The Abdus Salam International Centre for Theoretical Physics

United Nations Educational, Scientific and Cultural Organization

IAEA International Atomic Energy Agency

1875-4

First Latin American Regional Workshop on Distributed Laboratory Instrumentation in Physics

7 January - 4 February, 2008

Introduction to JAVA
- slides -

Carlos Kavka

INFN Sezione di Trieste
Area di Ricerca
Padriciano 99
34012 Trieste
Italy

Introduction to Java

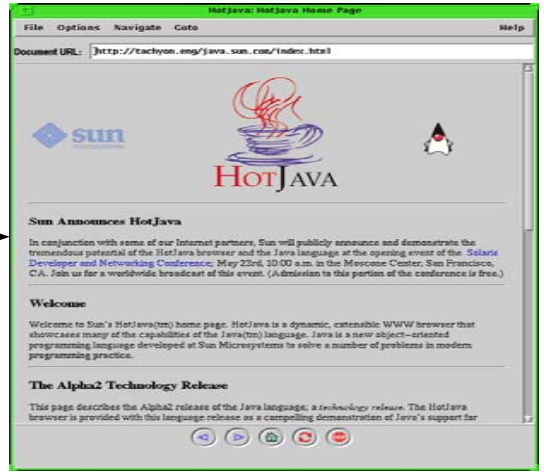
Carlos Kavka
INFN Sezione di Trieste
Area di Ricerca, Padriciano 99
34012, Trieste, Italia

First Latin American Workshop on Distributed Laboratory Instrumentation Systems
7 January 2008 - 1 February 2008
Universidad Austral de Chile
Valdivia, Chile

Introduction

- Java is a very powerful language that has generated a lot of interest in the last years.

Java



The Star 7 device

The HotJava browser

Oak

Java

netBeans

Introduction

- It is a general purpose concurrent object oriented language, with a syntax similar to C and C++, but omitting features that are complex and unsafe.

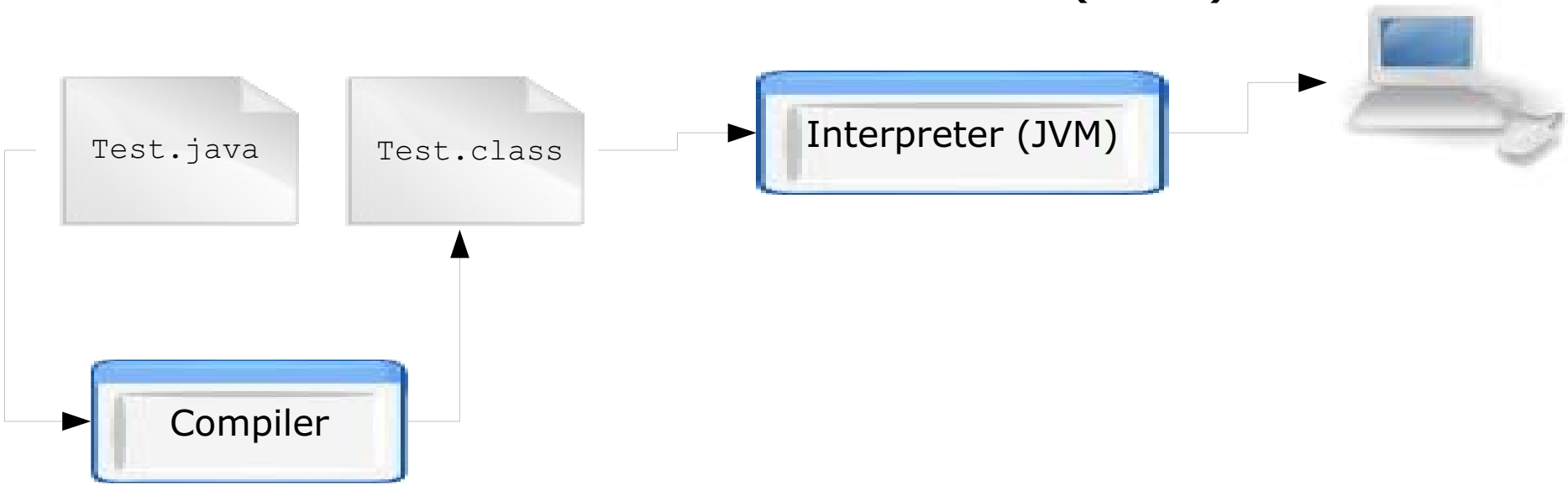
C++	Java
Backward compatile con C	Backward compatibility with previous Java versions
Execution efficiency	Developer productivity
Trusts the programmer	Protects the programmer
Arbitrary memory access possible	Memory access through objects
Concise expression	Explicit operation
Can arbitrarily override types	Type safety
Procedural or object oriented	Object oriented
Operator overloading	Meaning of operators immutable

Introduction

- The world wide web has popularized the use of Java, because programs can be transparently downloaded with web pages and executed in any computer with a Java capable browser.
- A Java application is a standalone Java program that can be executed independently of any web browser.
- A Java applet is a program designed to be executed under a Java capable browser.

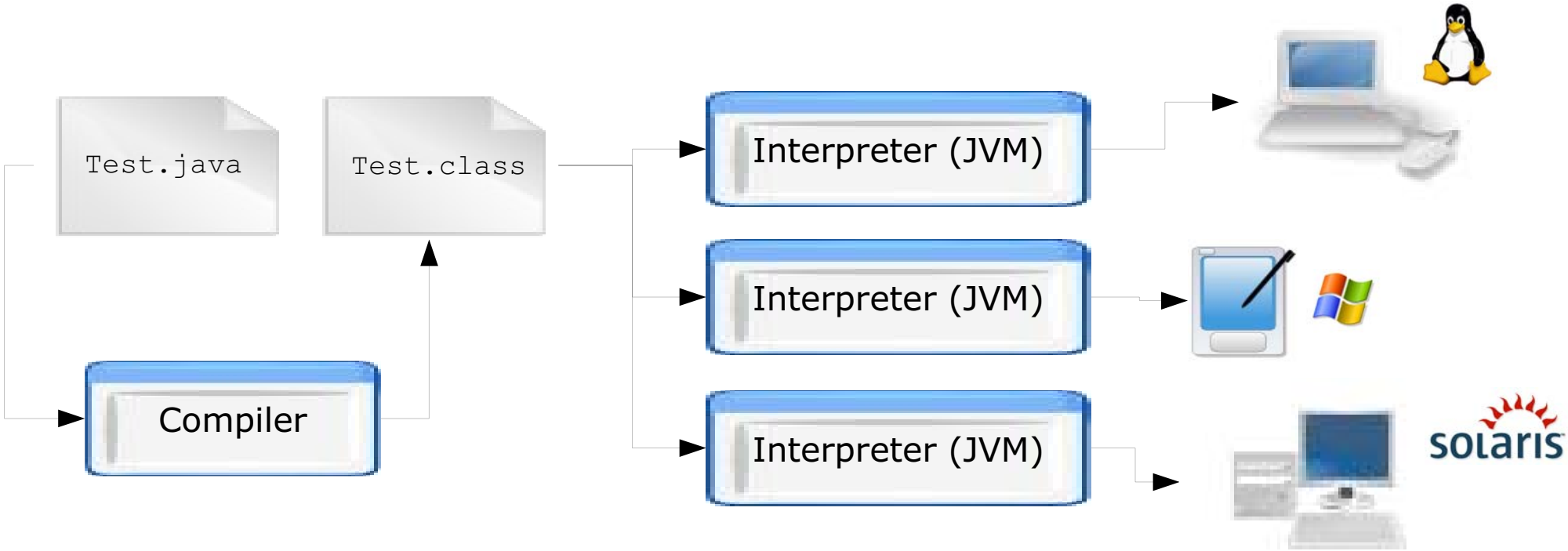
The Java platform

- Java programs are compiled to Java byte-codes, a kind of machine independent representation. The program is then executed by an interpreter called the Java Virtual Machine (JVM).



The Java platform

- The compiled code is independent of the architecture of the computer.
- The price to pay is a slower execution.



A first example

```java
/**
* Hello World Application
* Our first example
*/
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
```

Documentation

- The javadoc utility can be used to generate automatically documentation for the class.

```
/**
* My first <b>Test</b>
* @author Carlos Kavka
* @version 1.1
*/
public class HelloWorld {
  /**
  * @param args the command line arguments
  * @since 1.0
  */
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

Documentation

Main - Mozilla Firefox <2>

File Edit View Go Bookmarks Tools Help

Getting Started Latest Headlines

All Classes

Main

Method Summary

static void	main(java.lang.String[] args) Main function

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Main

public Main()

Creates a new instance of Main

Method Detail

main

public static void main(java.lang.String[] args)

Main function

Parameters:

args - the command line arguments

Since:

1.0

Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Find: Find Next Find Previous Highlight all Match case

Done

Fundamental types

- Java provides ten fundamental types:
  - integers: **byte**, **short**, **int** and **long**
  - floating point: **float** and **double**.
  - characters: **char**.
  - **boolean**
  - **void**
  - String

Variables

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

```
int x;
double f = 0.33;
char c = 'a';
String s = "abcd";

x = 55;
```

Literals

- The integer values can be written in decimal, hexadecimal, octal and long forms:

```
int x = 34;            // decimal value
int y = 0x3ef;         // hexadecimal
int z = 0772;          // octal
long m = 240395922L;   // long
```

- The floating point values are of type **double** by default:

```
double d = 6.28;       // 6.28 is a double value
float f = 6.28F;       // 6.28F is a float value
```

Literals

- The character values are specified with the standard C notation, with extensions for Unicode values:

```
char c = 'a';           // character lowercase a
char d = '\n';          // newline
char e = '\u2122'       // unicode character (TM)
```

- The boolean values are **true** and **false**:

```
boolean ready = true; // boolean value true
boolean late = false; // boolean value false
```

Constants

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

```
final double pi = 3.1415;      // constant PI
final int maxSize = 100;       // integer constant
final char lastLetter = 'z';   // last lowercase letter
final String word = "Hello";   // a constant string
```

Expressions

- Java provides a rich set of expressions:
  - Arithmetic
  - Bit level
  - Relational
  - Logical
  - Strings related

Arithmetic expressions

- Java provides the usual set of arithmetic operators:
  - addition (+)
  - subtraction (-)
  - division (/)
  - multiplication (*)
  - modulus (%)

Arithmetic operators

```
class Arithmetic {
  public static void main(String[] args) {
    int x = 12;
    int y = 2 * x;
    System.out.println(y);
    int z = (y - x) % 5;
    System.out.println(z);
    final float pi = 3.1415F;
    float f = pi / 0.62F;
    System.out.println(f);
  }
}
```

```
$ java Arithmetic
24
2
5.0669355
```

Arithmetic operators

- Shorthand operators are provided:

```
class ShortHand {
  public static void main(String[] args) {
    int x = 12;

    x += 5;                    // x = x + 5
    System.out.println(x);

    x *= 2;                    // x = x * 2
    System.out.println(x);
  }
}
```

```
$ java ShortHand
17
34
```

Arithmetic operators

- Pre and post operators are also provided:

```
class Increment {
  public static void main(String[] args) {
    int x = 12,y = 12;

    System.out.println(x++); // printed and then incremented
    System.out.println(x);

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

```
$ java Increment
12 13 13 13
```

Relational expressions

- Java provides the following relational operators:
  - equivalent (==)
  - not equivalent (!=)
  - less than (<)
  - greater that (>)
  - less than or equal (<=)
  - greater than or equal (>=)

- Important: relational expressions always return a **boolean** value.

Relational Expressions

```
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 operators

- Java provides the following operators:
  - and (&)
  - or (|)
  - not(~)
  - shift left (<<)
  - shift right with sign extension (>>)
  - shift right with zero extension (>>>).
- ***Important***: **char**, **short** and **byte** arguments are promoted to **int** before and the result is an **int**.

Bit level operators

```
class Bits {
  public static void main(String[] args) {
    int x = 0x16;                // 00000000000000000000000000010110
    int y = 0x33;                // 00000000000000000000000000110011

    System.out.println(x & y);// 00000000000000000000000000010010
    System.out.println(x | y);// 00000000000000000000000000110111
    System.out.println(~x);    // 11111111111111111111111111101001

    x &= 0xf;                    // 00000000000000000000000000000110
    System.out.println(x);       // 00000000000000000000000000000110

    short s = 7;                 // 0000000000000111
    System.out.println(~s);    // 1111111111111111111111111111111000
  }
}
```

Bit level operators

```
class Bits2 {
  public static void main(String[] args) {
    int x = 0x16;                  //00000000000000000000000000010110
    System.out.println(x << 3);//00000000000000000000000010110000

    int y = 0xfe;                  //00000000000000000000000011111110
    y >>= 4;                       //00000000000000000000000000001111
    System.out.println(y);         //00000000000000000000000000001111

    x = 9;                         //00000000000000000000000000001001
    System.out.println(x >> 3);//00000000000000000000000000000001
    System.out.println(x >>>3);//00000000000000000000000000000001

    x = -9;                        //11111111111111111111111111110111
    System.out.println(x >> 3);//11111111111111111111111111111110
    System.out.println(x >>>3);//00011111111111111111111111111110
  }
}
```

Logical operators

- Java provides the following operators:
  - and (&&)
  - or (||)
  - not(!)

- ***Important***: The logical operators can only be applied to **boolean** expressions and return a **boolean** value.

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
```

String operators

- Java provides many operators for Strings:
  - Concatenation (+)
  - many more...

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

- *Important*: Strings cannot be compared with == and !=.

String operators

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

    String s1 = "Hello" + " World!";
    System.out.println(s1);

    int i = 35,j = 44;
    System.out.println("The value of i is " + i +
                       " and the value of j is " + j);
  }
}
```

```
$ java Strings
Hello World!
The value of i is 35 and the value of j is 44
```

String operators

```
class Strings2 {
  public static void main(String[] args) {

    String s1 = "Hello";
    String s2 = "Hello";

    System.out.println(s1.equals(s2));
    System.out.println(s1.equals("Hi"));
  }
}
```

```
$ java Strings2
true
false
```

Casting

- Java performs a automatic type conversion in the values when there is no risk for data to be lost.

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

    int a = 'x';          // 'x' is a character
    long b = 34;          // 34 is an int
    float c = 1002;       // 1002 is an int
    double d = 3.45F;     // 3.45F is a float
  }
}
```

Casting

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

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

    long a = 34;
    int b = (int)a;          // a is a long
    double d = 3.45;
    float f = (float)d;      // d is a double
  }
}
```

Control structures

- Java provides the same set of control structures than C.

- *Important*: the value used in the conditional expressions must be a **boolean**.

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");
        System.out.println("and it is not 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++) {

      // start a new iteration if the counter is odd
      if (counter % 2 == 1) continue;

      // abandon the loop if the counter is equal to 8
      if (counter == 8) break;

      // print the value
      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;
```

Control structures (switch)

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

```
$ java Switch
number of days: 366
```

Arrays

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

```
int[] a;        // an unitialized array of integers
float[] b;      // an unitialized array of floats
String[] c;     // an unitialized array of Strings
```

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

```
int[] a = {13,56,2034,4,55};            // size: 5
float[] b = {1.23F,2.1F};               // size: 2
String[] c = {"Java","is","great"};     // size: 3
```

Arrays

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

```
int i = 3,j = 5;
double[] d;                // unitialized array of doubles

d = new double[i+j];       // array of 8 doubles
```

- Components of the arrays are initialized with default values:
  - 0 for numeric type elements,
  - '\0' for characters
  - **null** for references.

Arrays

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

```
a[2] = 1000; // modify the third element of a
```

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

```
int len = a.length; // get the size of the array
```

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
```

Command line arguments

- We have seen that the method **main** has to be defined as follows:

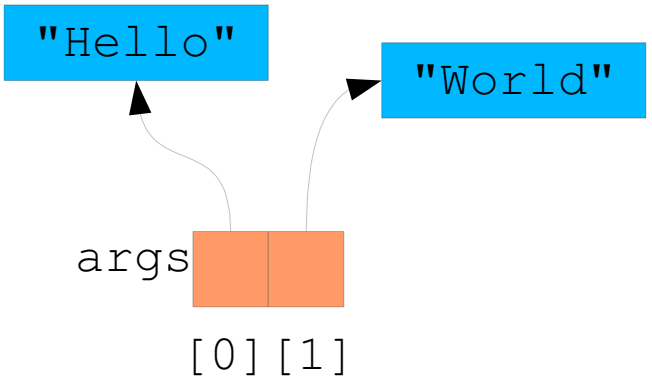
```
public static void main(String[] args)
```

- Through the array argument, the program can get access to the command line arguments

Command line arguments

```
class CommandArguments {
  public static void main(String[] args) {
    for(int i = 0;i < args.length;i++)
      System.out.println(args[i]);
  }
}
```

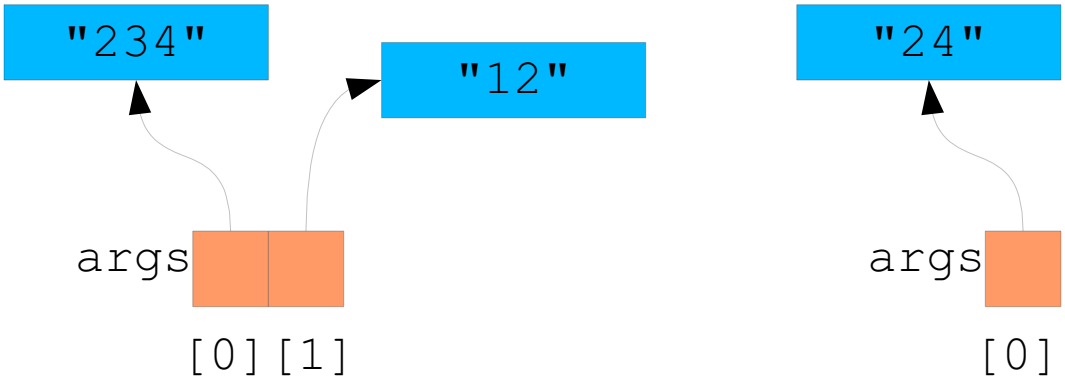
```
$ java CommandArguments Hello World
Hello
World

$ java CommandArguments

$ java CommandArguments I have 25 cents
I
have
25
cents
```



Command line arguments

```
class Add {
  public static void main(String[] args) {
    if (args.length != 2) {
      System.out.println("Error");
      System.exit(0);
    }
    int arg1 = Integer.parseInt(args[0]);
    int arg2 = Integer.parseInt(args[1]);
    System.out.println(arg1 + arg2);
  }
}
```

```
$ java Add 234 12
246

$ java Add 24
Error
```



Classes

- A class is defined in Java by using the class keyword and specifying a name for it:

```
class Book {
}
```

- New instances of the class can be created with new:

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

b3 = new Book();
```

Classes

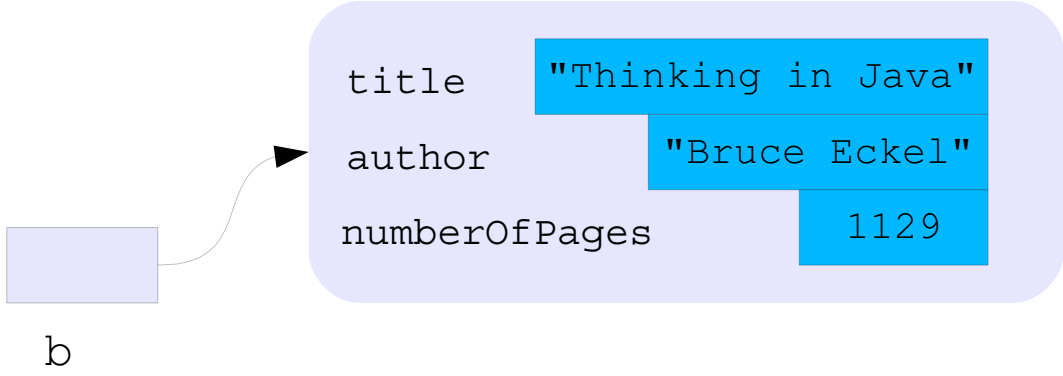
- Inside a class it is possible to define:
  - data elements, usually called instance variables
  - functions, usually called methods
- Class **Book** with instance variables:

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

- The instance variables can be accessed with the dot notation.

Classes

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




```
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);
  }
}
```

Constructors

- The constructors allow the creation of instances that are properly initialized.
- A constructor is a method that:
  - has the same name as the name of the class to which it belongs
  - has no specification for the return value, since it returns nothing.

Constructors

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

```
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);
  }
}
```

Default constructors

- Java provides a default constructor for the classes.

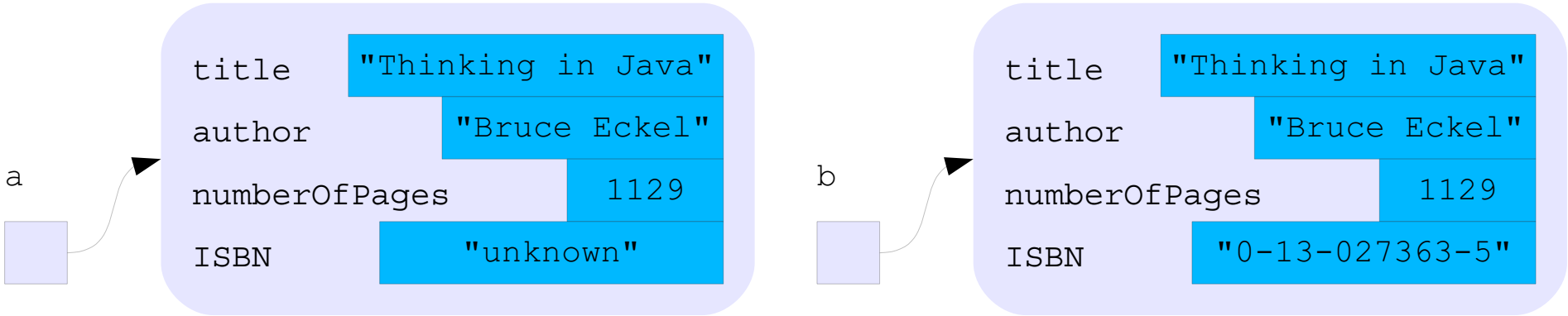
```
b = new Book();
```

- This default constructor is only available when no constructors are defined in the class.

Multiple constructors

- It is possible to define more than one constructor for a single class, only if they have different number of arguments or different types for the arguments.

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



Multiple constructors

```
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;
  }
}
```

Multiple constructors

```
class ExampleBooks3 {
  public static void main(String[] args) {
    Book b1,b2;

    b1 = new Book("Thinking in Java","Bruce Eckel",1129);
    System.out.println(b1.title + " : " + b1.author +
                " : " + b1.numberOfPages + " : " + b1.ISBN);
    b2 = new Book("Thinking in Java","Bruce Eckel",1129,
                  "0-13-027363-5");
    System.out.println(b2.title + " : " + b2.author +
                " : " + b2.numberOfPages + " : " + b2.ISBN);
  }
}
```

```
$ java ExampleBooks3
Thinking in Java : Bruce Eckel : 1129 : unknown
Thinking in Java : Bruce Eckel : 1129 : 0-13-027362-5
```

Methods

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

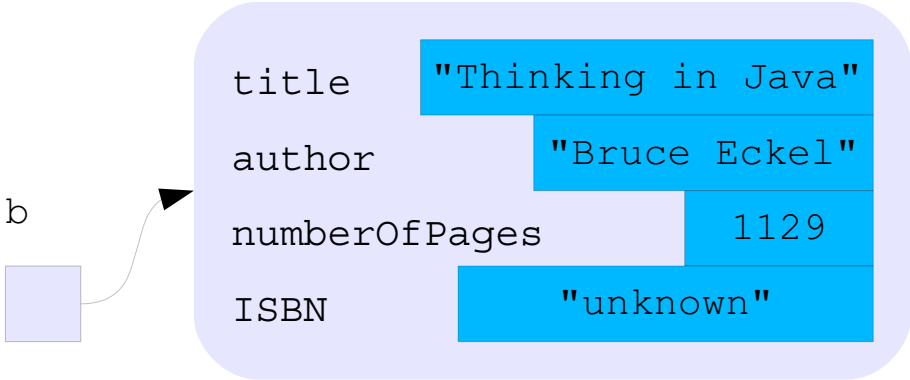
Methods

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

  ...

  // compute initials of author's name
  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;
  }
}
```

Methods

```
class ExampleBooks4 {
  public static void main(String[] args) {
    Book b;

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

```
$ java ExampleBooks4
Initials: B.E.
```

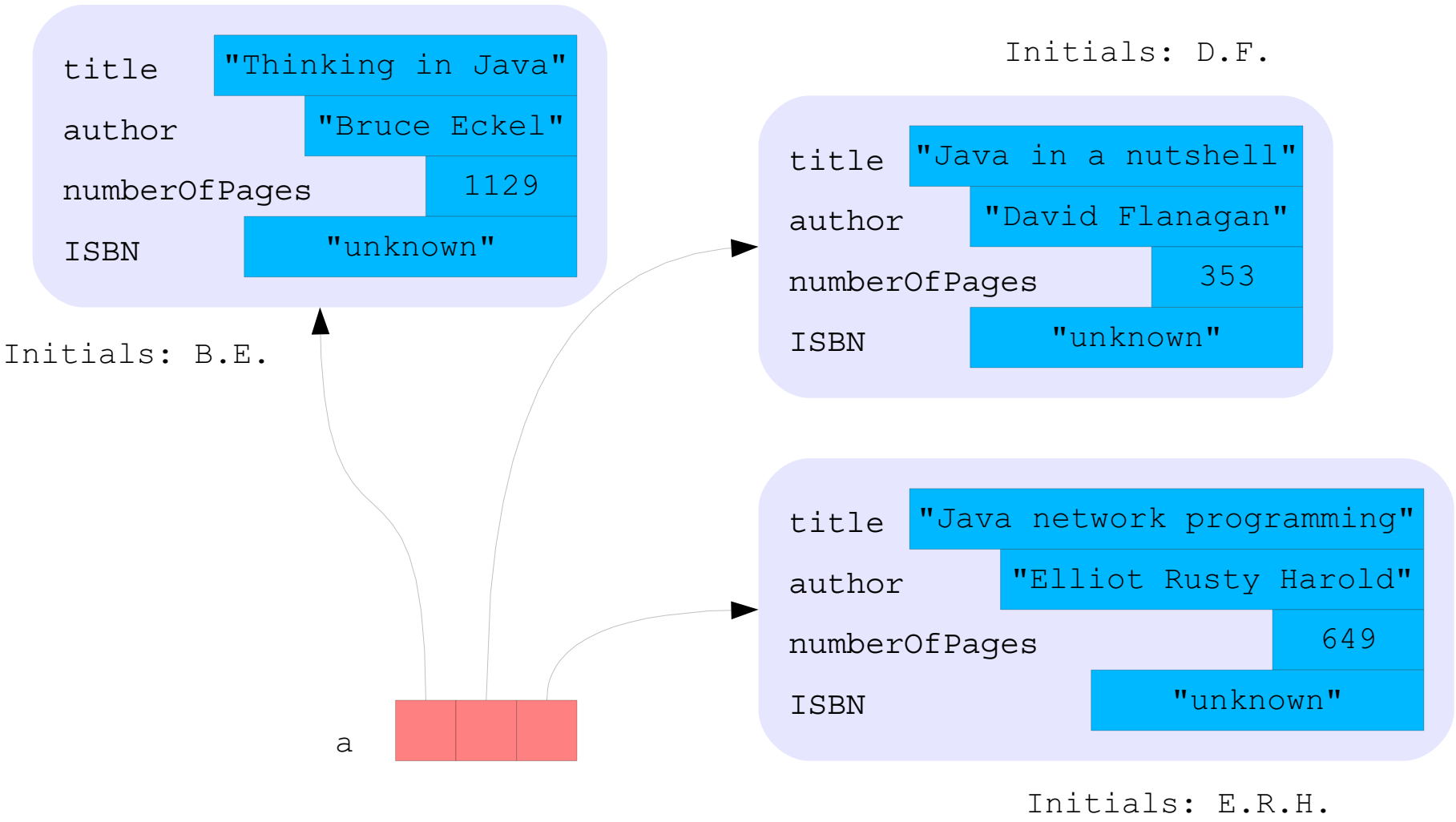


Methods

```
class ExampleBooks5 {
  public static void main(String[] args) {

    Book[] a = new Book[3];
    a[0] = new Book("Thinking in Java","Bruce Eckel",1129);
    a[1] = new Book("Java in a nutshell","David Flanagan",353);
    a[2] = new Book("Java network programming",
                    "Elliott Rusty Harold",649);

    for(int i = 0;i < a.length;i++)
      System.out.println("Initials: " + a[i].getInitials());
  }
}
```

```
$ java ExampleBooks5
Initials: B.E.
Initials: D.F.
Initials: E.R.H.
```

Methods

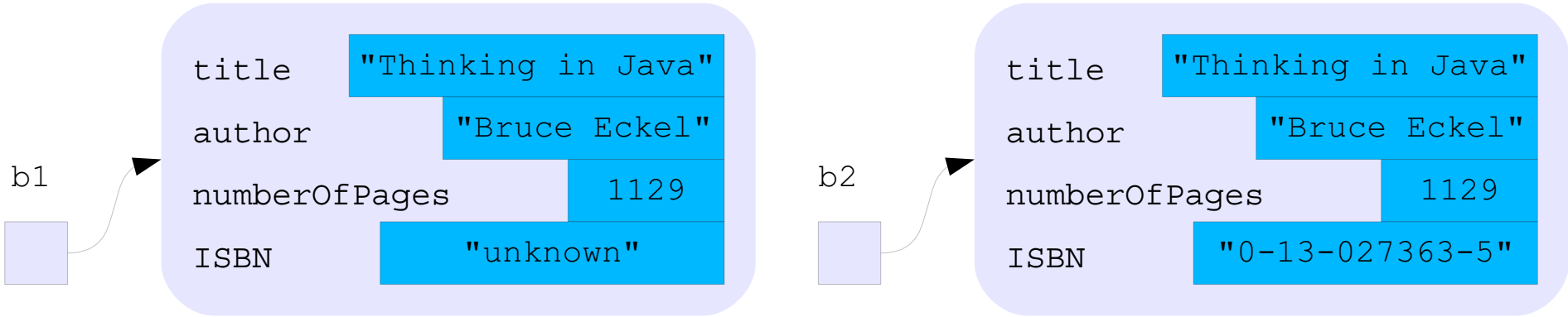

title
"Thinking in Java"
author
"Bruce Eckel"
numberOfPages
1129
ISBN
"unknown"
Initials: B.E.
Initials: D.F.
title
"Java in a nutshell"
author
"David Flanagan"
numberOfPages
353
ISBN
"unknown"
title
"Java network programming"
author
"Elliot Rusty Harold"
numberOfPages
649
ISBN
"unknown"
a
Initials: E.R.H.

Equality and equivalence

```
class ExampleBooks6 {
  public static void main(String[] args) {

    Book b1,b2;

    b1 = new Book("Thinking in Java","Bruce Eckel",1129);
    b2 = new Book("Thinking in Java","Bruce Eckel",1129);

    if (b1 == b2)
      System.out.println("The two books are the same");
    else
      System.out.println("The two books are different");
  }
}
```

```
$ java ExampleBooks6
The two books are different
```

Equality and equivalence



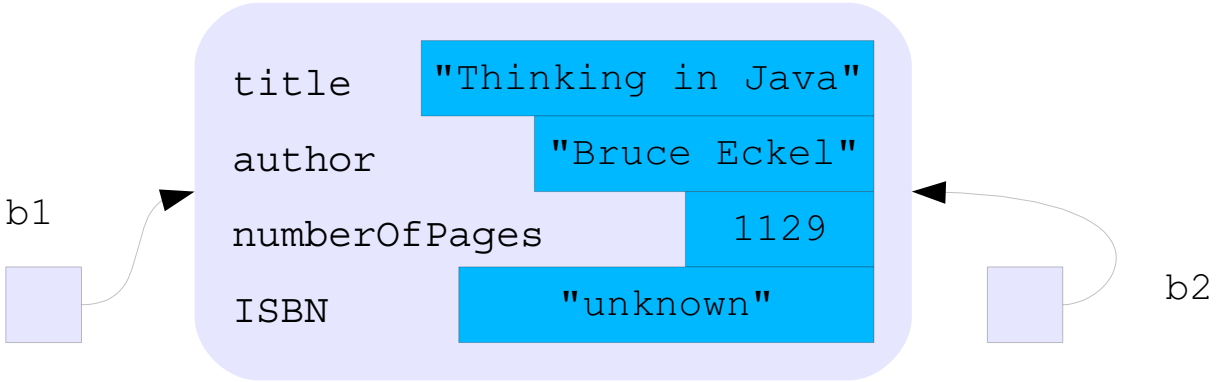

```
b1 = new Book("Thinking in Java","Bruce Eckel",1129);
b2 = new Book("Thinking in Java","Bruce Eckel",1129);

if (b1 == b2)
  System.out.println("The two books are the same");
else
  System.out.println("The two books are different");
```

Equality and equivalence

```
class ExampleBooks6a {
  public static void main(String[] args) {

    Book b1,b2;

    b1 = new Book("Thinking in Java","Bruce Eckel",1129);
    b2 = b1;

    if (b1 == b2)
      System.out.println("The two books are the same");
    else
      System.out.println("The two books are different");
  }
}
```

```
$ java ExampleBooks6a
The two books are the same
```

Equality and equivalence




```
b1 = new Book("Thinking in Java","Bruce Eckel",1129);
b2 = b1;

if (b1 == b2)
  System.out.println("The two books are the same");
else
  System.out.println("The two books are different");
```

Equality and equivalence

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

  ...

  // compare two books
  public boolean equals(Book b) {
    return (title.equals(b.title) &&
            author.equals(b.author) &&
            numberOfPages == b.numberOfPages &&
            ISBN.equals(b.ISBN));
  }
}
```

Equality and equivalence

```
class ExampleBooks7 {
  public static void main(String[] args) {

    Book b1,b2;

    b1 = new Book("Thinking in Java","Bruce Eckel",1129);
    b2 = new Book("Thinking in Java","Bruce Eckel",1129);

    if (b1.equals(b2))
      System.out.println("The two books are the same");
    else
      System.out.println("The two books are different");
  }
}
```

```
$ java ExampleBooks7
The two books are the same
```

Class variables

- Class variables are fields that belong to the class and do not exist in each instance.
- It means that there is always only one copy of this data member, independent of the number of the instances that were created.

Class variables

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

  ...

  public void setOwner(String name) {
    owner = name;
  }
  public String getOwner() {
    return owner;
  }
}
```

Class variables

```
class ExampleBooks8 {
  public static void main(String[] args) {

    Book b1,b2;
    b1 = new Book("Thinking in Java","Bruce Eckel",1129);
    b2 = new Book("Java in a nutshell","David Flanagan",353);
    b1.setOwner("Carlos Kavka");
    System.out.println("Owner of book b1: " + b1.getOwner());
    System.out.println("Owner of book b2: " + b2.getOwner());
  }
}
```

```
$ java ExampleBooks8
Owner of book b1: Carlos Kavka
Owner of book b2: Carlos Kavka
```

Class 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.

Class methods

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

  ...

  public static String description() {
    return "Book instances can store information on books";
  }
}
```

Class methods

```
class ExampleBooks9 {
  public static void main(String[] args) {

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

```
$ java ExampleBooks9
Book instances can store information on books
Book instances can store information on books
```

A static application

- All the examples we have seen till now define a class that contains a static method called main, where usually instances from other classes are created.
- It is possible to define a class with only static methods and static data members.

A static application

```
class AllStatic {
  static int x;
  static String s;

  public static String asString(int aNumber) {
    return "" + aNumber;
  }

  public static void main(String[] args) {
    x = 165;
    s = asString(x);
    System.out.println(s);
  }
}
```

```
$ java AllStatic
165
```

Instance initialization

- All data members in an object are guaranteed to have an initial value.
- There exists a default value for all primitive types:

type	initial value
byte	0
short	0
int	0
long	0
float	0.0F
double	0.0
char	’\0’
boolean	false
references	null

Instance initialization

```
class Values {
  int x;
  float f;
  String s;
  Book b;
}
```

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

    Values v = new Values();
    System.out.println(v.x);
    System.out.println(v.f);
    System.out.println(v.s);
    System.out.println(v.b);
  }
}
```

```
$ java InitialValues
0 0.0 null null
```

Instance initialization

```
class Values {
  int x = 2;
  float f = inverse(x);
  String s;
  Book b;
  Values(String str) { s = str; }
  public float inverse(int value) { return 1.0F / value; }
}
```

```
class InitialValues2 {
  public static void main(String[] args) {
    Values v = new Values("hello");
    System.out.println("" + v.x + "\t" + v.f);
    System.out.println("" + v.s + "\t" + v.b);
  }
}
```

```
$ java InitialValues2
2 0.5
hello null
```

This keyword **this**

- 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 keyword **this**

- For example, the method setOwner in the previous Book class could have been defined as follows:

```
public Book setOwner(String name) {
  owner = name;
  return this;
}
```

```
Book b1 = new Book("Thinking in Java","Bruce Eckel",1129);
System.out.println(b1.setOwner("Carlos Kavka").getInitials());
System.out.println(b1.getOwner());
```

```
B.E.
Carlos Kavka
```

The keyword **this**

- The class Book has two constructors:

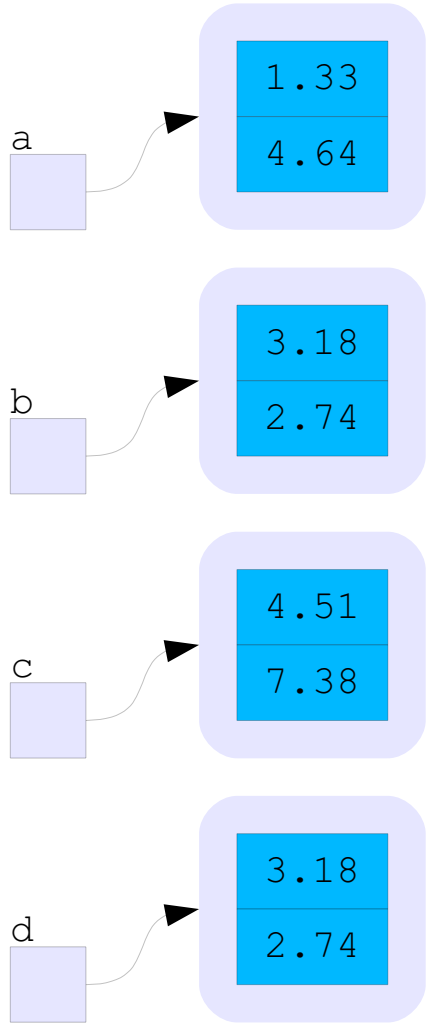
```
Book(String tit,String aut,int num) {
  title = tit; author = aut; numberOfPages = num;
  ISBN = "unknown";
}
Book(String tit,String aut,int num,String isbn) {
  title = tit; author = aut; numberOfPages = num;
  ISBN = isbn;
}
```

- The second can be defined in terms of the first one:

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

An example: complex class

```
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
```

An example: complex class

```
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 = Complex(1.33,4.64)
```

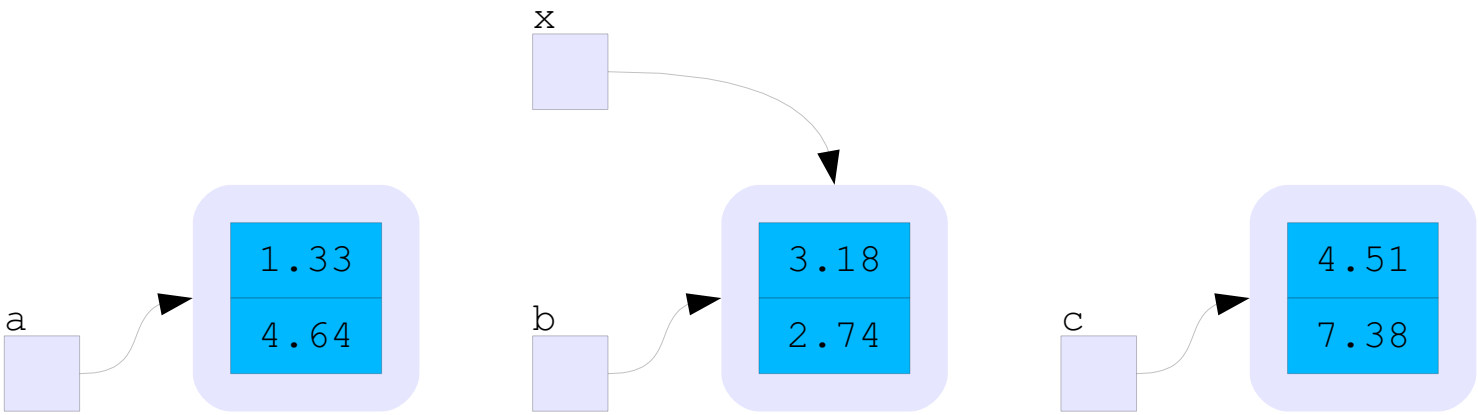
```
double realPart = a.getReal()
```

```
double imPart = a.getImmaginary()
```

An example: complex class

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

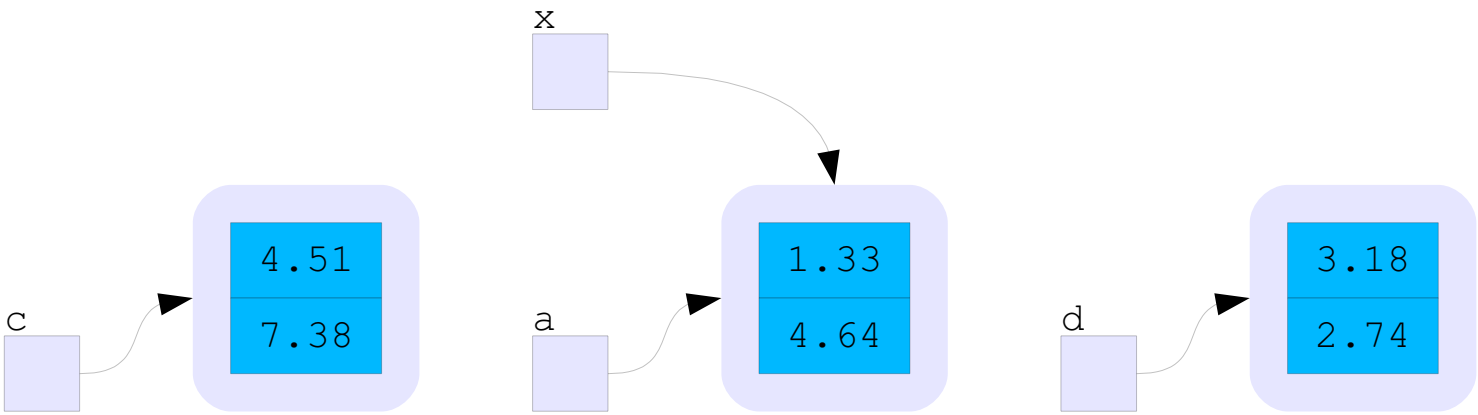
```
Complex c = a.add(b);
```



An example: complex class

```
// substract two complex numbers
public Complex sub(Complex c) {
  return new Complex(real - c.real,im - c.im);
}
```

```
Complex d = c.sub(a);
```

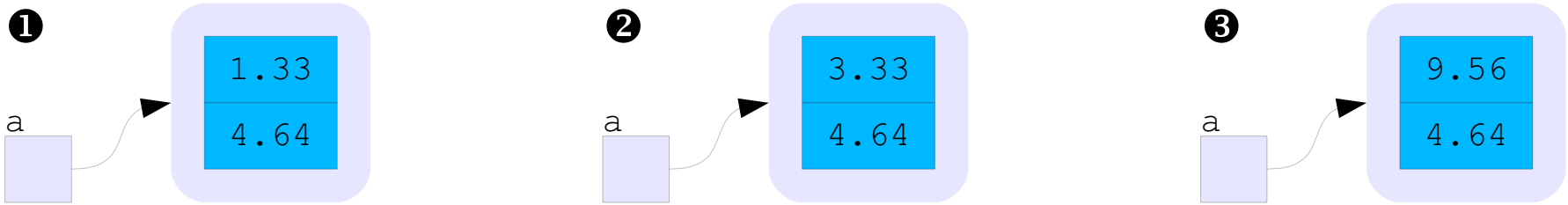


An example: complex class

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

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

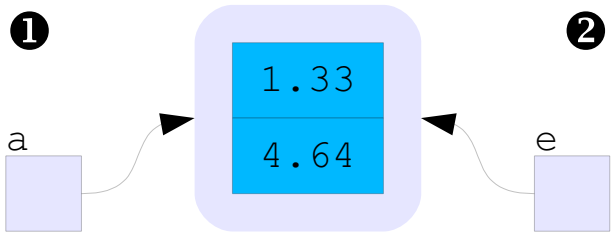
```
❶ Complex a = new Complex(1.33,4.64);
❷ a.addReal(2.0);
❸ a.addReal(3.0).addReal(3.23);
```



An example: complex class

- We must be careful if we want to create one complex number as a copy of the other:

```
❶ Complex a = new Complex(1.33,4.64);
❷ Complex e = a;
```



What will be the effect of

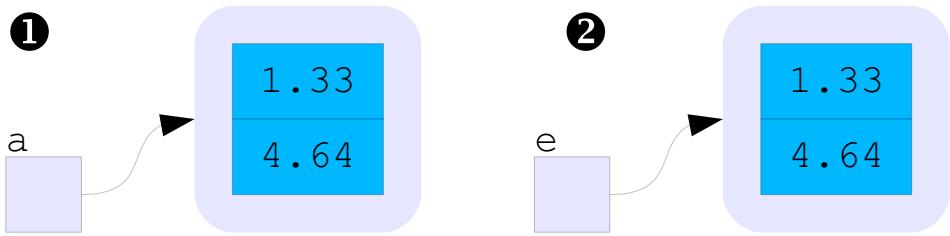
`e.addReal(5.6);` ?

An example: complex class

- We can define a new constructor to avoid the problem:

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

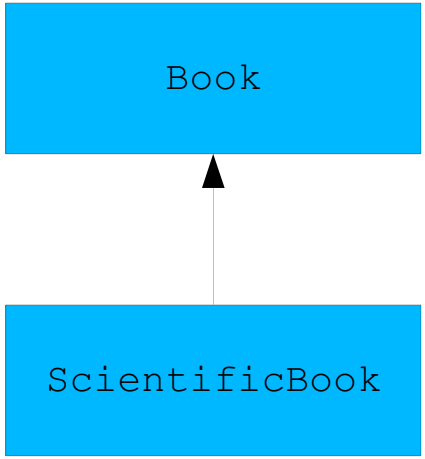
```
❶ Complex a = new Complex(1.33,4.64);
❷ Complex e = new Complex(a);
```



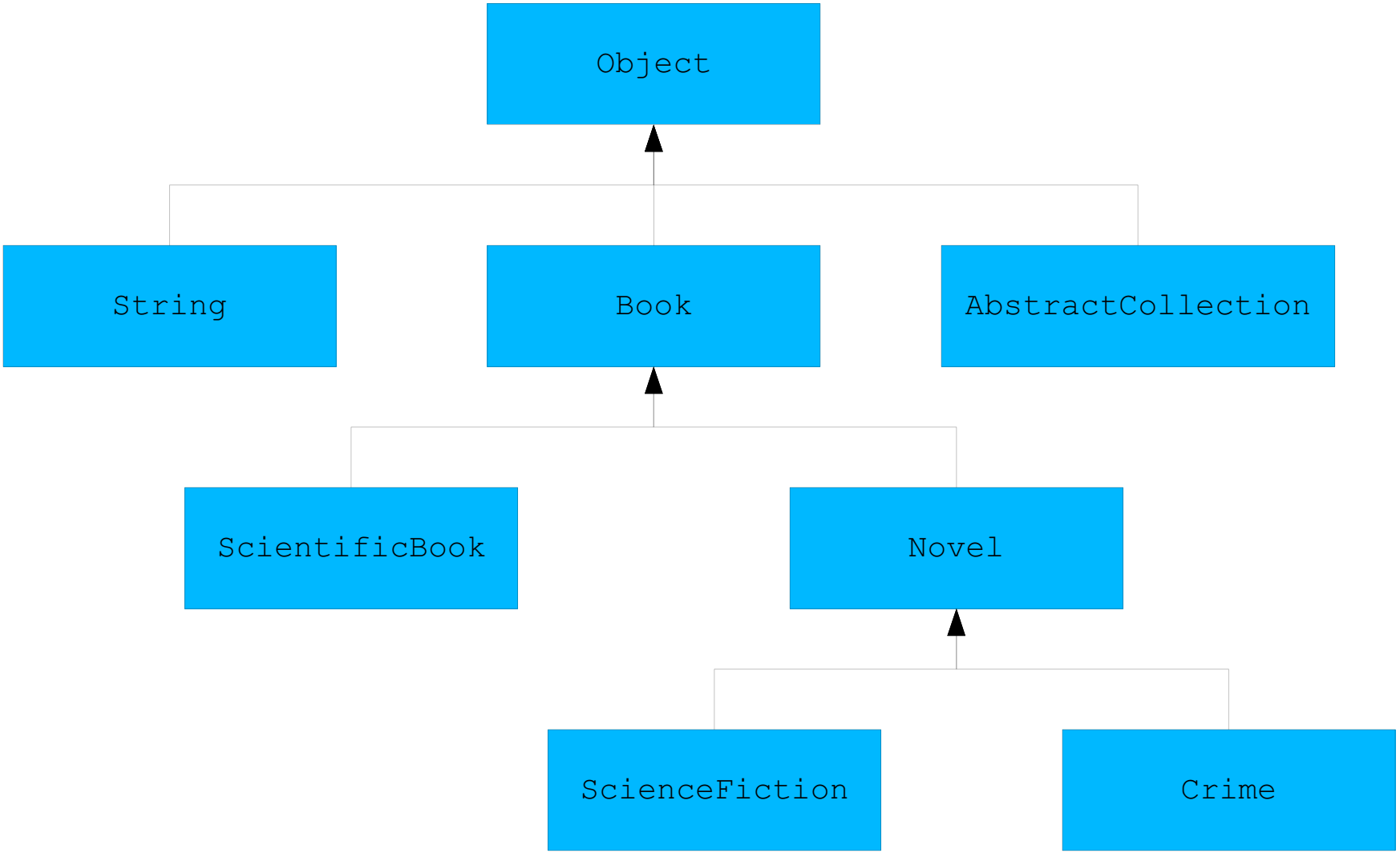
Inheritance

- Inheritance allows to define new classes by reusing other classes, specifying just the differences.
- It is possible to define a new class (subclass) by saying that the class must be *like* other class (superclass):

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

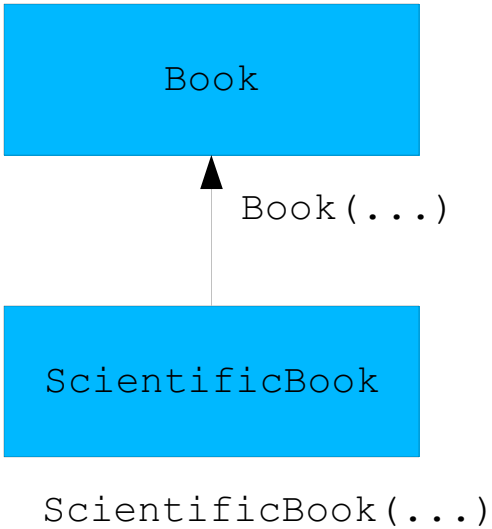


Inheritance (hierarchy)



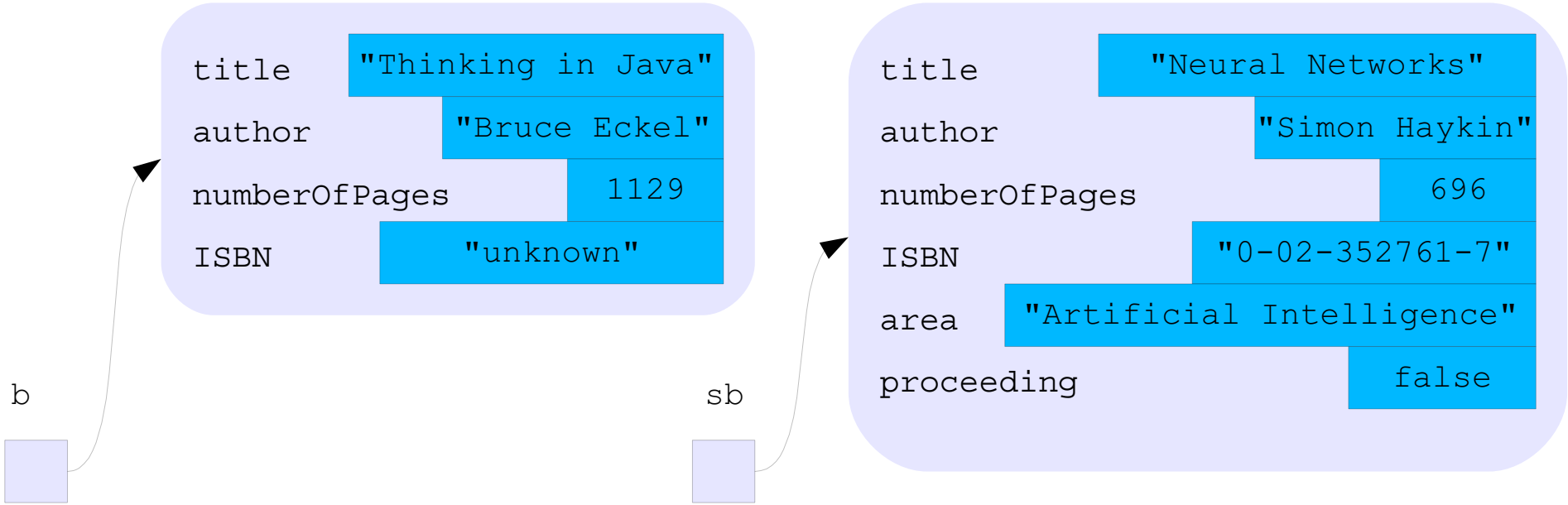
Inheritance (constructors)

```
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;
  }
}
```




```
ScientificBook sb;

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

Inheritance (constructors)

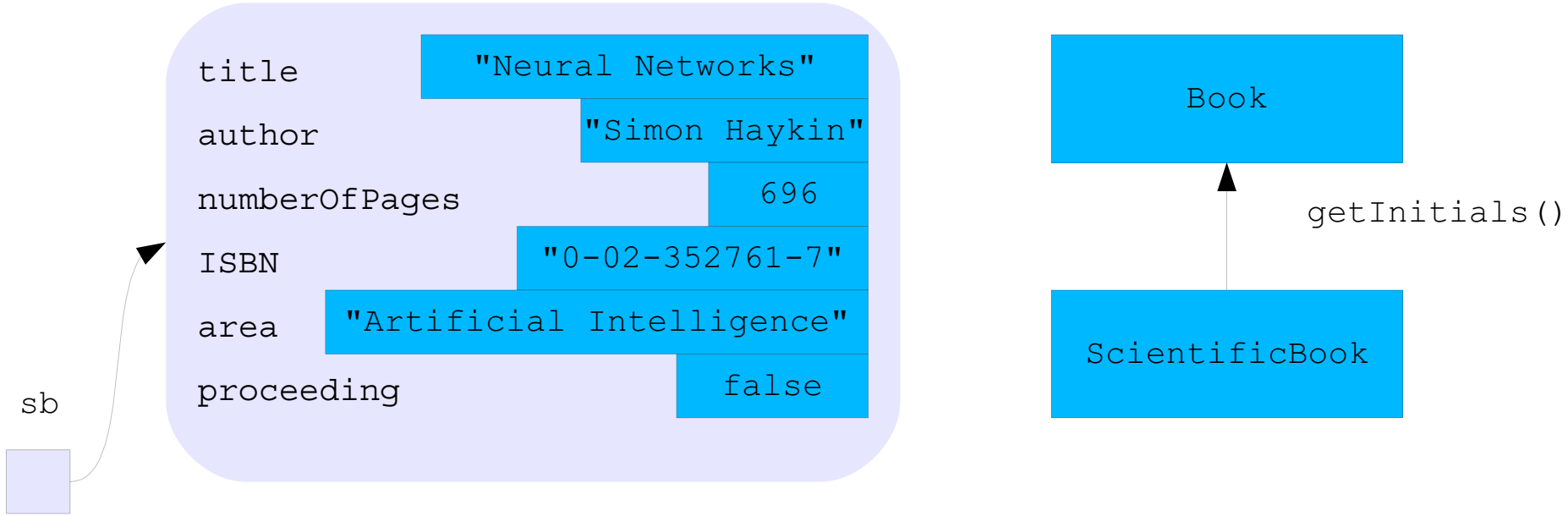



```
Book b = new Book("Thinking in Java","Bruce Eckel",1129);

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

Inheritance (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.

Inheritance (inheriting methods)

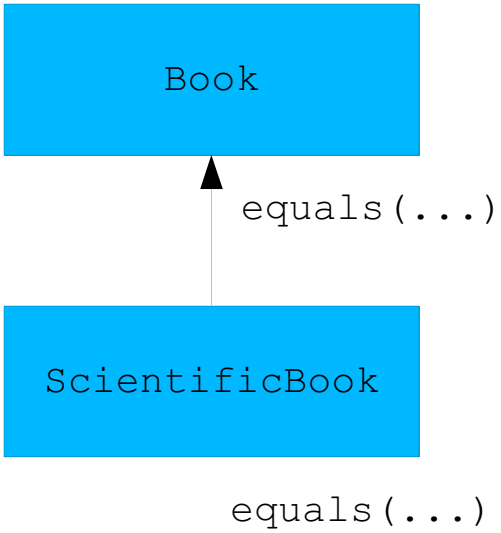



```
ScientificBook sb;

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

S.H.

Inheritance (overriding 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;
  }

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



Inheritance (overriding methods)

- Two possible solutions:

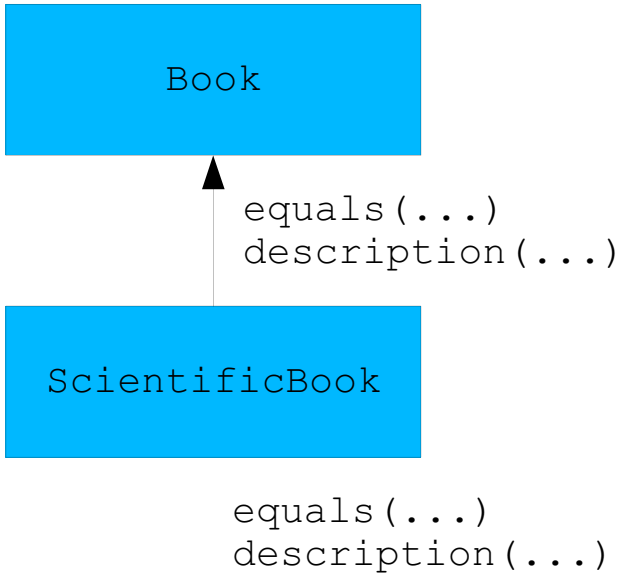
```
public boolean equals(ScientificBook b){
  return super.equals(b) && area.equals(b.area)
         && proceeding == b.proceeding;
}
```

```
public boolean equals(ScientificBook b) {
  return (title.equals(b.title) && author.equals(b.author)
         && numberOfPages == b.numberOfPages
         && ISBN.equals(b.ISBN) && area.equals(b.area)
         && proceeding == b.proceeding;
}
```

Which one is better ?

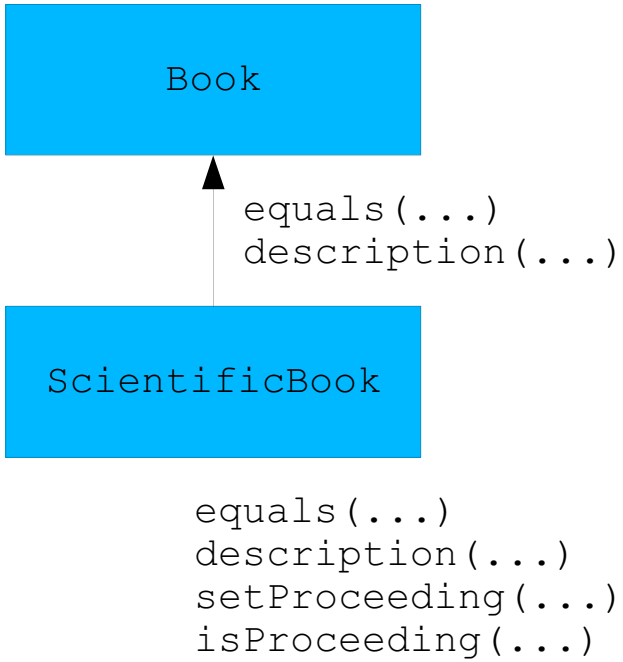
Inheritance (overriding methods)

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

  ScientificBook(String tit, String aut,
       int num, String isbn, String a) {
    ...
  }

  public boolean equals(ScientificBook b){
    ...
  }

  public static String description() {
    return "ScientificBook instances can" +
           " store information on " +
           " scientific books";
  }
}
```



Inheritance (new 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;
  }
  ...

  public void setProceeding() {
    proceeding = true;
  }

  public boolean isProceeding() {
    return proceeding;
  }
}
```



Inheritance (new methods)

```
class TestScientificBooks {
  public static void main(String[] args) {
    ScientificBook sb1,sb2;

    sb1 = new ScientificBook("Neural Networks","Simon Haykin",
                             696,"0-02-352761-7",
                             "Artificial Intelligence");
    sb2 = new ScientificBook("Neural Networks","Simon Haykin",
                             696,"0-02-352761-7",
                             "Artificial Intelligence");
    sb2.setProceeding();
    System.out.println(sb1.getInitials());
    System.out.println(sb1.equals(sb2));
    System.out.println(sb2.description());
  }
}
```

```
$ java TestScientificBooks
S.H.      false
ScientificBook instances can store information on scientific books
```

instanceof

- **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
```

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 and **getClass**()

```
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
class Book
class ScientificBook
true true false true
```

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.
- If not specified, classes are inserted into the *default* package.

Packages

- 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());
  }
}
```

```
$ java TestDate
Wed Oct 25 09:48:54 CEST 2006
```

Packages

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

```
package myBook;

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

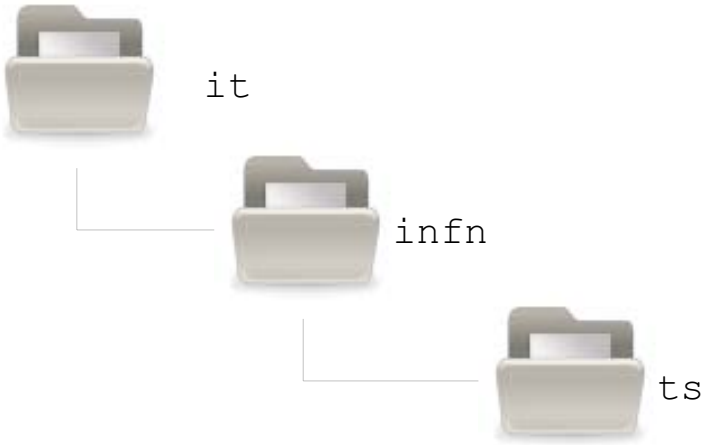
Book.java

ExampleBooks.java

```
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

- Files have to be stored in special directories accessible on the class path ($CLASSPATH):



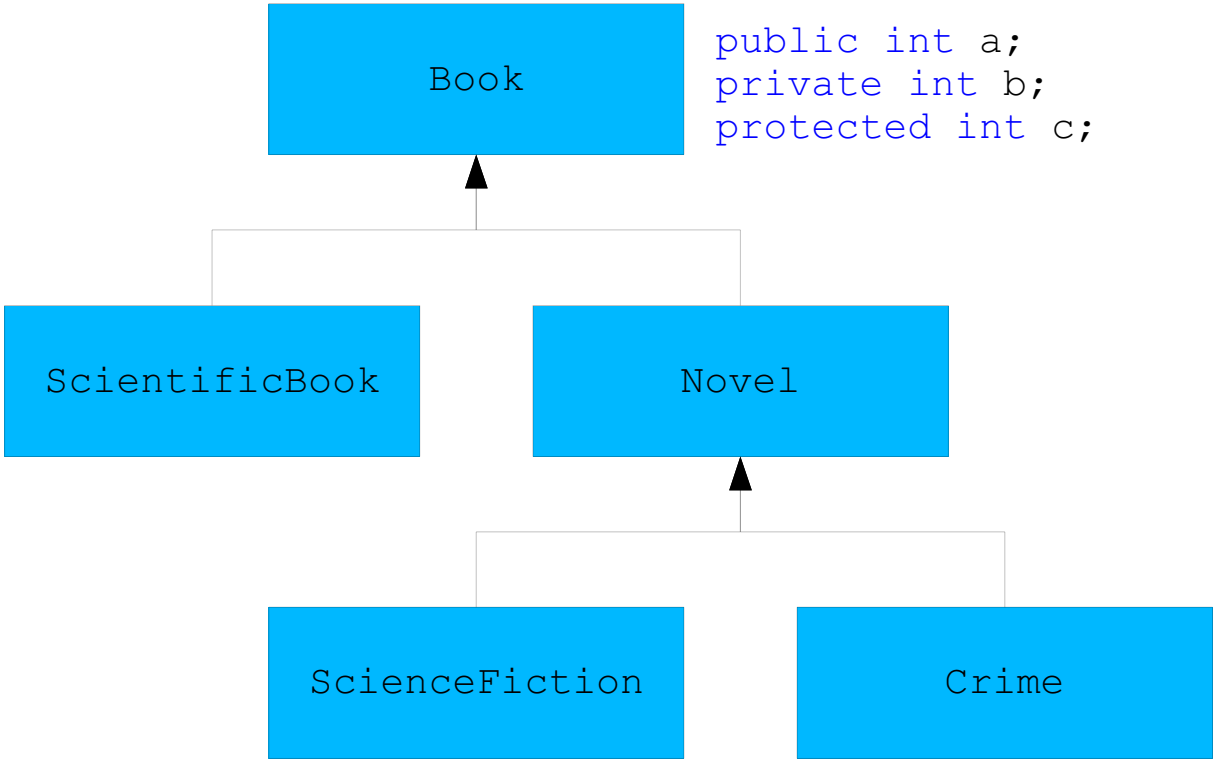

```
package it.infn.ts;

class Book {
    ...
}
```

Example of use:

```
import it.infn.ts.Book;

class TestBook {
    ...
    Book b = new Book(...);
    ...
}
```

Access control

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



Access control

- The default access allows full access from all classes that belong to the same package.
- For example, it is possible to set the proceeding condition of a scientific book in two ways:

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

- or by just accessing the data member:

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

Access control

- Usually we do not want direct access to a data member in order to guarantee encapsulation:

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

- Now, the proceeding condition can only be asserted with the message:

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

Access control

- The same access control can be applied to methods.

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

  private boolean initialized() {

    return title != null && author != null &&
           numberOfPages != 0 && area != null;
  }
}
```

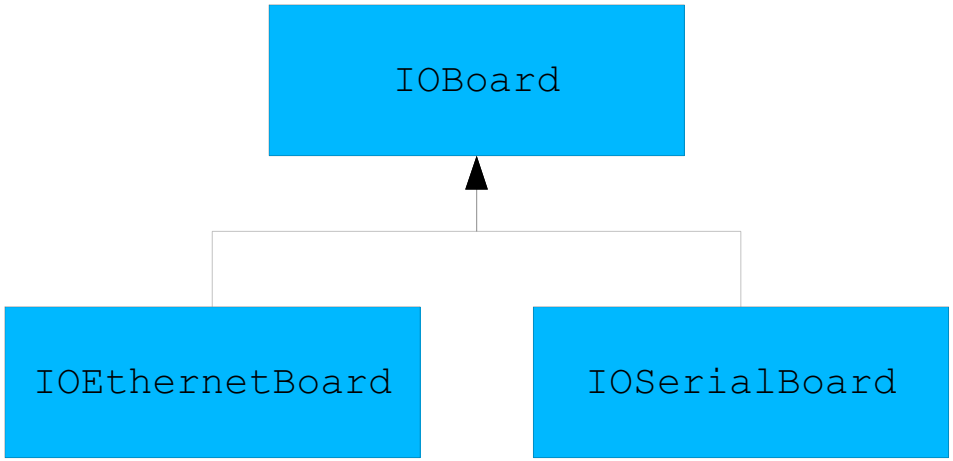
Where can initialized() be called from ?

Final and abstract

- The modifiers **final** and **abstract** can be applied to classes and methods:
  - **final**:
    - A final class does not allow subclassing.
    - A final method cannot be redefined in a subclass.
  - **abstract**:
    - An abstract class is a class that cannot be instantiated.
    - An abstract method has no body, and it must be redefined in a subclass.

Final and abstract

- An example: the class IOBoard and its subclasses.



Final and abstract

```
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();
}
```

Final and abstract

```
class IOSerialBoard extends IOBoard {
  int port;

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

Final and abstract

```
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

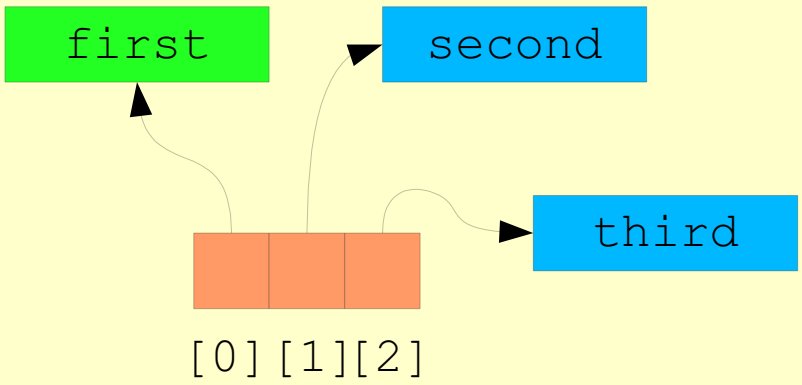
- Creation of a serial board instance:

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

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

Polymorphism

```
class TestBoards2 {
  public static void main(String[] args) {
    IOBoard[] board = new IOBoard[3];

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

    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
  - no instances can be created from interfaces.

Interfaces

- An interface for specifying IO boards behavior:

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

- An interface for specifying *nice* behavior:

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

Interfaces

```
class IOSerialBoard2 implements IOBoardInterface {
  int port;

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

Interfaces

- A class can implement more than one interface.

```
class IOSerialBoard2 implements IOBoardInterface,
                                NiceBehavior {

....
}
```

Which methods should it implement ?

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));
  }
}
```

```
$ 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:

```
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 TestExceptions3 {
  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");
  }
}
```

```
$ java TestExceptions3
No such position
```

Exceptions

- It is possible to send messages to an exception object:

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

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

Exceptions

- We can add multiple catch blocks and a finally clause:

```
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!");
    }
  }
}
```

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!");
    }
  }
```

```
String sentence = "A test sentence.";
MultipleCatch mc = new MultipleCatch();
mc.printInfo(sentence);
```

```
First: A Last: e
done!
```

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!");
    }
  }
```

```
String sentence = "A test sentence";
MultipleCatch mc = new MultipleCatch();
mc.printInfo(sentence);
```

```
Wrong sentence, no dot?
done!
```

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!");
    }
  }
```

```
String sentence = null;
MultipleCatch mc = new MultipleCatch();
mc.printInfo(sentence);
```

```
Non valid string
done!
```

Exceptions

- There exists a set of predefined exceptions that can be caught.
- In some cases it is compulsory to catch exceptions.
- It is also possible to express the interest to not to catch even compulsory exceptions.

Input - Output

- Input output in Java is rather complicated.
- However, input output from files, devices, memory or web sites is performed in the same way.
- It is based on the idea of streams:
  - An *input stream* is a data source that can be accessed in order to get data.
  - An *output stream* is a data sink, where data can be written.

Input - Output

- Streams can be classified in:
  - byte streams
    - provides support also for fundamental types.
  - character streams
    - Unicode, but with OS character support.
- Streams can be:
  - non buffered
  - buffered

Input - Output

byte oriented stream

```
import java.io.*;

class WriteBytes {
  public static void main(String[] args) {
    int data[] = { 10,20,30,40,255 };

    FileOutputStream f;
    try {
      f = new FileOutputStream("file1.data");
      for(int i = 0;i < data.length;i++)
        f.write(data[i]);
      f.close();
    } catch (IOException e) {
      System.out.println("Error with files:"+e.toString());
    }
  }
}
```

Input - Output

byte oriented stream

```
import java.io.*;

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

    FileInputStream f;
    try {
      f = new FileInputStream("file1.data");
      int data;
      while((data = f.read()) != -1)
        System.out.println(data);
      f.close();
    } catch (IOException e) {
      System.out.println("Error with files:"+e.toString());
    }
  }
}
```

```
$ java ReadBytes
10 20 30 40 255
```

Input - Output

byte oriented stream

```
import java.io.*;

class WriteArrayBytes {
  public static void main(String[] args) {
    byte data[] = { 10,20,30,40,-128 };

    FileOutputStream f;
    try {
      f = new FileOutputStream("file1.data");
      f.write(data,0,data.length);
      f.close();
    } catch (IOException e) {
      System.out.println("Error with files:"+e.toString());
    }
  }
}
```

Input - Output

buffered byte oriented stream

```
import java.io.*;

class WriteBufferedBytes {
  public static void main(String[] args) {
    int data[] = { 10,20,30,40,255 };
    FileOutputStream f;
    BufferedOutputStream bf;

    try {
      f = new FileOutputStream("file1.data");
      bf = new BufferedOutputStream(f);
      for(int i = 0;i < data.length;i++)
        bf.write(data[i]);
      bf.close();
    } catch (IOException e) {
      System.out.println("Error with files:"+e.toString());
    }
  }
}
```

Input - Output

buffered byte oriented stream

```
import java.io.*;

class ReadBufferedBytes {
  public static void main(String[] args) {
    FileInputStream f; BufferedInputStream bf;
    try {
      f = new FileInputStream("file1.data");
      bf = new BufferedInputStream(f);
      int data;
      while((data = bf.read()) != -1)
        System.out.println(data);
      bf.close();
    } catch (IOException e) {
      System.out.println("Error with files:"+e.toString());
    }
  }
}
```

```
$ java ReadBufferedBytes
10 20 30 40 255
```

Input - Output

- A data buffered byte oriented stream can deal with data in small pieces (fundamental types).
- The following messages are provided:
  - **readBoolean**() **writeBoolean**(*boolean*)
  - **readByte** () **writeByte**(*byte*)
  - **readShort**() **writeShort**(*short*)
  - **readInt**() **writeInt**(*int*)
  - **readLong**() **writeLong**(*long*)
  - **readFloat**() **writeFloat**(*float*)
  - **readDouble**() **writeDouble**(*double*)

Input - Output

data buffered byte oriented stream

```
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 with files:"+e.toString());
  }
}
```

Input - Output

data buffered byte oriented stream

```
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 with files:"+e.toString());
  }
}
```

```
$ java ReadData
10.3
20.65
8.45
-4.12
true
```

Input - Output

- The character oriented streams can be used to read and write characters.
- There exists three methods that can be used to write data into this kind of streams:
  - **write**(*String*,*int*,*int*)
  - **write**(*char*[],*int*,*int*)
  - **newLine**()

Input - Output

buffered character oriented stream

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

Input - Output

buffered character oriented stream

```
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 with files:"+e.toString());
  }
}
```

```
$ java ReadText
HelloWorld!
nice!
```

Input - Output

standard input

```
import java.io.*;

class StandardInput {
  public static void main(String[] args) {
    InputStreamReader isr;
    BufferedReader br;
    try {
      isr = new InputStreamReader(System.in);
      br = new BufferedReader(isr);
      String line;
      while ((line = br.readLine()) != null)
        System.out.println(line);
    } catch (IOException e) {
      System.out.println("Error with standard input");
  }
}
```

correct lecture notes

Input - Output

standard input with scanner

```
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 with standard input");
  }
}
```

```
$ java ReadWithScanner
11
9
^D
20
```

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.

Threads

```
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");
      }
    }
  }
}
```

Threads

```
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
...
```

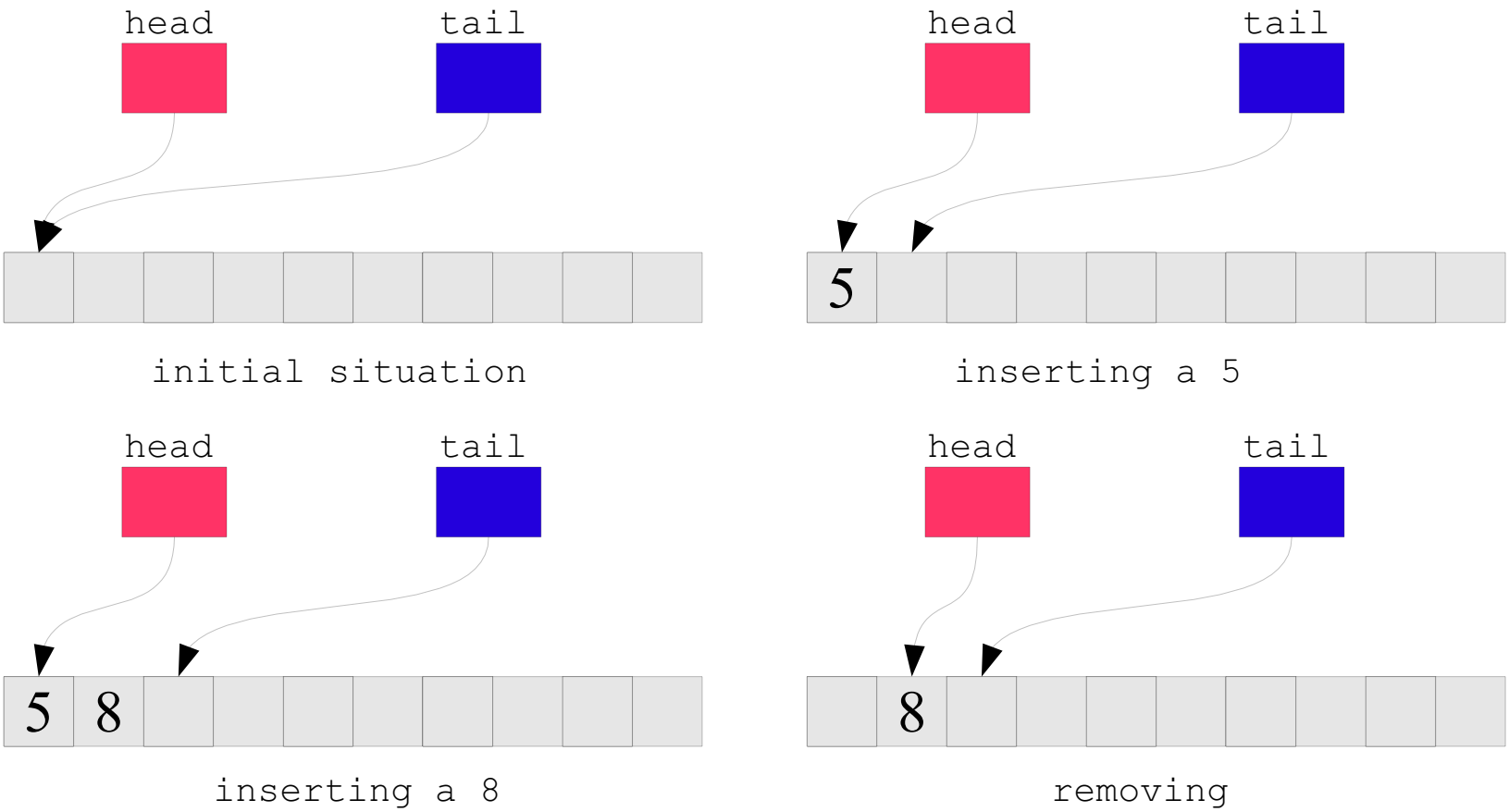
Threads

- A typical producer - consumer application:

```
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();
  }
}
```

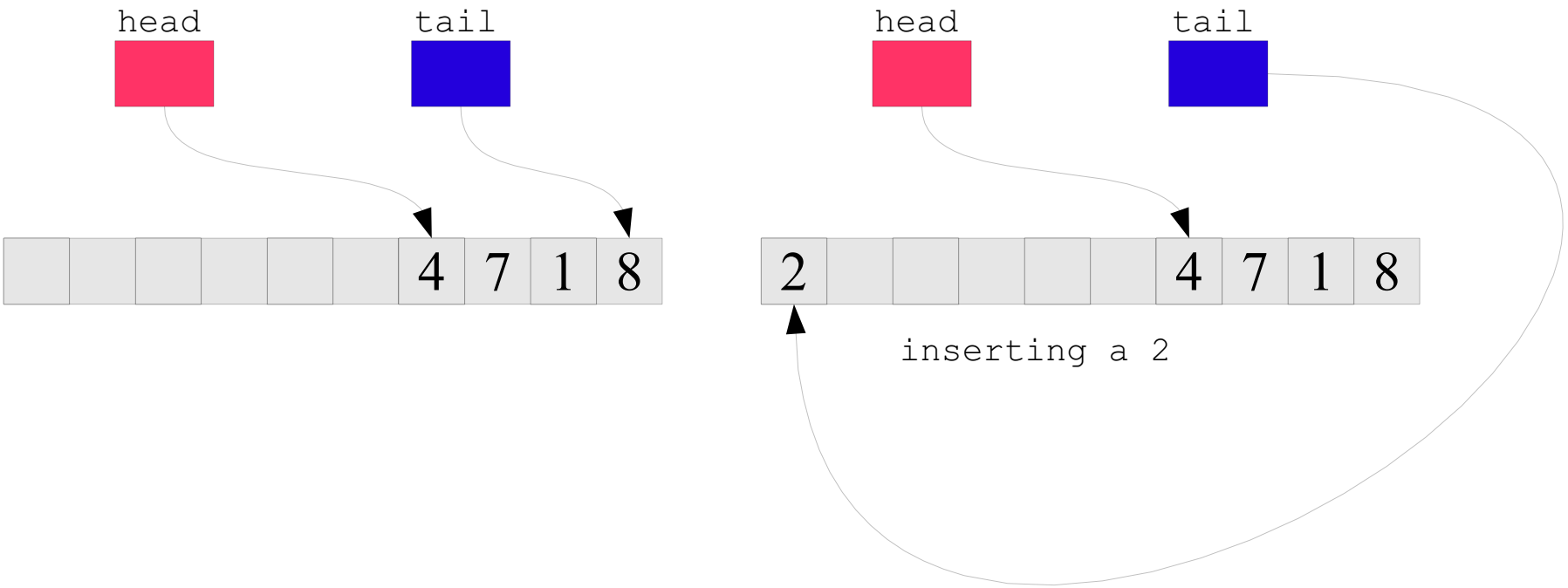
Threads

- Insertion and removal of elements in the buffer:



Threads

- Going beyond the limit of the buffer:



Threads

```
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);
    }
  }
}
```

Threads

```
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;
  }
}
```

However... it does not work

Threads

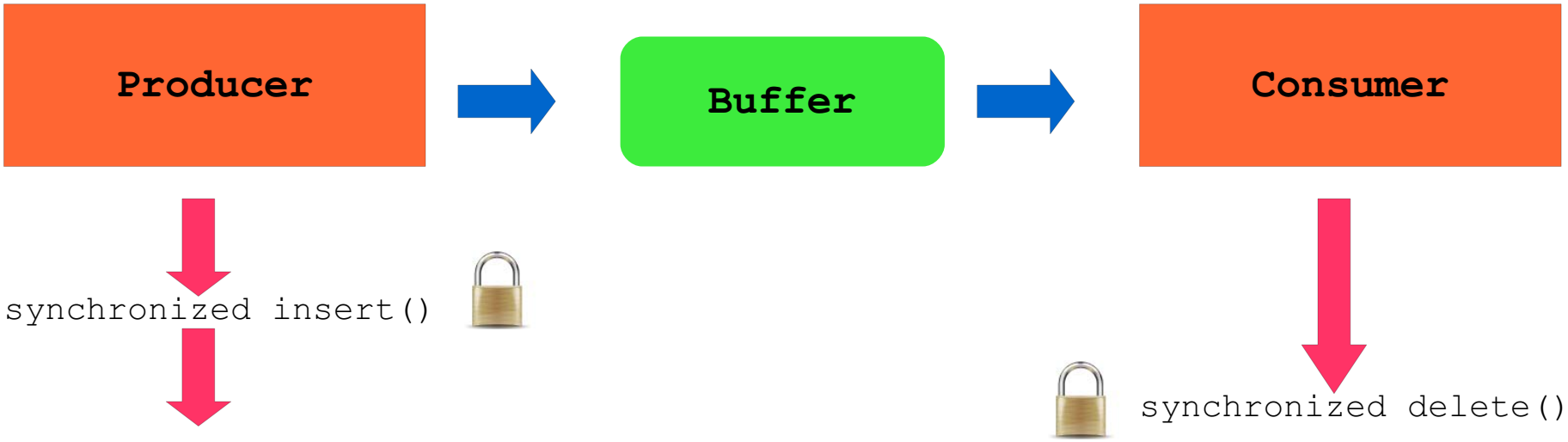
- The implementation does not work!

  - 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.

Threads

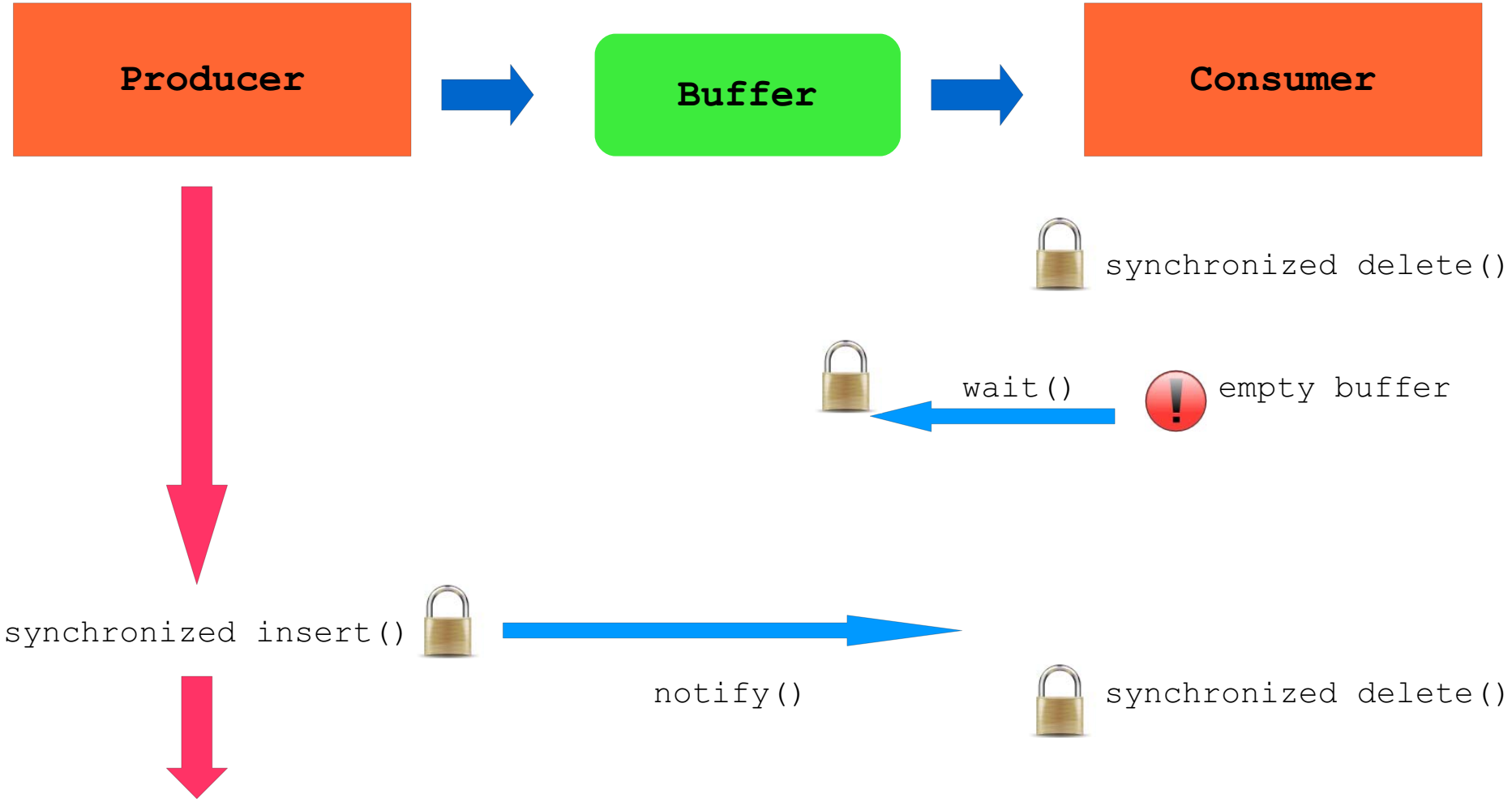
- Synchronized access to a critical resource can be achieved with **synchronized** method:
  - They are not allowed to be executed concurrently on the same instance.
  - Each instance has a lock, used to synchronize the access.



Threads

- Threads are synchronized with **wait** and **notify**:
  - The message **wait** puts the calling thread to sleep, releasing the lock.
  - The message **notify** awakens a waiting thread on the corresponding lock.

Threads


Producer
Buffer
Consumer
synchronized delete()
wait()
empty buffer
synchronized insert()
notify()
synchronized delete()

Threads

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

Threads

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

Jar files

- When we were compiling the example of the Producer and Consumer, four class files were generated:

```
$ ls *.class

Buffer.class
Consumer.class
ProducerConsumer.class
Producer.class
```

- In order to distribute the executable application it is necessary to copy the four files.

Jar files

- A **JAR** (Java ARchive) file can be created and manipulated by the command jar.
- In order to create a **JAR** file, it is necessary to define a manifest file, which contains information on the files included.
- The command jar creates a default manifest file in the directory **META-INF** with name **MANIFEST.MF**, just below the current directory.

Jar files

- The creation of the JAR file can be done as follows:

```
$ jar cmf mylines.txt ProducerConsumer.jar \
          ProducerConsumer.class Producer.class \
          Consumer.class Buffer.class
```

- with:

```
$ cat mylines.txt
Main-Class: ProducerConsumer
```

- The application can be executed as follows:

```
$ java -jar ProducerConsumer.jar
```

Jar files

- It contents can be displayed as follows:

```
$ jar tf ProducerConsumer.jar
META-INF/
META-INF/MANIFEST.MF
ProducerConsumer.class
Producer.class
Consumer.class
Buffer.class
```

- Note that a manifest file was added:

```
Manifest-Version: 1.0
Main-Class: ProducerConsumer
Created-By: 1.2.2 (Sun Microsystems Inc.)
```

Ant

- Ant is a building tool that provides support to compile, pack, deploy and document Java applications.
- In some sense, its functionality is similar to the make command, except that the approach is completely different.
- The specifications for the ant command are defined in term of XML sentences.
- Ant can be extended in an object oriented sense.

Ant

- An example of a **build.xml** file:

```
<?xml version="1.0"?>
<!-- first build file -->
<project name="HelloWorld" default="build" basedir=".">
<target name="build">
<javac srcdir="." />
</target>
</project>
```

```
# ant
Buildfile: build.xml
build:
[javac] Compiling 1 source file
BUILD SUCCESSFUL
Total time: 3 seconds
```

Ant

- A *project* specifies three elements
  - *name*
  - *target*
  - *basedir*
- A *target* specifies five elements:
  - *name*
  - *depends*
  - *if*
  - *unless*
  - *description*

Ant

- Properties are like variables:

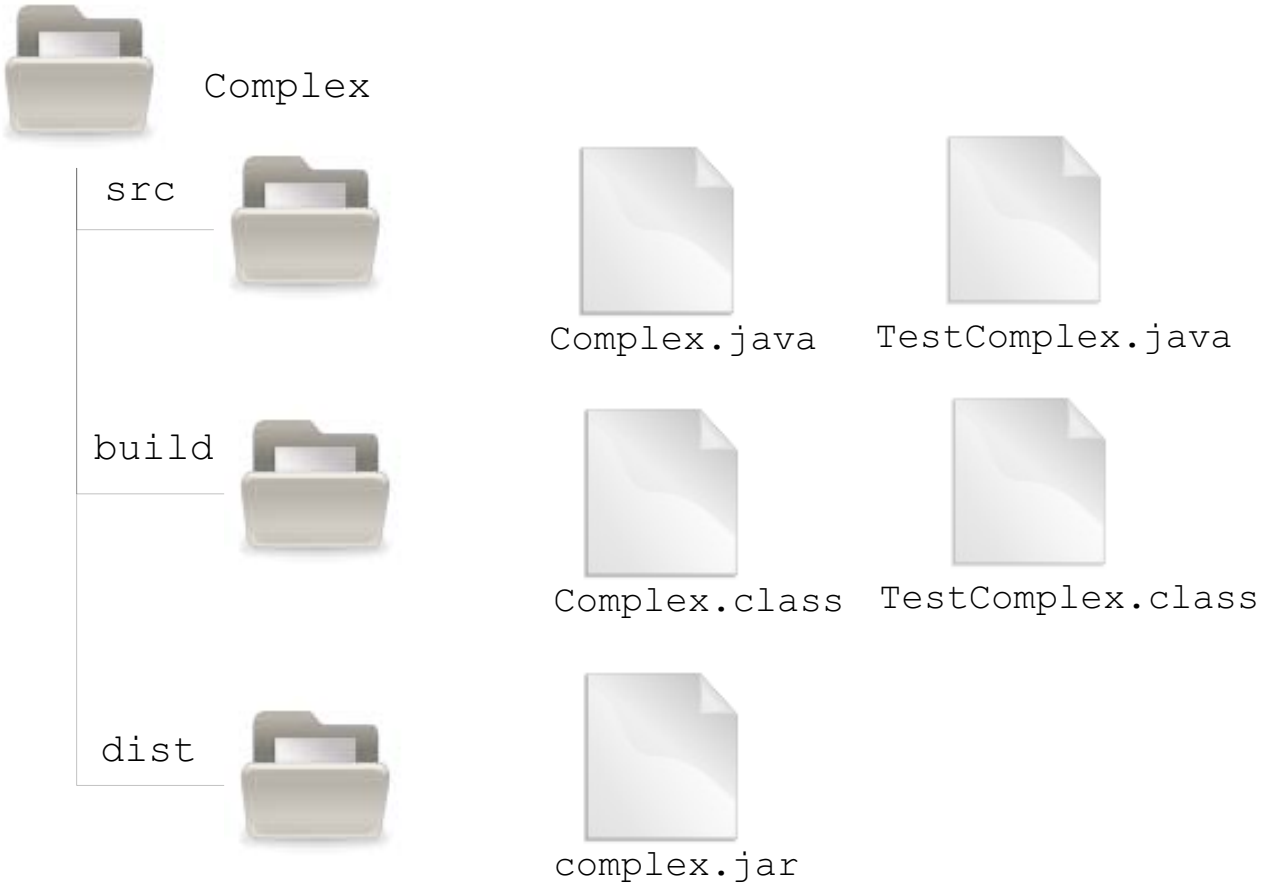
```
<property name="conditionOK" value="yes"/>
<property name="src-dir" location="src"/>
```

- The values can be obtained by placing the property name between **${** and **}**.

```
<javac srcdir="${src-dir}"/>
```

Ant

- An example:



four tasks:
- clean
- init
- build
- dist

Ant

```
<?xml version="1.0"?>

<!-- first build file -->

<project name="Complex" default="dist" basedir=".">

  <!-- set global properties -->

  <property name="src-dir" location="src"/>
  <property name="build-dir" location="build"/>
  <property name="dist-dir" location="dist"/>

  <target name="init" description="initial task">

    <!-- Create the build directory -->

    <mkdir dir="${build-dir}"/>
  </target>
```

Ant

```
<target name="build" depends="init"
        description="compile task">
  <javac srcdir="${src-dir}" destdir="${build-dir}"/>
</target>

<target name="dist" depends="build"
        description="build distribution" >

  <mkdir dir="${dist-dir}"/>

  <jar jarfile="${dist-dir}/complex.jar"
       basedir="${build-dir}">
     <include name="*.class"/>
     <manifest>
       <attribute name="Main-Class" value="TestComplex"/>
      </manifest>
  </jar>
</target>
```

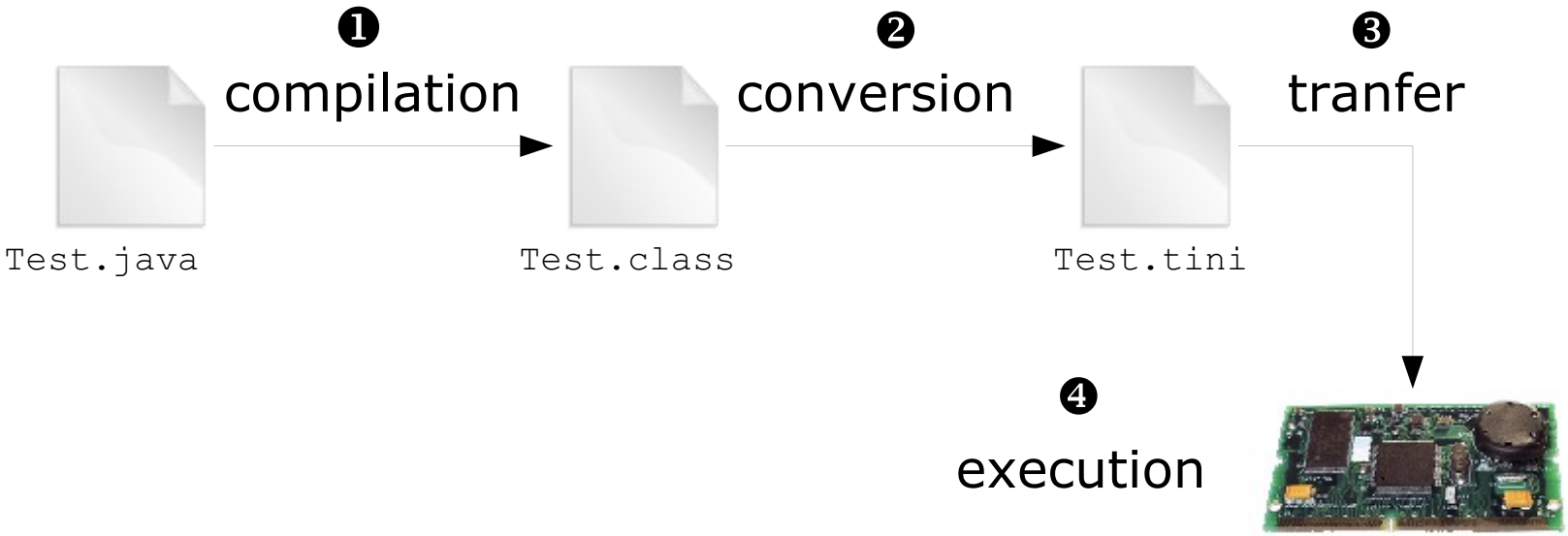
Ant

```
  <target name="clean" description="clean up">
    <delete dir="${build-dir}"/>
    <delete dir="${dist-dir}"/>
  </target>

</project>
```

```
$ ant
Buildfile: build.xml
init:
   [mkdir] Created dir: ComplexNumbers/build
build:
   [javac] Compiling 2 source files to ComplexNumbers/build
dist:
   [mkdir] Created dir: ComplexNumbers/dist
     [jar] Building jar: ComplexNumbers/dist/complex.jar
BUILD SUCCESSFUL
Total time: 11 seconds
```

Java on the TINI

- In order to be able to execute an application developed in Java on the TINI, it is necessary to follow a four steps process:



Java on the TINI

- Step 1: compilation

```
$ javac HelloWorld.java
```

- Step 2: conversion

```
$ java -classpath /tini/bin/tini.jar TINIConvertor \
       -f HelloWorld.class \
       -d /tini/bin/tini.db -o HelloWorld.tini
```

Java on the TINI

- Step 3: transfer

```
$ ftp tini
Connected to tini.
220 Welcome to slush. (Version 1.17) Ready for user login.
User (tini:(none)): root
331 root login allowed. Password required.
Password:
230 User root logged in.
ftp> bin
200 Type set to Binary
ftp> put HelloWorld.tini
200 PORT Command successful.
150 BINARY connection open, putting HelloWorld.tini
226 Closing data connection.
ftp: 183 bytes sent in 0.00 Seconds.
ftp> bye
```

Java on the TINI

- Step 4: execution

```
# telnet tini
Connected to tini.
Escape character is '^]'.
Welcome to slush. (Version 1.17)
tini00a93c login: root
tini00a93c password:

TINI /> java HelloWorld.tini
HelloWorld
```

TiniAnt

```
<?xml version="1.0" encoding="UTF-8"?>

<project name="HelloWorld" default="convert" basedir=".">

  <taskdef name="tini" classname="net.geeba.ant.Tini"/>

  <property name="tini.dir" value="/tini"/>
  <property name="tini.db" value="${tini.dir}/bin/tini.db"/>
  <property name="tini.classes"
            value="${tini.dir}/bin/tiniclasses.jar"/>
  <property name="tini.jar"
            value="${tini.dir}/bin/tini.jar"/>

  <target name="init" description="initialize">
    <mkdir dir="build"/>
  </target>
```

TiniAnt

```
  <target name="build" depends="init" description="compile">
    <javac srcdir="src" destdir="build"
           bootclasspath="${tini.classes}"/>
  </target>

  <target name="convert" depends="build" description="convert">
    <tini outputfile="HelloWorld.tini" database="${tini.db}"
          classpath="${tini.jar}">
      <convert dir="build"/>
    </tini>
  </target>

  <target name="clean" description="clean">
    <delete dir="build"/>
    <delete file="HelloWorld.tini"/>
  </target>
</project>
```

TiniAnt

```
$ ant
Buildfile: build.xml
init:
   [mkdir] Created dir: HelloWorldAnt/build
build:
   [javac] Compiling 1 source file to HelloWorldAnt/build
convert:
   [tini] TINIConvertor (KLA)
   [tini] Version 1.24 for TINI 1.1 (Beta 2 and later ONLY!!!)
   [tini] Built on or around March 20, 2002
   [tini] Copyright (C) 1996 - 2002 Dallas Semiconductor Corp.
   [tini] Loading class HelloWorldAnt/build/HelloWorld.class
          from file HelloWorldAnt/build/HelloWorld.class
   [tini] Getting UNMT...there are 0 user native methods
   [tini] Class HelloWorld, size 125, CNUM 8000, TH Contrib: 19
   [tini] Initial length of the application: 125
   [tini] Output file size : 472
   [tini] Number of string table entries: 1
BUILD SUCCESSFUL
Total time: 8 seconds
```