Principi della programmazione ad oggetti

Gli oggetti sono entità dinamiche, generate e distrutte nel corso dell'esecuzione del programma.

Gli oggetti sono definiti mediante *classi*. Possiedono dati (*attributi*) ed eseguono operazioni (*metodi*) su richiesta di altri oggetti.

Interazione tra due oggetti: nello svolgimento di una sua operazione un oggetto può richiedere l'esecuzione di un'operazione (con eventuali parametri) da parte di un altro oggetto; il chiamante aspetta la conclusione.

Un oggetto conosce un altro oggetto o perché gli viene passato come parametro di un'operazione o perché ha un'associazione con esso.

Si possono stabilire *associazioni* (legami) tra oggetti. Si dice *navigazione* il muoversi da un oggetto ad altri oggetti seguendo le loro associazioni.

classe Punto

```
public class Punto {
      int x = 0; // attributi
      int y = 0;
      public Punto (int a, int b) { // costruttore
            x = a;
            y = b;
      }
      public void sposta (int a, int b) { // metodo
            x = a;
            y = b;
      }
```

classe Rettangolo

```
public class Rettangolo {
      int larghezza;  // attributi
      int altezza;
      Punto origine; // associazione con il punto origine
      public Rettangolo (int x, int y, int l, int h) {
            origine = new Punto(x, y);
            larghezza  = l;
            altezza = h;
      }
      public void sposta (int a, int b) { // metodo
            origine.sposta(a, b); // interazione tra oggetti
      }
      public void print() {
            System.out.println("r: x=" + origine.x + " y="
            + origine.y + " l=" + larghezza + " a=" + altezza); }}
```

main

```
public class Main {
      public static void main(String[] args) {
            Rettangolo r = new Rettangolo(100, 200, 10, 20);
            r.print();
            r.sposta(1000,2000);
            r.print();
      }
}
```

Risultato

```
r: x=100 y=200 l=10 a=20
r: x=1000 y=2000 l=10 a=20
```

Classification of OO languages (Wegner)

- **Object-Based** (Ada). The language has specific constructs to manage objects

- **Class-Based** (CLU). + each object belongs to a class

- **Object-Oriented** (Simula, Smalltalk). + classes support inheritance

- **Strongly-Typed Object-Oriented** (C++, Java, C#). + the language is strongly typed

Java application

Using a text editor, create a file named *HelloWorldApp*.java with the following Java code:

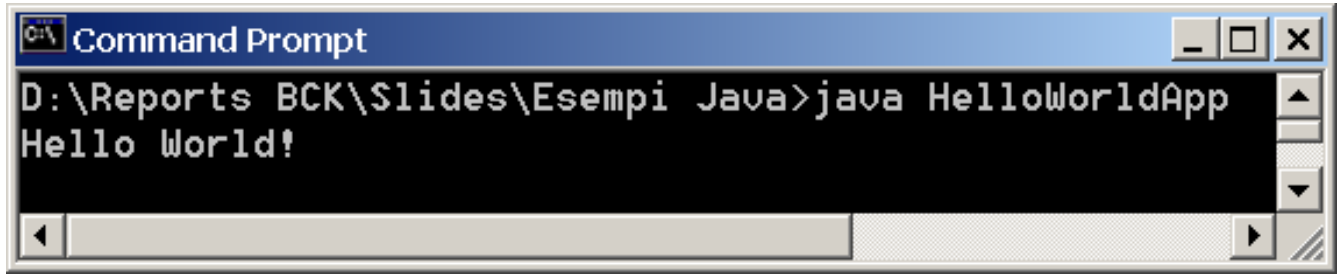
```
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //Display the string.
    }
}
```

Java application

Compile the source file using the Java compiler.

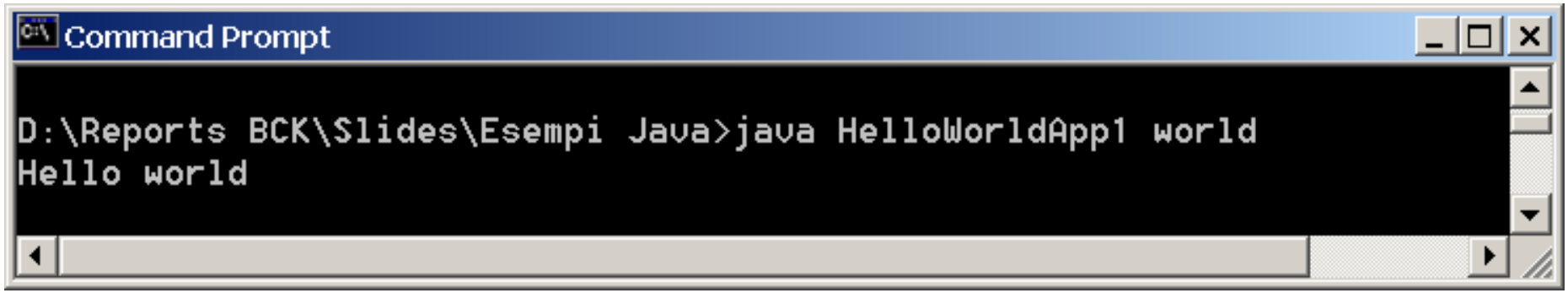
If the compilation succeeds, the compiler creates a file named HelloWorldApp.**class** in the same directory (folder) as the Java source file (HelloWorldApp.java). This class file contains Java bytecodes, which are platform-independent codes interpreted by the Java runtime system.

Run the program using the Java interpreter.



Java application

```
class HelloWorldApp1 {
    public static void main(String[] args) {
        System.out.println("Hello " + args[0]);
  }
}
```



```
int anInt = 4;
```

Literal	**Data Type**
178	int
8864L	long
37.266	double
37.266D	double
87.363F	float
26.77e3	double
'c'	char
true	boolean
false	boolean

Arithmetic operators

Operator	Use	Description
+	op1 + op2	Adds op1 and op2
-	op1 - op2	Subtracts op2 from op1
*	op1 * op2	Multiplies op1 by op2
/	op1 / op2	Divides op1 by op2
%	op1 % op2	Computes the remainder of dividing op1 by op2
++	op++	Increments op by 1; evaluates to the value of op before it was incremented
++	++op	Increments op by 1; evaluates to the value of op after it was incremented
--	op--	Decrements op by 1; evaluates to the value of op before it was decremented
--	--op	Decrements op by 1; evaluates to the value of op after it was decremented

Relational operators

Operator	**Use**	**Returns `true if`**
`>`	`op1 > op2`	`op1 is greater than op2`
`>=`	`op1 >= op2`	`op1 is greater than or equal to op2`
`<`	`op1 < op2`	`op1 is less than op2`
`<=`	`op1 <= op2`	`op1 is less than or equal to op2`
`==`	`op1 == op2`	`op1 and op2 are equal`
`!=`	`op1 != op2`	`op1 and op2 are not equal`

You can use the following conditional operators to form multi-part decisions.

Operator	Use	Returns true if
&&	op1 && op2	op1 and op2 are both true, conditionally evaluates op2
||	op1 || op2	either op1 or op2 is true, conditionally evaluates op2
!	! op	op is false
&	op1 & op2	op1 and op2 are both true, always evaluates op1 and op2
|	op1 | op2	either op1 or op2 is true, always evaluates op1 and op2
^	op1 ^ op2	if op1 and op2 are different--that is if one or the other of the operands is true but not both

Operator	Use	Operation
>>	op1 >> op2	shift bits of op1 right by distance op2
<<	op1 << op2	shift bits of op1 left by distance op2
>>>	op1 >>> op2	shift bits of op1 right by distance op2 (unsigned)

Operator	Use	Operation
&	op1 & op2	bitwise and
|	op1 | op2	bitwise or
^	op1 ^ op2	bitwise xor
~	~op2	bitwise complement

Package

```
package g;
public class Punto {
        int x = 0; // attributi
        int y = 0;
...
```

```
package g;
public class Rettangolo {
        int larghezza;
        int altezza;
...
```

```
import g.*;
public class Main {
        public static void main(String[] args) {
                Rettangolo r = new Rettangolo(100, 200, 10, 20);
                r.print();
                r.sposta(1000,2000);
                r.print();
        }}
```

Stringhe

classi String e StringBuffer

Classes String and StringBuffer

```
public class Stringhe {
public static String reverseIt(String source) {
       int i, len = source.length();
       StringBuffer dest = new StringBuffer(len);
       for (i = (len - 1); i >= 0; i--) {
              dest.append(source.charAt(i));
       }
       return dest.toString();
}
public static void main(String[] args) {
       System.out.println(reverseIt("alfabeto")); // otebafla
}
}
```

Class String

Strings are constant; their values cannot be changed after they are created.

```
String c = "abc".substring(2,3);
```

```
public char charAt(int index)
```

```
public boolean equalsIgnoreCase(String anotherString)
```

```
public int compareTo(String anotherString)
```

```
public boolean startsWith(String prefix, int toffset)
```

```
public int indexOf(int ch)
```

```
public String substring(int beginIndex, int endIndex)
```

```
public int length()
```

public final class **String**

extends Object

implements Serializable

The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class. Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

String str = "abc";

public **String**(String value)

Allocates a new string that contains the same sequence of characters as the string argument.

public **String**(char value[])

Allocates a new String so that it represents the sequence of characters currently contained in the character array argument.

public **String**(StringBuffer buffer)

Allocates a new string that contains the sequence of characters currently contained in the string buffer argument.

public int **length**()

Returns the length of this string. The length is equal to the number of 16-bit Unicode characters in the string.

public char **charAt**(int index)

Returns the character at the specified index. An index ranges from 0 to length() - 1.

public boolean **equals**(Object anObject)

Compares this string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.