificBook  
true true false true
```

Access control

It is possible to **control the access** to methods and variables from other classes with the modifiers: public, private, protected



Access control

Currently, it is possible to set the proceeding condition of a scientific book in two ways

```
sb1.setProceeding();
```

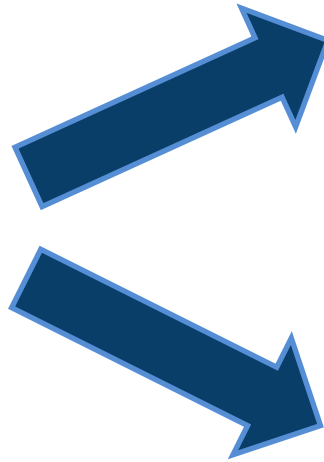
```
sb1.proceeding = true;
```

However, direct access to a data member should not be allowed in order to guarantee **encapsulation!**

```
class ScientificBook extends Book {  
    private boolean proceeding = false;  
    ...  
}
```

```
sb1.setProceeding();    // fine  
sb1.proceeding = true;  // wrong
```

The modifiers
final and abstract
can be applied to
both classes and
methods:



A **final class** does not
allow subclassing

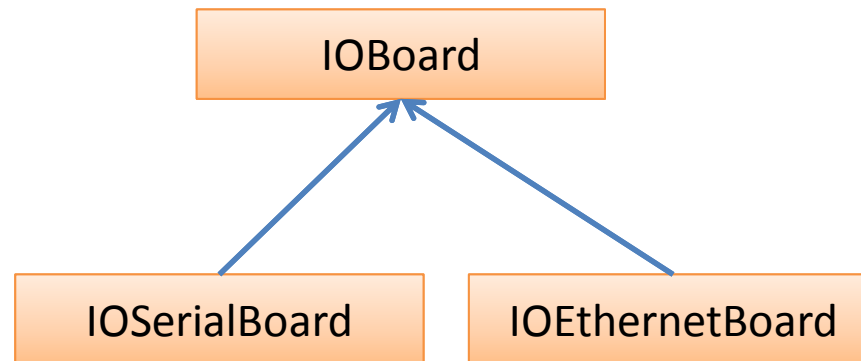
A **final method** cannot be
redefined in a subclass

An **abstract class** cannot
be instantiated

An **abstract methods** has
no body, and must be
redefined in a subclass

Final and abstract: an example

the class IOBoard and its subclasses



IOBoard is a **container** for the common behavior of the other boards

Final and abstract: an example

```
abstract class IOBoard {  
    String name;  
    int numErrors = 0;  
  
    IOBoard(String s) {  
        System.out.println("IOBoard constructor");  
        name = s;  
    }  
    final public void anotherError() {  
        numErrors++;  
    }  
    final public int getNumErrors() {  
        return numErrors;  
    }  
    abstract public void initialize();  
    abstract public void read();  
    abstract public void write();  
    abstract public void close();  
}
```

The method
anotherError() is final,
cannot be redefined in
subclasses

The other methods are
abstract, subclasses
must implement them

Final and abstract: an example

```
class IOBoard extends IOBoard {  
    int port;  
  
    IOBoard(String s,int p) {  
        super(s); port = p;  
        System.out.println("IOBoard constructor");  
    }  
    public void initialize() {  
        System.out.println("initialize method in IOBoard");  
    }  
    public void read() {  
        System.out.println("read method in IOBoard");  
    }  
    public void write() {  
        System.out.println("write method in IOBoard");  
    }  
    public void close() {  
        System.out.println("close method in IOBoard");  
    }  
}
```

Final and abstract: an example

```
class IOEthernetBoard extends IOBoard {  
    long networkAddress;  
  
    IOEthernetBoard(String s, long netAdd) {  
        super(s); networkAddress = netAdd;  
        System.out.println("IOEthernetBoard constructor");  
    }  
    public void initialize() {  
        System.out.println("initialize method in IOEthernetBoard");  
    }  
    public void read() {  
        System.out.println("read method in IOEthernetBoard");  
    }  
    public void write() {  
        System.out.println("write method in IOEthernetBoard");  
    }  
    public void close() {  
        System.out.println("close method in IOEthernetBoard");  
    }  
}
```

Final and abstract: an example

```
class TestBoards1 {  
    public static void main(String[] args) {  
        IOBoard serial = new IOBoard("my first port", 0x2f8);  
        serial.initialize();  
        serial.read();  
        serial.close();  
    }  
}
```

```
$ java TestBoards1  
IOBoard constructor  
IOBoard constructor  
initialize method in IOBoard  
read method in IOBoard  
close method in IOBoard
```

Polymorphism

- ✓ It is one of the **most important concepts** in Object Oriented Programming
- ✓ A solution is **polymorphic** if the same interface can be used to control a number of different implementations.
- ✓ Example: the **power-on** interface to request the same operation on a number of very different devices



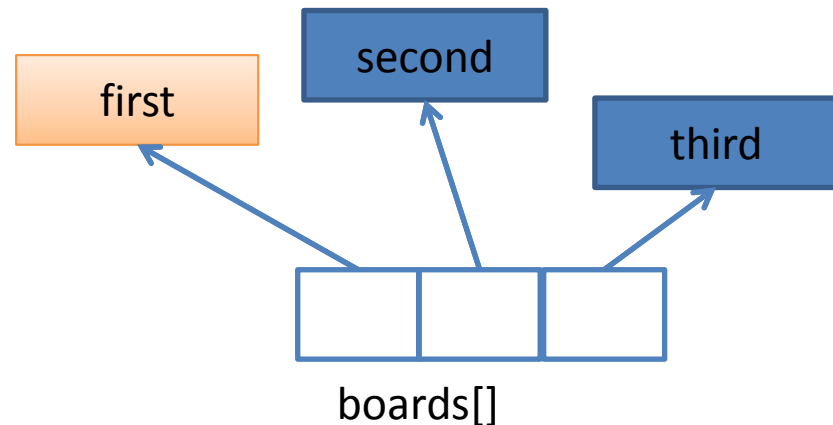
Polymorphism

An array of boards can be defined with IOBoard

```
IOBoard[] board = new IOBoard[3];  
  
board[0] = new IOBoard("my first port",0x2f8);  
board[1] = new IOEthernetBoard("my second port",0x3ef8dda8);  
board[2] = new IOEthernetBoard("my third port",0x3ef8dda9);
```

Operations are executed based on its
corresponding implementation

```
for(int i = 0;i < 3;i++)  
    board[i].initialize();  
  
for(int i = 0;i < 3;i++)  
    board[i].read();  
  
for(int i = 0;i < 3;i++)  
    board[i].close();
```



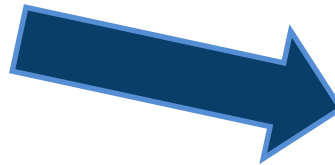
Interfaces

An interface describes what classes should do,
without specifying how they should do it.

An interface looks
like a class
definition where



All fields are static and final



All methods have no body and
are public

```
interface IOBoardInterface {  
    public void initialize();  
    public void read();  
    public void write();  
    public void close();  
}
```

```
interface NiceBehavior {  
    public String getName();  
    public String getGreeting();  
    public void sayGoodBye();  
}
```

Interfaces

```
class IOBoard implements IOBoardInterface, Nice Behavior {  
    int port;  
  
    public void initialize() { ... }  
    public void read() { ... }  
    public void write() { ... }  
    public void close() { ... }  
  
    public String getName() { ... }  
    public String getGreeting() { ... }  
    public void sayGoodBye() { ... }  
}
```

Note that there is **no**
inheritance

Note a class can
implement **more**
than one interface

Packages

A **package** is a structure in which classes can be organized.

It can contain **any number of classes**, usually related by purpose or by inheritance.

The **standard classes** in the system are organized in packages

```
import java.util.*; // or import java.util.Date

class TestDate {
    public static void main(String[] args) {
        System.out.println(new Date());
    }
}
```


Packages

```
package myBook;

class Book {
    String title;
    String author;
    int numberOfPages;
}
```

Package name is defined by using the keyword package as the first instruction

```
package myBook;

class ExampleBooks {
    public static void main(String[] args) {

        Book b = new Book();
        b.title = "Thinking in Java";
        b.author = "Bruce Eckel";
        b.numberOfPages = 1129;
        System.out.println(b.title + " : " +
            b.author + " : " + b.numberOfPages);
    }
}
```

Packages

```
package my.workshop.myBook;
```

```
class Book {  
    String title;  
    String author;  
    int numberOfPages;  
}
```

```
package my.workshop.myBook;
```

```
class ExampleBooks {  
    public static void main(String[] args) {  
  
        Book b = new Book();  
    }  
}
```



There is a **correspondence** between the package name and the directory structure where the classes are located

Java for Android Applications Development

Part IV: Exceptions and Input - Output

Explore new perspectives

ESTECO

Exceptions

The usual behavior on runtime errors is to
abort the execution

```
class TestExceptions1 {  
    public static void main(String[] args) {  
  
        String s = "Hello";  
        System.out.print(s.charAt(10));  
    }  
}
```

For example,
here there is an
error in the
charAt() call

```
$ java TestExceptions1  
Exception in thread "main"  
java.lang.StringIndexOutOfBoundsException:  
String index out of range: 10  
at java.lang.String.charAt(String.java:499)  
at TestExceptions1.main(TestExceptions1.java:11)
```

Exceptions

The exception can be **trapped**
by using a try-catch block

```
class TestExceptions2 {  
    public static void main(String[] args) {  
  
        String s = "Hello";  
        try {  
            System.out.print(s.charAt(10));  
        } catch (Exception e) {  
            System.out.println("No such position");  
        }  
    }  
}
```

```
$ java TestExceptions2  
No such position
```

Exceptions

It is possible to specify **interest** on a particular exception

```
class TestExceptions4 {  
    public static void main(String[] args) {  
  
        String s = "Hello";  
        try {  
            System.out.print(s.charAt(10));  
        } catch (StringIndexOutOfBoundsException e) {  
            System.out.println("No such position");  
            System.out.println(e.toString());  
        }  
    }  
}
```

And also **send**
messages to an
exception object

```
$ java TestExceptions4  
No such position  
java.lang.StringIndexOutOfBoundsException:  
String index out of range: 10
```

Exceptions

```
class MultipleCatch {  
    public void printInfo(String sentence) {  
        try {  
            // get first and last char before the dot  
            char first = sentence.charAt(0);  
            char last = sentence.charAt(sentence.indexOf(".") - 1);  
            String out = String.format("First: %c Last: %c", first, last);  
            System.out.println(out);  
        } catch (StringIndexOutOfBoundsException e1) {  
            System.out.println("Wrong sentence, no dot?");  
        } catch (NullPointerException e2) {  
            System.out.println("Non valid string");  
        } finally {  
            System.out.println("done!");  
        }  
    }  
}
```

It is possible to add **multiple** catch blocks and a finally clause

- ✓ Java provides strong I/O capabilities based on the concepts of **streams**
- ✓ A stream is an **abstraction** that produces and consumes data
- ✓ Streams can be **byte-oriented** or **character-oriented**
- ✓ Can be applied to **any kind of device**

Byte oriented streams

```
import java.io.*;

class WriteData {
    public static void main(String[] args) {

        double data[] = { 10.3,20.65,8.45,-4.12 };

        FileOutputStream f;
        BufferedOutputStream bf;
        DataOutputStream ds;

        try {

            f = new FileOutputStream("file1.data");
            bf = new BufferedOutputStream(f);
            ds = new DataOutputStream(bf);
```

```
            ds.writeInt(data.length);
            for(int i = 0;i < data.length;i++)
                ds.writeDouble(data[i]);
            ds.writeBoolean(true);
            ds.close();

        } catch (IOException e) {
            System.out.println("Error:" + e);
        }
    }
}
```

The file will be a sequence of
bytes

Please note that **no structure** is defined in
the file

Byte oriented streams

```
import java.io.*;

class ReadData {
    public static void main(String[] args) {

        FileOutputStream f;
        BufferedOutputStream bf;
        DataOutputStream ds;

        try {

            f = new FileInputStream("file1.data");
            bf = new BufferedInputStream(f);
            ds = new DataInputStream(bf);
```

```
            int length = ds.readInt();
            for(int i = 0; i < length; i++)
                System.out.println(ds.readDouble());
            System.out.println(ds.readBoolean());
            ds.close();

        } catch (IOException e) {
            System.out.println("Error :"+e);
        }
    }
}
```

The reader has to know the file structure!

Character oriented streams

```
import java.io.*;

class WriteText {
    public static void main(String[] args) {
        FileWriter f;
        BufferedWriter bf;
        try {
            f = new FileWriter("file1.text");
            bf = new BufferedWriter(f);
            String s = "Hello World!";
            bf.write(s,0,s.length());
            bf.newLine();
            bf.write("Java is nice!!!",8,5);
            bf.newLine();
            bf.close();
        } catch (IOException e) {
            System.out.println("Error with
files:"+e.toString());
        }
    }
}
```

```
import java.io.*;

class ReadText {
    public static void main(String[] args) {
        FileReader f;
        BufferedReader bf;
        try {
            f = new FileReader("file1.text");
            bf = new BufferedReader(f);
            String s;
            while ((s = bf.readLine()) != null)
                System.out.println(s);
            bf.close();
        } catch (IOException e) {
            System.out.println("Error:"+ e);
        }
    }
}
```

Character oriented streams

```
import java.io.*;

class ReadWithScanner {
    public static void main(String[] args) {

        try {
            Scanner sc = new Scanner(System.in);
            int sum = 0;
            while (sc.hasNextInt()) {
                int anInt = sc.nextInt();
                sum += anInt;
            }
            System.out.println(sum);
        } catch (IOException e) {
            System.out.println("Error!");
        }
    }
}
```

Uses the predefined
stream for **standard**
input

Character oriented streams

Can be used also to access other devices, **not only** files

```
URL url = new URL("http://www.google.com");
URLConnection conn = url.openConnection();

InputStreamReader reader = new InputStreamReader(conn.getInputStream());
BufferedReader in = new BufferedReader(reader);

String s;
while ((s = in.readLine()) != null) {
    System.out.println(s);
}
in.close();
```

```
<!doctype html><html itemscope="itemscope"
itemtype="http://schema.org/WebPage"><head> .....
```

Java for Android Applications Development

Part V: Threads

Explore new perspectives

ESTECO

Threads

- ✓ It is possible to run **concurrently** different tasks called threads.
- ✓ The threads can **communicate** between themselves
- ✓ Their access to shared data can be **synchronized**
- ✓ **Two implementation possibilities:**
extend thread or implement runnable interface

Sub-classing the Thread class

```
class CharThread extends Thread {  
    char c;  
    CharThread(char aChar) {  
        c = aChar;  
    }  
    public void run() {  
        while (true) {  
            System.out.println(c);  
            try {  
                sleep(100);  
            } catch (InterruptedException e) {  
                System.out.println("Interrupted");  
            }  
        }  
    }  
}
```

```
class TestThreads {  
    public static void main(String[] args) {  
        CharThread t1 = new CharThread('a');  
        CharThread t2 = new CharThread('b');  
  
        t1.start();  
        t2.start();  
    }  
}
```

```
$ java TestThreads  
a  
b  
a  
b  
...
```


Runnable interface

```
class CharThread implements Runnable {  
    char c;  
    CharThread(char aChar) {  
        c = aChar;  
    }  
    public void run() {  
        while (true) {  
            System.out.println(c);  
            try {  
                Thread.sleep(100);  
            } catch (InterruptedException e) {  
                System.out.println("Interrupted");  
            }  
        }  
    }  
}
```

Now the class can
extend other classes

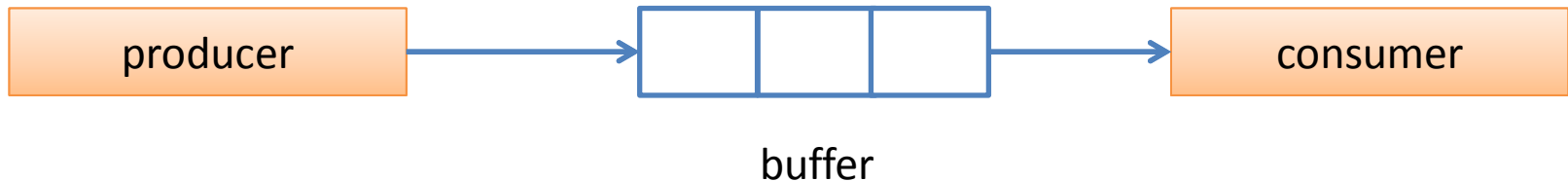
Note that sleep is **not**
inherited any more!

An example

```
class ProducerConsumer {  
    public static void main(String[] args) {  
        Buffer buffer = new Buffer(10);  
        Producer prod = new Producer(buffer);  
        Consumer cons = new Consumer(buffer);  
  
        prod.start();  
        cons.start();  
    }  
}
```

The producer and the
consumer are
implemented with
threads

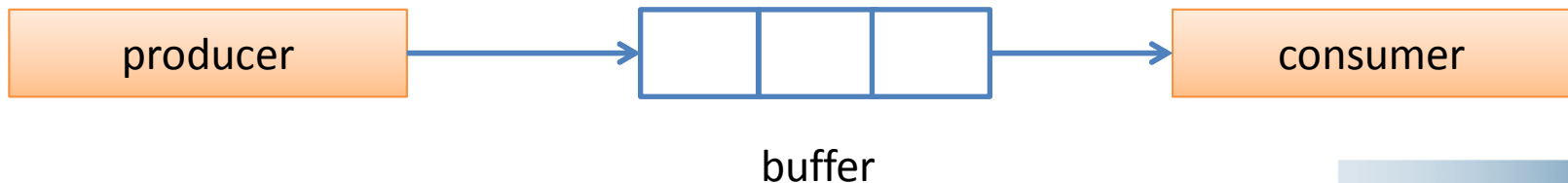
The buffer is **shared**
between the two threads



An example: the producer and consumer

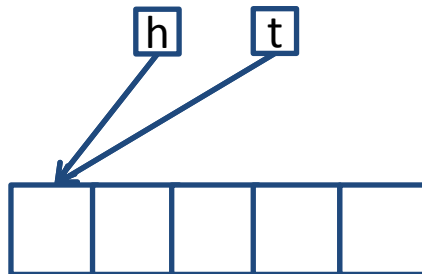
```
class Producer extends Thread {  
  
    Buffer buffer;  
  
    public Producer(Buffer b) {  
        buffer = b;  
    }  
  
    public void run() {  
        double value = 0.0;  
        while (true) {  
            buffer.insert(value);  
            value += 0.1;  
        }  
    }  
}
```

```
class Consumer extends Thread {  
  
    Buffer buffer;  
  
    public Consumer(Buffer b) {  
        buffer = b;  
    }  
  
    public void run() {  
        while(true) {  
            char element = buffer.delete();  
            System.out.println(element);  
        }  
    }  
}
```

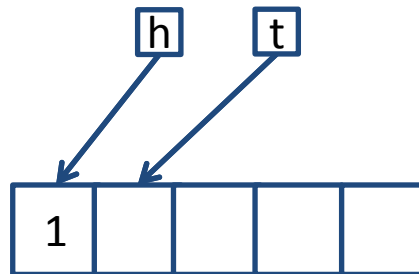


An example: the circular buffer

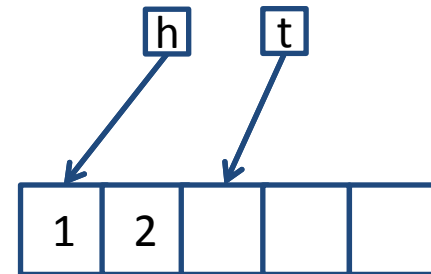
Insertion of elements in the buffer:



Initial situation

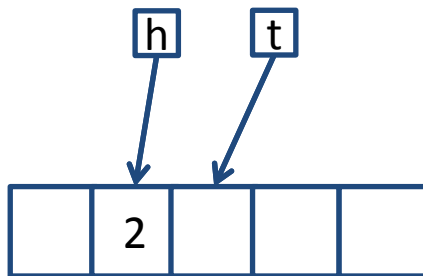


insert(1)



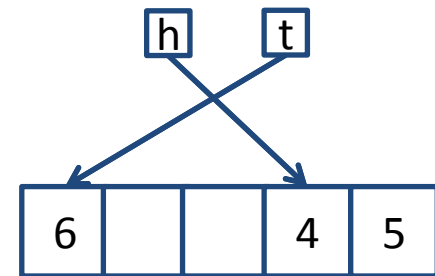
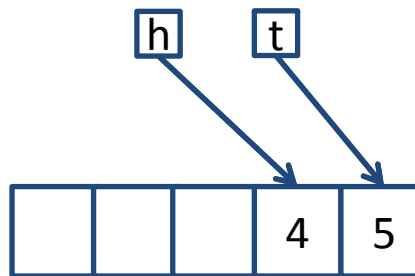
insert(2)

Remove one element:



remove()

Going beyond the limit of the buffer:



insert(6)

An example: the buffer

```
class Buffer {  
    double buffer[];  
    int head = 0, tail = 0, size = 0, numElements = 0;  
  
    public Buffer(int s) {  
        buffer = new double[s];  
        size = s;  
    }  
    public void insert(double element) {  
        buffer[tail] = element; tail = (tail + 1) % size;  
        numElements++;  
    }  
    public double delete() {  
        double value = buffer[head]; head = (head + 1) % size;  
        numElements--;  
        return value;  
    }  
}
```

An example: problems

However, the implementation **does not work!**.

- The methods insert() and delete() operate **concurrently** over the same structure.
- The method insert() does not check if there is **at least one slot free** in the buffer
- the method delete() does not check if there is **at least one piece of data available** in the buffer.

There is a need for
synchronization

Synchronization

- ✓ Synchronized access to a critical resource can be achieved with **synchronized methods**
- ✓ Each instance has a **lock**, used to synchronize the access.
 - ✓ **Synchronized methods** are not allowed to be executed concurrently on the same instance.

An example: synchronized methods

```
public synchronized void insert(double element) {  
  
    while (numElements == size) {  
        try {  
            wait();  
        } catch (InterruptedException e) {  
            System.out.println("Interrupted");  
        }  
    }  
  
    buffer[tail] = element;  
    tail = (tail + 1) % size;  
    numElements++;  
    notify();  
}
```

The method goes to **sleep** (and release the lock) if buffer is full

At the end, it **awakes** producer(s) which can be sleeping waiting for the lock

Synchronized access to the critical resource is achieved with a **synchronized** method:

An example: synchronized methods

```
public synchronized double delete() {  
  
    while (numElements == 0) {  
        try {  
            wait();  
        } catch (InterruptedException e) {  
            System.out.println("Interrupted");  
        }  
    }  
  
    double value = buffer[head];  
    head = (head + 1) % size;  
    numElements--;  
    notify();  
    return value;  
}
```

Synchronized access
to the critical resource
is achieved with a
synchronized
method:

Explore new perspectives

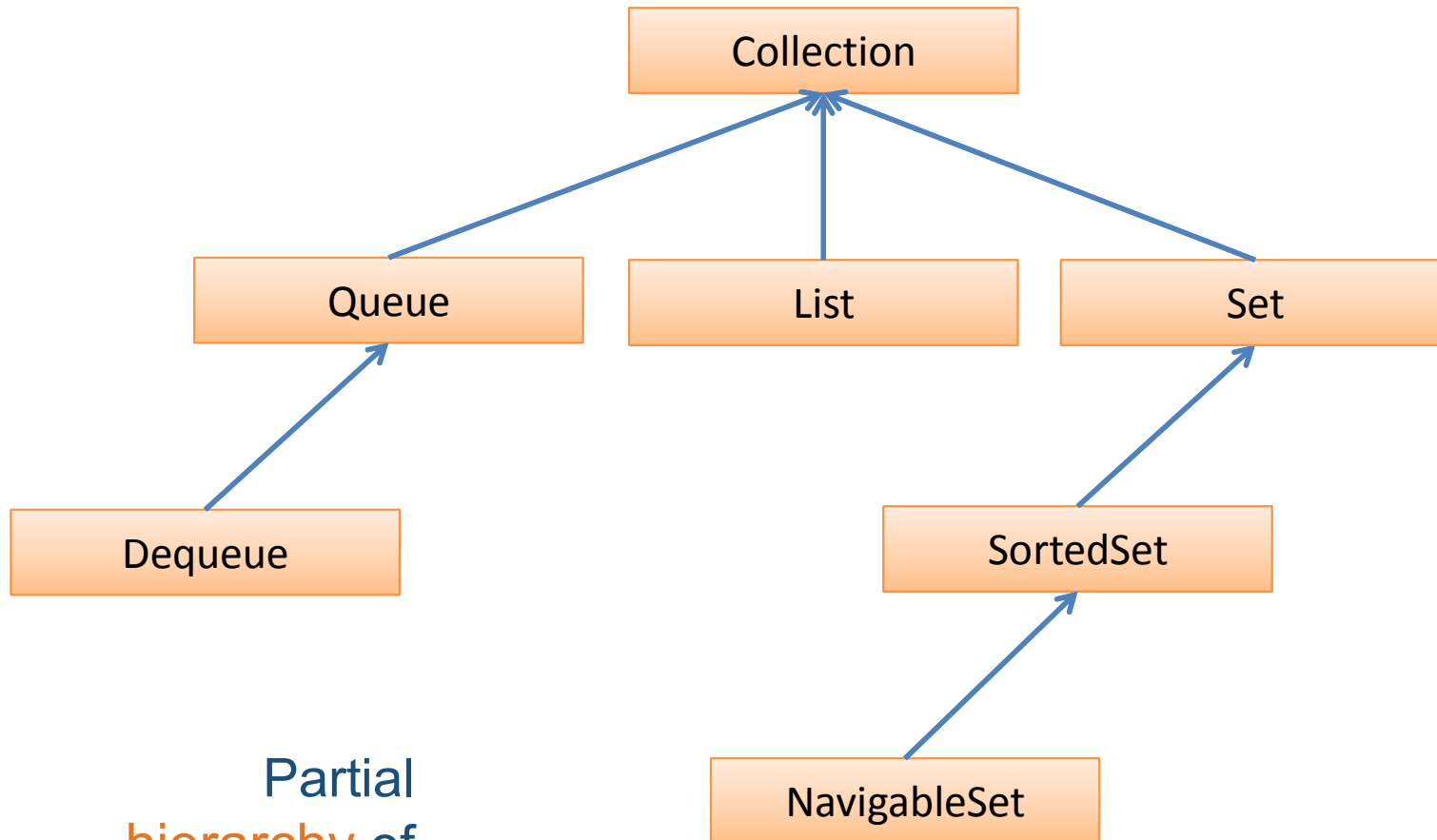
ESTECO

Java for Android Applications

Part VI: Collections and Generics

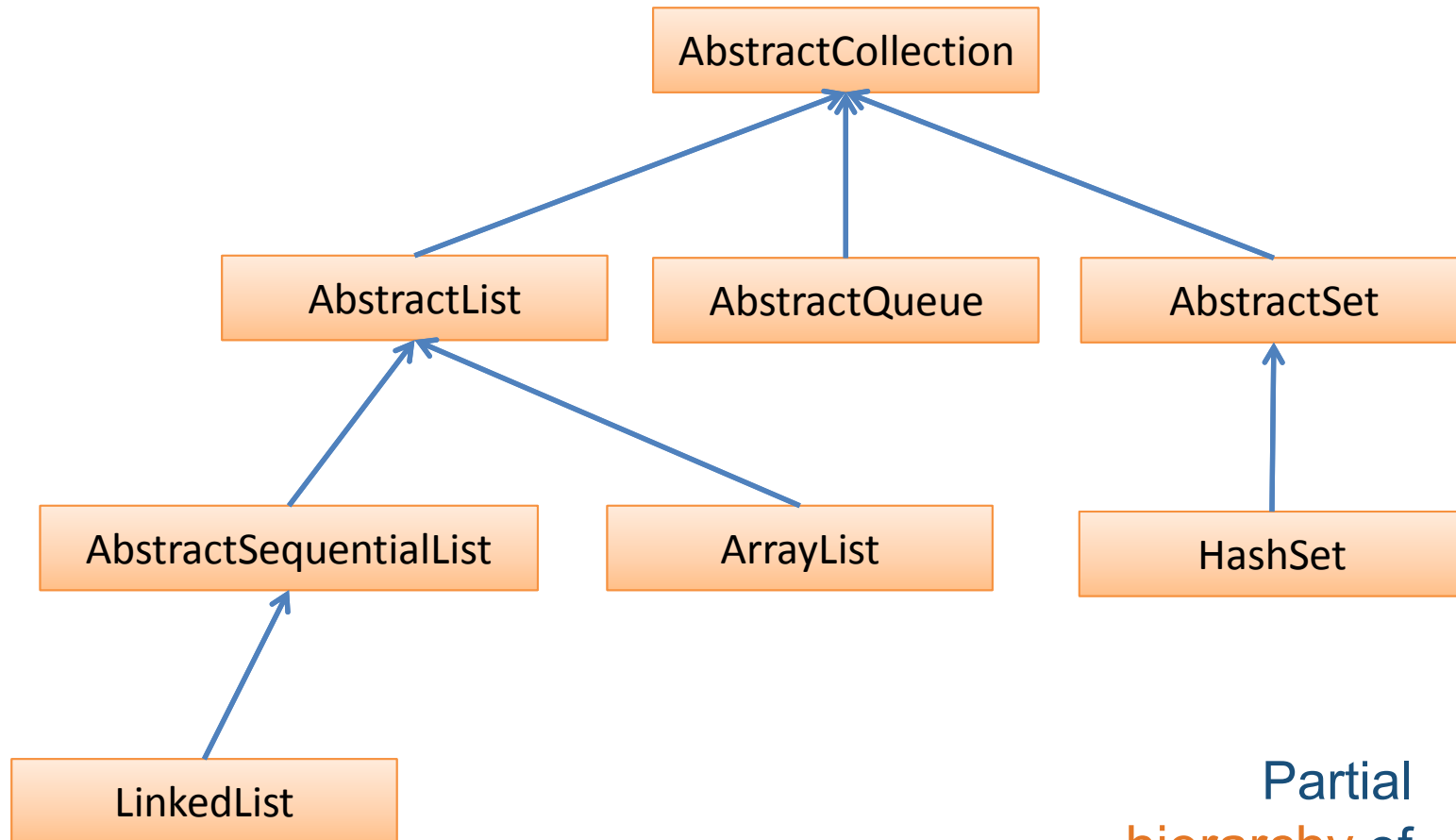
- ✓ The **framework** provides state-of-the-art technology for managing groups of objects
- ✓ A **highly sophisticated** hierarchy of interfaces and classes
- ✓ Java programmers **must know and use it**

Interfaces



Partial
hierarchy of
interfaces

Classes



Partial
hierarchy of
classes

An example with ArrayList

Creation and insertion

```
ArrayList<String> list = new ArrayList<String>();  
  
list.add("red");  
list.add("blue");  
list.add("white");
```

```
for(String x : list) {  
    System.out.println(x);  
}
```

Traversing the structure

Removing elements

```
list.remove(2);  
list.remove("white");
```

An example with LinkedList

Creation and insertion

```
LinkedList<String> list = new LinkedList<String>();  
  
list.add("red");  
list.addFirst("blue");  
list.add(1, "white");
```

```
for(String x : list) {  
    System.out.println(x);  
}
```

Traversing the structure

Removing elements

```
list.Last();  
list.remove("white");
```

An example with HashMap

Creation and insertion

```
HashMap<String, Integer> map = new HashMap<>();  
  
map.put("temperature", 22);  
map.put("humidity", 65);
```

```
int temp = map.get("temperature");
```

Accessing a value

Getting keys

```
for(String x : map.keySet()) {  
    System.out.println(map.get(x));  
}
```


- ✓ Generics allows to build **parameterized types**:
 - ✓ **create** classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter.
- ✓ Improve **type safety** when compared with Objects

An example

```
public class Stack<BaseType> {  
    LinkedList<BaseType> data;  
  
    Stack() {  
        data = new LinkedList<BaseType>();  
    }  
  
    public void push(BaseType e) {  
        data.addFirst(e);  
    }  
  
    public BaseType pop() {  
        return data.removeFirst();  
    }  
  
    public int size() {  
        return data.size();  
    }  
}
```

A generic stack

```
public static void main(String[] args) {  
    Stack<Integer> stack = new Stack<>();  
    stack.push(22);  
    stack.push(66);  
    System.out.println(stack.pop());  
}
```

Be careful, no
checking is done
when removing
elements!

Bounded classes

The generic class can be
restricted

```
public class Stack<BaseType extends Number> {  
    ...  
}
```

This specifies that BaseType can
only be replaced by Number, or
subclasses of Number.

Wildcard arguments

Let's defined a new methods to compare the size of two stacks:

```
public class Stack<BaseType extends Number> {  
    ...  
    public boolean equalSize(Stack<BaseType> other) {  
        return size() == other.size();  
    }  
}
```

```
Stack<Integer> stack1 = new Stack<>();  
stack1.push(22);  
stack1.push(66);  
  
Stack<Float> stack2 = new Stack<>();  
stack2.push(3.1);  
  
boolean equalSize = stack1.equalSize(stack2);
```

However, it **does not work** if types are different!

Wildcard arguments

A new method with wildcards to compare the size of two stacks:

```
public class Stack<BaseType extends Number> {  
    ...  
    public boolean equalSize(Stack<?> other) {  
        return size() == other.size();  
    }  
}
```

```
Stack<Integer> stack1 = new Stack<>();  
stack1.push(22);  
stack1.push(66);  
  
Stack<Float> stack2 = new Stack<>();  
stack2.push(3.1);  
  
boolean equalSize = stack1.equalSize(stack2);
```

It works now

Comparator interface for Collections

Classes that implements the **comparable interface** can be “compared” by Collection methods

```
public interface Comparable<T extends Object> {  
    public int compareTo(T t);  
}
```

Note the **bounded** generic declaration!

Comparator interface for Collections

```
class Book implements Comparable<Book> {  
  
    ...  
  
    public int compareTo(Book aBook) {  
        return numberOfPages - aBook.numberOfPages;  
    }  
}
```

Books can be
compared now!

```
Book b1 = new Book("Thinking in Java", "Bruce Eckel", 1129);  
Book b2 = new Book("Java in a nutshell", "David Flanagan", 353);  
  
ArrayList<Book> list = new ArrayList<Book>();  
  
list.add(b1); list.add(b2);  
  
Collections.sort(list);  
  
for (Book x : list) {  
    System.out.println(x.title);  
}
```



Thank you!

