Introduction to Java

Carlos Kavka
Departamento de Informática
Universidad Nacional de San Luis
San Luis, Argentina
email: ckavka@unsl.edu.ar

Seventh College on Microprocessor-based Real-Time Systems in Physics
The Abdus Salam ICTP, Trieste, Italia
October 28 – November 22, 2002

Introduction

- Java is a very powerful language that has generated a lot of interest in the last years.
- 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.
- The world wide web has popularized the use of Java, because programs written in this language can be transparently downloaded with the web pages and executed in any computer with a Java capable browser.

Some definitions

- 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.

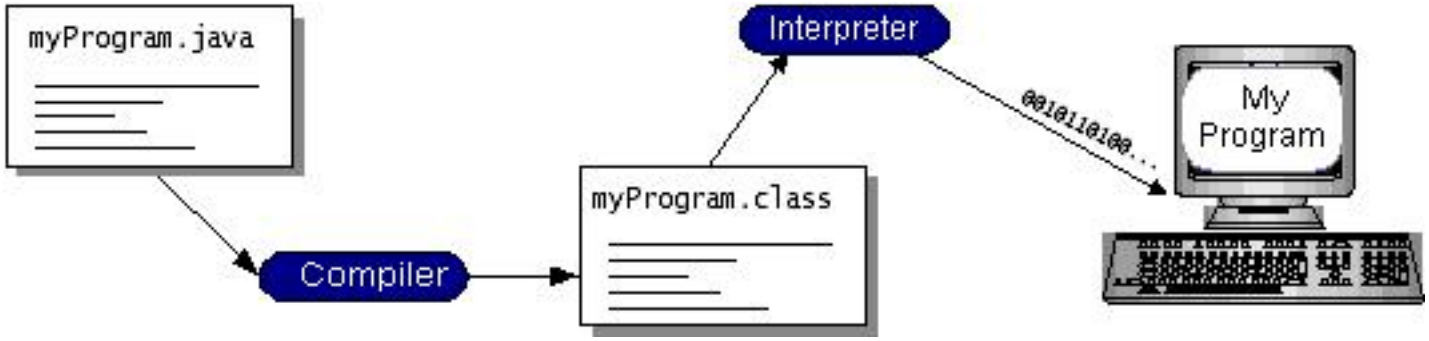
These lecture notes

- These lecture notes assume that you have some familiarity with C.
- All examples used in this notes are available for experimentation.
- There is a page that complements these lecture notes at:

  `http://www.ictp.trieste.it/~ckavka/Java`

The Java platform

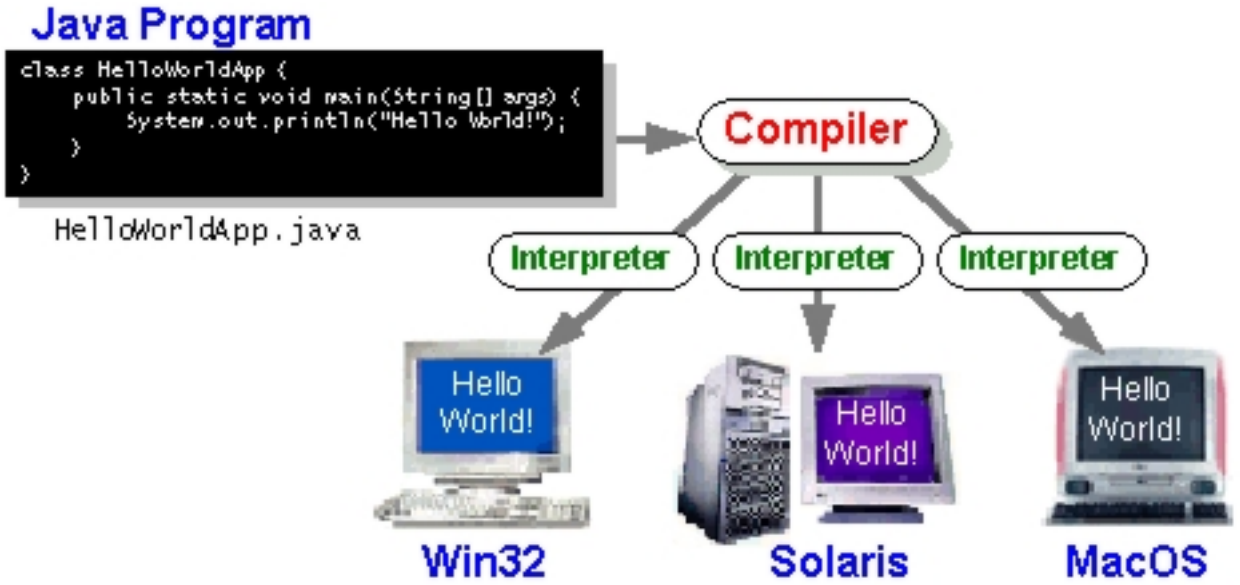
- Java programs are compiled to Java byte-codes, a kind of machine independent representation. The program is then executed by an interpreter called the Java Virtual Machine (JVM) [1].



[1] Ilustration from Sun Java Tutorial at http://java.sun.com

Introduction

- Its main advantage is the fact that the compiled code is independent of the architecture of the computer) [2].



- The price to pay is a slower execution.

[2] Ilustration from Sun Java Tutorial at http://java.sun.com

A first example

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

public class HelloWorld {

  public static void main(String[] args) {

    System.out.println("Hello World!");  // display output

  }
}
```

Development cycle

- Creation of the source file:

```
# emacs HelloWorld.java
```

- Compilation:

```
# javac HelloWorld.java
# ls
  HelloWorld.java
  HelloWorld.class
```

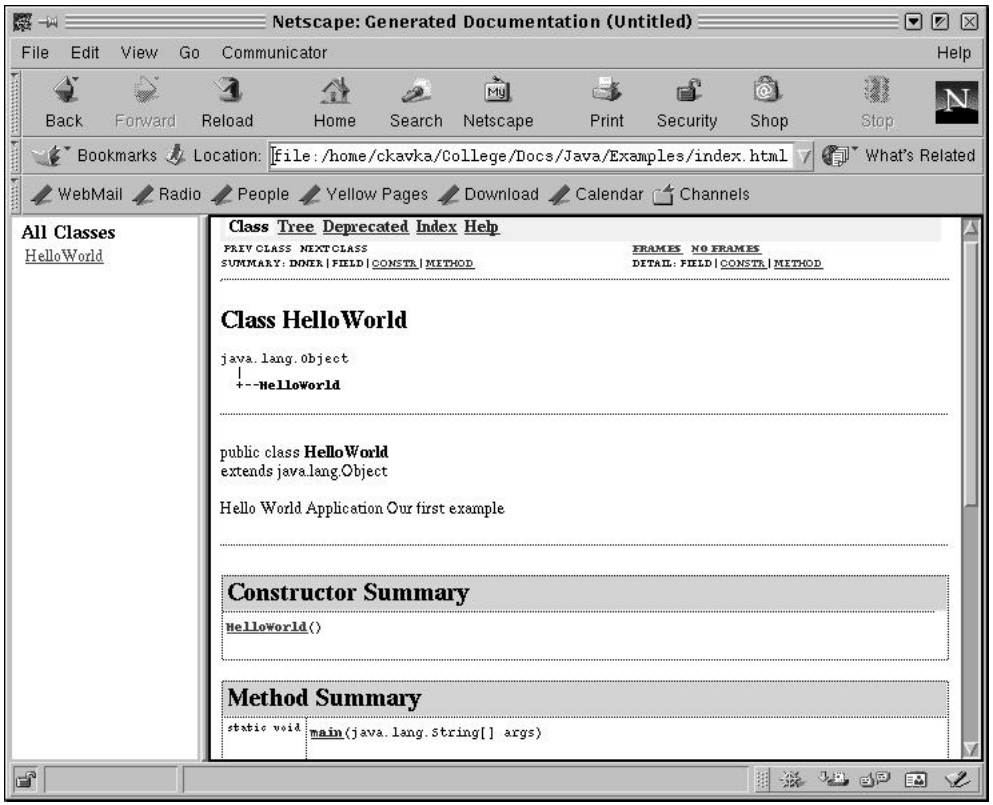
- Execution:

```
# java HelloWorld
Hello World!
```

Documentation

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

```
# javadoc HelloWorld
```



Basic types

- Java provides ten primitive types:
  - four types of integers
  - two types of floating point numbers
  - characters
  - booleans
  - the special type void
  - strings.

Variables

- The variables can be declared specifying its type and name. They can be initialized in the point of declaration, or a value can be assigned later with the assignment expression, as it is shown below:

```
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, as shows the next example:

```
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. In order to specify a float value, we have to add the letter F at the end, as it is shown below:

```
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 the exception that unicode values can be specified with \u:

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

- The boolean values are true and false. They are the only values that can be assigned to boolean variables:

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

Constants

- The declaration for constants is very similar to the declaration of variables. It has to include the word final in front. The specification of the initial value is compulsory, as it is shown in the examples below:

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

- Of course, their values cannot be modified.

Expressions

- Java provides a rich set of operators in order to use them in expressions.
- Expressions can be classified as:
  - arithmetic
  - bit level
  - relational
  - logical
  - specific for strings

Arithmetic operators

- 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);
  }
}
```

The output produced by the execution of the application is:

```
24
2
5.0669355
```

Arithmetic operators

- The shorthand operators composed of the assignment operator and a binary operator are also present.

```
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);
  }
}
```

The output produced by the execution of the application is:

```
17
34
```

Arithmetic operators

- The pre and post increment and decrement operators are also provided:

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

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

The output produced by the execution of the application is:

```
12
13
13
13
```

Relational operators

- Java provides the standard set of relational operators:
  - ⋆ equivalent (==)
  - ⋆ not equivalent (!=)
  - ⋆ less than (<)
  - ⋆ greater than (>)
  - ⋆ less than or equal (<=)
  - ⋆ greater than or equal (>=).
- The relational expressions always return a boolean value.

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);
  }
}
```

The output produced by the execution of the application is:

```
true
false
true
```

Bit level operators

- Java provides a set of operators that can manipulate bits directly.
- Some operators do boolean algebra on bits:
  - ⋆ and (&)
  - ⋆ or (|)
  - ⋆ not(~)
- and others perform bits shifting:
  - ⋆ shift left (<<)
  - ⋆ shift right with sign extension (>>)
  - ⋆ shift right with zero extension (>>>).
- This operators operate on integral types. If the argument is a char, short or byte, it is promoted to int 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);    // 11111111111111111111111111111000
 }
}
```

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 logical operators:
  - ⋆ and (&&)
  - ⋆ or (||)
  - ⋆ not (!).
- 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);
  }
}
```

The output produced by the execution of the application is:

```
true
false
true
```

String operators

- Java provides a complete set of operations on Strings.
- We will consider now just the concatenation operator (+).
- This operator combines two strings, and produces a new one with the characters from both arguments.
- If the expression begins with a string and uses the + operator, then the next argument is converted to a string.

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);
  }
}
```

The output produced by the execution of the application is:

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

Casting

- Java performs a automatic type conversion in the values when there is no risk for data to be lost. This is the usual case for widening conversions:

```
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 (narrowing conversions) 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 as C.
- The main difference is that the value used in the conditional expressions must be a boolean value, and cannot be an integer.

The if control statement

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

The output produced by the execution of the application is:

```
letter: x
```

The while control statement

```
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("It is necessary to increment it "
                       + counter + " times");
  }
}
```

The output produced by the execution of the application is:

```
It is necessary to increment it 22 times
```

The for control statement

```
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("It is necessary to increment it "
                       + counter + " times");
  }
}
```

The output produced by the execution of the application is:

```
It is necessary to increment it 22 times
```

Break and 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.");
  }
}
```

The output produced by the execution of the application is:

```
0    2    3    6    done.
```

The switch control statement

```
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;
```

The switch control statement

```
        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);
  }
}
```

Arrays

- In Java it is possible to declare arrays that 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
```

- The declaration does not specify a size for the arrays. In fact, the declaration does not even allocate space for them.

- The size can be specified by initializing the arrays in the declaration:

```
int[] a = {13,56,2034,4,55};              // size: 5
float[] b = {1.23F,2.1};                  // 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. In this case, the size of the array can be computed even at execution time:

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

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

- The components of the array are initialized with default values: 0 for numeric type elements, '\0' for characters and null for references.

Arrays

- The array can be accessed by using an integer index that can take values from 0 the the size of the array 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 arrays.

```
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 floats with this size
    float[] d = new float[sum];
    // assign some values
    for(int i = 0;i < d.length;i++)
      d[i] = 1.0F / i;

    // print the values in odd positions
    for(int i = 1;i < d.length;i += 2)
      System.out.println("d[" + i + "]=" + d[i]);
  }
}
```

Arrays

The output produced by the execution of the application is:

```
d[1]=1.0
d[3]=0.33333334
d[5]=0.2
d[7]=0.14285715
d[9]=0.11111111
```

Arrays

- It is also possible to declare multidimensional arrays with a similar approach.
- As an example, the following line declares a matrix of integers that can be used to store 50 elements, organized in 10 rows of 5 columns.

```
int a[][] = new int[10][5];
```

Command line arguments

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

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

- It takes one argument that is defined as an array of strings. Through this array, the program can get command line arguments, typed when the program is submitted to the java virtual machine for execution.

Command line arguments

```
class CommandArguments {

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

Sample executions of the application follows:

```
# java CommandArguments Hello World
Hello
World
# java CommandArguments
# java CommandArguments I have 25 cents
I
have
25
cents
```

Command line arguments

- Even if, in the last example, the argument 25 is an integer, it is considered as the string "25", which is stored in args[2].

- It is possible to convert a string that contains a valid integer into an int value by using the class method parseInt that belongs to the class Integer.

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);
  }
}
```

Sample executions of the application follows:

```
# java Add 2 4
6
# java Add 4
Error
# java Add 33 22
55
```

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, as follows:

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

- or in two steps, with exactly the same meaning:

```
Book b3;

b3 = new Book();
```

Classes

- Inside a class it is possible to define:
  - ⋆ data members, usually called fields
  - ⋆ member functions, usually called methods
- We can then add data members to the class as follows:

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

- The fields can be accessed with the dot notation, which consists of the use of a dot (.) between the name of the instance and the name of the field we want to access.

Classes

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

The output produced by the execution of the application is:

```
Thinking in Java : Bruce Eckel : 1129
```

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

Constructors

The output produced by the execution of this application is:

```
Thinking in Java : Bruce Eckel : 1129
```

Constructors

- Java provides a default constructor for the classes. This is the one it was called in the example ExampleBooks before, without arguments:

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

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

- 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.

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;
  }
}
```

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);
  }
}
```

The output of the execution of the application is:

```
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 also.

Methods

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

  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());
  }
}
```

The output of the execution of the application is:

```
Initials: B.E.
```

Methods

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

    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",
                    "Elliotte Rusty Harold",649);
    for(int i = 0;i < a.length;i++)
      System.out.println("Initials: " + a[i].getInitials());
  }
}
```

The output of the execution of the application is:

```
Initials: B.E.
Initials: D.F.
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");
  }
}
```

The output of the execution of the application is:

```
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");
  }
}
```

The output of the execution of the application is:

```
The two books are the same
```

Equality and equivalence

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

  Book(String tit,String aut,int num) { ... }
  Book(String tit,String aut,int num,String isbn) { ... }
  public String getInitials() { ... }

  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");
  }
}
```

The output of the execution of the application is:

```
The two books are the same
```

Static data members

- Static data members (or 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.

Static data members

```
class Book {
  String title;
  String author;
  int numberOfPages;
  String ISBN;
  static String owner;
  Book(String tit,String aut,int num) { ...  }
  Book(String tit,String aut,int num,String isbn) { ...  }
  public String getInitials() { ... }
  public boolean equals(Book b) { ... }

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

  public String getOwner() {
    return owner;
  }
}
```

Static data members

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

The output of the execution of the application is:

```
Carlos Kavka
Carlos Kavka
```

Static data 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.

Static data methods

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

  Book(String tit,String aut,int num) { ... }
  Book(String tit,String aut,int num,String isbn) { ... }
  public String getInitials() { ... }
  public boolean equals(Book b) { ... }
  public void setOwner(String name) { ... }
  public String getOwner() { ... }
  public static String description() {
    return "Book instances can store information on books";
  }
}
```

Static data 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());
  }
}
```

The output of the execution of the application is:

```
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);
  }
}
```

The output of the execution of the application is:

```
165
```

Data members initialization

- All data members in an object are guaranteed to have an initial value.
- There exists a default value for all primitive types:
  - byte : 0
  - short : 0
  - int :0
  - long : 0
  - float : 0.0F
  - double : 0.0
  - char : '\0'
  - boolean: false
  - references: null

Data members 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);
  }
}
```

The output of the execution of the application is:

```
0   0.0   null   null
```

Data members 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);
    System.out.println(v.f);
    System.out.println(v.s);
    System.out.println(v.b);
  }
}
```

Data members initialization

The output of the execution of the application is:

```
2
0.5
hello
null
```

The 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;
}
```

- With this definition of the method, it can be used as follows:

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

The output of the execution of this example is:

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

The keyword this

For example, in the definition of the class Book there were 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 one can be defined in a shorter way by calling the first constructor:

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

An example: the complex number 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("a+b = " + c.getReal() + " " +
                       c.getImaginary());

    Complex d = c.sub(a);
    System.out.println("c+d = " + d.getReal() + " " +
                       d.getImaginary());
  }
}
```

The output of the execution of this application should be something like:

```
a+b = 4.51 7.38       c+d = 3.18 2.74
```

An example: the complex number class

```
public 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;
  }
}
```

An example: the complex number class

```
/** This method returns a new complex number wich is
 *  the result of the addition of the receptor and the
 *  complex number passed as argument
 */

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

/** This method returns a new complex number wich is
 *  the result of the substraction of the receptor and the
 *  complex number passed as argument
 */

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

An example: the complex number class

- Let's suppose we want to define a method addReal that increments just the real part of the receptor of the message with the value passed as argument.

```
a.addReal(2.0);
a.addReal(3.0).addReal(3.23);
```

This can be done as follows:

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

An example: the complex number class

- We must be careful if we want to create one complex number as a copy of the other, since the next assignment expression will not do it:

```
Complex e = a;
```

- This will make just e to be a reference to the same object referenced by a. This means that if we increment e, then a will be incremented also.

An example: the complex number class

- In order to create a new complex number, we should use a constructor, as follows:

```
Complex e = new Complex(a);
```

- It is necessary then to define a constructor that takes one complex number as argument.

- An interesting way to define it follows:

```
/** This constructor creates a complex number as a copy
 *  of the complex number passed as argument
 */

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

Inheritance

- Inheritance allows to define new classes by reusing other classes.
- It is possible to define a new class (called subclass) by saying that the class must be "like" other class (called base class) by using the word "extends" followed by the name of the base class.
- The definition of the new class specifies the differences with the base class.

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

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;
  }
}
```

A scientific book can be defined as follows:

```
ScientificBook sb;

sb = new ScientificBook("Neural Networks, A Comprehensive
         Foundation","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.
- However, methods defined above in this hierarchy can also be called.
- 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 of classes till it is found.

Inheritance (Methods)

- Some methods can be reused:

```
ScientificBook sb;

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

System.out.println(sb.getInitials());
```

The output of the execution of this example is:

```
S.H.
```

Inheritance (Methods)

- New methods that calls methods in the base class can be defined:

```
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 (Methods)

- Or it can be defined in this way:

```
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 (title.equals(b.title) && author.equals(b.author) &&
            numberOfPages == b.numberOfPages &&
            ISBN.equals(b.ISBN) && area.equals(b.area) &&
            proceeding == b.proceeding;
  }
}
```

Inheritance (Methods)

- Methods that overrides methods defined in the base class can also be defined:

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

Inheritance (Methods)

- New methods can also be defined:

```
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() { ... }

  public void setProceeding() {
    proceeding = true;
  }

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

Inheritance

```
class TestScientificBooks {
  public static void main(String[] args) {
   ScientificBook sb1,sb2;
   sb1 = new ScientificBook("Neural Networks, A Comprehensive"+
              " Foundation","Simon Haykin",696,"0-02-352761-7",
              "Artificial Intelligence");
   sb2 = new ScientificBook("Neural Networks, A Comprehensive"+
              " Foundation","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());
  }
}
```

The output of the execution of this example is:

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

Instanceof and getClass methods

```
class TestClass {
  public static void main(String[] args) {
   Book b1 = new Book("Thinking in Java","Bruce Eckel",1129);
   ScientificBook sb1 = new ScientificBook("Neural Networks, "+
       "A Comprehensive Foundation","Simon Haykin",696,"0-02-352761-7",
       "Artificial Intelligence");

   System.out.println(b1.getClass());
   System.out.println(sb1.getClass());
   System.out.println(b1 instanceof Book);
   System.out.println(sb1 instanceof Book);
   System.out.println(b1 instanceof ScientificBook);
   System.out.println(sb1 instanceof ScientificBook);
  }
}
```

The output of the execution of this example is:

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

Packages

- A package is a structure in which classes can be organized.
- A package can contain any number of classes, usually related by purpose or by inheritance.
- The standard classes in the system are organized in packages.

Packages

```
/**
 * Test Date Class
 */

import java.util.*;

class TestDate {

  public static void main(String[] args) {

    System.out.println(new Date());
  }
}
```

The output of the application (when I was executing it) was:

```
Thu Nov 08 19:14:18 GMT-02:00 2001
```

Packages

- New packages can be defined by using the statement package with the name of the package we are going to define as argument:

```
package mypackage;
```

- They can be imported by other classes with the import statement.

- There is a name convention, that we will not be covering here, in order to define packages that can be shared with the Java community.

Access control

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

Access control

- There exists a default access which is the one we have been using in most examples, that 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 from the class TestScientificBook as follows:

```
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. In this case we can use the modifier private.

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

- In this case, the direct access to the data member proceeding is not allowed from other classes, and the condition of a scientific book to be a proceeding can only be asserted by sending the message setProceeding:

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

- The same applies to methods: A private method can only be called from other methods in its own class.

Access control

- The public modifier allows full access from all other classes without restrictions. This is the usual way in which methods are defined so the messages they implement can be sent to objects of its class from all other classes.

- The protected modifier allows access to data members and methods from subclasses and from all classes in the same package.

Final and abstract

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

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

```
class TestBoards1 {

  public static void main(String[] args) {

    IOSerialBoard serial = new IOSerialBoard("my first port",
                                             0x2f8);
    serial.initialize();
    serial.read();
    serial.close();
  }
}
```

The output of the execution of this application is:

```
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 IOTcpIpBoard("my second port",0x3ef8dda8);
    board[2] = new IOTcpIpBoard("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 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.
- As an example:

```
/**
 * IO board interface
 */

interface IOBoardInterface {

  public void initialize();
  public void read();
  public void write();
  public void close();
}
```

Interfaces

```
class IOSerialBoard2 implements IOBoardInterface {
  int port;
  IOSerialBoard2(int p) {
    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

- The class can be used as follows:

```
class TestBoards3 {

  public static void main(String[] args) {

    IOSerialBoard2 serial = new IOSerialBoard2(0x2f8);

    serial.initialize();
    serial.read();
    serial.close();
  }
}
```

Interfaces

- The following interface defines how classes with *nice behaviour* must behave:

```
interface NiceBehavior {

  public String getName();
  public String getGreeting();
  public void sayGoodBye();
}
```

- A class that expresses its desire to be *nice* can be defined as follows:

```
class IOSerialBoard3 implements IOBoardInterface,
                                 NiceBehavior {
  ...
}
```

Exceptions

- The usual behavior when there is a runtime error in an application is to abort the execution.

```
class TestExceptions1 {

  public static void main(String[] args) {
    String s = "Hello";

    System.out.print(s.charAt(10));
  }
}
```

- The execution stops with the following message:

```
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

- This error, or exception in Java terminology, can be caught and some processing can be done:

```
class TestExceptions2 {

  public static void main(String[] args) {
    String s = "Hello";

    try {
      System.out.println(s.charAt(10));
    } catch (Exception e) {
      System.out.println("No such position");
    }
  }
}
```

The output of the execution of the application is:

```
No such position
```

Exceptions

- If we are interested just in process the exception for index out of bounds for strings, we can do it in this way:

```
class TestExceptions3 {

  public static void main(String[] args) {
    String s = "Hello";

    try {
      System.out.println(s.charAt(10));
    } catch (StringIndexOutOfBoundsException e) {
      System.out.println("No such position");
    }
  }
}
```

The output of the execution of the application is:

```
No such position
```

Exceptions

- There exists messages that can be sent to an exception object.

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

The output of the execution of the application is:

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

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

- The input output system in Java is rather complicated.
- One advantage is the fact that input output from files, devices, memory or web sites is performed in the same way.
- It is based on the idea of streams:
  - ⋆ A 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.

Byte oriented streams (writing)

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

Byte oriented streams (reading)

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

The output of the execution of the application is:

```
10    20    30    40    255
```

Byte oriented streams (writing)

```
import java.io.*;

class WriteArrayBytes {

  public static void main(String[] args) {
    byte data[] = { 10,20,30,40,50 };

    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());
    }
  }
}
```

Buffered byte oriented streams (writing)

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

Buffered byte oriented streams (reading)

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

The output of the execution of the application is:

```
10    20    30    40    255
```

Data buffered byte oriented streams

- A data buffered byte oriented stream can be used to work with data in small pieces corresponding to the primitive types.

- The following messages can be used to read and write data:
  - readBoolean() writeBoolean(boolean)
  - readByte () writeByte(byte)
  - readShort() writeShort(short)
  - readInt() writeInt(int)
  - readLong() writeLong(long)
  - readFloat() writeFloat(float)
  - readDouble() writeDouble(double)

Data buffered byte oriented streams (writing)

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

Data buffered byte oriented streams (reading)

```
class ReadData {
  public static void main(String[] args) {
    FileInputStream f;
    BufferedInputStream bf;
    DataInputStream 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());
    }
  }
}
```

Data buffered byte oriented streams (reading)

The output of the execution of the application is:

```
10.3
20.65
8.45
-4.12
true
```

Character oriented streams

- 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()

Character oriented streams (writing)

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

Character oriented streams (reading)

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

Standard Input

- Standard input can be read by using an input buffered stream:

```
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()).length() != 0)
        System.out.println(line);
    } catch(IOException e) {
        System.out.println("Error in standard input");
    }
  }
}
```

Standard input

- The input output exception can be thrown at method level:

```
class StandardInputWithThrows {

  public static void main(String[] args) throws IOException {
    InputStreamReader isr;
    BufferedReader br;

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

    String line;
    while ((line = br.readLine()).length() != 0)
      System.out.println(line);
  }
}
```

Threads

- It is possible to run concurrently different tasks called threads.
- Each thread can be seen as an independently running task.
- The threads can communicate between themselves and 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();
  }
}
```

The output of the execution of the application is:

```
a
b
a
b
a
...
```

The Producer and Consumer example

```
/**
 * Producer Consumer class Application
 */

class ProducerConsumer {

  public static void main(String[] args) {

    Buffer buffer = new Buffer(20);

    Producer prod = new Producer(buffer);
    Consumer cons = new Consumer(buffer);

    prod.start();
    cons.start();
  }
}
```

The Producer and Consumer example

```
/**
 * Producer class Application
 */
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;
    }
  }
}
```

The Producer and Consumer example

```
/**
 * Consumer class Application
 */

class Consumer extends Thread {
  Buffer buffer;

  public Consumer(Buffer b) {
    buffer = b;
  }

  public void run() {

    while(true) {
      System.out.println(buffer.delete());
    }
  }
}
```

The Producer and Consumer example

```
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;
  }
}
```

The Producer and Consumer example

- This implementation does not work for two reasons:
  - ⋆ 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, and the method delete does not check if there is at least one piece of data available in the buffer.

Synchronized methods

- The synchronized methods are not allowed to be executed concurrently on the same instance.
- Each instance has a lock, that is used to synchronize the access.

Synchronized methods

- The solution to the first problem is to define the methods as follows:

```
public synchronized void insert(double element) {

  buffer[tail] = element;
  tail = (tail + 1) % size;
  numElements++;
}

public synchronized double delete() {

  double value = buffer[head];
  head = (head + 1) % size;
  numElements--;
  return value;
}
```

Wait and notify

- Subclasses of Thread can send the messages wait and notify only from synchronized methods.
- The message wait puts the calling thread to sleep, releasing the lock.
- The message notify awakens a waiting thread on the corresponding lock.

Wait and notify

```
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();
}
```

Wait and notify

```
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 ProducerConsumer, four class files were generated, as the following command shows:

```
# 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

- Java provides a mechanism to pack and compress files into one file, in order to make the process of distribution of applications easier. This compressed file is called a JAR (Java ARchive) file.
- A JAR 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. The manifest file contains information on the files included in the JAR file.
- 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 a JAR file for this application 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
```

JAR files

- It is possible to see the contents of the JAR file just created by using the option t 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. Its content is:

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

JAR files

- The application included in the JAR file can be executed as follows:

```
# java -jar ProducerConsumer.java
```