

# *Programmazione ad oggetti*

*Giorgio Bruno*

*Dip. Automatica e Informatica  
Politecnico di Torino*

*Tel. 011 5647003, email: [giorgio.bruno@polito.it](mailto:giorgio.bruno@polito.it)*

***a.a. 2006-2007***

Java opens a lot of doors. No development environment has ever combined so many good things in one package. Java is

- Object-oriented
- Safe
- Robust
- Secure
- Network-enabled
- Graphical
- Multithreaded
- Distributed
- Platform independent

<http://java.sun.com/developer/onlineTraining/index.html>



# *Argomenti*

Caratteristiche di base

Inheritance, interfacce, JDK

Collezioni

Gestione eccezioni

I/O e file

Grafica, swing

Applet e thread

Esempi

Testo consigliato:

K. Arnold, J. Gosling, D. Holmes,  
Il linguaggio Java – Manuale  
ufficiale. Pearson, 2006.

# *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;  
    }  
}
```

```
public class Rettangolo {  
    int larghezza; // attributi  
    int altezza;
```

*classe Rettangolo*

```
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:

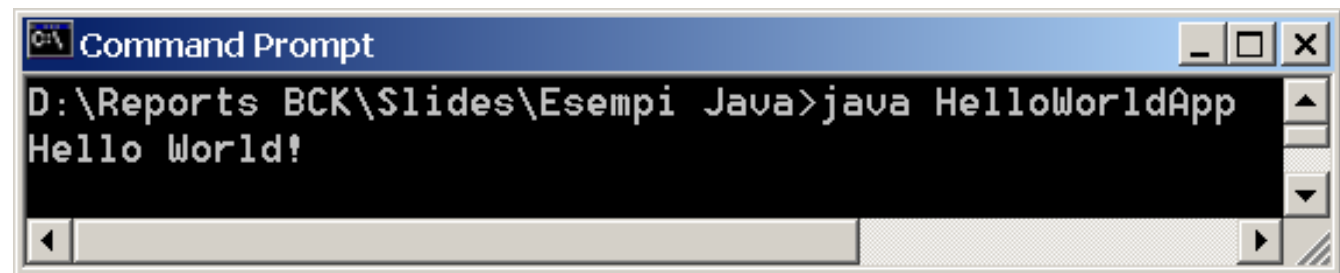
```
class HelloWorldApp {  
    public static void main(String[] args) {  
        System.out.println("Hello World!"); //Display the string.  
    }  
}
```

# *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.

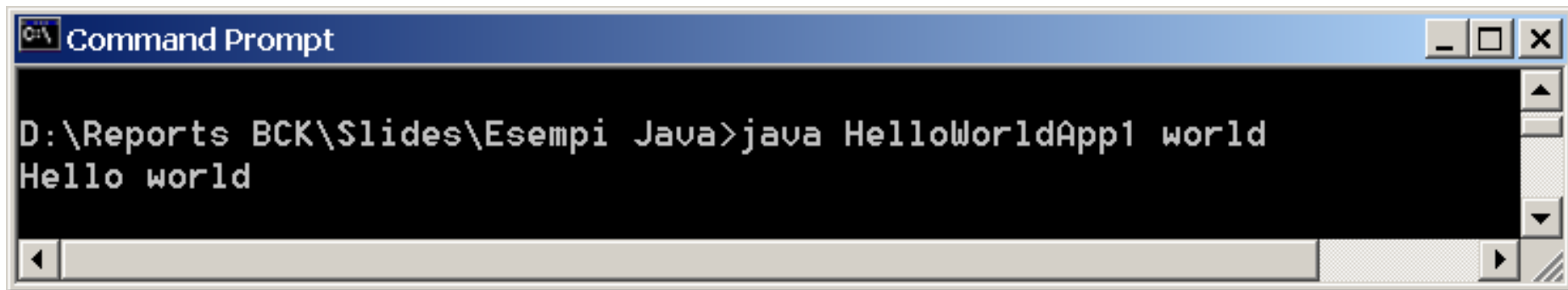
Run the program using the Java interpreter.

A screenshot of a Windows Command Prompt window. The title bar is blue and says "Command Prompt". The window has standard Windows window controls (minimize, maximize, close) on the right. The command prompt shows the directory path "D:\Reports BCK\Slides\Esempi Java" followed by the command "java HelloWorldApp". The output of the command is "Hello World!". There is a scroll bar at the bottom of the window.

```
Command Prompt
D:\Reports BCK\Slides\Esempi Java>java HelloWorldApp
Hello World!
```

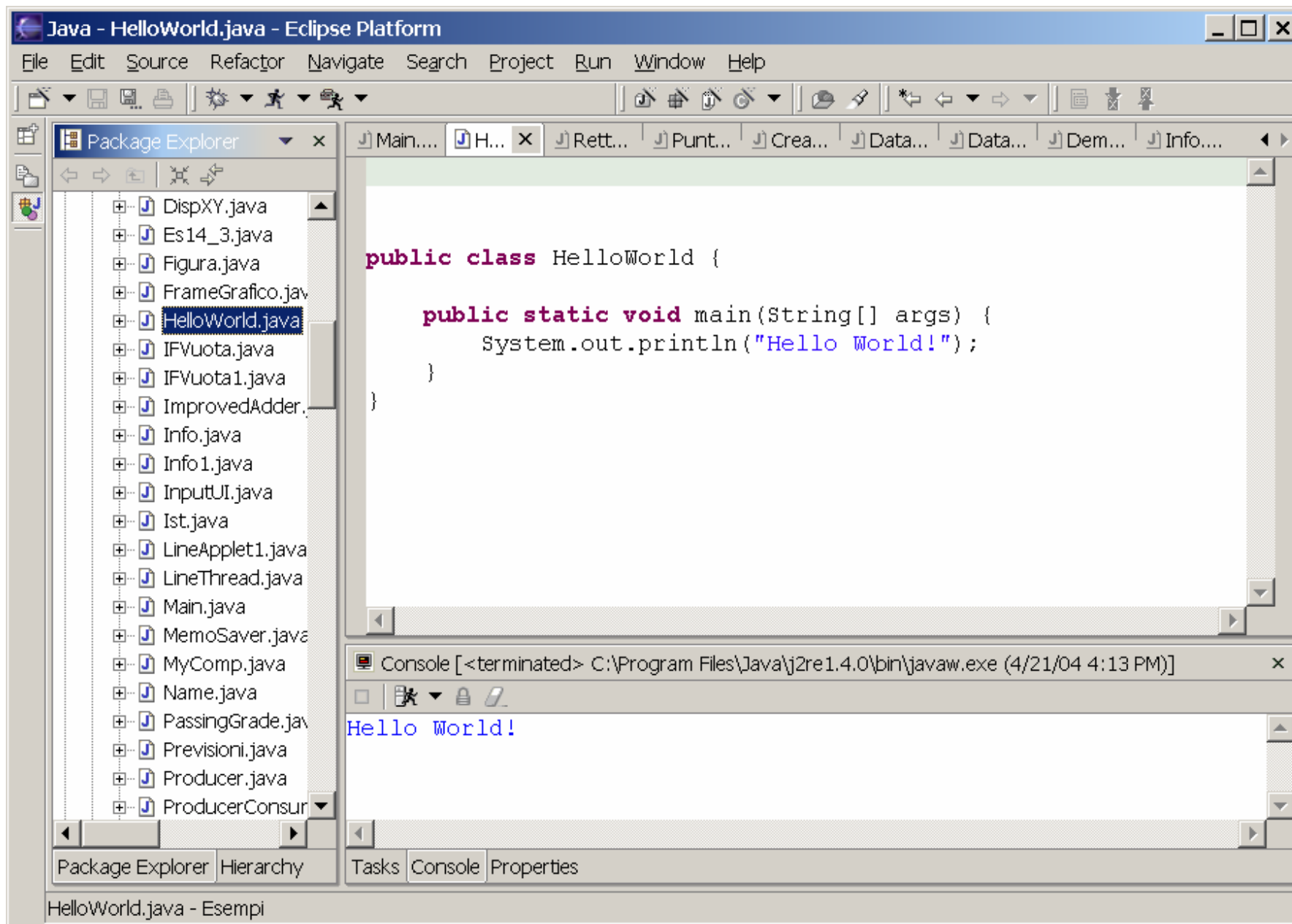
# *Java application*

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

A screenshot of a Windows Command Prompt window. The title bar is blue and reads "Command Prompt". The window has standard Windows window controls (minimize, maximize, close) on the right. The command prompt shows the directory "D:\Reports BCK\Slides\Esempi Java" and the command "java HelloWorldApp1 world" being executed. The output "Hello world" is displayed on the line below the command. A horizontal scrollbar is visible at the bottom of the window.

```
D:\Reports BCK\Slides\Esempi Java>java HelloWorldApp1 world  
Hello world
```

# Eclipse



The BasicsDemo program that follows adds the numbers from 1 to 10 and displays the result.

```
public class BasicsDemo {  
    public static void main(String[] args) {  
        int sum = 0;  
        for (int current = 1; current <= 10; current++) {  
            sum += current;  
        }  
        System.out.println("Sum = " + sum);  
    }  
}
```

The output from this program is:

Sum = 55

# *Variables*

When you declare a variable, you explicitly set the variable's name and data type. The Java programming language has two categories of data types: **primitive** and **reference**. A variable of primitive type contains a value. This table shows all of the primitive data types along with their sizes and formats:

<b>Keyword</b>	<b>Description</b>	<b>Size/Format</b>
<i>(integers)</i>		
byte	Byte-length integer	8-bit two's complement
short	Short integer	16-bit two's complement
int	Integer	32-bit two's complement
long	Long integer	64-bit two's complement
<i>(real numbers)</i>		
float	Single-precision floating point	32-bit IEEE 754
double	Double-precision floating point	64-bit IEEE 754
<i>(other types)</i>		
char	A single character	16-bit Unicode character
boolean	A boolean value (true or false)	true or false

- The location of a variable declaration implicitly sets the variable's scope, which determines what section of code may refer to the variable by its simple name. There are four categories of scope: *member variable scope*, *local variable scope*, *parameter scope*, and *exception-handler parameter scope*.
- You can provide an initial value for a variable within its declaration by using the assignment operator (=).
- You can declare a variable as **final**. The value of a final variable cannot change after it's been initialized.



```
int anInt = 4;
```

<b>Literal</b>	<b>Data Type</b>
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*

<b>Operator</b>	<b>Use</b>	<b>Returns true if</b>
>	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.

<b>Operator</b>	<b>Use</b>	<b>Returns true if</b>
&&	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

<b>Operator</b>	<b>Use</b>	<b>Operation</b>
>>	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)
<b>Operator</b>	<b>Use</b>	<b>Operation</b>
&	op1 & op2	bitwise and
	op1   op2	bitwise or
^	op1 ^ op2	bitwise xor
~	~op2	bitwise complement

Operator	Use	Equivalent to	<i>Assignment</i>
+=	op1 += op2	op1 = op1 + op2	
-=	op1 -= op2	op1 = op1 - op2	
*=	op1 *= op2	op1 = op1 * op2	
/=	op1 /= op2	op1 = op1 / op2	
%=	op1 %= op2	op1 = op1 % op2	
&=	op1 &= op2	op1 = op1 & op2	
=	op1  = op2	op1 = op1   op2	
^=	op1 ^= op2	op1 = op1 ^ op2	
<<=	op1 <<= op2	op1 = op1 << op2	
>>=	op1 >>= op2	op1 = op1 >> op2	
>>>=	op1 >>>= op2	op1 = op1 >>> op2	

?:	op1 ? op2 : op3	If op1 is true, returns op2. Otherwise, returns op3.
[]	<i>type</i> []	Declares an array of unknown length, which contains <i>type</i> elements.
[]	<i>type</i> [ op1 ]	Creates an array with op1 elements. Must be used with the new operator.
[]	op1[ op2 ]	Accesses the element at op2 index within the array op1. Indices begin at 0.
.	op1.op2	Is a reference to the op2 member of op1.
()	op1( <i>params</i> )	Declares or calls the method named op1 with the specified parameters.
(type)	(type) op1	Casts (converts) op1 to type.
new	new op1	Creates a new object or array.
<b>instanceof</b>	op1 instanceof op2	Returns true if op1 is an instance of op2

# *Control statements*

```
while (boolean expression) {  
    statement(s)  
}
```

```
do {  
    statement(s)  
} while (expression);
```

```
for (initialization ; termination ; increment) {  
    statement(s)  
}
```



# *Argomenti*

Classi e oggetti, packages, visibilità

Proprietà di classe

Classi String e StringBuffer, classi wrapper, arrays

Inheritance, polimorfismo, dynamic binding

Classi astratte, interfacce

Classe Object, esempi di clonazione, classe Class

Classi annidate, interfaccia Enumeration

Patterns

# *Classi e oggetti*

classi geometriche, Point e Rectangle

classe utente

## *Costruttori*

```
public class Rettangolo {  
    int larghezza; // attributi  
    int altezza;  
    Punto origine;  
    public Rettangolo (int x, int y, int l, int h) {  
        origine = new Punto(x, y); // associazione con il  
                                     // punto origine  
        larghezza  = l;  
        altezza = h;}  
    public Rettangolo(Punto p, int w, int h) {  
        origine = p;  
        larghezza = w;  
        altezza = h;}  
    public Rettangolo(int w, int h) {  
        this(new Punto(0, 0), w, h);}  
}
```

# Garbage collection

An object is eligible for garbage collection when there are **no more references** to that object. References that are held in a variable are usually dropped when the variable goes out of scope. Or, you can explicitly drop an object reference by setting the variable to the special value **null**. Remember that a program can have multiple references to the same object; all references to an object must be dropped before the object is eligible for garbage collection.

The Java runtime environment has a **garbage collector** that periodically frees the memory used by objects that are no longer referenced. The garbage collector does its job automatically, although, in some situations, you may want to run the garbage collection explicitly by calling the **gc** method in the `System` class. For instance, you might want to run the garbage collector after a section of code that creates a large amount of garbage or before a section of code that needs a lot of memory.

Before an object gets garbage-collected, the garbage collector gives the object an opportunity to clean up after itself through a call to the object's `finalize` method. This process is known as *finalization*.

Most programmers don't have to worry about implementing the `finalize` method. In rare cases, however, a programmer might have to implement a `finalize` method to release resources, such as native peers, that aren't under the control of the garbage collector.

# *Packages*

To make classes easier to find and use, to avoid naming conflicts, and to control access, programmers bundle groups of related classes into packages.

**Definition:** A package is a collection of related classes and interfaces that provides access protection and namespace management.

To create a package, you simply put a class or interface in it. To do this, you put a package statement at the top of the source file in which the class or interface is defined. For example, the following code appears in the source file Circle.java and puts the Circle class in the graphics package:

```
package graphics;
```

```
public class Circle extends Graphic implements Draggable {
```

However, if you are trying to use a member from a different package and that package has not been imported, then you must use the member's *long name*, which includes the package name. This is the long name for the Rectangle class declared in the graphics package in the previous example:

```
graphics.Rectangle
```

To import a specific member into the current file, put an import statement at the beginning of your file before any class or interface definitions (but after the package statement, if there is one). Here's how you would import the Circle class from the graphics package created in the previous section:

```
import graphics.Circle;
```

Now you can refer to the Circle class by its short name:

```
Circle myCircle = new Circle();
```

```
import graphics.*;
```

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

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

## *Package*

```
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();  
    }  
}
```

# Controlling Access to Members of a Class

The following chart shows the access level permitted by each specifier.

<b>Specifier</b>	<b>class</b>	<b>subclass</b>	<b>package</b>	<b>world</b>
private	X			
protected	X	X	X	
public	X	X	X	X
(package)	X		X	

Usare metodi **accessor** e **mutator** per leggere o scrivere dati privati.



## *Proprietà di classe*

```
package g;

public class Punto {
    private static int nPunti = 0;
    public static int getNPunti() {return nPunti;}
    public static Punto origine = new Punto(0,0);
    int x = 0; // attributi
    int y = 0;
    public Punto (int a, int b) { // costruttore
        x = a;          y = b;
        nPunti++;
    }
    public void sposta (int a, int b) { // metodo
        x = a; y = b;
    }
}
```

```
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();
        System.out.println(Punto.getNPunti());
    }
}
```

r: x=100 y=200 l=10 a=20

r: x=1000 y=2000 l=10 a=20

2

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")); // otebafle  
    }  
}
```

## *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.

public boolean **equalsIgnoreCase**(String anotherString)

Compares this String to another 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, where case is ignored.

public int **compareTo**(String anotherString)

Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. Returns: the value 0 if the argument string is equal to this string; a value less than 0 if this string is lexicographically less than the string argument; and a value greater than 0 if this string is lexicographically greater than the string argument.

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

Tests if this string starts with the specified prefix.

public int **hashCode**() Returns a hashcode for this string.

public int **indexOf**(int ch)

Returns the index within this string of the first occurrence of the specified character or -1 if the character does not occur.

public int **indexOf**(String str)

Returns the index within this string of the first occurrence of the specified substring.



public String **substring**(int beginIndex)

Returns a new string that is a substring of this string. The substring begins at the specified index and extends to the end of this string.

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

Returns a new string that is a substring of this string. The substring begins at the specified beginIndex and extends to the character at index endIndex - 1. Throws: StringIndexOutOfBoundsException if the beginIndex or the endIndex is out of range.

public String **replace**(char oldChar, char newChar)

Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

public String **toLowerCase**()

Converts this String to lowercase.

public String **trim**()

Removes white space from both ends of this string.

public static String **valueOf**(int i) Returns the string representation of the int argument.

public static String **valueOf**(float f) Returns the string representation of the float argument.

# *Class StringBuffer*

public final class **StringBuffer**

extends Object

implements Serializable

A string buffer implements a mutable sequence of characters. String buffers are safe for use by multiple threads. The methods are synchronized where necessary so that all the operations on any particular instance behave as if they occur in some serial order.

String buffers are used by the compiler to implement the binary string concatenation operator `+`. For example, the code:

`x = "a" + 4 + "c"` is compiled to the equivalent of:

`x = new StringBuffer().append("a").append(4).append("c") .toString()` The principal operations on a **StringBuffer** are the **append** and **insert** methods, which are overloaded so as to accept data of any type.

**public StringBuffer(int length)**

Constructs a string buffer with no characters in it and an initial capacity specified by the **length** argument. Throws: NegativeArraySizeException

```
String palindrome = "Dot saw I was Tod";  
int len = palindrome.length();  
String anotherPalindrome = "Niagara. O roar again!";  
char aChar = anotherPalindrome.charAt(9); //car. O  
String anotherPalindrome = "Niagara. O roar again!";  
String roar = anotherPalindrome.substring(11, 15); //da 11 a 15: roar
```

```
StringBuffer sb = new StringBuffer("Drink Java!");  
sb.insert(6, "Hot ");  
System.out.println(sb.toString());
```

This code snippet prints: Drink Hot Java!

Another useful StringBuffer modifier is **setCharAt**, which replaces the character at a specific location in the StringBuffer with the character specified in the argument list.

You can use **valueOf** to convert variables of different types to Strings. For example, to print the value of pi:

```
System.out.println(String.valueOf(Math.PI));
```

## Converting Strings to Numbers

The `String` class itself does not provide any methods for converting a `String` to a floating point, integer, or other numerical type. However, four of the "type wrapper" classes (`Integer`, `Double`, `Float`, and `Long`) provide a class method named **`valueOf`** that converts a `String` to an object of that type.

```
String piStr = "3.14159";  
Float pi = Float.valueOf(piStr);
```

# *Wrapper Classes*

Boolean, Character, Double, Float, Integer, Long

ClassType(type) - constructor - Character c1 = new Character('x');

type typeValue() - char c2 = c1.charValue();

String toString()

static ClassType valueOf(String s) - genera un oggetto dalla stringa

# *Oggetti wrapper*

```
public class Wrapper {  
  
    public static void main(String[] args) {  
        Integer i0 = new Integer(10);  
        Integer i1 = new Integer("10");  
        Integer i2 = Integer.valueOf("10");  
        int int0 = i0.intValue();  
  
        int int1 = Integer.parseInt("10");  
        String s1 = new String("10");  
        String s2 = String.valueOf(10);  
        System.out.println((i1==i2) + " " + i1.equals(i2)); //false true  
    }  
}
```

# *Class Integer*

java.lang.Object

| +----java.lang.Number

| +----java.lang.Integer

public final class Integer

extends Number

The Integer class wraps a value of the primitive type int in an object. An object of type Integer contains a single field whose type is int. In addition, this class provides several methods for converting an int to a String and a String to an int, as well as other constants and methods useful when dealing with an int.

public static final int **MIN\_VALUE**

The smallest value of type int.

public static final int **MAX\_VALUE**

The largest value of type int.

public **Integer**(int value)

Constructs a newly allocated Integer object that represents the primitive int argument.

public **Integer**(String s) throws NumberFormatException

Constructs a newly allocated Integer object that represents the value represented by the string. The string is converted to an int value as if by the valueOf method.



public static String **toBinaryString**(int i)

Creates a string representation of the integer argument as an unsigned integer in base 2.

public static int **parseInt**(String s) throws NumberFormatException

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign '-' to indicate a negative value.

public static Integer **valueOf**(String s) throws NumberFormatException

Returns a new Integer object initialized to the value of the specified String. Throws an exception if the String cannot be parsed as an int. The radix is assumed to be 10.

public int **intValue**() Returns the value of this Integer as an int.

public long **longValue**() Returns the value of this Integer as a long.

public String **toString**() Returns a String object representing this Integer's value.

public boolean **equals**(Object obj)

Compares this object to the specified object. The result is true if and only if the argument is not null and is an Integer object that contains the same int value as this object.

Number classes:

- Number
- Byte
- Double
- Float
- Integer
- Long
- Short
- BigDecimal
- BigInteger

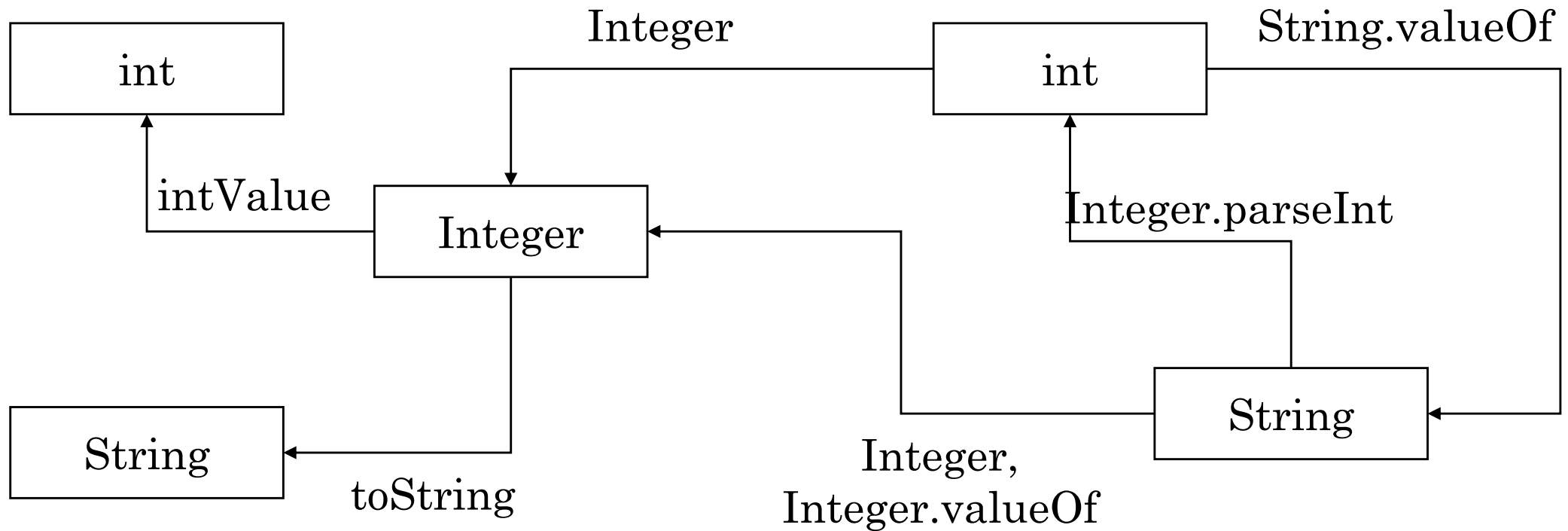
Wrappers for other data types (for completeness):

- Boolean
- Character
- Void

Beyond basic arithmetic:

- Math

# *Conversioni*



# *Arrays*

Here's a simple program, called ArrayDemo, that creates the array, puts some values in it, and displays the values.

```
public class ArrayDemo {  
    public static void main(String[] args) {  
        int[] anArray;           // declare an array of integers  
  
        anArray = new int[10]; // create an array of int  
  
        // assign a value to each array element and print  
        for (int i = 0; i < anArray.length; i++) {  
            anArray[i] = i;  
            System.out.print(anArray[i] + " ");  
        }  
        System.out.println();  
    }  
}
```

To get the size of an array, you write

*arrayname*.**length**

Be careful: Programmers new to the Java programming language are tempted to follow `length` with an empty set of parenthesis. This doesn't work because `length` is not a method. **length is a property** provided by the Java platform for all arrays.

The Java programming language provides a shortcut syntax for creating and initializing an array. Here's an example of this syntax:

```
boolean[] answers = {true, false, true, true, false};
```

Here's a small program, [ArrayOfStringsDemo](#) that creates an array containing three string objects then prints the strings in all lower case letters.

```
public class ArrayOfStringsDemo {  
    public static void main(String[] args) {  
        String[] anArray = { "String One", "String Two",  
"String Three" };  
  
        for (int i = 0; i < anArray.length; i++) {  
            System.out.println(anArray[i].toLowerCase());  
        }  
    }  
}
```

This program creates and populates the array in a single statement.

The following program, [ArrayCopyDemo](#), uses `arraycopy` to copy some elements from the `copyFrom` array to the `copyTo` array.

```
public class ArrayCopyDemo {  
    public static void main(String[] args) {  
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',  
                             'i', 'n', 'a', 't', 'e', 'd' };  
        char[] copyTo = new char[7];  
  
        System.arraycopy(copyFrom, 2, copyTo, 0, 7);  
        System.out.println(new String(copyTo));  
    }  
}
```

The `arraycopy` method call in this example program begins the copy at element number 2 in the source array.

Output: **caffein**



As with arrays of objects, you must explicitly create the sub-arrays within an array. So if you don't use an initializer, you need to write code like the following,

```
public class ArrayOfArraysDemo2 {  
    public static void main(String[] args) {  
        int[][] aMatrix = new int[4][];  
  
        //populate matrix  
        for (int i = 0; i < aMatrix.length; i++) {  
            aMatrix[i] = new int[5];    //create sub-array  
            for (int j = 0; j < aMatrix[i].length; j++) {  
                aMatrix[i][j] = i + j;  
            }  
        }  
    }  
}
```

You must specify the length of the primary array when you create the array. You can leave the length of the sub-arrays unspecified until you create them.

# Matrici

```
public class ArrayOfArraysDemo2 {  
    public static void main(String[]  
args) {  
        int[][] aMatrix = new int[2][];  
  
        for (int i = 0; i <  
aMatrix.length; i++) {  
            aMatrix[i] = new int[3];  
  
            //create sub-array  
  
            for (int j = 0; j <  
aMatrix[i].length; j++) {  
                aMatrix[i][j] = i + j;    }  
            }  
  
            printMatrix(aMatrix);  
            System.out.println();  
  
            int[][] aMatrix1 = { {1,2}, {3,4} };  
            printMatrix(aMatrix1);  
            System.out.println();  
  
            int[][] aMatrix2 = new int[2][3];  
            printMatrix(aMatrix2);  
        }  
    }  
}
```

```
public static void  
printMatrix(int[][] aMatrix1) {  
    for (int i = 0; i < aMatrix1.length;  
i++) {  
        for (int j = 0; j <  
aMatrix1[i].length; j++) {  
            System.out.println(" " + i + "  
" + j + ":" + aMatrix1[i][j]);  
        }  
    }  
}
```

0 0:0	0 0:1	0 0:0
0 1:1	0 1:2	0 1:0
0 2:2	1 0:3	0 2:0
1 0:1	1 1:4	1 0:0
1 1:2		1 1:0
1 2:3		1 2:0

# *Ereditarietà*

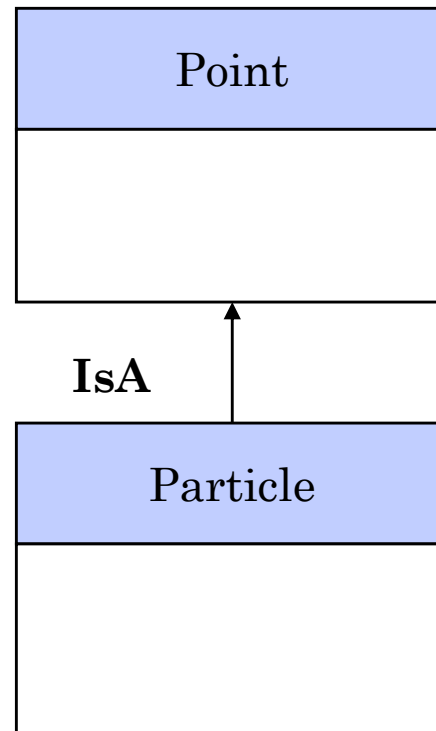
Una classe può ereditare le proprietà di un'altra classe, può ridefinirle e aggiungerne altre.

Polimorfismo

Dynamic binding

Down-cast

# *Ereditarietà*



```
public int getX()
public int getY()
public Point(int x, int y)
public void move(int x, int y)
public String toString()
```

```
public int getM()
public void setM(int m)
public Particle(int x, int y, int m)
public String toString()
```

Particle eredita `getX`, `getY`, `move`; ridefinisce `toString`

# *Point*

```
package g;

public class Point {
    private int x,y;
    public int getX() {return x;}
    public int getY() {return y;}
    public Point(int x, int y) {this.x = x; this.y = y;}
    public void move(int x, int y) {this.x = x; this.y = y;}
    public String toString(){return "x = " + x + " y = " + y;}
}
```

# *Particle*

```
package g;  
  
public class Particle extends Point {  
    private int m;  
    public int getM() {return m;}  
    public void setM(int m) {this.m = m;}  
    public Particle(int x, int y, int m) {super(x,y); this.m = m;}  
    public String toString(){return super.toString() + " m = " + m;}  
}
```

## Note

La prima istruzione del costruttore di Particle deve essere la chiamata di un costruttore della classe di base.

Le proprietà della classe di base non private sono accessibili mediante il rif. super.

```
import g.*;
public class DemoG {
public static void main(String[] args) {
Point p; Particle pa;
Point p1 = new Point(23, 94);
Particle pa1 = new Particle(10,20,100);
pa1.move(1,2); // metodo ereditato
System.out.println(pa1); // chiama pa1.toString()
```

x = 1 y = 2 m = 100

*Uso*

*x = 23 y = 94*

p = p1; // p punta ad un point

System.out.println(p.toString()); // chiama toString() di Point

p = pa1; // p punta ad una Particle

System.out.println(p.toString()); // chiama toString() di Particle

*x = 1 y = 2 m = 100*

// polimorfismo: prima p punta ad un point, poi ad una Particle

// p è definito di tipo Point, ma può puntare anche ad oggetti di classe derivata da Point

// dynamic binding: la versione di toString() da chiamare dipende dal

// tipo dell'oggetto effettivamente puntato

p.setM(10); // errore, p è definito di tipo Point, quindi non vede setM

pa = p; // errore, se pa puntasse ad un point la chiamata di pa.setM

// lecita sintatticamente, produrrebbe un errore a run-time

pa = (Particle)p; // down-cast, se il programmatore insiste

pa.setM(10); // funziona perché pa punta ad una Particle

System.out.println(pa);

*x = 1 y = 2 m = 10*



```
public static void test() {  
    Point[] v = {new Point(10,20), new Particle(1,2,100),  
    new Point(0,1)};  
    for (int i = 0; i < v.length; i++){System.out.println(v[i]);}  
}
```

x = 10 y = 20

x = 1 y = 2 m = 100

x = 0 y = 1

## *Omonimie*

One interesting feature of Java member variables is that a class can access a hidden member variable through its superclass. Consider the following superclass and subclass pair:

```
class A {  
    Integer aNumber;  
}  
class B extends A {  
    Float aNumber;  
}
```

The aNumber variable in B hides aNumber in A. But you can access A's aNumber from B with

**super.aNumber**

# *Inheritance*

A subclass inherits all of the members in its superclass that are accessible to that subclass unless the subclass explicitly hides a member variable or overrides a method. Note that constructors are not members and are not inherited by subclasses.

The following list itemizes the members that are inherited by a subclass:

- Subclasses inherit those superclass members declared as public or protected.
- Subclasses inherit those superclass members declared with no access specifier as long as the subclass is in the same package as the superclass.
- Subclasses don't inherit a superclass's member if the subclass declares a member with the same name. In the case of member variables, the member variable in the subclass hides the one in the superclass. In the case of methods, the method in the subclass overrides the one in the superclass.

# *Inheritance*

A subclass cannot override methods that are declared final in the superclass (by definition, final methods cannot be overridden).

Also, a subclass cannot override methods that are declared static in the superclass. In other words, a subclass cannot override a class method. A subclass can hide a static method in the superclass by declaring a static method in the subclass with the same signature as the static method in the superclass.

# *Costruzione dell'oggetto di base*

```
public Particle(int x, int y, int m) {super(x,y); this.m = m;}
```

La prima istruzione nel costruttore di una classe derivata deve essere la chiamata di un costruttore della classe di base; se manca, il compilatore inserisce la chiamata del costruttore di default.

Un costruttore può chiamare, come prima istruzione, un altro costruttore (della stessa classe) nel modo seguente: `this(...)`

Una classe derivata può chiamare un metodo (ad es. `M1`) della classe di base, che ha ridefinito con un suo metodo, nel modo seguente: `super.M1(...)`

Una catena di `super`, come `super.super.M1(...)`, è illegale.

## *Classi astratte*

public **abstract** class **Number** extends Object

**public abstract** double doubleValue();

Returns the value of the specified number as a double.

public abstract float floatValue();

Returns the value of the specified number as a float.

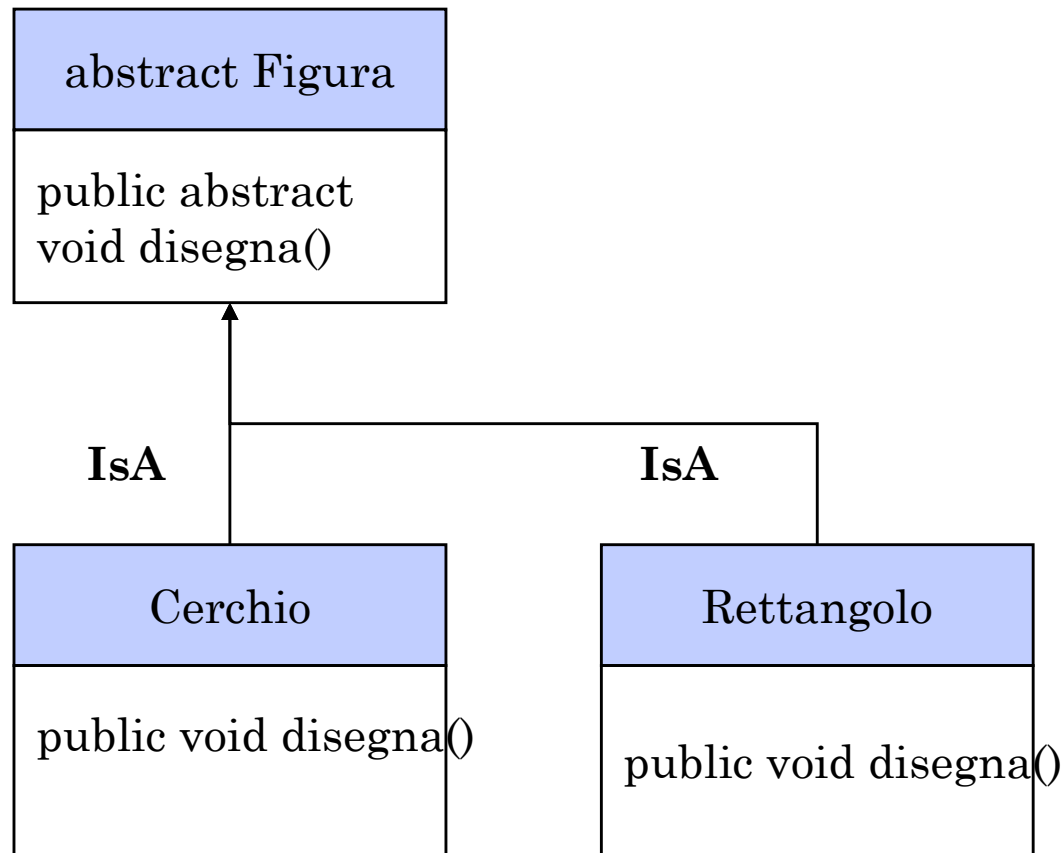
public abstract int **intValue**();

Returns the value of the specified number as an int.

public abstract long longValue();

Returns the value of the specified number as a long.

# *Gerarchia di classi*



```
public static void test() {  
    Figura[] v = {new Cerchio(), new Rettangolo()};  
    for (int i = 0; i < v.length; i++){ (v[i]).disegna();}
```

# *Interfacce*

Oggetti di classi differenti possono interpretare ruoli simili attraverso un'interfaccia comune.

Un'interfaccia è simile ad una classe astratta e contiene dichiarazioni (intestazioni) di metodi.

```
package g;  
  
public interface Movable {  
    void move(int x, int y);  
}  
  
public class Point implements Movable { ...  
public class Rectangle implements Movable { ...
```



```
import g.*;
public class DemoG {
public static void main(String[] args) {
Point p; Particle pa; Movable m;
Point p1 = new Point(23, 94);
Particle pa1 = new Particle(10,20,100);
Rectangle r1 = new Rectangle (1000,2000);
m = p1; m.move(1,2);
m = r1; m.move(1,2);
m = pa1; m.move(1,2);
}
```

public interface **Enumeration**

## *Interface Enumeration*

boolean **hasMoreElements()**

Tests if this enumeration contains more elements.

Object **nextElement()**

Returns the next element of this enumeration.

**Throws:** NoSuchElementException if no more elements exist.

An object that implements the Enumeration interface generates a series of elements, one at a time. Successive calls to the **nextElement** method return successive elements of the series.

```
for (Enumeration e = ... ; e.hasMoreElements() ;) {  
    System.out.println(e.nextElement());  
}
```

Nota: `NoSuchElementException` estende `RuntimeException`

# *Class Object*

The Object class sits at the top of the class hierarchy tree in the Java platform. Every class in the Java system is a descendent, direct or indirect, of the Object class. This class defines the basic state and behavior that all objects must have, such as the ability to compare oneself to another object, to convert to a string, to wait on a condition variable, to notify other objects that a condition variable has changed, and to return the class of the object.

Your classes may want to override the following Object methods.

**clone**

**equals, hashCode**

finalize - chiamato prima della garbage collection

**toString**

Your class cannot override these Object methods (they are final):

getClass, notify, notifyAll, wait

# *Class Object*

public final native Class getClass()

Returns the object of type Class that represents the runtime class of the object.

public native int hashCode()

Returns a hash code value for the object. This method is supported for the benefit of hashtables such as those provided by java.util.Hashtable.

public boolean equals(Object obj)

Compares two Objects for equality.

protected native Object clone() throws CloneNotSupportedException

Creates a new object of the same class as this object. It then initializes each of the new object's fields by assigning it the same value as the corresponding field in this object. No constructor is called. The clone method of class Object will only clone an object whose class indicates that it is willing for its instances to be cloned. A class indicates that its instances can be cloned by declaring that it implements the Cloneable interface.

public String toString()

Returns a string representation of the object. In general, the toString method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.

protected void **finalize**() throws Throwable

Called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the finalize method to dispose of system resources or to perform other cleanup.

# *Esempi di clonazione*

# *Point*

```
package g;

public class Point implements Cloneable {
    private int x,y;

    public int getX() {return x;}
    public int getY() {return y;}
    public Point(int x, int y) {this.x = x; this.y = y;}
    public void move(int x, int y) {this.x = x; this.y = y;}
    public String toString(){
        return "x = " + x + " y = " + y;}
    public boolean equals(Object o) {
        if (!(o instanceof Point)) return false;
        Point p = (Point)o; return x == p.x && y == p.y;}
    public Object clone() {
        try {return super.clone(); }
        catch (CloneNotSupportedException e) {return null;}}}
```

```
import g.Point;

public class DemoG {
    public static void main(String[] args) {
        Point p1 = new Point(23, 94);
        Point p2 = (Point) p1.clone();
        p2.move(10,20);

        System.out.println(p1.toString()); // x = 23 y = 94
        System.out.println(p2); // x = 10 y = 20
        Point p3 = new Point(10, 20);
        System.out.println(p3.equals(p2)); // true
        System.out.println(p3.equals(p1)); // false
    }
}
```



```
package g;
```

```
public class Rectangle implements Cloneable{
```

*Rectangle*

```
    private int width = 20; private int height = 10;
```

```
    private Point origin;
```

```
    public Rectangle() {origin = new Point(0, 0);}
```

```
    public Rectangle(int w, int h) {this(new Point(0, 0), w, h);}
```

```
    public Rectangle(Point p, int w, int h) {
```

```
        origin = p; width = w; height = h;}
```

```
    public String toString() {return origin.toString()
```

```
        + " w = " + width + " h = " + height;}
```

```
    public void move(int x, int y) {origin.move(x,y);}
```

```
    public int area() {return width * height;}
```

```
    public Object clone() {
```

```
        try {Rectangle r = (Rectangle)super.clone();
```

```
            r.origin = (Point)origin.clone(); return r;}
```

```
        catch (CloneNotSupportedException e) {return null;}}}
```

```
import g.*;

public class DemoG1 {

public static void main(String[] args) {

    Rectangle r1 = new Rectangle(100, 200);
    Rectangle r2 = (Rectangle) r1.clone();
    r2.move(10,20);
    System.out.println(r1.toString());   x = 0 y = 0 w = 100 h = 200
    System.out.println(r2.toString());   x = 10 y = 20 w = 100 h = 200
}}
```

The **getClass** method is a final method that returns a runtime representation of the class of an object. This method returns a Class object.

Once you have a Class object you can query it for various information about the class, such as its name, its superclass, and the names of the interfaces that it implements. The following method gets and displays the class name of an object:

```
void PrintClassName(Object obj) {  
    System.out.println("The Object's class is " +  
        obj.getClass().getName());  
}
```

One handy use of a Class object is to create a new instance of a class without knowing what the class is at compile time. The following sample method creates a new instance of the same class as obj, which can be any class that inherits from Object (which means that it could be any class):

```
Object createNewInstanceOf(Object obj) {  
    return obj.getClass().newInstance();  
}
```

```
public Object newInstance() throws InstantiationException,  
    IllegalAccessException
```

## *Uso di Class*

```
import g.*;

public class DemoG2 {

public static void main(String[] args) {

    Rectangle r1 = new Rectangle(100,200);

    Class c1 = r1.getClass();

    System.out.println(c1.getName()); // g.Rectangle

try{

    Rectangle r2 = (Rectangle)c1.newInstance();

    System.out.println(r2);    // x = 0 y = 0 w = 20 h = 10

    Class c2 = Class.forName("g.Particle");

    System.out.println(c2);    // class g.Particle

    System.out.println(c2.getSuperclass()); // class g.Point

} catch(Exception e) {System.out.println(e);}
```

## *hashCode e toString*

```
public class Info {  
    int val;  
    public Info(int val) {this.val = val;}  
}  
  
public class InfoTest {  
    public static void main(String[] args) {  
        Info a = new Info(10);  
        Info b = new Info(10);  
        System.out.println(a.hashCode());  
        System.out.println(b.hashCode());  
        System.out.println(a);  
        System.out.println(b);  
    }  
}
```

17523401

8567361

Info@10b62c9

Info@82ba41

```
public class Info {
```

```
    int val;
```

```
    public Info(int val) {this.val = val;}
```

```
    public int hashCode() {return val;}
```

```
    public String toString() {return String.valueOf(val);}
```

```
}
```

```
public class InfoTest {
```

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

```
        Info a = new Info(10);
```

```
        Info b = new Info(10);
```

```
        System.out.println(a.hashCode());
```

```
        System.out.println(b.hashCode());
```

```
        System.out.println(a);
```

```
        System.out.println(b);
```

```
    }}
```

*hashCode e toString*

10

10

10

10

# *Nested Classes*

Java lets you define a class as a member of another class. Such a class is called a *nested class* and is illustrated here:

```
class EnclosingClass{  
    . . .  
    class ANestedClass {  
        . . .  
    }  
}
```

**Definition:** A nested class is a class that is a member of another class.

# *StringList*

```
import java.util.*;

public class StringList {
    private String[] a = {"s1", "s2", "s3"};
    public Enumeration enumerator() {return new StringEnum();}
    private class StringEnum implements Enumeration {
        int i = 0;
        public boolean hasMoreElements() {return (i < a.length);}
        public Object nextElement() {
            if (!hasMoreElements())
                throw new NoSuchElementException();
            else return a[i++];
        }
    }
}
```



```
import java.util.*;

public class StringListMain {

    public static void main(String[] args) {

        StringList s = new StringList();

        for (Enumeration e = s.enumerator(); e.hasMoreElements();) {

            String s1 = (String)e.nextElement();

            System.out.println(s1);}

        Enumeration e = s.enumerator();

        for (; e.hasMoreElements();) {

            String s1 = (String)e.nextElement();

            System.out.println(s1);}

        String s1 = (String)e.nextElement();}}
```

s1  
s2  
s3  
s1  
s2  
s3

```
Exception in thread "main" java.util.NoSuchElementException
at StringList$StringEnum.nextElement(StringList.java:9)
at StringListMain.main(StringListMain.java:14)
```

- `java.lang`: wrappers for basic data types and strings; threads; mathematical operations; exceptions; the `System` class; and the `Object` base class.
- `java.io`: Classes for controlling file and stream operations.
- `java.util`: a class for manipulating dates; basic data structure classes, such as hash tables and stacks; a tool for parsing Strings.
- `java.applet`: This package contains the `Applet` class.
- `java.awt`: The AWT (Abstract Window Toolkit) package provides simple and advanced tools used to direct the visual interface of a Java applet.
- `java.awt.image`: This "sub-package" of AWT provides the underpinnings for using images in Java.
- `java.awt.peer`: This package is used to tie AWT widgets to the underlying platform.
- `java.awt.net`: The classes in this package are used to write Java network programs. This package includes sockets classes for both the client and the server and classes for referencing URLs.

# *Class System*

**System.runFinalization();** This method calls the finalize methods on all objects that are waiting to be garbage collected.

**System.gc();** Running the Garbage Collector.

```
byte[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a', 't', 'e', 'd' };
```

```
byte[] copyTo = new byte[7];
```

```
System.arraycopy(copyFrom, 2, copyTo, 0, 7);
```

```
long clickTime = System.currentTimeMillis();
```

```
System.exit(-1); // status code <> 0 terminazione anormale
```

# *Classe Math*

Math Class (final con metodi statici)

come ad esempio static double sin(double a)

```
public long uniform(long l, long h) {  
    double d = Math.random() ;  
    return Math.round(l + d * (h - l)); }  
}
```

# *Collections*

# Collections

A collection (sometimes called a *container*) is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve and manipulate data, and to transmit data from one method to another.

Collections typically represent data items that form a natural group, like a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a collection of name-to-phone-number mappings). A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain three things:

**Interfaces:** abstract data types representing collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages like Java, these interfaces generally form a hierarchy.

**Implementations:** concrete implementations of the collection interfaces. In essence, these are *reusable data structures*.

**Algorithms:** methods that perform useful computations, like searching and sorting, on objects that implement collection interfaces. These algorithms are said to be *polymorphic* because the same method can be used on many different implementations of the appropriate collections interface. In essence, algorithms are *reusable functionality*.

# *Collezioni*

Interface	Implementation				Historical
Set	HashSet		TreeSet		
List		ArrayList		LinkedList	Vector Stack
Map	HashMap		TreeMap		Hashtable Properties

**Collection**

**Iterator,  
ListIterator**

## *Collection*

```
+add(element : Object) : boolean  
+addAll(collection : Collection) : boolean  
+clear() : void  
+contains(element : Object) : boolean  
+containsAll(collection : Collection) : boolean  
+equals(object : Object) : boolean  
+hashCode() : int  
+iterator() : Iterator  
+remove(element : Object) : boolean  
+removeAll(collection : Collection) : boolean  
+retainAll(collection : Collection) : boolean  
+size() : int  
+toArray() : Object[]  
+toArray(array : Object[]) : Object[]
```

*Collection*



**Set**

```
+add(element : Object) : boolean  
+addAll(collection : Collection) boolean  
+clear() : void  
+contains(element : Object) : boolean  
+containsAll(collection : Collection) : boolean  
+equals(object : Object) : boolean  
+hashCode() : int  
+iterator() : Iterator  
+remove(element : Object) : boolean  
+removeAll(collection : Collection) : boolean  
+retainAll(collection : Collection) : boolean  
+size() : int  
+toArray() : Object[]  
+toArray(array : Object[]) : Object[]
```

# List

## *List*

```
+add(element : Object) : boolean  
+add(index : int, element : Object) : void  
+addAll(collection : Collection) : boolean  
+addAll(index : int, collection : Collection) : boolean  
+clear() : void  
+contains(element : Object) : boolean  
+containsAll(collection : Collection) : boolean  
+equals(object : Object) : boolean  
+get(index : int) : Object  
+hashCode() : int  
+indexOf(element : Object) : int  
+iterator() : Iterator  
+lastIndexOf(element : Object) : int  
+listIterator() : ListIterator  
+listIterator(startIndex : int) : ListIterator  
+remove(element : Object) : boolean  
+remove(index : int) : Object  
+removeAll(collection : Collection) : boolean  
+retainAll(collection : Collection) : boolean  
+set(index : int, element : Object) : Object  
+size() : int  
+subList(fromIndex : int, toIndex : int) : List  
+toArray() : Object[]  
+toArray(array : Object[]) : Object[]
```

## *Map*

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

The Iterator interface is shown below:

```
public interface Iterator {  
    boolean hasNext();  
    Object next();  
    void remove();}
```

## *Iterator*

The hasNext method is identical in function to Enumeration.hasMoreElements, and the next method is identical in function to Enumeration.nextElement. The remove method removes from the underlying Collection the last element that was returned by next. The remove method may be called only once per call to next, and throws an exception if this condition is violated. Note that Iterator.remove is the only safe way to modify a collection during iteration; the behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress.

The following snippet shows you how to use an Iterator to filter a Collection, that is, to traverse the collection, removing every element that does not satisfy some condition:

```
static void filter(Collection c) {  
    for (Iterator i = c.iterator(); i.hasNext(); )  
        if (!cond(i.next()))  
            i.remove();}
```

# Sets

A Set is a Collection that cannot contain duplicate elements. Set models the mathematical *set* abstraction. The Set interface extends Collection and contains *no* methods other than those inherited from Collection.

The JDK contains two general-purpose Set implementations. HashSet, which stores its elements in a hash table, is the best-performing implementation. TreeSet, which stores its elements in a red-black tree, guarantees the order of iteration.

Here's a simple but useful Set idiom. Suppose you have a Collection, *c*, and you want to create another Collection containing the same elements, but with all duplicates eliminated. The following one-liner does the trick:

```
Collection noDups = new HashSet(c);
```

# Sets

```
import java.util.*;

public class TestSet1 {

    public static void main(String[] args) {

        Set elementiDistinti = new HashSet();
        Set elementiRipetuti = new TreeSet();

        String[] v = {"alfa", "beta", "gamma", "alfa", "delta", "gamma"};

        for (int i = 0; i < v.length; i++)

            if (!elementiDistinti.add(v[i])) elementiRipetuti.add(v[i]);

        System.out.println("elementi distinti: " + elementiDistinti);
        System.out.println("elementi ripetuti: " + elementiRipetuti);

    }

}
```

```
elementi distinti: [gamma, delta, beta, alfa]
elementi ripetuti: [alfa, gamma]
```

# Sets

```
import java.util.*;

class Info1 {int val;

    public Info1(int val) {this.val = val;}

    public String toString() {return String.valueOf(val);}}

public class TestSet2 {

    public static void main(String[] args) {

        Set elementiDistinti = new HashSet();

        Set elementiRipetuti = new HashSet(); Info1 a = new Info1(1);

        Info1[] v = {a, new Info1(2), new Info1(1), a};

        for (int i = 0; i < v.length; i++)

            if (!elementiDistinti.add(v[i])) elementiRipetuti.add(v[i]);

        System.out.println("elementi distinti: " + elementiDistinti);

        System.out.println("elementi ripetuti: " + elementiRipetuti);}}
```

```
elementi distinti: [1, 1, 2]
elementi ripetuti: [1]
```

Per scartare gli elementi  
ripetuti occorre implementare  
equals e hashCode

```
public class Info {  
    int val;  
    public Info(int val) {this.val = val;}  
    public int hashCode() {return val;}  
    public boolean equals(Object o)  
    {  
        if (!(o instanceof Info)) return false;  
        Info x = (Info)o; return x.val == val;}  
    public String toString() {return String.valueOf(val);}  
}
```



```
import java.util.*;
public class TestSet3 {
    public static void main(String[] args) {
        Set elementiDistinti = new HashSet();
        Set elementiRipetuti = new HashSet();
        Info a = new Info(1);
        Info[] v = {a, new Info(2), new Info(1), a};
        for (int i = 0; i < v.length; i++)
            if (!elementiDistinti.add(v[i])) elementiRipetuti.add(v[i]);
        System.out.println("elementi distinti: " + elementiDistinti);
        System.out.println("elementi ripetuti: " + elementiRipetuti);}}
// elementi distinti: [2, 1]
// elementi ripetuti: [1]
```

Un HashSet aggiunge un nuovo elemento se il suo hashCode è diverso da quelli già presenti oppure se equals è falso per tutti gli oggetti di pari hashCode.

Per avere gli elementi ordinati occorre usare un TreeSet e implementare l'interfaccia Comparable (e quindi il metodo compareTo).

```
public class Info implements Comparable {  
    int val;  
    public Info(int val) {this.val = val;}  
    public int hashCode() {return val;}  
    public boolean equals(Object o)  
        {if (!(o instanceof Info)) return false;  
         Info x = (Info)o; return x.val == val;}  
    public int compareTo(Object x) {  
        Info info1 = (Info)x;  
        if (val < info1.val) return -1;  
        if (val == info1.val) return 0; else return 1;}  
    public String toString() {return String.valueOf(val);}  
}
```

# Sets

```
import java.util.*;

public class TestSet3 {

    public static void main(String[] args) {

        Set elementiDistinti = new TreeSet();
        Set elementiRipetuti = new HashSet();

        Info a = new Info(1);
        Info[] v = {a, new Info(2), new Info(1), a};
        for (int i = 0; i < v.length; i++)
            if (!elementiDistinti.add(v[i])) elementiRipetuti.add(v[i]);

        System.out.println("elementi distinti: " + elementiDistinti);
        System.out.println("elementi ripetuti: " + elementiRipetuti);}}

// elementi distinti: [1, 2]
// elementi ripetuti: [1]
```

# *Iterator*

```
import java.util.*;

public class TestSet3 {

    public static void main(String[] args) {

        Set elementiDistinti = new TreeSet();
        Set elementiRipetuti = new HashSet();

        Info a = new Info(1);
        Info[] v = {a, new Info(2), new Info(1), a};

        for (int i = 0; i < v.length; i++)
            if (!elementiDistinti.add(v[i])) elementiRipetuti.add(v[i]);

        //System.out.println("elementi distinti: " + elementiDistinti);
        //System.out.println("elementi ripetuti: " + elementiRipetuti);

        for (Iterator i = elementiDistinti.iterator(); i.hasNext();) {
            Info x = (Info)i.next(); System.out.println(x);}

        // 1
        // 2
    }
}
```

## *toArray*

```
import java.util.*;

public class TestSet3 {

    public static void main(String[] args) {

        Set elementiDistinti = new TreeSet();
        Set elementiRipetuti = new HashSet();

        Info a = new Info(1);

        Info[] v = {a, new Info(2), new Info(1), a};

        for (int i = 0; i < v.length; i++)

            if (!elementiDistinti.add(v[i])) elementiRipetuti.add(v[i]);

        Object[] v1 = elementiDistinti.toArray();

        for (int i = 0; i < v1.length; i++) System.out.println(v1[i]);

        Info[] v2 = new Info[elementiDistinti.size()];

        elementiDistinti.toArray(v2);

        // 1

        // 2
    }
}
```

# *Bulk operations*

The bulk operations are particularly well suited to Sets: they perform standard set-algebraic operations. Suppose *s1* and *s2* are Sets. Here's what the bulk operations do:

**s1.containsAll(s2):** Returns true if *s2* is a subset of *s1*. (For example, set *s1* is a subset of *s2* if set *s2* contains all the elements in *s1*.)

**s1.addAll(s2):** Transforms *s1* into the union of *s1* and *s2*. (The union of two sets is the set containing all the elements contained in either set.)

**s1.retainAll(s2):** Transforms *s1* into the intersection of *s1* and *s2*. (The intersection of two sets is the set containing only the elements that are common in both sets.)

**s1.removeAll(s2):** Transforms *s1* into the (asymmetric) set difference of *s1* and *s2*. (For example, the set difference of *s1* - *s2* is the set containing all the elements found in *s1* but not in *s2*.)

**s1.clear():** Removes all elements.

The `addAll`, `removeAll`, and `retainAll` methods all return true if the target Collection was modified in the process of executing the operation.

## *Bulk operations*

To calculate the union, intersection, or set difference of two sets (s1, s2) non-destructively (without modifying either set), the caller must copy one set before calling the appropriate bulk operation. The resulting idioms are shown below:

```
Set union = new HashSet(s1);  
union.addAll(s2);
```

```
Set intersection = new HashSet(s1);  
intersection.retainAll(s2);
```

```
Set difference = new HashSet(s1);  
difference.removeAll(s2);
```



# *Bulk operations*

As a simple example of the power of the bulk operations, consider the following idiom to remove all instances of a specified element, `e` from a `Collection`, `c`:

```
c.removeAll(Collections.singleton(e));
```

More specifically, suppose that you want to remove all of the null elements from a `Collection`:

```
c.removeAll(Collections.singleton(null));
```

This idiom uses `Collections.singleton`, which is a static factory method that returns an immutable `Set` containing only the specified element.

Let's revisit the FindDups example program above. Suppose you want to know which words in the argument list occur only once and which occur more than once, but you do not want any duplicates printed out repeatedly. This effect can be achieved by generating *two sets, one containing every word in the argument list, and the other containing only the duplicates*. The words that occur only once are the set difference of these two sets, which we know how to compute.

```
import java.util.*;

public class FindDups2 {
    public static void main(String args[]) {
        Set uniques = new HashSet();
        Set dups = new HashSet();
        for (int i=0; i<args.length; i++)
            if (!uniques.add(args[i]))
                dups.add(args[i]);
        uniques.removeAll(dups); // Destructive set-difference

        System.out.println("Unique words:      " + uniques);
        System.out.println("Duplicate words: " + dups);    }}


```

A screenshot of a Windows Command Prompt window. The title bar is blue with the text "Command Prompt". The window shows the command "F:\Java\Prove>\jdk1.3\bin\java FindDups2 alfa beta gamma alfa" being executed. The output is "Unique words: [gamma, beta]" and "Duplicate words: [alfa]". The prompt "F:\Java\Prove>" is visible at the bottom.

```
MS-DOS Command Prompt
F:\Java\Prove>\jdk1.3\bin\java FindDups2 alfa beta gamma alfa
Unique words: [gamma, beta]
Duplicate words: [alfa]
F:\Java\Prove>
```

# List

A List is an ordered Collection (sometimes called a *sequence*). Lists may contain duplicate elements. In addition to the operations inherited from Collection, the List interface includes operations for:

**Positional Access:** manipulate elements based on their numerical position in the list.

**Search:** search for a specified object in the list and return its numerical position.

**List Iteration:** extend Iterator semantics to take advantage of the list's sequential nature.

**Range-view:** perform arbitrary *range operations* on the list.

The JDK contains two general-purpose List implementations. ArrayList, which is generally the best-performing implementation, and LinkedList which offers better performance under certain circumstances.

## *LinkedList*

In addition, `LinkedList` adds several methods for working with the elements at the ends of the list.

By using these new methods, you can easily treat the `LinkedList` as a stack, queue, or other end-oriented data structure.

```
LinkedList queue = ...;  
queue.addFirst(element);  
Object object = queue.removeLast();
```

```
LinkedList stack = ...;  
stack.addFirst(element);  
Object object = stack.removeFirst();
```

<http://java.sun.com/developer/onlineTraining/collections/Collection.html>

# *Liste*

```
import java.util.*;

public class ListExample {

    public static void main(String args[]) {

        List list = new ArrayList();

        list.add("Bernadine"); list.add("Elizabeth"); list.add("Gene");
        list.add("Elizabeth"); list.add("Clara");

        System.out.println(list); // [Bernadine, Elizabeth, Gene, Elizabeth, Clara]
        System.out.println("2: " + list.get(2)); // 2: Gene
        System.out.println("0: " + list.get(0)); // 0: Bernadine

        LinkedList queue = new LinkedList();

        queue.addFirst("Bernadine"); queue.addFirst("Elizabeth");
        queue.addFirst("Gene"); queue.addFirst("Elizabeth"); queue.addFirst("Clara");
        System.out.println(queue); // [Clara, Elizabeth, Gene, Elizabeth, Bernadine]
        queue.removeLast(); queue.removeLast();
        System.out.println(queue); // [Clara, Elizabeth, Gene]
    }
}
```

The following idiom concatenates one list to another:

```
list1.addAll(list2);
```

Here's a non-destructive form of this idiom, which produces a third List consisting of the second list appended to the first:

```
List list3 = new ArrayList(list1);  
list3.addAll(list2);
```

Here's a little function to swap two indexed values in a List.

```
private static void swap(List a, int i, int j) {  
    Object tmp = a.get(i);  
    a.set(i, a.get(j));  
    a.set(j, tmp);  
}
```

```
public interface ListIterator extends Iterator {  
    boolean hasNext();  
    Object next();  
  
    boolean hasPrevious();  
    Object previous();  
  
    int nextIndex();  
    int previousIndex();  
  
    void remove();  
    void set(Object o);  
    void add(Object o);  
}
```

Here's the standard idiom for iterating backwards through a list:

```
for (ListIterator i=l.listIterator(l.size()); i.hasPrevious();) {  
    Foo f = (Foo) i.previous();  
    ...  
}
```

The `ListIterator` interface provides two additional operations to modify the list: **set** and **add**. The `set` method "overwrites" the last element returned by `next` or `previous` with the specified element. It is demonstrated by the following polymorphic algorithm to replace all occurrences of one specified value with another:

```
public static void replace(List l, Object val, Object newVal) {  
    for (ListIterator i = l.listIterator(); i.hasNext();)   
        if (val==null ? i.next()==null : val.equals(i.next()))  
            i.set(newVal);  
}
```

The only bit of trickiness in this example is the equality test between `val` and `i.next`. We have to special-case a `val` value of `null` in order to prevent a `NullPointerException`.



The add method inserts a new element into the list, immediately before the current cursor position. This method is illustrated in the following polymorphic algorithm *to replace all occurrences of a specified value with the sequence of values contained in the specified list*:

```
public static void replace(List l, Object val, List newVals) {  
  
    for (ListIterator i = l.listIterator(); i.hasNext(); ) {  
        if (val==null ? i.next()==null : val.equals(i.next()))  
        {  
            i.remove();  
            for (Iterator j = newVals.iterator(); j.hasNext(); )  
                i.add(j.next());  
        }  
    }  
}
```

The range-view operation, **subList**(int fromIndex, int toIndex), returns a List *view* of the portion of this list whose indices range from fromIndex, inclusive, to toIndex, exclusive. For example, the following idiom removes a range of elements from a list:

```
list.subList(fromIndex, toIndex).clear();
```

Similar idioms may be constructed to search for an element in a range:

```
int i = list.subList(fromIndex, toIndex).indexOf(o);  
int j = list.subList(fromIndex, toIndex).lastIndexOf(o);
```

Note that the above idioms return the index of the found element in the subList, not the index in the backing List.

Most of the polymorphic algorithms in the Collections class apply specifically to List. Having all of these algorithms at your disposal makes it very easy to manipulate lists.

**sort(List)**: Sorts a List using a merge sort algorithm, which provides a fast, *stable* sort. (A stable sort is one that does not reorder equal elements.)

**shuffle(List)**: Randomly permutes the elements in a List.

**reverse(List)**: Reverses the order of the elements in a List.

**fill(List, Object)**: Overwrites every element in a List with the specified value.

**copy(List dest, List src)**: Copies the source List into the destination List.

**binarySearch(List, Object)**: Searches for an element in an ordered List using the binary search algorithm.

A Map is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value.

```
public interface Map {  
    // Basic Operations  
    Object put(Object key, Object value);  
    Object get(Object key);  
    Object remove(Object key);  
    boolean containsKey(Object key);  
    boolean containsValue(Object value);  
    int size();  
    boolean isEmpty();  
    // Bulk Operations  
    void putAll(Map t);  
    void clear(); }  

```

The JDK contains two new general-purpose Map implementations. HashMap, which stores its entries in a hash table, is the best-performing implementation. TreeMap, which stores its entries in a red-black tree, guarantees the order of iteration.

```
Map copy = new HashMap(m);
```

Here's a simple program to generate a *frequency table* of the words found in its argument list. The frequency table maps each word to the number of times it occurs in the argument list.

```
import java.util.*;
public class Freq {
    private static final Integer ONE = new Integer(1);
    public static void main(String args[]) {
        Map m = new HashMap();
        // Initialize frequency table from command line
        for (int i=0; i<args.length; i++) {
            Integer freq = (Integer) m.get(args[i]);
            m.put(args[i], (freq==null ? ONE :
                new Integer(freq.intValue() + 1)));
        }
        System.out.println(m.size()+" distinct words detected:");
        System.out.println(m);    }}
}
```

A screenshot of a Windows Command Prompt window. The title bar is blue with the text 'Command Prompt' and standard window controls. The command prompt shows the directory 'F:\Java\Prove' and the command to run 'java Freq alfa beta gamma alfa'. The output shows '3 distinct words detected:' followed by the frequency table '{gamma=1, beta=1, alfa=2}'. The prompt returns to 'F:\Java\Prove>'.

```
MS-DOS Command Prompt
F:\Java\Prove>\jdk1.3\bin\java  Freq alfa beta gamma alfa
3 distinct words detected:
{gamma=1, beta=1, alfa=2}
F:\Java\Prove>
```

```
// Collection Views
    public Set keySet();
    public Collection values();
    public Set entrySet();
    // Interface for entrySet elements
    public interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
    }
}
```

The Collection-view methods allow a Map to be viewed as a Collection in three ways:

**keySet:** the Set of keys contained in the Map.

**values:** The Collection of values contained in the Map. This Collection is not a Set, as multiple keys can map to the same value.

**entrySet:** The Set of key-value pairs contained in the Map. The Map interface provides a small nested interface called Map.Entry that is the type of the elements in this Set.

Here's an example illustrating the standard idiom for iterating over the keys in a Map:

```
for (Iterator i=m.keySet().iterator(); i.hasNext(); )  
    System.out.println(i.next());
```

For starters, suppose you want to know whether one Map is a submap of another, that is, whether the first Map contains all of the key-value mappings in the second. The following idiom does the trick:

```
if (m1.entrySet().containsAll(m2.entrySet())) { ... }
```

Along similar lines, suppose that you want to know if two Map objects contain mappings for all the same keys:

```
if (m1.keySet().equals(m2.keySet())) { ... }
```

Suppose you want to remove all the key-value pairs that one Map has in common with another:

```
m1.entrySet().removeAll(m2.entrySet());
```

Suppose you want to remove from one Map all the keys that have mappings in another:

```
m1.keySet().removeAll(m2.keySet());
```



## *Note su mappe*

```
Map s1 = new HashMap();  
s1.put(new Info(1), new Integer(1));  
s1.put(new Info(1), new Integer(2));  
System.out.println(s1);           // {1=2}  
  
s1 = new HashMap(); s1.put(new Info(1), new Integer(1));  
s1.put(new Info(2), new Integer(2));  
s1.put(new Info(5), new Integer(5));  
s1.put(new Info(4), new Integer(4));  
System.out.println(s1);           // {5=5, 4=4, 2=2, 1=1}  
  
s1 = new TreeMap(); s1.put(new Info(1), new Integer(1));  
s1.put(new Info(2), new Integer(2));  
s1.put(new Info(5), new Integer(5));  
s1.put(new Info(4), new Integer(4));  
System.out.println(s1);           // {1=1, 2=2, 4=4, 5=5}
```

# Object Ordering

A List l may be sorted as follows:

```
Collections.sort(l);
```

If the list consists of String elements, it will be sorted into lexicographic (alphabetical) order. If it consists of Date elements, it will be sorted into chronological order. How does Java know how to do this? It's magic! Well, no. Actually String and Date both implement the Comparable interface.

If you try to sort a list whose elements do not implement Comparable, Collections.sort(list) will throw a ClassCastException

The Comparable interface consists of a single method:

```
public interface Comparable {  
    public int compareTo(Object o);  
}
```

The compareTo method compares the receiving object with the specified object, and returns a negative integer, zero, or a positive integer as the receiving object is less than, equal to, or greater than the specified Object. If the specified object cannot be compared to the receiving object, the method throws a ClassCastException.

Here's a *class representing a person's name* that implements Comparable:

```
import java.util.*;
public class Name implements Comparable {
    private String  firstName, lastName;
    public Name(String firstName, String lastName) {
        if (firstName==null || lastName==null)
            throw new NullPointerException();
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public String firstName()    {return firstName;}
    public String lastName()     {return lastName;}
    public boolean equals(Object o) {
        if (!(o instanceof Name))
            return false;
        Name n = (Name)o;
        return n.firstName.equals(firstName) &&
            n.lastName.equals(lastName);
    }
}
```

```
public int hashCode() {  
    return 31*firstName.hashCode() + lastName.hashCode();  
}  
public String toString() {return firstName + " " + lastName;}  
  
public int compareTo(Object o) {  
    Name n = (Name)o;  
    int lastCmp = lastName.compareTo(n.lastName);  
    return (lastCmp!=0 ? lastCmp :  
            firstName.compareTo(n.firstName));  
}  
}
```

Just to show that it all works, here's a little program that builds a list of Name objects and sorts them:

```
import java.util.*;
class NameSort {
    public static void main(String args[]) {
        Name n[] = {
            new Name("John", "Lennon"),
            new Name("Karl", "Marx"),
            new Name("Groucho", "Marx"),
            new Name("Oscar", "Grouch")
        };
        List l = Arrays.asList(n);
        Collections.sort(l);
        System.out.println(l);
    }
}
```

If you run this program, here's what it prints:

```
[Oscar Grouch, John Lennon, Groucho Marx, Karl Marx]
```

# *Comparator*

OK, so now you know about natural ordering. But what if you want to sort some objects in some order other than their natural order? Or what if you want to sort some objects that don't implement Comparable? To do either of these things, you'll need to provide a Comparator. A Comparator is simply an object that encapsulates an ordering. Like the Comparable interface, the Comparator interface consists of a single method:

```
public interface Comparator {  
    int compare(Object o1, Object o2);  
}
```

The compare method compares its two arguments, returning a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second. If either of the arguments has an inappropriate type for the Comparator, the compare method throws a ClassCastException.

Suppose you have a class called `EmployeeRecord`:

```
public class EmployeeRecord implements Comparable
{
    public Name name();
    public int employeeNumber();
    public Date hireDate();
    ...
}
```

Let's assume that the natural ordering of `EmployeeRecord` objects is Name-ordering (as defined in the previous example) on employee name. Unfortunately the boss has asked us for a list of employees in order of seniority.

```
import java.util.*;

class EmpSort {
    static final Comparator SENIORITY_ORDER = new Comparator() {
        public int compare(Object o1, Object o2) {
            EmployeeRecord r1 = (EmployeeRecord) o1;
            EmployeeRecord r2 = (EmployeeRecord) o2;
            return r2.hireDate().compareTo(r1.hireDate());
        }
    };

    static final Collection employees = ... ; // Employee Database

    public static void main(String args[]) {
        List emp = new ArrayList(employees);
        Collections.sort(emp, SENIORITY_ORDER);
        System.out.println(emp);
    }
}
```



# *Total ordering*

The way to do this is to do a two-part comparison (like we did for Name) where the first part is the one that we're actually interested in (in this case, the hire-date), and the second part is attribute that uniquely identifies the object. In this case, the employee number is the obvious attribute to use as the second part. Here's the Comparator that results:

```
static final Comparator SENIORITY_ORDER = new Comparator() {  
    public int compare(Object o1, Object o2) {  
        EmployeeRecord r1 = (EmployeeRecord) o1;  
        EmployeeRecord r2 = (EmployeeRecord) o2;  
        int dateCmp = r2.hireDate().compareTo(r1.hireDate());  
        if (dateCmp != 0) return dateCmp;  
        return (r1.employeeNumber() < r2.employeeNumber() ? -1 :  
  
        (r1.employeeNumber() == r2.employeeNumber() ? 0 : 1));  
    }  
};
```

The two general purpose Set implementations are HashSet and TreeSet. It's very straightforward to decide which of these two to use. HashSet is much faster (constant time vs. log time for most operations), but offers no ordering guarantees. If you need to use the operations in the SortedSet, or in-order iteration is important to you, use TreeSet. Otherwise, use HashSet. The initial capacity may be specified using the int constructor. To allocate a HashSet whose initial capacity is 17 (101 is the default value): `Set s= new HashSet(17);`

The two general purpose List implementations are ArrayList and LinkedList. Most of the time, you'll probably use ArrayList. It offers constant time positional access, and it's just plain fast, because it does not have to allocate a node object for each element in the List, and it can take advantage of the native method `System.arraycopy` when it has to move multiple elements at once. Think of ArrayList as Vector without the synchronization overhead. ArrayList has one tuning parameter, the *initial capacity*.

The two general purpose Map implementations are HashMap and TreeMap. The situation for Map is *exactly* analogous to Set. If you need SortedMap operations or in-order Collection-view iteration, go for TreeMap; otherwise, go for HashMap. Everything else in the Set section also applies to Map so just re-read it.

The *polymorphic algorithms* described in this section are pieces of reusable functionality provided by the JDK. All of them come from the Collections class. All take the form of static methods whose first argument is the collection on which the operation is to be performed. The great majority of the algorithms provided by the Java platform operate on List objects, but a couple of them (min and max) operate on arbitrary Collection objects.

## Sorting

```
import java.util.*;
public class Sort {
    public static void main(String args[]) {
        List l = Arrays.asList(args);
        //Returns a fixed-size list backed by the specified array.
        Collections.sort(l);
        System.out.println(l);    }}
```

## *Algorithms*

The second form of sort takes a Comparator in addition to a List and sorts the elements with the Comparator.

# Shuffling

The shuffle algorithm does the opposite of what sort does: it destroys any trace of order that may have been present in a List. There are two forms of this operation. The first just takes a List and uses a default source of randomness. The second requires the caller to provide a Random object to use as a source of randomness.

```
import java.util.*;
```

```
public class Shuffle {  
    public static void main(String args[]) {  
        List l = new ArrayList();  
        for (int i=0; i<args.length; i++)  
            l.add(args[i]);  
        Collections.shuffle(l, new Random());  
        System.out.println(l);    } }  
//[9, 0, 8, 2, 7, 1, 3, 6, 4, 5]
```

## Routine Data Manipulation

The Collections class provides three algorithms for doing routine data manipulation on List objects. All of these algorithms are pretty straightforward:

**reverse:** Reverses the order of the elements in a List.

```
(public static void reverse(List l))
```

**fill:** Overwrites every element in a List with the specified value. This operation is useful for re-initializing a List.

```
(public static void fill(List list, Object o))
```

**copy:** Takes two arguments, a destination List and a source List, and copies the elements of the source into the destination, overwriting its contents. The destination List must be at least as long as the source. If it is longer, the remaining elements in the destination List are unaffected.

```
(public static void copy(List dest, List src))
```

## Searching

The `binarySearch` algorithm searches for a specified element in a sorted List using the *binary search* algorithm.

The following idiom, usable with both forms of the `binarySearch` operation, looks for the specified search key, and inserts it at the appropriate position if it's not already present:

```
int pos = Collections.binarySearch(l, key);  
if (pos < 0)  
    l.add(-pos-1, key);
```

## Date, Calendar

### Date()

Allocates a **Date** object and initializes it so that it represents the time at which it was allocated, measured to the nearest millisecond.

## Random

## StringTokenizer

## Vector, Stack, Hashtable

## Uso di Random

```
Random randGen = new Random();  
// intero casuale tra 0 e 99  
int i = Math.abs(randGen.nextInt())%100;
```

# *Calendar*

```
import java.util.Calendar;
import java.util.GregorianCalendar;
public class GestioneDate {
    public static void main(String[] args) {
        Calendar c1 = new GregorianCalendar(); // oggi
        Calendar c2 = new GregorianCalendar(2004,12,25);
        // confronti con after e before
        if (c1.after(c2) ...
        System.out.println(c1.getTime()); //Wed Apr 21 17:58:08 CEST 2004
```



# *Class Random*

public class **Random** extends Object implements Serializable

An instance of this class is used to generate a stream of pseudorandom numbers. The class uses a 48-bit seed, which is modified using a linear congruential formula.

public **Random()**

Creates a new random number generator. Its seed is initialized to a value based on the current time.

public **Random(long seed)** Creates a new random number generator using a single long seed.

public float **nextFloat()**

Returns the next pseudorandom, uniformly distributed float value between 0.0 and 1.0 from this random number generator's sequence.

public int **nextInt()**

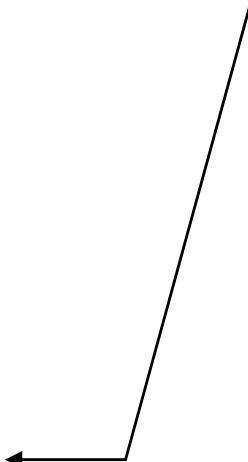
Returns the next pseudorandom, uniformly distributed int value from this random number generator's sequence.

# *StringTokenizer*

```
String CurrentLine = "a: b: c: d ";  
StringTokenizer st = new StringTokenizer(CurrentLine, ":", false);  
while (st.hasMoreTokens())  
    System.out.println(st.nextToken() + "|");
```

a|  
b|  
c|  
d |

il token non contiene il  
delimitatore



public class **StringTokenizer** extends Object implements Enumeration

public **StringTokenizer**(String str, String delim, boolean returnTokens)

Constructs a string tokenizer for the specified string. The characters in the delim argument are the delimiters for separating tokens. If the returnTokens flag is true, then the delimiter characters are also returned as tokens. Each delimiter is returned as a string of length one. If the flag is false, the delimiter characters are skipped and only serve as separators between tokens.

public boolean **hasMoreTokens**() Tests if there are more tokens available from this tokenizer's string.

public String **nextToken**() Returns the next token from this string tokenizer. Throws: NoSuchElementException if there are no more tokens in this tokenizer's string.

public boolean **hasMoreElements**()

Returns the same value as the hasMoreTokens method. It exists so that this class can implement the Enumeration interface.

public Object **nextElement**()

Returns the same value as the nextToken method, except that its declared return value is Object rather than String. It exists so that this class can implement the Enumeration interface. Throws: NoSuchElementException if there are no more tokens in this tokenizer's string.

public int **countTokens**() Calculates the number of times that this tokenizer's nextToken method can be called before it generates an exception.

## **Architectural Patterns**

An *architectural pattern* expresses a fundamental structural organization or schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.

## **Design Patterns**

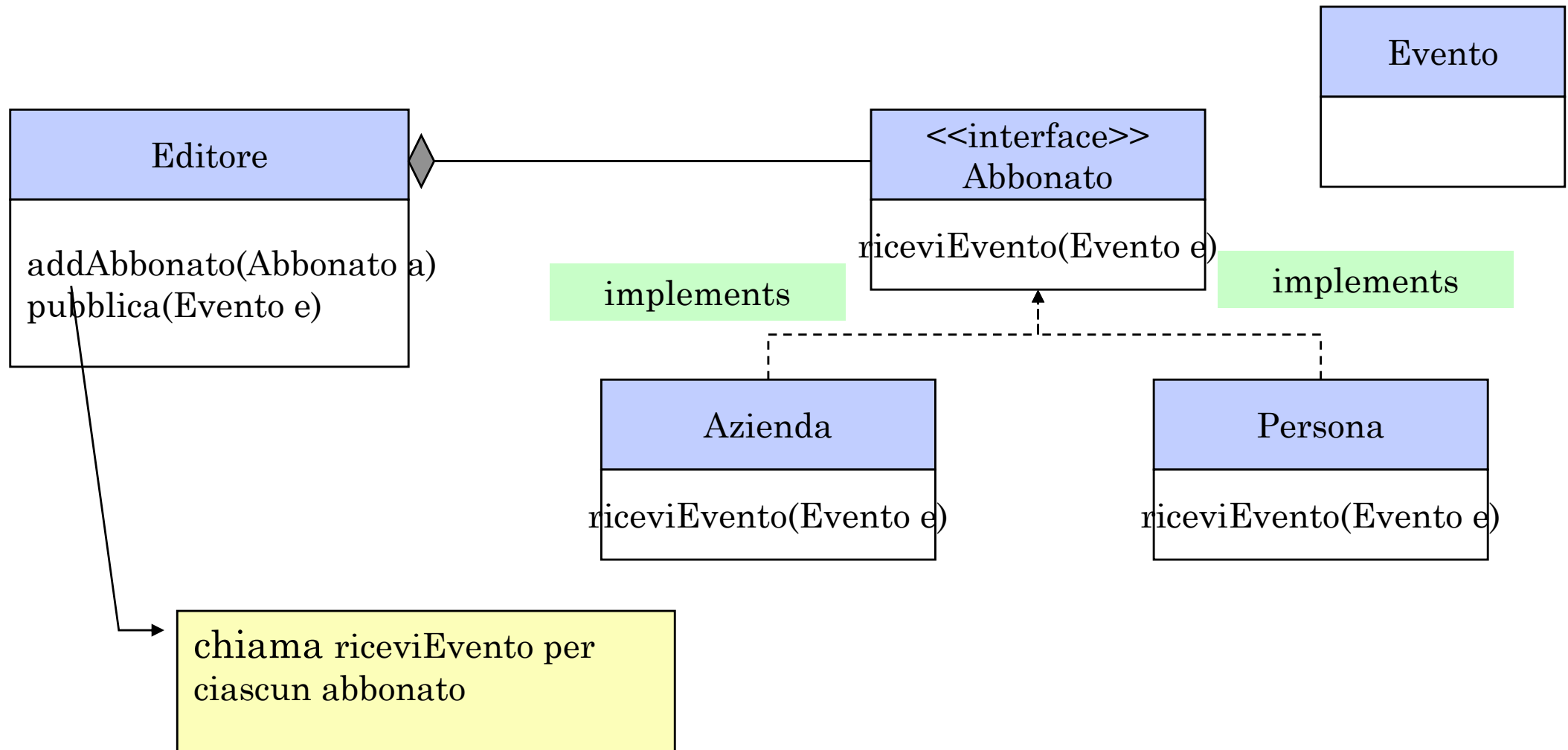
A *design pattern* provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes commonly recurring structure of communicating components that solves a general design problem within a particular context.

## **Idioms**

An *idiom* is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

# *Pattern publish/subscribe*

Esiste una classe che produce eventi di tipo Evento ed esistono altre classi che desiderano ricevere tali eventi.



# *Implementazione*

```
package abbonamenti;
```

```
public interface Abbonato {void riceviEvento(Evento e);}
```

```
public class Evento {private String cont;
```

```
    public Evento(String s) {cont=s;}
```

```
    public String toString() {return cont;}}
```

```
public class Persona implements Abbonato {private String nome;
```

```
    public Persona (String s){nome = s;}
```

```
    public void riceviEvento(Evento e) {
```

```
        System.out.println(nome + ": " + e);}}
```

```
public class Azienda implements Abbonato {private String nome;
```

```
    public Azienda (String s){nome = s;}
```

```
    public void riceviEvento(Evento e) {
```

```
        System.out.println(nome + ": " + e);}}
```

# *Implementazione*

```
import java.util.*;

public class Editore {
    private String nome;
    private ArrayList abbonati = new ArrayList();
    public Editore (String s){nome = s;}
    public void addAbbonato(Abbonato a){abbonati.add(a);}
    public void pubblica(Evento e){
        for (ListIterator i = abbonati.listIterator(); i.hasNext(); ) {
            Abbonato a = (Abbonato) i.next(); a.riceviEvento(e);}
    }
}
```

```
import abbonamenti.*;

public class TestAbbonati {

    public static void main(String[] args) {

        Evento e = new Evento("Evento1");

        Persona p1 = new Persona("p1");

        Azienda a1 = new Azienda("a1");

        Editore e1 = new Editore("e1");

        e1.addAbbonato(p1); e1.addAbbonato(a1);

        e1.pubblica(e);

    }
}
```

p1: Evento1

a1: Evento1



# *Singleton pattern*

Garantisce che di una classe esista un'unica istanza.

## *Singleton pattern*

```
package audioclip;

public class AudioclipManager {
    private static AudioclipManager instance = new AudioclipManager();
    private Audioclip prevClip;
    private AudioclipManager() { } // costruttore privato
    public static AudioclipManager getInstance() {return instance;}
    public void play(Audioclip clip) {
        if (prevClip != null) prevClip.stop();
        prevClip = clip; clip.play();}
    public void stop() {
        if (prevClip != null) {prevClip.stop(); prevClip = null;}}}

package audioclip;

public class Audioclip {
    void play() {System.out.println("play");}
    void stop() {System.out.println("stop");}}
```

```
import audioclip.*;

public class TestAudioclip {

public static void main(String[] args) {

Audioclip a1 = new Audioclip();
Audioclip a2 = new Audioclip();
AudioclipManager m1 = AudioclipManager.getInstance();
// il costruttore non è visibile
m1.play(a1); m1.play(a2);  m1.stop();
}
}
```

play

stop

play

stop

*Exceptions*

# *Exceptions*

The term *exception* is shorthand for the phrase "exceptional event." It can be defined as follows:

**Definition:** An *exception* is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

The runtime system searches backwards through the call stack, beginning with the method in which the error occurred, until it finds a method that contains an appropriate *exception handler*.

# *Eccezioni*

```
public class ProvaEccezioni {
```

```
    public static void main(String[] args) {  
        System.out.println(m1());  
    }
```

```
    static int m1() {  
        int i = Integer.parseInt("abc");  
        return i;  
    }  
}
```

```
java.lang.NumberFormatException: abc  
at java.lang.Integer.parseInt(Integer.java:426)  
at java.lang.Integer.parseInt(Integer.java:476)  
at ProvaEccezioni.m1(ProvaEccezioni.java:7)  
at ProvaEccezioni.main(ProvaEccezioni.java:4)  
Exception in thread "main"
```

# *Eccezioni*

```
public class ProvaEccezioni1 {  
    public static void main(String[] args) {  
        System.out.println(m1("abc"));  
    }  
  
    static int m1(String s1){  
        int i;  
        try {  
            i = Integer.parseInt(s1);  
        } catch(NumberFormatException e){i = -1;}  
        return i;  
    }  
}
```

# *Eccezioni*

```
public class ProvaEccezioni {  
  
    public static void main(String[] args) {  
        try{  
            System.out.println(m1("abc"));  
        } catch(NumberFormatException e){System.out.println("E");}}  
  
    static int m1(String s1){  
        int i = Integer.parseInt(s1);  
        return i;  
    }  
}
```



# *Eccezioni*

La classe dell'eccezione deve essere Throwable oppure una classe derivata.

Un blocco catch può contenere un blocco try/catch.

```
catch (Exception e) {  
    try{  
        System.out.println("Caught exception " + e.getMessage());  
        int total = 0; int correct = 10;  
        int n = correct/total;}  
    catch (Exception f){  
        System.out.println("Caught exception B " + f.getMessage());}  
}}
```

# *Eccezioni*

```
public static void main(String[] args) {  
    try{  
        try{  
            int total = 0; int correct = 10;  
            int n = correct/total;  
        } catch (Exception e) {  
            System.out.println("Caught exception " + e.getMessage());  
            int total = 0; int correct = 10;  
            int n = correct/total;  
        }  
    } catch (Exception f) {  
        System.out.println("Caught exception B " + f.getMessage());}}}
```

# *Grouping Error Types and Error Differentiation*

Because all exceptions that are thrown within a Java program are first-class objects, grouping or categorization of exceptions is a natural outcome of the class hierarchy. Java exceptions must be instances of Throwable or any Throwable descendant. As for other Java classes, you can create subclasses of the Throwable class and subclasses of your subclasses. Each "leaf" class (a class with no subclasses) represents a specific type of exception and each "node" class (a class with one or more subclasses) represents a group of related exceptions.

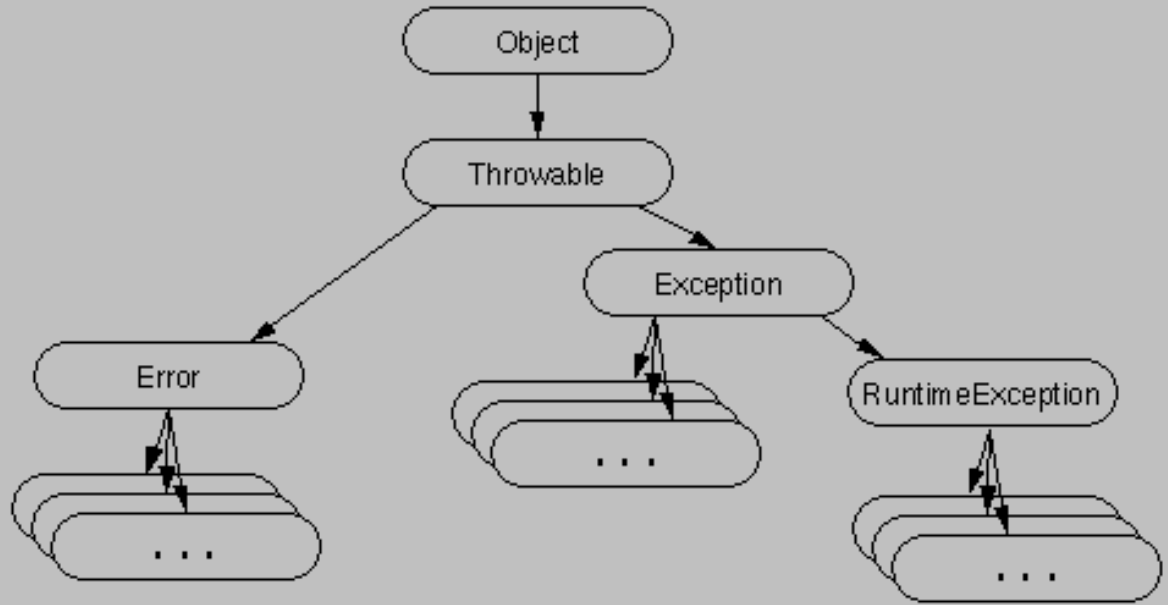
The Throwable Class and Its Subclasses - Microsoft Internet Explorer

File Edit View Go Favorites Help

Address <http://java.sun.com/nav/read/Tutorial/java/exceptions/throwable.html>

Back Links

This diagram illustrates the class hierarchy of the Throwable class and its most significant subclasses.



```
graph TD; Object --> Throwable; Throwable --> Error; Throwable --> Exception; Exception --> RuntimeException; Error --> E1[...]; Exception --> E2[...]; RuntimeException --> R1[...];
```

As you can see from the diagram, Throwable has two direct descendants: Error and Exception.

**Errors**

When a dynamic linking failure or some other "hard" failure in the virtual machine occurs, the virtual machine throws an Error. Typical Java programs should not

Done

*Eccezioni*

**RuntimeException:**  
il compilatore non  
forza il catch

# *Runtime Exceptions*

Runtime exceptions represent problems that are detected by the runtime system. This includes *arithmetic* exceptions (such as when dividing by zero), *pointer* exceptions (such as trying to access an object through a null reference), and *indexing* exceptions (such as attempting to access an array element through an index that is too large or too small).

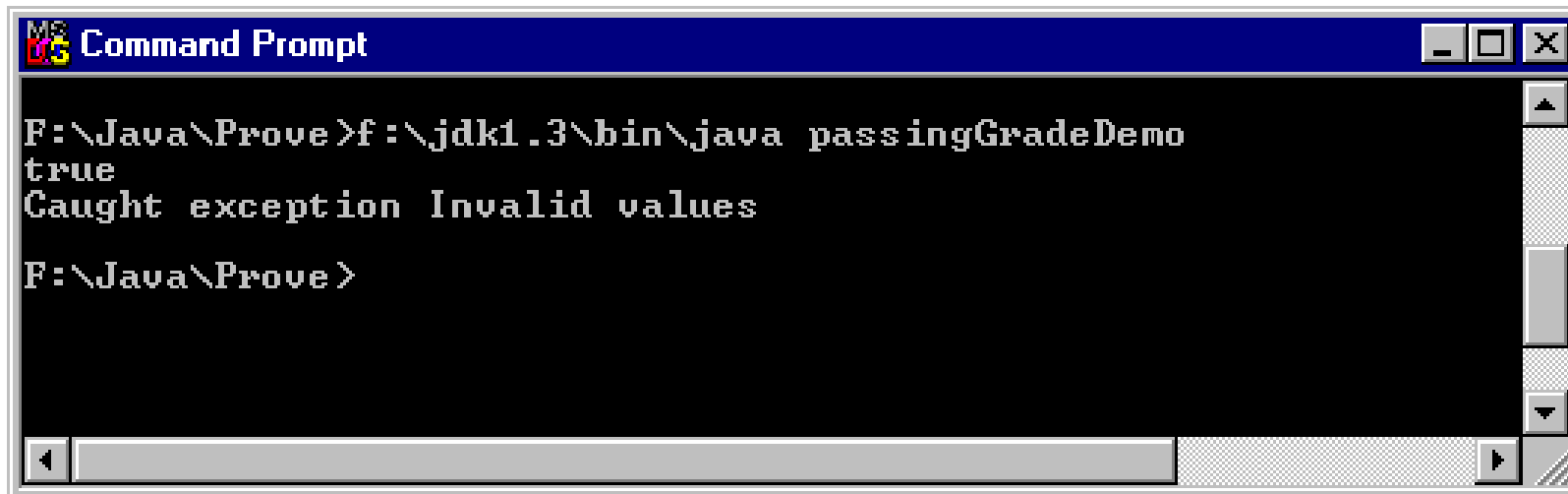
# *Eccezioni*

L'uso delle eccezioni evita di attribuire significati di errore ai risultati dei metodi e costringe il chiamante ad intervenire.

```
static boolean passingGrade(int correct, int total)
throws Exception {
    boolean Result = false;
    if (correct > total) throw new Exception ("Invalid values");
    if ((float)correct / (float)total > 0.7) Result = true;
    return Result;
}
```

```
public static void main (String[] args) {  
    try{  
        System.out.println(passingGrade(60,80));  
        System.out.println(passingGrade(80,60));  
        System.out.println(passingGrade(40,80)); //never gets executed  
    } catch (Exception e) {  
        System.out.println("Caught exception " + e.getMessage());  
    }  
}
```

Output :

A screenshot of a Windows Command Prompt window. The title bar is blue with the text "MS-DOS Command Prompt". The window has standard Windows window controls (minimize, maximize, close) in the top right corner. The command prompt shows the following text:  
F:\Java\Prove>f:\jdk1.3\bin\java passingGradeDemo  
true  
Caught exception Invalid values  
F:\Java\Prove>  
The text is displayed in a monospaced font on a black background. There is a scrollbar on the right side of the window.

**Question:** Is the following code legal?

```
try {  
    ...  
} finally {  
    ...  
}
```

**Question:** What exceptions can be caught by the following handler?

```
...  
} catch (Exception e) {  
    ...  
} catch (ArithmeticException a) {  
    ...  
}
```



**Question:** Match each situation in the first column with an item in the second column. (error, checked exception , runtime exception , no exception)

1 `int[] A; A[0] = 0;`

2 The Java VM starts running your program, but the VM can't find the Java platform classes. (The Java platform classes reside in `classes.zip` or `rt.jar`.)

3 A program is reading a stream and reaches the end of stream marker.

4 Before closing the stream and after reaching the end of stream marker, a program tries to read the stream again.

**Answer:**

- 3 (runtime exception).
- 1 (error).
- 4 (no exception). When you read a stream, you expect there to be an end of stream marker. You should use exceptions to catch unexpected behavior in your program.
- 2 (checked exception).

The concept of standard input and output streams is a C library concept that has been assimilated into the Java environment. There are three standard streams, all of which are managed by the `java.lang.System` class:

**Standard input--referenced by `System.in`**

Used for program input, typically reads input entered by the user. *I/O*

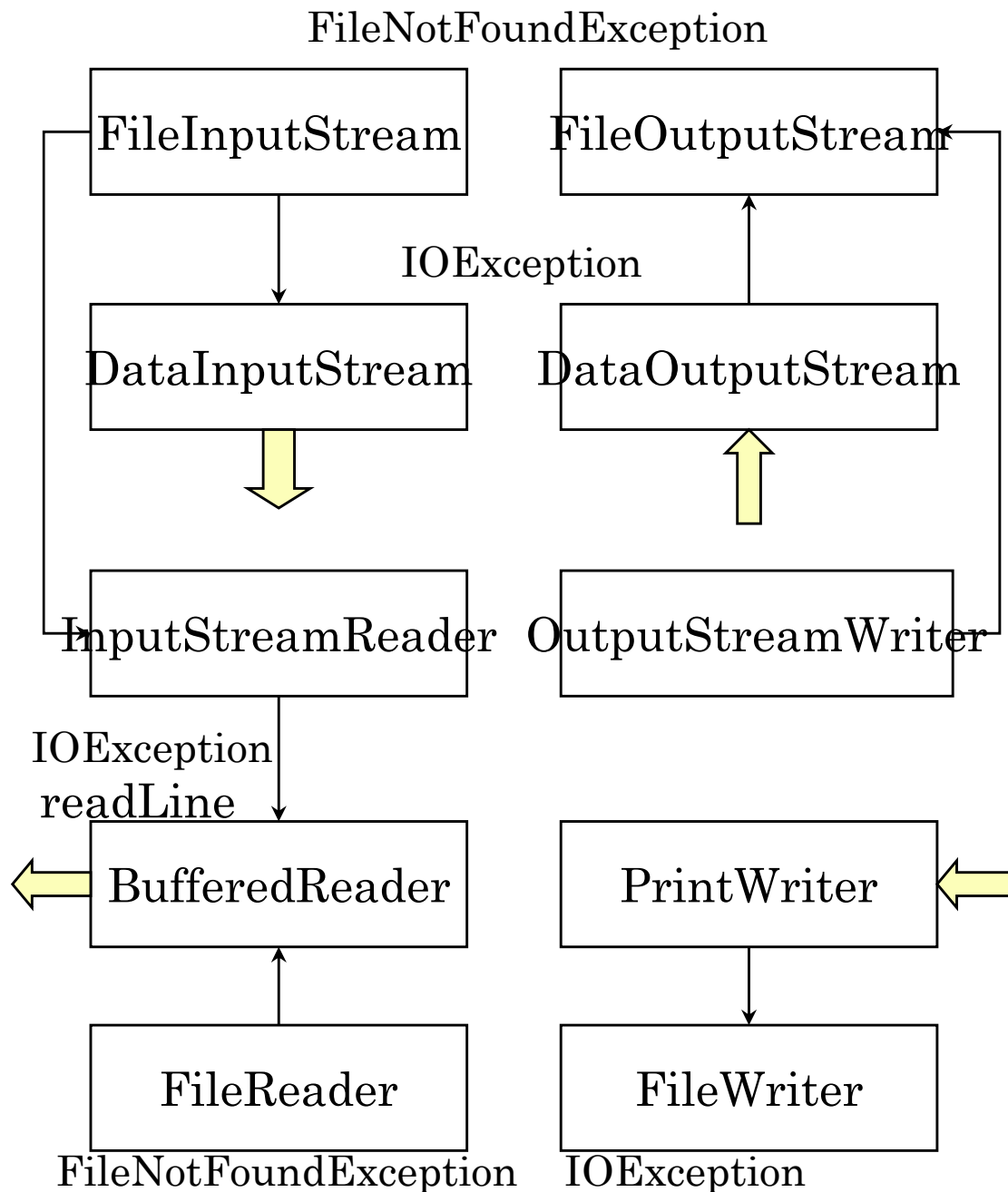
**Standard output--referenced by `System.out`**

Used for program output, typically displays information to the user.

**Standard error--referenced by `System.err`**

Used to display error messages to the user.

# Classi di IO



I/O su file di blocchi di byte

I/O di tipi primitivi  
readInt, readDouble,  
readUTF

writeInt, writeDouble,  
writeUTF

bridge tra byte e caratteri

caratteri - testo

file di caratteri

FileNotFoundException extends  
IOException

# *Binary File I/O*

```
DataOutputStream out = new FileOutputStream(new  
FileOutputStream("invoice1.txt"));
```

```
DataInputStream in = new DataInputStream(new  
FileInputStream("invoice1.txt"));
```

Le classi `FileOutputStream` e `FileInputStream` sono usate per l'apertura dei file; possono produrre eccezioni.

Le classi `DataOutputStream` e `DataInputStream` sono usate per scrivere/leggere valori di tipi predefiniti.

# *How to Use `DataInputStream` and `DataOutputStream`*

```
import java.io.*;

public class DataIOTest {

    public static void main(String[] args) throws IOException {

        DataOutputStream out = new DataOutputStream(new
FileOutputStream("D://Temp//invoice1.txt"));

        double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
        int[] units = { 12, 8, 13, 29, 50 };
        String[] descs = { "Java T-shirt", "Java Mug", "Duke Juggling Dolls",
            "Java Pin", "Java Key Chain" };

        for (int i = 0; i < prices.length; i++) {
            out.writeDouble(prices[i]);
            out.writeInt(units[i]);
            out.writeUTF(descs[i]); //UTF = Unicode Transformation Format
        }

        out.close();
    }
}
```

```
// read it in again

DataInputStream in = new DataInputStream(new
FileInputStream("D://Temp//invoice1.txt"));

double price; int unit; String desc; double total = 0.0;

try {
    while (true) {
        price = in.readDouble();
        unit = in.readInt();
        desc = in.readUTF();

        System.out.println("You've ordered " + unit + " units of
" + desc + " at $" + price);

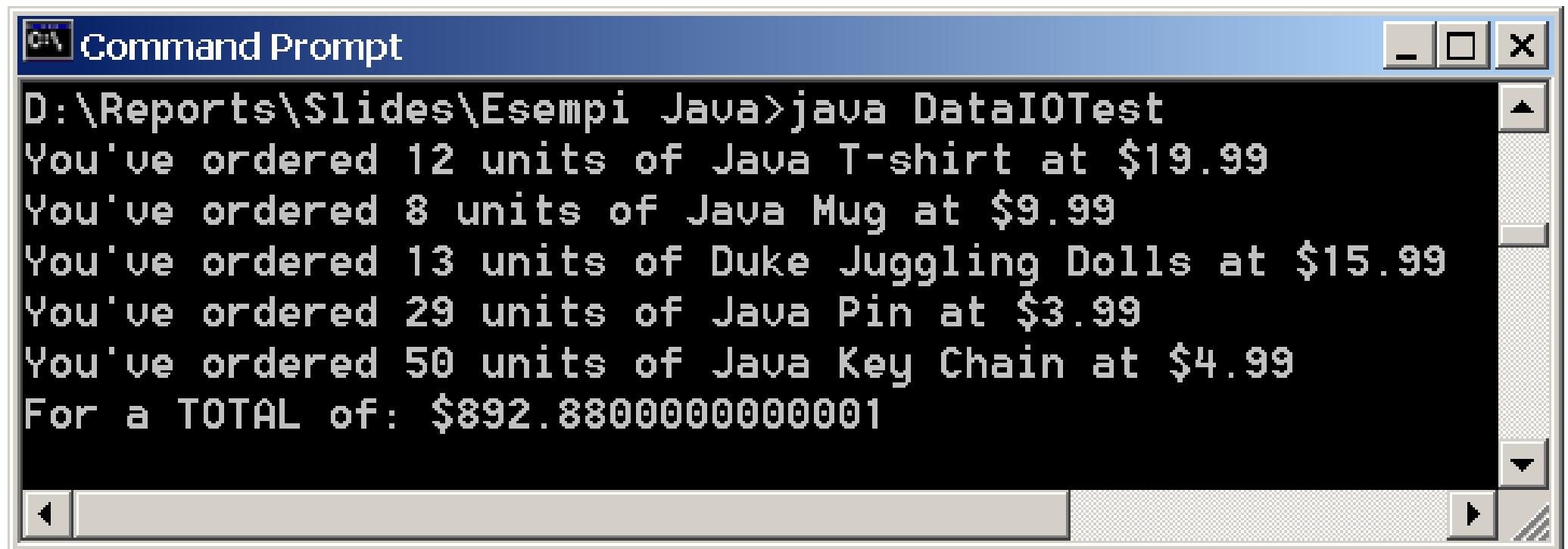
        total = total + unit * price;
    }
} catch (EOFException e) {
}

System.out.println("For a TOTAL of: $" + total);
in.close();    }}
```

This page shows you how to use the java.io [DataInputStream](#) and [DataOutputStream](#) classes. It features an example, [DataIOTest](#), that reads and writes invoices for Java merchandise (sales price, number of units ordered, description of the item):

19.99	12	Java T-shirt
9.99	8	Java Mug

When you run the DataIOTest program you should see the following output:



```
Command Prompt
D:\Reports\Slides\Esempi Java>java DataIOTest
You've ordered 12 units of Java T-shirt at $19.99
You've ordered 8 units of Java Mug at $9.99
You've ordered 13 units of Duke Juggling Dolls at $15.99
You've ordered 29 units of Java Pin at $3.99
You've ordered 50 units of Java Key Chain at $4.99
For a TOTAL of: $892.88000000000001
```

# *Text I/O*

```
import java.io.*;

public class DataIOTest1 {
    public static void main(String[] args) throws IOException {
        BufferedReader keyboard = new BufferedReader(new
InputStreamReader(System.in));

        //read user's input
        String line1 = keyboard.readLine();
        String line2 = keyboard.readLine();

        PrintWriter out = new PrintWriter(new
FileOutputStream("D://Temp//testo1.txt"));

        out.println(line1); out.println(line2); out.close();

        // read it in again

        BufferedReader in = new BufferedReader(new
        FileReader("D://Temp//testo1.txt"));

        line1 = in.readLine(); line2 = in.readLine();
        System.out.println(line1); System.out.println(line2);
        in.close();          }}

```

file  
testo1

hello  
world



# *Text I/O*

Lettura da keyboard

```
BufferedReader keyboard =  
new BufferedReader(new InputStreamReader(System.in));  
//read user's input  
String line1 = keyboard.readLine();
```

Scrittura su file

```
PrintWriter out =  
new PrintWriter(new FileOutputStream("D://Temp//testo1.txt"));  
out.println(line1);  
oppure  
new PrintWriter(new FileWriter("D://Temp//testo1.txt"));
```

Lettura da file

```
BufferedReader in =  
new BufferedReader(new FileReader("D://Temp//testo1.txt"));  
line1 = in.readLine();
```

# *Class FileReader*

public class **FileReader** extends InputStreamReader

Convenience class for reading character files. The constructors of this class assume that the default character encoding and the default byte-buffer size are appropriate. To specify these values yourself, construct an `InputStreamReader` on a `FileInputStream`.

public `FileReader`(String fileName) throws FileNotFoundException

public class **InputStreamReader**  
extends Reader

## *Class InputStreamReader*

An `InputStreamReader` is a bridge from byte streams to character streams: It reads bytes and translates them into characters according to a specified character encoding. The encoding that it uses may be specified by name, or the platform's default encoding may be accepted.

public `InputStreamReader`(InputStream in)

Create an `InputStreamReader` that uses the default character encoding.

public int `read()` throws IOException

Read a single character. Returns: The character read, or -1 if the end of the stream has been reached. Throws: IOException If an I/O error occurs.

public int `read(char cbuf[], int off, int len)` throws IOException

Read characters into a portion of an array.

public boolean `ready()` throws IOException

Tell whether this stream is ready to be read. An `InputStreamReader` is ready if its input buffer is not empty, or if bytes are available to be read from the underlying byte stream.

public void `close()` throws IOException Close the stream.

# *Class **BufferedReader***

public class **BufferedReader** extends Reader

Read text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines. The buffer size may be specified, or the default size may be used. The default is large enough for most purposes.

public **BufferedReader**(Reader in)

Create a buffering character-input stream that uses a default-sized input buffer.

public String **readLine()** throws IOException

Read a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed. Returns: A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached

public void **close()** throws IOException

Close the stream.

public class **FileWriter** extends OutputStreamWriter *Class FileWriter*

Convenience class for writing character files. The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable. To specify these values yourself, construct an `OutputStreamWriter` on a `FileOutputStream`.

public `FileWriter`(String fileName) throws IOException

public `FileWriter`(String fileName, boolean append) throws IOException

public class **FileOutputStream** extends OutputStream

A file output stream is an output stream for writing data to a `File` or to a `FileDescriptor`.

public `FileOutputStream`(String name) throws IOException

Creates an output file stream to write to the file with the specified name.

public `FileOutputStream`(String name, boolean append) throws IOException

# *Class PrintWriter*

public class **PrintWriter** extends Writer

Print formatted representations of objects to a text-output stream. This class implements all of the print methods found in `PrintStream`. It does not contain methods for writing raw bytes, for which a program should use unencoded byte streams. Unlike the `PrintStream` class, if automatic flushing is enabled it will be done only when one of the `println()` methods is invoked, rather than whenever a newline character happens to be output. The `println()` methods use the platform's own notion of line separator rather than the newline character.

Methods in this class never throw I/O exceptions. The client may inquire as to whether any errors have occurred by invoking `checkError()`.

public **PrintWriter**(Writer out)

Create a new `PrintWriter`, without automatic line flushing.

public void **print**(int i)

Print an integer.

public void **print**(float f)

Print a float.

public void **print**(String s)

Print a `String`.

public void **println**()

Finish the line.

public void **println**(int x)

Print an integer, and then finish the line.

public void **println**(String x) Print a `String`, and then finish the line.

public void **close**() Close the stream.

# *Class DataInputStream*

public class **DataStream**

extends FilterInputStream

implements DataInput

A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a data output stream to write data that can later be read by a data input stream.

public DataInputStream(InputStream in)

Creates a new data input stream to read data from the specified input stream.



public final char **readChar()** throws [IOException](#)

Reads a Unicode character from this data input stream. This method reads two bytes from the underlying input stream. If the bytes read, in order, are b1 and b2, where  $0 \leq b1$ ,  $b1 \leq 255$ , then the result is equal to:  $(\text{char})((b1 \ll 8) \mid b2)$

This method blocks until either the two bytes are read, the end of the stream is detected, or an exception is thrown. Throws: [EOFException](#) if this input stream reaches the end before reading two bytes. Throws: [IOException](#) if an I/O error occurs.

public final int **readInt()** throws [IOException](#)

Reads a signed 32-bit integer from this data input stream. This method reads four bytes from the underlying input stream.

public final float **readFloat()** throws [IOException](#) Reads a float from this data input stream.

public final [String](#) **readLine()** throws [IOException](#)

**Note: readLine() is deprecated.** *This method does not properly convert bytes to characters. As of JDK 1.1, the preferred way to read lines of text is via the `BufferedReader.readLine()` method.*

public static final [String](#) **readUTF([DataInput](#) in)** throws [IOException](#)

Reads in a string from the specified data input stream. The string has been encoded using a modified UTF-8 format.

# *Class DataOutputStream*

public class **DataOutputStream** extends FilterOutputStream implements DataOutput

A data input stream lets an application write primitive Java data types to an output stream in a portable way. An application can then use a data input stream to read the data back in.

public **DataOutputStream**(OutputStream out)

Creates a new data output stream to write data to the specified underlying output stream.

public final void **writeInt**(int v) throws IOException

Writes an int to the underlying output stream as four bytes, high byte first.

public final void **writeFloat**(float v) throws IOException

Converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the underlying output stream as a 4-byte quantity, high byte first.

public final void **writeUTF**(String str) throws IOException

Writes a string to the underlying output stream using UTF-8 encoding in a machine-independent manner.

# *Class FileInputStream*

public class **FileInputStream** extends InputStream

A file input stream is an input stream for reading data from a File or from a FileDescriptor.

public **FileInputStream**(String name) throws FileNotFoundException

Creates an input file stream to read from a file with the specified name.

public int **read**(byte b[], int off, int len) throws IOException

Reads up to len bytes of data from this input stream into an array of bytes. This method blocks until some input is available.

public native void **close**() throws IOException

Closes this file input stream and releases any system resources associated with the stream.

# *Class FileOutputStream*

public class **FileOutputStream** extends OutputStream

A file output stream is an output stream for writing data to a File or to a FileDescriptor.

public **FileOutputStream**(String name) throws IOException

Creates an output file stream to write to the file with the specified name.

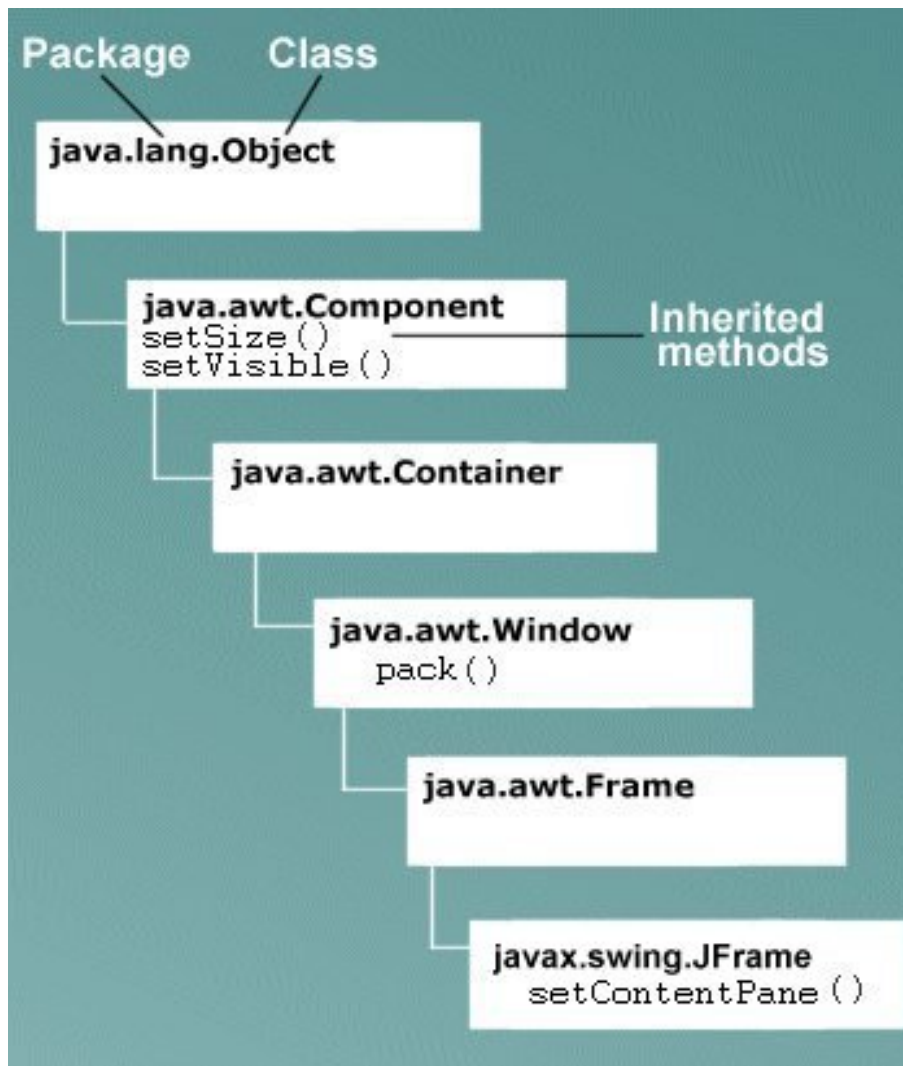
public void **write**(byte b[], int off, int len) throws IOException

Writes len bytes from the specified byte array starting at offset off to this file output stream.

public native void **close**() throws IOException

Closes this file output stream and releases any system resources associated with this stream.

# *awt e swing*



Event-driven programming



SwingApplication creates four commonly used Swing components:

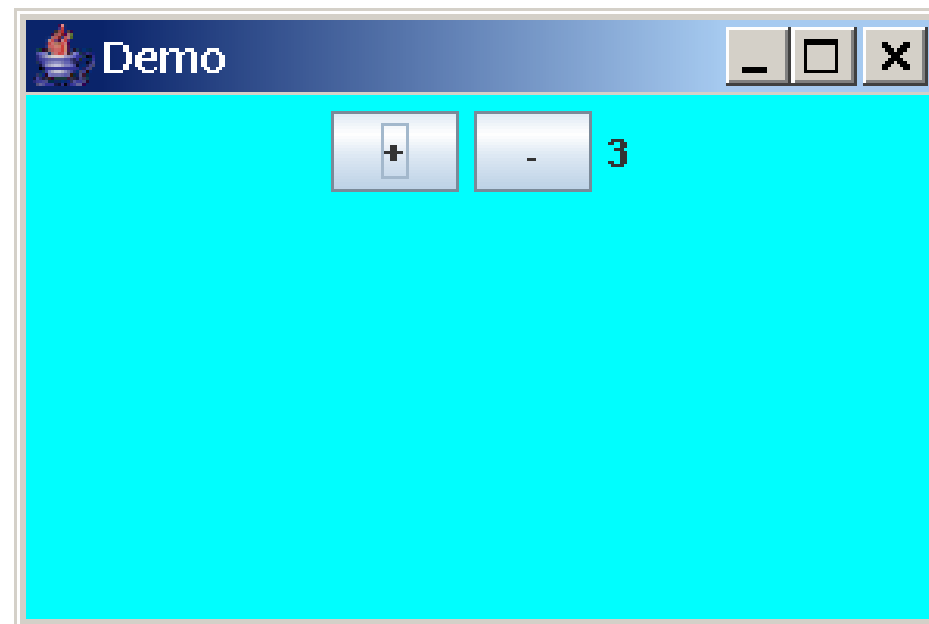
- a *frame*, or main window (JFrame)
- a *panel*, sometimes called a *pane* (JPanel)
- a button (JButton)
- a label (JLabel)

The frame is a *top-level container*. It exists mainly to provide a place for other Swing components to paint themselves. The other commonly used top-level containers are dialogs (JDialog) and applets (JApplet).

The panel is an *intermediate container*. Its only purpose is to simplify the positioning of the button and label. Other intermediate Swing containers, such as scroll panes (JScrollPane) and tabbed panes (JTabbedPane), typically play a more visible, interactive role in a program's GUI.

The button and label are *atomic components* -- components that exist not to hold random Swing components, but as self-sufficient entities that present bits of information to the user. Often, atomic components also get input from the user. The Swing API provides many atomic components, including combo boxes (JComboBox), text fields (JTextField), and tables (JTable).

# *Esempio*



# *ButtonDemo*

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ButtonDemo extends JFrame implements ActionListener {
    public static final int WIDTH = 300;
    public static final int HEIGHT = 200;
    int val = 0;
    JLabel labelV;

    public static void main(String[] args) {
        ButtonDemo gui = new ButtonDemo();
        gui.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        gui.setVisible(true);
    }
}
```



## *ButtonDemo*

```
public ButtonDemo()  
{  
    setSize(WIDTH, HEIGHT);  
    setTitle("Demo");  
    Container contentPane = getContentPane();  
    contentPane.setBackground(Color.cyan);  
    contentPane.setLayout(new FlowLayout());  
    JButton buttonP = new JButton("+");  
    buttonP.addActionListener(this);  
    contentPane.add(buttonP);  
    JButton buttonM = new JButton("-");  
    buttonM.addActionListener(this);  
    contentPane.add(buttonM);  
    JLabel labelV = new JLabel("0");  
    contentPane.add(labelV);  
}
```

# *ButtonDemo*

```
public void actionPerformed(ActionEvent e)
{
    if (e.getActionCommand().equals("+"))
        labelV.setText(String.valueOf(++val));
    else if (e.getActionCommand().equals("-"))
        labelV.setText(String.valueOf(--val));
}
}
```

java.awt.event

Interface *ActionListener* - All Superinterfaces: *EventListener*

void **actionPerformed**(ActionEvent e)                      Invoked when an action occurs.

java.awt.event

Class **ActionEvent**

String getActionCommand()

Returns the command string associated with this action.

int getModifiers()

Returns the modifier keys held down during this action event.

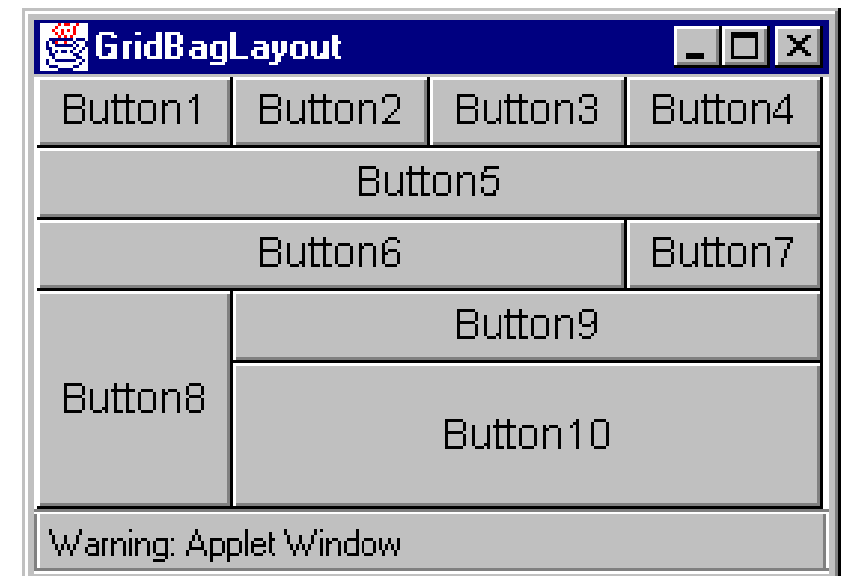
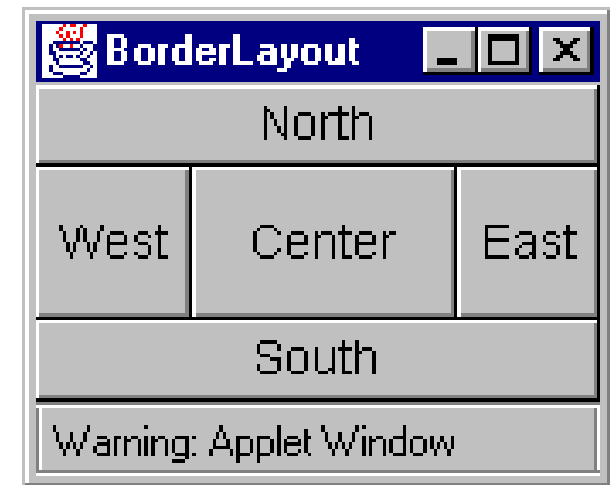
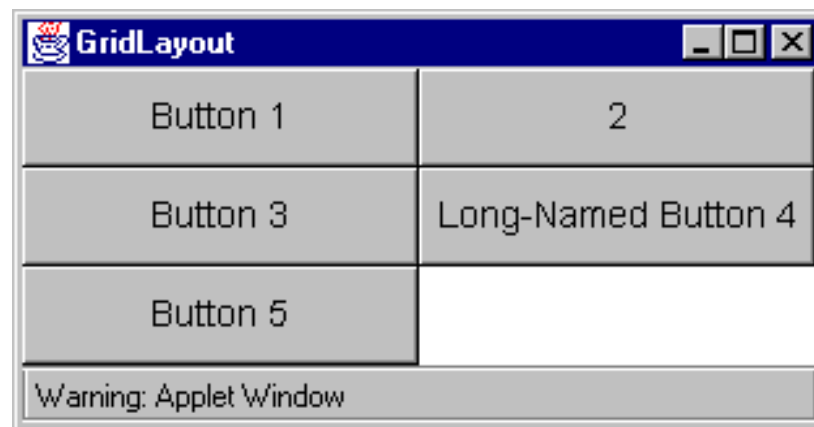
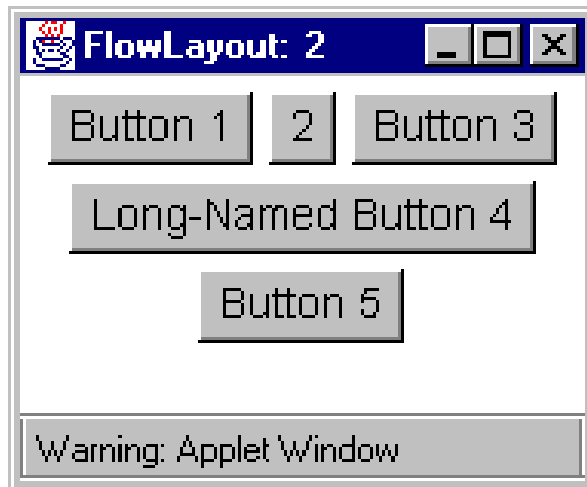
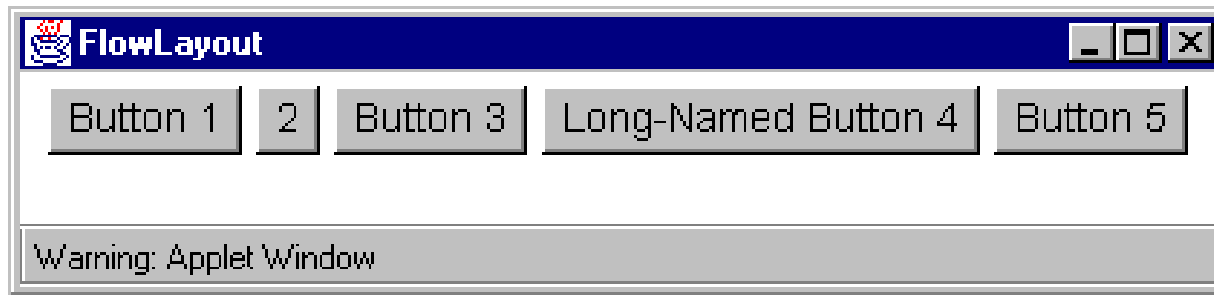
long getWhen()

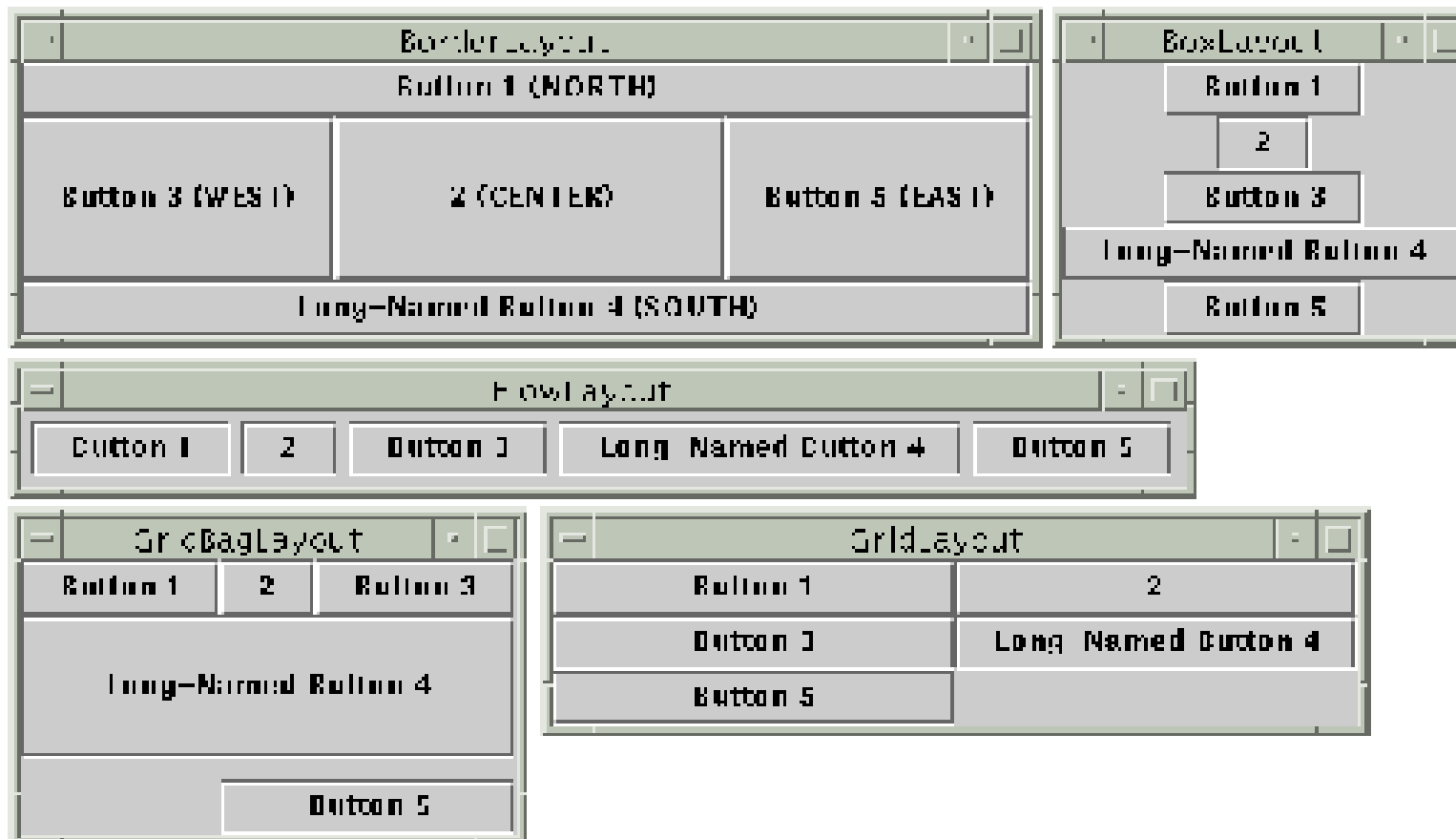
Returns the timestamp of when this event occurred.

String paramString()

Returns a parameter string identifying this action event.

# *Layout principali*





*Layout management* is the process of determining the size and position of components. By default, each container has a *layout manager* -- an object that performs layout management for the components within the container. The Java platform supplies five commonly used layout managers: BorderLayout, BoxLayout, FlowLayout, GridBagLayout, and GridLayout.

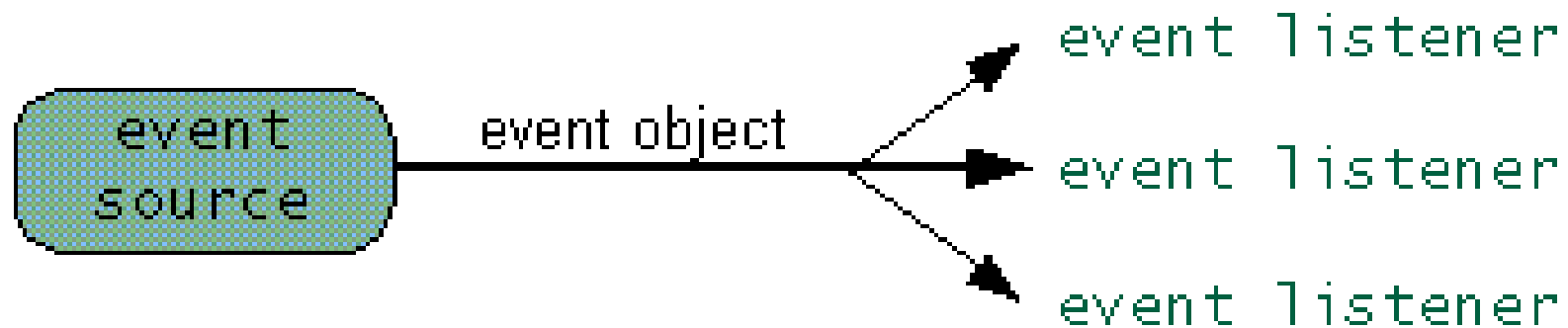
# *Event handling*

Every time the user types a character or pushes a mouse button, an event occurs. Any object can be notified of the event. All it has to do is implement the appropriate interface and be registered as an *event listener* on the appropriate *event source*.

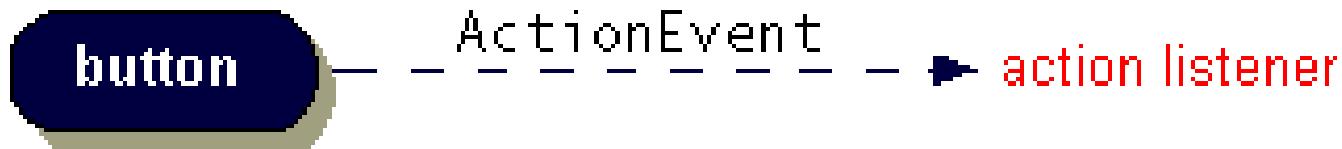
Swing components can generate many kinds of events.

<b>Act that results in the event</b>	<b>Listener type</b>
User clicks a button, presses Return while typing in a text field, or chooses a menu item	ActionListener
User closes a frame (main window)	WindowListener
User presses a mouse button while the cursor is over a component	MouseListener
User moves the mouse over a component	MouseMotionListener
Component becomes visible	ComponentListener
Component gets the keyboard focus	FocusListener
Table or list selection changes	ListSelectionListener

Each event is represented by an object that gives information about the event and identifies the event source. Event sources are typically components, but other kinds of objects can also be event sources. As the following figure shows, each event source can have multiple listeners registered on it. Conversely, a single listener can register with multiple event sources.



**Caption: Multiple listeners can register to be notified of events of a particular type from a particular source.**



**Caption: When the user clicks a button, the button's action listeners are notified.**

Event-handling code executes in a single thread, the *event-dispatching thread*. This ensures that each event handler will finish executing before the next one executes. For instance, the `actionPerformed` method in the preceding example executes in the event-dispatching thread. Painting code also executes in the event-dispatching thread. This means that while the `actionPerformed` method is executing, the program's GUI is frozen -- it won't repaint or respond to mouse clicks, for example.



# *Painting*

When a Swing GUI needs to paint itself -- whether for the first time, in response to becoming unhidden, or because it needs to reflect a change in the program's state -- it starts with the highest component that needs to be repainted and works its way down the containment hierarchy. This process is orchestrated by the AWT painting system, and made more efficient and smooth by the Swing repaint manager and double-buffering code. In this way, each component paints itself before any of the components it contains. This ensures that the background of a JPanel, for example, is visible only where it isn't covered by painting performed by one of the components it contains.

**1. background  
(if opaque)**



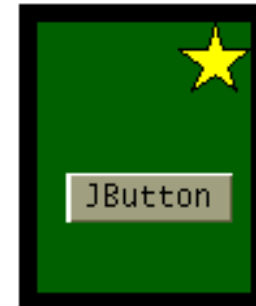
**2. custom  
painting  
(if any)**



**3. border  
(if any)**



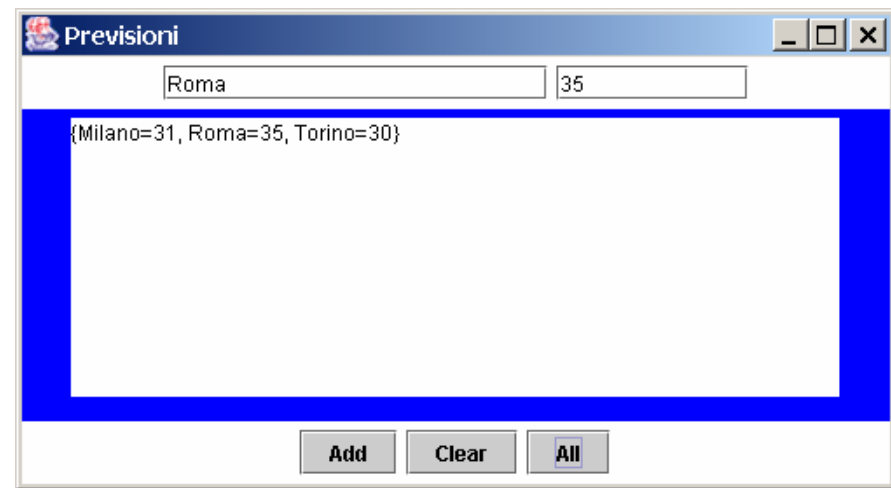
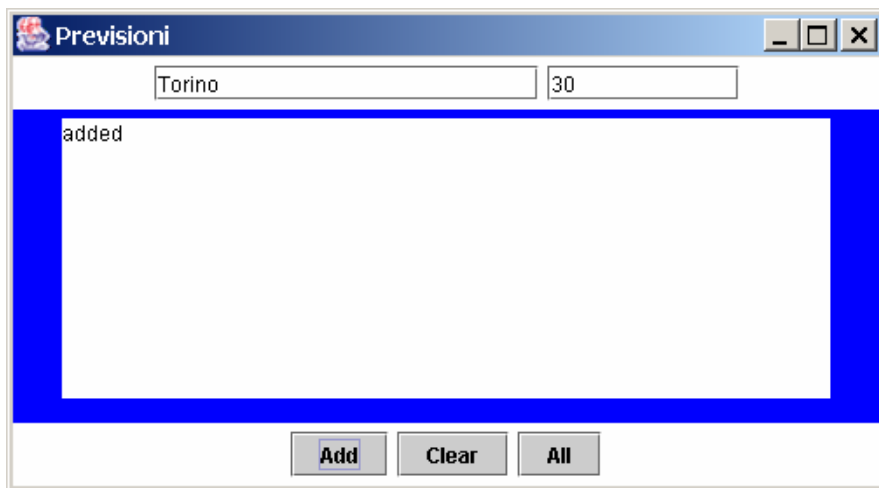
**4. children  
(if any)**



# *Previsioni*

Si progetti un'interfaccia grafica che consenta l'inserimento di una serie di temperature per varie città; le temperature sono valori interi. Il comando Clear cancella gli elementi già inseriti; il comando All visualizza l'elenco ordinato per città.

(1) Esempio di inserimento di una coppia città – temperatura (se l'inserimento è accettato appare added nell'area di testo). (2) Esempio di risultato del comando All.



# *Previsioni*

```
import javax.swing.*; import java.awt.*; import java.awt.event.*;
import java.util.Map; import java.util.TreeMap;
public class Previsioni extends JFrame implements ActionListener
{
    public static final int WIDTH = 600;
    public static final int HEIGHT = 300;
    public static final int LINES = 10;
    public static final int CHAR_PER_LINE = 40;
    private JTextArea theText; private JTextField townTF;
    private JTextField tTF; private Map tMap = new TreeMap();
    public static void main(String[] args)
    {Previsioni gui = new Previsioni(); gui.setVisible(true);
      gui.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);}
```

# *Previsioni*

```
private void addButton(String label, JPanel panel, ActionListener  
listener){  
    JButton button = new JButton(label);  
    button.addActionListener(listener); panel.add(button);}
```

```
private JTextField addTF(String label, int length, JPanel panel){  
    JTextField tf = new JTextField(label,length);  
    tf.setBackground(Color.white); panel.add(tf); return tf;}
```

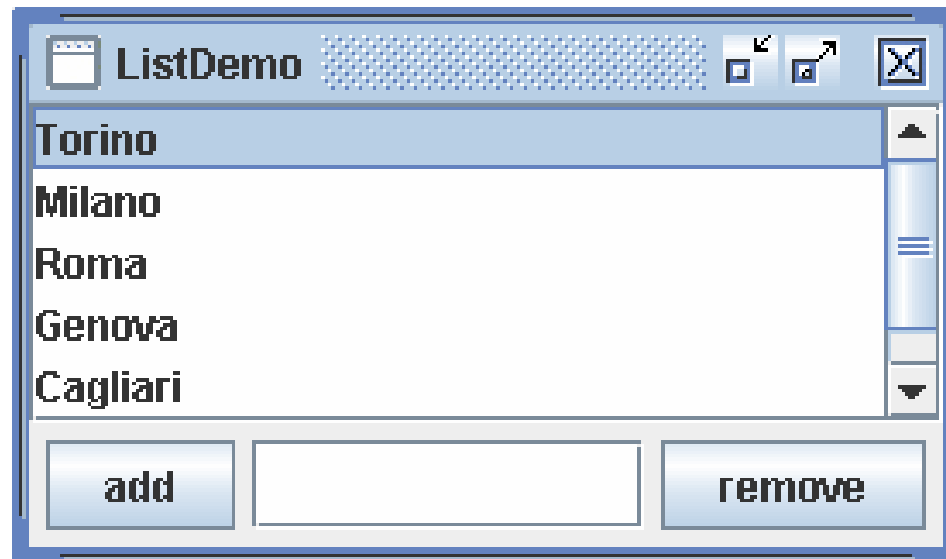
## *Previsioni*

```
public Previsioni() {setSize(WIDTH, HEIGHT);  
    setTitle("Previsioni");  
    Container contentPane = getContentPane();  
    JPanel buttonPanel = new JPanel(); buttonPanel.setBackground(Color.white);  
    addButton("Add",buttonPanel,this); addButton("Clear",buttonPanel,this);  
    addButton("All",buttonPanel,this);  
    contentPane.add(buttonPanel, BorderLayout.SOUTH);  
    JPanel textPanel = new JPanel(); textPanel.setBackground(Color.blue);  
    theText = new JTextArea(LINES, CHAR_PER_LINE);  
    theText.setBackground(Color.white); textPanel.add(theText);  
    contentPane.add(textPanel, BorderLayout.CENTER);  
    JPanel textFPanel = new JPanel(); textFPanel.setBackground(Color.white);  
    townTF = addTF(" ", 20, textFPanel); tTF = addTF(" ", 10, textFPanel);  
    contentPane.add(textFPanel, BorderLayout.NORTH); }
```

## *Previsioni*

```
public void actionPerformed(ActionEvent e) {  
    String actionCommand = e.getActionCommand();  
    theText.setText("");  
    if (actionCommand.equals("Add")){  
        String town = townTF.getText();  
        if (town != null && !town.equals("")){  
            try{tMap.put(town, Integer.valueOf(tTF.getText()));  
                theText.setText("added");}  
            catch(NumberFormatException ex){}  
        }  
    }  
    else if (actionCommand.equals("Clear"))  
        tMap.clear();  
    else if (actionCommand.equals("All")){  
        theText.setText(tMap.toString());}} }
```

# *ListDemo*



```
import javax.swing.*;
import java.awt.*;

// esempio Sun rivisto

public class ListDemo extends JFrame {
    JButton removeButton; JList list;
    DefaultListModel listModel;
    JTextField text;

    public static void main(String[] args) {
        //Make sure we have nice window decorations.
        JFrame.setDefaultLookAndFeelDecorated(true);
        ListDemo frame = new ListDemo();
        frame.setTitle("ListDemo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //Display the window.
        frame.pack(); frame.setVisible(true);
    }
}
```



```
public ListDemo() {  
    listModel = new DefaultListModel();  
    listModel.addElement("Torino");listModel.addElement("Milano");  
    listModel.addElement("Roma");listModel.addElement("Genova");  
    listModel.addElement("Cagliari");listModel.addElement("Firenze");  
    //      Create the list and put it in a scroll pane.  
    list = new JList(listModel);  
    list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);  
    list.setSelectedIndex(0);  
    list.setVisibleRowCount(5);  
    JScrollPane listScrollPane = new JScrollPane(list);  
    JButton addButton = new JButton("add");  
    JButton removeButton = new JButton("remove");  
    text = new JTextField(10);
```

```
ListDemoListener listener =  
    new ListDemoListener(removeButton, list, listModel, text);  
addButton.addActionListener(listener);  
removeButton.addActionListener(listener);  
//      Create a panel that uses BoxLayout.  
JPanel buttonPane = new JPanel();  
buttonPane.setLayout(new BoxLayout(buttonPane, BoxLayout.LINE_AXIS));  
buttonPane.add(addButton);  
buttonPane.add(Box.createHorizontalStrut(5));  
buttonPane.add(text);  
buttonPane.add(Box.createHorizontalStrut(5));  
buttonPane.add(removeButton);  
buttonPane.setBorder(BorderFactory.createEmptyBorder(5, 5, 5, 5));  
add(listScrollPane, BorderLayout.CENTER);  
add(buttonPane, BorderLayout.PAGE_END);}}
```

# *ListDemoListene r*

```
import javax.swing.*;

import java.awt.*; import java.awt.event.*;

public class ListDemoListener implements ActionListener {
    JButton removeButton; JList list;
    DefaultListModel listModel;
    JTextField text;

    public ListDemoListener(JButton button, JList l,
                             DefaultListModel lM, JTextField t) {
        removeButton = button; list = l;
        listModel = lM; text = t;
    }
}
```

## *ListDemoListener*

```
public void actionPerformed(ActionEvent e) {  
    if (e.getActionCommand().equals("remove")) {  
        int index = list.getSelectedIndex();  
        listModel.remove(index); int size = listModel.getSize();  
        if (size == 0) removeButton.setEnabled(false);  
        else { //Select an index.  
            if (index == listModel.getSize()) index--;  
            list.setSelectedIndex(index);  
            list.ensureIndexIsVisible(index);  
        }  
    }  
}
```

## *ListDemoListener*

```
} else {          // add
    String name = text.getText();
    if (name.equals("") || listModel.contains(name)) {
        Toolkit.getDefaultToolkit().beep();
        text.requestFocusInWindow(); text.selectAll(); return;
    }
    int index = list.getSelectedIndex(); //get selected index
    if (index == -1) { //no selection, so insert at beginning
        index = 0;
    } else { //add after the selected item
        index++;
    }
}
```

# *ListDemoListener*

```
listModel.insertElementAt(name, index);  
text.requestFocusInWindow();  
text.setText("");  
// Select the new item and make it visible.  
list.setSelectedIndex(index);  
list.ensureIndexIsVisible(index);  
if (listModel.getSize() > 0) removeButton.setEnabled(true);  
}  
}  
}
```

# *Threads*

public class **Thread** extends Object implements Runnable

## Threads

A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently. Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new **Thread** object, the new thread has its priority initially set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon.

When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which typically calls the method named **main** of some designated class). The Java Virtual Machine continues to execute threads until either of the following occurs:

- The **exit** method of class **Runtime** has been called and the security manager has permitted the exit operation to take place.
- All threads that are not daemon threads have died, either by returning from the call to the **run** method or by performing the **stop** method.

There are two ways to create a new thread of execution. One is to declare a class to be a subclass of **Thread**. This subclass should override the **run** method of class **Thread**. The other way to create a thread is to declare a class that implements the **Runnable** interface.



# *Scrittura dei Threads*

1. Si definisce una sottoclasse di Thread e si scrive la logica nel metodo **run** (che è chiamato allo **start** del thread).
2. Se una classe implementa l'interfaccia **Runnable** (ossia ha un metodo run) si può associare ad un thread un suo oggetto.

Class X implements Runnable { ... }

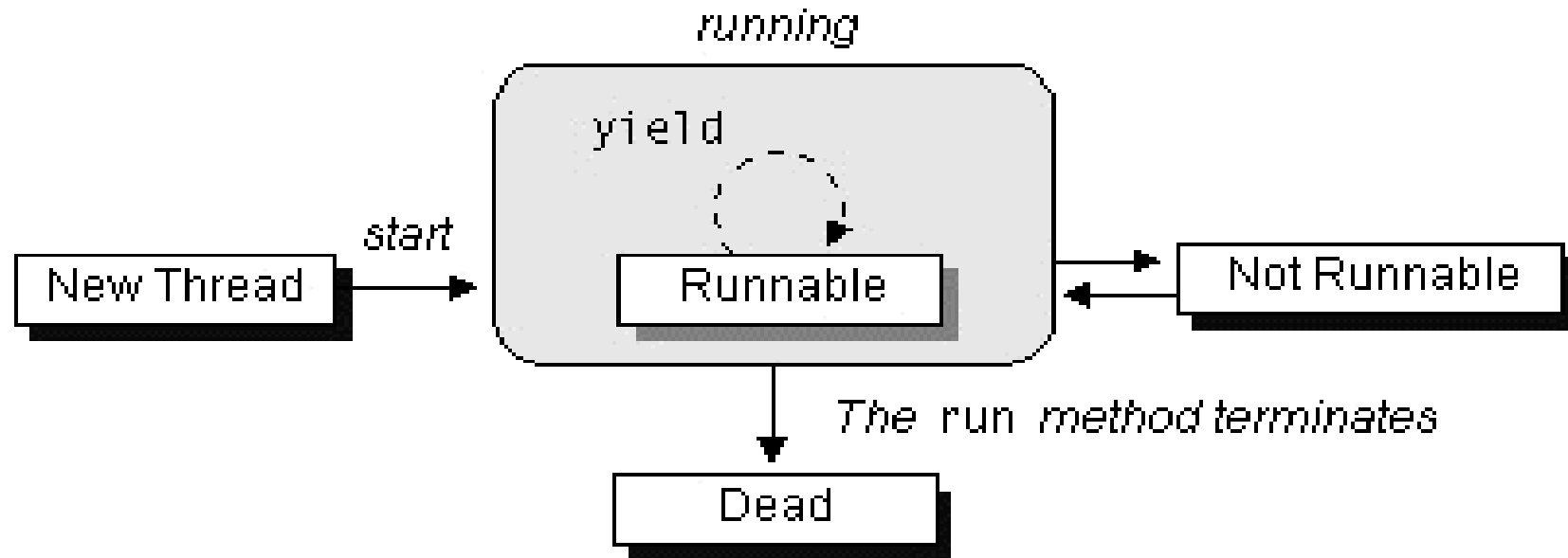
Runnable R = new X(...);

Thread T = new Thread(R);

T.start();

**Definition:** A thread is a single sequential flow of control within a program.

```
public interface Runnable {  
    public void run(); }
```



A thread becomes Not Runnable when one of these events occurs:

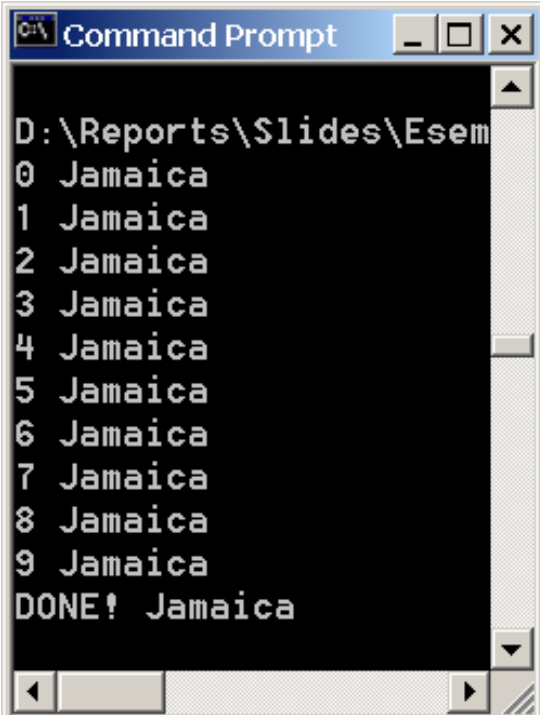
- Its sleep method is invoked.
- The thread calls the wait method to wait for a specific condition to be satisfied.
- The thread is blocking on I/O.

```

public class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str); // sets the Thread's name
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((long)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + getName());
    }
}

public static void main (String[] args) {
    new SimpleThread("Jamaica").start();
}
}

```



```

Command Prompt
D:\Reports\Slides\Esem
0 Jamaica
1 Jamaica
2 Jamaica
3 Jamaica
4 Jamaica
5 Jamaica
6 Jamaica
7 Jamaica
8 Jamaica
9 Jamaica
DONE! Jamaica

```

```
class TwoThreadsTest {
    public static void main (String[] args) {
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();}}
class SimpleThread extends Thread {
    public SimpleThread(String str) {super(str);}
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {sleep((int)(Math.random() * 1000));}
            catch (InterruptedException e) {}}
        System.out.println("DONE! " + getName());}}
```

## *Threads*

### **Output:**

0 Jamaica

0 Fiji

1 Fiji

...

Done!

Jamaica

A thread becomes ***Not Runnable*** when one of these events occurs:

- Its sleep method is invoked.
- The thread calls the wait method to wait for a specific condition to be satisfied.
- The thread is blocking on I/O.

**Thread.sleep(1000);**

### *Stopping a Thread*

A thread arranges for its own death by having a run method that terminates naturally.

# *SimpleThread1*

```
public class SimpleThread1 extends Thread {  
    public SimpleThread1(String str) {super(str); }  
    public void run() {  
        int i = 0;  
        try {  
            while (true){  
                System.out.println(i++ + " " + getName());  
                sleep((long)(Math.random() * 1000));  
            }  
        } catch (InterruptedException e) {  
            System.out.println("Interrupted " + getName());  
        }  
        System.out.println("DONE! " + getName());}}}
```

# *TestSimpleThread1*

```
import java.util.Timer;
import java.util.TimerTask;

public class TestSimpleThread1 extends TimerTask {
    SimpleThread1 st1;
    Timer timer = new Timer();

    public TestSimpleThread1 (SimpleThread1 st1, int seconds){
        this.st1 = st1; timer.schedule(this, seconds*1000); }

    public void run() {
        timer.cancel(); //Terminate the timer thread
        st1.interrupt();
        System.out.println("Time's up!"); }

    public static void main(String[] args) {
        SimpleThread1 st1 = new SimpleThread1("Jamaica");
        st1.start();
        TestSimpleThread1 tst1 = new TestSimpleThread1(st1, 15);}}
```

# *Stampe*

...

27 Jamaica

28 Jamaica

29 Jamaica

30 Jamaica

31 Jamaica

32 Jamaica

33 Jamaica

Interrupted Jamaica

DONE! Jamaica

Time's up!



public class **Timer** extends [Object](#)

A facility for threads to schedule tasks for future execution in a background thread. Tasks may be scheduled for one-time execution, or for repeated execution at regular intervals.

Corresponding to each Timer object is a single background thread that is used to execute all of the timer's tasks, sequentially. Timer tasks should complete quickly. If a timer task takes excessive time to complete, it "hogs" the timer's task execution thread. This can, in turn, delay the execution of subsequent tasks, which may "bunch up" and execute in rapid succession when (and if) the offending task finally completes.

After the last live reference to a Timer object goes away *and* all outstanding tasks have completed execution, the timer's task execution thread terminates gracefully (and becomes subject to garbage collection). However, this can take arbitrarily long to occur. By default, the task execution thread does not run as a *daemon thread*, so it is capable of keeping an application from terminating. If a caller wants to terminate a timer's task execution thread rapidly, the caller should invoke the timer's cancel method.

If the timer's task execution thread terminates unexpectedly, for example, because its stop method is invoked, any further attempt to schedule a task on the timer will result in an `IllegalStateException`, as if the timer's cancel method had been invoked.

This class is thread-safe: multiple threads can share a single Timer object without the need for external synchronization.

This class does *not* offer real-time guarantees: it schedules tasks using the `Object.wait(long)` method.

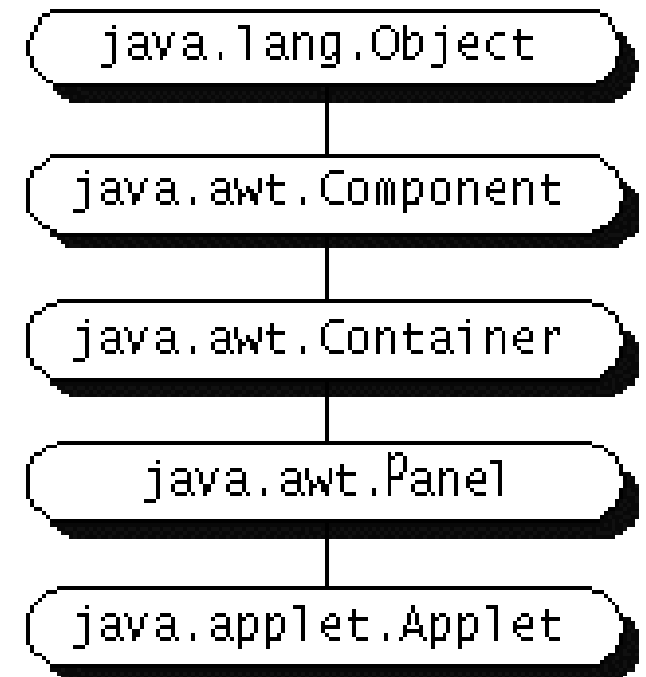
Implementation note: This class scales to large numbers of concurrently scheduled tasks (thousands should present no problem). Internally, it uses a binary heap to represent its task queue, so the cost to schedule a task is  $O(\log n)$ , where  $n$  is the number of concurrently scheduled tasks.

public abstract class **TimerTask** extends [Object](#) implements [Runnable](#)

A task that can be scheduled for one-time or repeated execution by a Timer.

# *Applets*

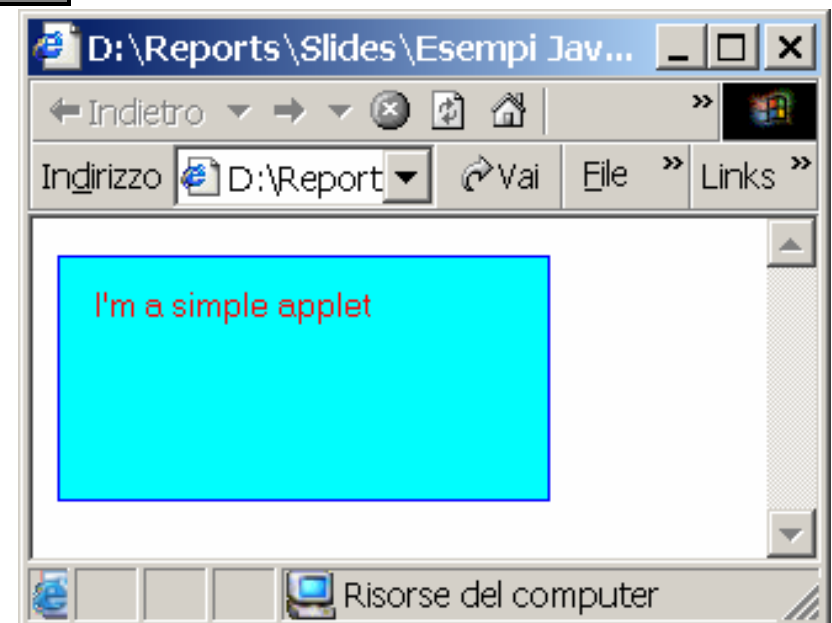
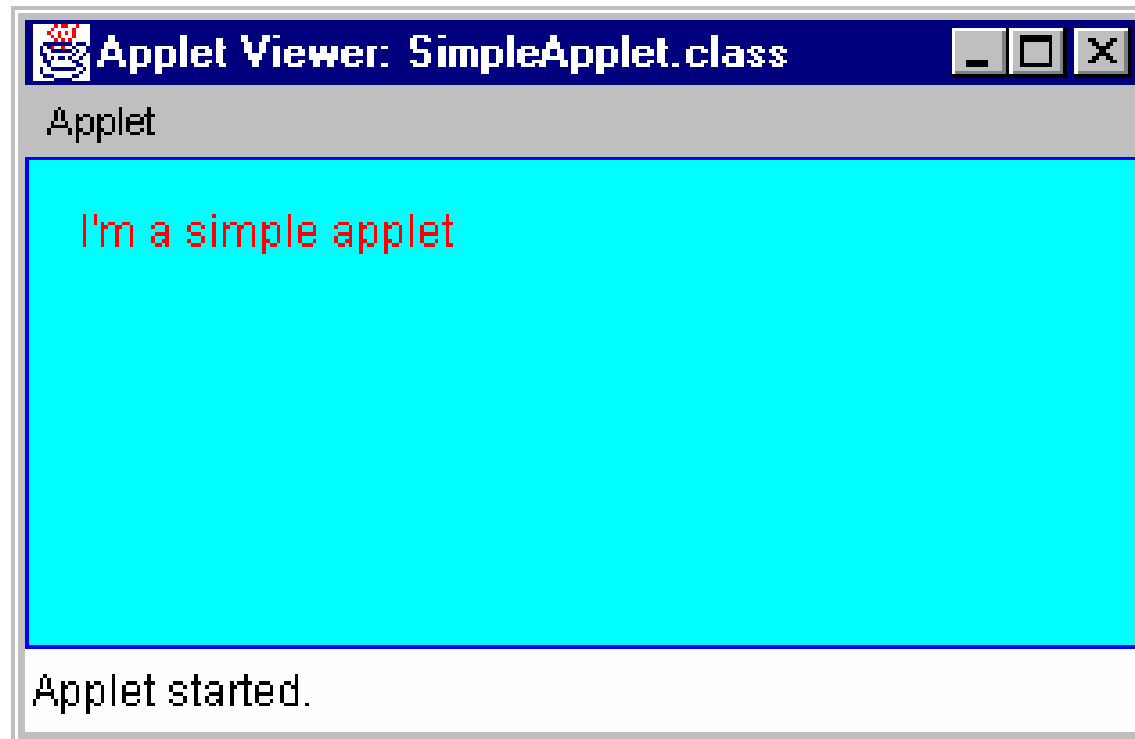
Every applet is implemented by creating a subclass of the Applet class. The following figure shows the inheritance hierarchy of the Applet class. This hierarchy determines much of what an applet can do and how, as you'll see on the next few pages.



**Panel** is the simplest container class. A panel provides space in which an application can attach any other component, including other panels. The default layout manager for a panel is the **FlowLayout** layout manager.

A *component* is an object having a graphical representation that can be displayed on the screen and that can interact with the user. Examples of components are the buttons, checkboxes, and scrollbars of a typical graphical user interface. The Component class is the abstract superclass of the nonmenu-related Abstract Window Toolkit components.

# *Applets*



```
import java.applet.Applet;  
import java.awt.Graphics;  
import java.awt.Color;  
public class SimpleApplet extends Applet{  
    String text = "I'm a simple applet";  
    public void init() {  
        setBackground(Color.cyan);    }  
    public void start() {}  
    public void stop() {}  
    public void destroy() {}  
    public void paint(Graphics g){  
        //System.out.println("Paint");  
        g.setColor(Color.blue);  
        g.drawRect(0, 0, getSize().width -1, getSize().height -1);  
        g.setColor(Color.red);  
        g.drawString(text, 15, 25);  
    }  
}
```

To see the applet in action, you need an HTML file with the Applet tag as follows:

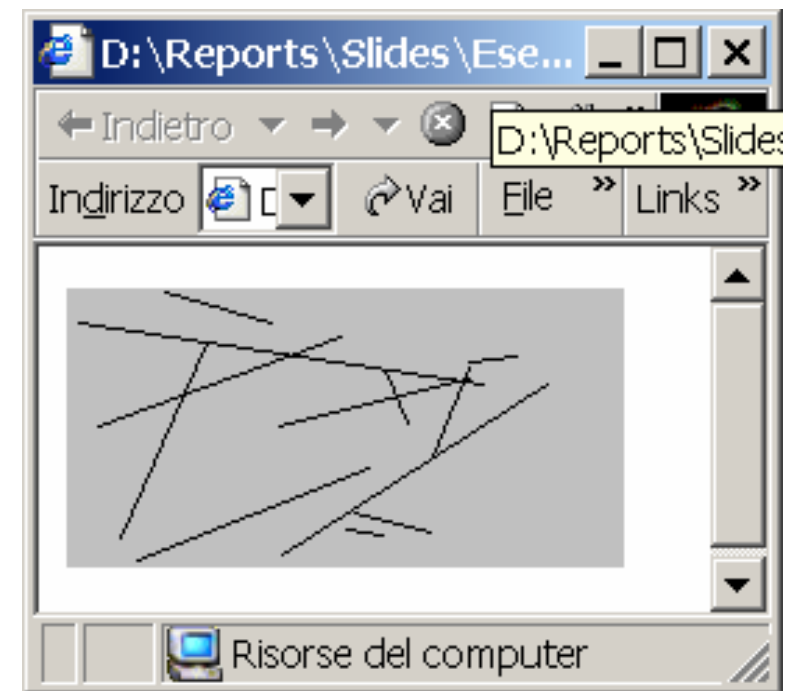
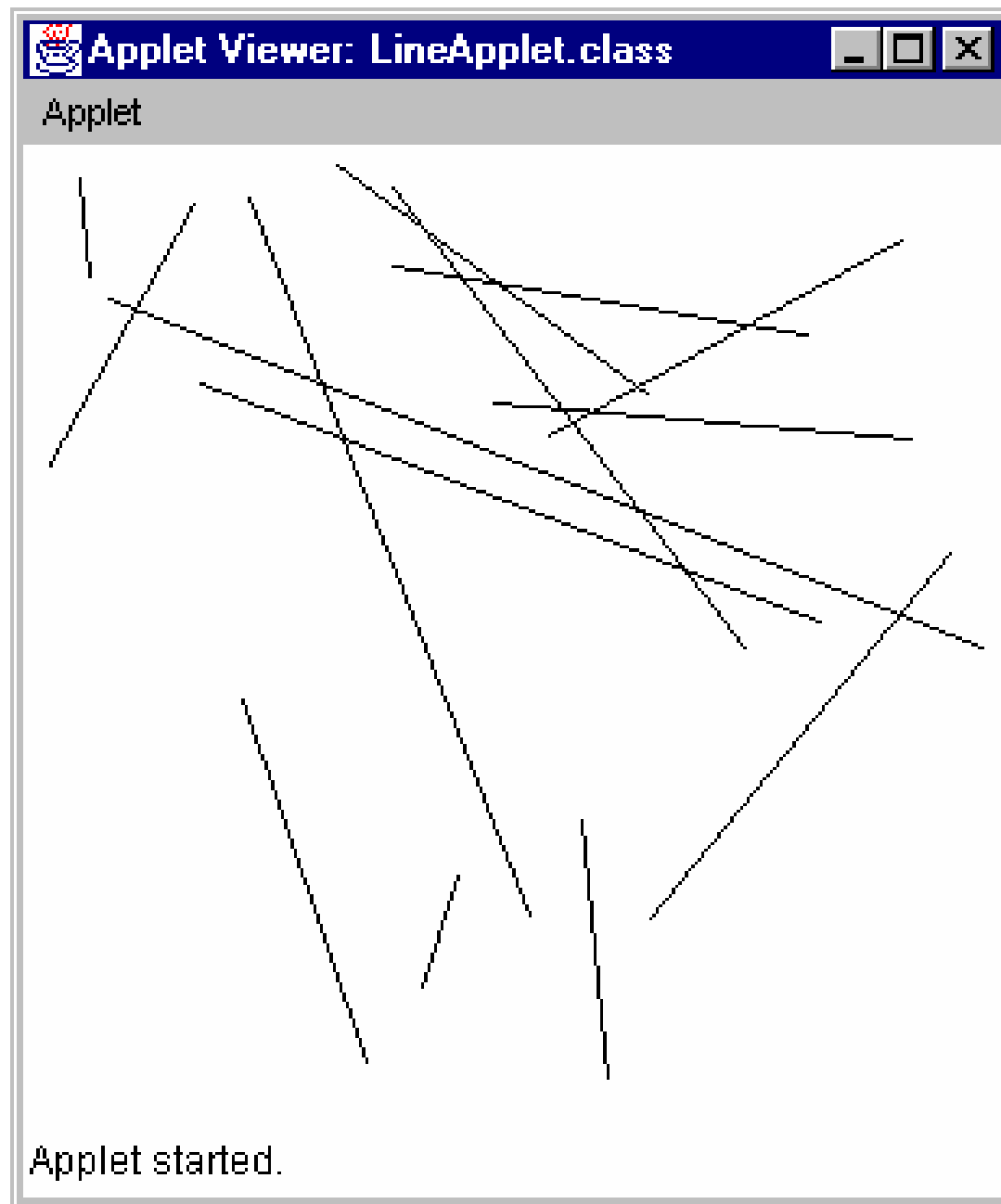
```
<HTML>  
<BODY>  
<APPLET CODE=SimpleApplet.class WIDTH=200 HEIGHT=100>  
</APPLET>  
</BODY>  
</HTML>
```

**The init Method:** The init method is called when the applet is first created and loaded by the underlying software.

**The start Method:** The start method is called when the applet is visited such as when the end user goes to a web page with an applet on it. After the start method executes, the event thread calls the paint method to draw to the applet's Panel.

**The stop and destroy Methods:** The stop method stops the applet when the applet is no longer on the screen such as when the end user goes to another web page. The destroy method is called when the browser exits.

The Graphics object passed to the paint method defines a *graphics context* for drawing on the Panel. The Graphics object has methods for graphical operations such as setting drawing colors, and drawing graphics, images, and text.





protected void **processMouseEvent**(MouseEvent e)

Processes mouse events occurring on this component by dispatching them to any registered `MouseListener` objects.

This method is not called unless mouse events are enabled for this component. Mouse events are enabled when one of the following occurs:

- A `MouseListener` object is registered via `addMouseListener`.
- Mouse events are enabled via `enableEvents`.

public synchronized void **addMouseListener**(MouseListener l)

Adds the specified mouse listener to receive mouse events from this component.

public interface **MouseListener**

**mouseClicked**(`MouseEvent`) Invoked when the mouse has been clicked on a component.

**mouseEntered**(`MouseEvent`) Invoked when the mouse enters a component.

**mouseExited**(`MouseEvent`) Invoked when the mouse exits a component.

**mousePressed**(`MouseEvent`) Invoked when a mouse button has been pressed on a component.

**mouseReleased**(`MouseEvent`) Invoked when a mouse button has been released on a component.

## *LineApplet1*

```
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener; import java.applet.Applet;

// This is an applet that creates a Thread that
// randomly draws lines on its workspace...

public class LineApplet1 extends Applet implements MouseListener {
    LineThread t;

    // Set the size of the applet, create and start the thread
    public void init() {resize(300,300); addMouseListener(this);
        t = new LineThread(this); t.start();}

    // Click the mouse to kill the thread...

    public void mousePressed(MouseEvent arg0) {
        if (t != null) {t.interrupt(); t = null;} ;}

    public void mouseClicked(MouseEvent arg0) {}

    public void mouseReleased(MouseEvent arg0) {}

    public void mouseEntered(MouseEvent arg0) {}

    public void mouseExited(MouseEvent arg0) {}}
```

# *LineThread*

```
import java.awt.*; import java.applet.Applet;
class LineThread extends Thread {
Applet a; // Thread needs to know the applet...
public LineThread(Applet a) {this.a = a;} // stores the applet
public void run() { // Run the thread. Lines everywhere!
    // Get dimension data about the applet...
    double width = (double) a.getSize().width;
    double height = (double) a.getSize().height;
```

# *LineThread*

```
try{
    while (true) {
        Graphics g = a.getGraphics();
        g.drawLine((int)(width * Math.random()),
            (int)(height * Math.random()),
            (int)(width * Math.random()),
            (int)(height * Math.random()) );
        sleep(1000); }
    } catch (InterruptedException e){
        System.out.println("thread interrupted");}
System.out.println("thread terminated");
    } }
```

# *Esempi*

# *Tema n. 1*

```
import competenze.*;
public class TestCompetenze {
public static void main(String[] args) {
String[] v; Org Org1 = new Org();
try{
Competenza c1 = Org1.newCompetenza("java", "programmazione java");
Org1.newCompetenza("c++", "programmazione c++");
Org1.newCompetenza("uml", "modellazione ad oggetti");
Org1.newCompetenza("cs", "progetto sistemi client/server");
Org1.newCompetenza("er", "modellazione er");
String[] v1 = {"java"};
String[] v2 = {"uml", "cs", "java"};
String[] v3 = {"uml", "er"};
String[] v4 = {"uml", "cs", "java", "c++"};
Attività a1 = Org1.newAttività("Analisi s1", v1);
Attività a2 = Org1.newAttività("Progetto s1", v2);
```

# *Tema n. 1*

```
Utente u1 = Org1.newUtente("U1", v3);
Org1.newUtente("U2", v2);
Org1.newUtente("U3", v4);
System.out.println(a2); // Progetto s1[cs, java, uml]
System.out.println(u1); // U1[er, uml]
} catch (OrgEccezione e){System.out.println(e.getMessage());}
System.out.println(Org1.elencoCompetenzeRichieste());
// [java: nA=2 nU=2, cs: nA=1 nU=2, uml: nA=1 nU=3, c++: nA=0 nU=1, er: nA=0 nU=1]

System.out.println(Org1.elencoCompetenzePossedute());
// [uml: nA=1 nU=3, cs: nA=1 nU=2, java: nA=2 nU=2, c++: nA=0 nU=1, er: nA=0 nU=1]

}}
```

# *Tema n. 1*

Si implementino le classi usate nel programma di test precedente; tali classi si trovano nel package competenze.

Note

`newCompetenza nomeCompetenza, descrizione:` genera un'eccezione se `nomeCompetenza` è null oppure è ripetuto;

`newAttività nomeAttività, elenco competenze richieste:` genera un'eccezione se `nomeAttività` è null o è ripetuto oppure se una competenza nell'elenco non è stata definita o è già stata richiesta per l'attività corrente;

`newUtente nomeUtente, elenco competenze possedute:` genera un'eccezione come nel caso precedente;

`elencoCompetenzeRichieste:` stampa l'elenco delle competenze ordinate in modo decrescente in base al numero delle attività richiedenti; a parità di n. richiedenti vale l'ordine alfabetico delle competenze; si usi un oggetto comparatore;

`elencoCompetenzePossedute:` stampa l'elenco delle competenze ordinate in base al numero degli utenti possessori; stesse modalità del caso precedente;

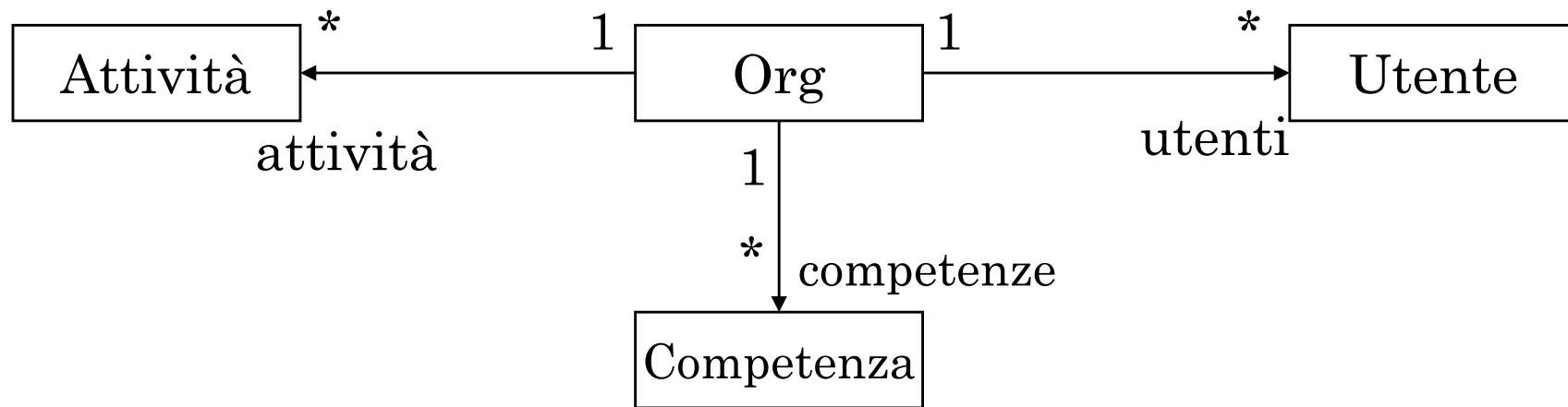
Suggerimento

In ogni oggetto `Competenza` si registrino il n. di attività che la richiedono e il n. di utenti che la posseggono.

Eccezioni: competenza nulla/duplicata/inesistente/già indicata,  
attività nulla/duplicata, utente nullo/duplicato



# *Progetto*



Attributi:

Competenza: String nome, String descrizione, int nA, int nU.

Attività: String nome, TreeSet<String> competenze.

Utente: nome, TreeSet<String> competenze.

Costruttori:

Competenza(String nome, String descrizione)

Attività(String nome, TreeSet<String> competenze)

Utente(String nome, TreeSet<String> competenze)

# *Competenza*

```
package competenze;  
public class Competenza {  
    private String nome; private String descrizione;  
    private int nA = 0; private int nU = 0;  
    Competenza(String nome, String descrizione)  
    {this.nome = nome; this.descrizione = descrizione;}  
    public String toString() {  
        return nome + ":" + " nA=" + nA + " nU=" + nU;}  
    void incrNA() {nA++;}  
    void incrNU() {nU++;}  
    String getNome() {return nome;}  
    int getNA() {return nA;}  
    int getNU() {return nU;}  
}
```

# *Attività*

```
package competenze;

import java.util.*;

public class Attività {
    private String nome;
    private TreeSet<String> competenze;

    Attività(String nome, TreeSet<String> competenze) {
        this.nome = nome; this.competenze = competenze;
    }

    public String toString(){return nome + competenze;}
}
```

# *Utente*

```
package competenze;

import java.util.*;

public class Utente {
    private String nome;
    private TreeSet<String> competenze;
    Utente(String nome, TreeSet<String> competenze) {
        this.nome = nome; this.competenze = competenze;
    }
    public String toString(){return nome + competenze;}}
```

```
package competenze;

import java.util.*;

public class Org {
    private Map<String,Competenza> competenze = new HashMap<String,Competenza>();
    private Map<String,Attività> attività = new HashMap<String,Attività>();
    private Map<String,Utente> utenti = new HashMap<String,Utente>();
    public Competenza newCompetenza(String nome, String descrizione)
    throws OrgEccezione {
        if (nome == null) throw new OrgEccezione("competenza nulla");
        if (competenze.containsKey(nome))
            throw new OrgEccezione("competenza duplicata:" + nome);
        Competenza c = new Competenza(nome, descrizione);
        competenze.put(nome, c); return c;
    }
}
```

```

public Attività newAttività (String nome, String[] v) throws OrgEccezione {
    TreeSet<String> s = new TreeSet<String>();
    if (nome == null) throw new OrgEccezione("attività nulla");
    if (attività.containsKey(nome)) throw new OrgEccezione("attività duplicata:" + nome);
    for (String c:v){
        if (competenze.get(c) == null) throw new OrgEccezione("competenza inesistente:" + c);
        else if (!s.add(c)) throw new OrgEccezione("competenza già richiesta:" + c);
    }
    for (String c:v){competenze.get(c).incrNA();}
    Attività a = new Attività(nome, s); attività.put(nome, a); return a;}

public Utente newUtente (String nome, String[] v) throws OrgEccezione {
    TreeSet<String> s = new TreeSet<String>();
    if (nome == null) throw new OrgEccezione("utente nullo");
    if (utenti.containsKey(nome)) throw new OrgEccezione("utente duplicato:" + nome);
    for (String c:v){ come sopra }
    for (String c:v){competenze.get(c).incrNU();}
    Utente u = new Utente(nome, s); utenti.put(nome, u); return u;}

```

*Org*

```
public String elencoCompetenzeRichieste () {  
    Comparator<Competenza> comparator = new Comparator<Competenza>() {  
        public int compare(Competenza c1, Competenza c2) {  
            if (c2.getNA() == c1.getNA()) return c1.getNome().compareTo(c2.getNome());  
            if (c2.getNA() > c1.getNA()) return 1; else return -1;};  
        List<Competenza> l1 = new ArrayList<Competenza>(competenze.values());  
        Collections.sort (l1, comparator); return l1.toString();}  
    }  
  
    public String elencoCompetenzePossedute () {  
        Comparator<Competenza> comparator = new Comparator<Competenza>() {  
            public int compare(Competenza c1, Competenza c2) {  
                if (c2.getNU() == c1.getNU()) return c1.getNome().compareTo(c2.getNome());  
                if (c2.getNU() > c1.getNU()) return 1; else return -1;};  
                List<Competenza> l1 = new ArrayList<Competenza>(competenze.values());  
                Collections.sort (l1, comparator); return l1.toString();  
            }  
        }  
    }
```

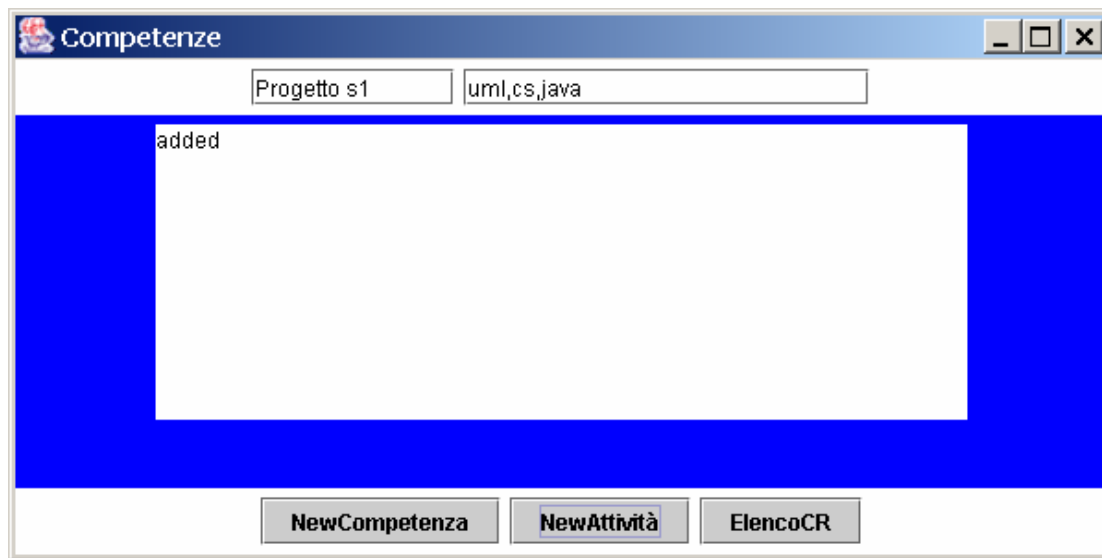
# *OrgEccezione*

package competenze;

**public class OrgEccezione** extends Exception {

public **OrgEccezione**(String s) {super(s);}}





# GUI

Si progetti un'interfaccia grafica che consenta l'inserimento delle competenze e delle attività relative al package di cui al punto 1. Per inserire una competenza si scrive il nome nella prima casella, la descrizione nella seconda e si preme il pulsante NewCompetenza; per inserire un'attività si scrive il nome nella prima casella, l'elenco delle competenze separate da virgole nella seconda e si preme il pulsante NewAttività; per avere l'elenco delle competenze richieste si preme il pulsante ElencoCR. L'interfaccia è realizzata dalla classe Gui che usa il package competenze del punto 1. L'area di testo centrale è usata per mostrare le eccezioni dei primi due comandi e il risultato del terzo.

Nel caso del comando NewAttività si usi uno StringTokenizer per estrarre le competenze dalla stringa (ad es. "uml,cs,java") letta dalla seconda casella di testo per poi collocarle nel vettore da passare al metodo NewAttività della classe Org.

Si indichino gli attributi della classe Gui e si scriva il metodo actionPerformed.

Attributi di Gui

```
private JTextArea ta;  
private JTextField tf1;  
private JTextField tf2;  
private Org org1 = new Org();
```

```
public void actionPerformed(ActionEvent e) {  
    String actionCommand = e.getActionCommand();  
    ta.setText(" ");  
    if (actionCommand.equals("NewCompetenza")) {  
        String n = tf1.getText();  
        String d = tf2.getText();  
        try { org1.newCompetenza(n, d); ta.setText("added"); }  
        catch (OrgEccezione ex) { ta.setText(ex.getMessage()); }  
    }
```

```
else if (actionCommand.equals("NewAttività")) {
    String n = tf1.getText();
    String s = tf2.getText();
    StringTokenizer st = new StringTokenizer(s, ",");
    String[] v = new String[st.countTokens()];
    for (int i = 0; st.hasMoreTokens(); i++) {
        v[i] = st.nextToken();
    }
    try {org1.newAttività(n, v); ta.setText("added");}
    catch (OrgEccezione ex) {ta.setText(ex.getMessage());}
}

else if (actionCommand.equals("ElencoCR")) {
    ta.setText(org1.elencoCompetenzeRichieste());
}
}
```

## *Tema n. 2*

```
import biblioteca.*;

public class TestPrestiti {

public static void main(String[] args) {

Biblioteca bib = new Biblioteca();

try{    Libro l1 = bib.newLibro("titolo1", 1);
        bib.newLibro("titolo2", 2);
        bib.newLibro("titolo3", 3);
        Utente u1 = bib.newUtente("utente1", 1);
        bib.newUtente("utente2", 3);

        int c1 = bib.newPrestito("utente1", "titolo1");
        bib.restituzione("utente1", c1);

        int c2 = bib.newPrestito("utente1", "titolo3");
        int c3 = bib.newPrestito("utente2", "titolo1");

    }catch(BiblioEccezione e){System.out.println(e.getMessage());}

System.out.println(bib.graduatoriaLibri());

//[titolo1:2, titolo3:1, titolo2:0]

System.out.println(bib.prestitiInCorso());}}

//[titolo1:100(utente2), titolo3:103(utente1)]
```

Si implementino le classi usate nel programma di test precedente; tali classi si trovano nel package biblioteca. Tutti gli attributi di tali classi devono essere privati.

**newLibro** titolo, nVolumi: genera nVolumi disponibili per il libro dal titolo dato; produce un'eccezione se titolo è null oppure è ripetuto o nVolumi  $\leq 0$ . Ogni volume ha un numero progressivo univoco a partire da 100.

**newUtente** nome, maxVolumi: genera un utente con il nome dato e stabilisce il n. max di volumi che può prendere in prestito simultaneamente; produce un'eccezione se nome è null oppure è ripetuto o maxVolumi  $\leq 0$ .

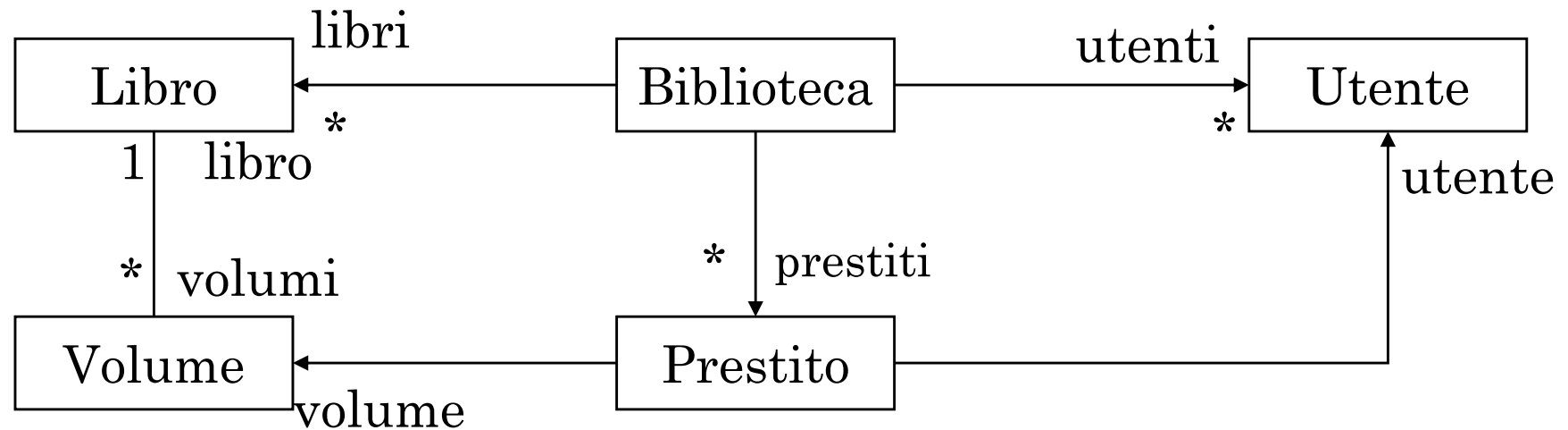
**newPrestito** utente, titolo: genera un prestito e fornisce il codice del volume; produce un'eccezione se utente/titolo è null oppure non esiste o l'utente ha raggiunto il max numero di libri in prestito o non c'è alcun volume disponibile. Un prestito è una coppia di riferimenti: volume, utente.

**restituzione** utente, codice; restituisce il volume con il codice dato; produce un'eccezione se utente è null oppure non esiste o il codice non si riferisce ad un volume preso in prestito dall'utente.

**graduatoriaLibri** fornisce l'elenco dei libri ordinati in base al n. dei prestiti totali e a parità di tale numero in base al titolo; si usi un comparator.

**prestitiInCorso** fornisce l'elenco dei prestiti in corso ordinati in base al titolo e al codice del volume.

# *Progetto*



## Attributi

Libro: String titolo, int nPrestiti.

Volume: int codice, boolean disp.

Utente: String nome, int maxVolumi, int nPrestiti.

cardinalità 1 di  
default

# *Utente*

```
package biblioteca;  
  
public class Utente {  
    private String nome; private int maxVolumi;  
    private int nPrestiti = 0;  
  
    Utente(String nome, int maxVolumi)  
        {this.nome = nome; this.maxVolumi = maxVolumi;}  
  
    String getNome() {return nome;}  
  
    void incPrestiti() {nPrestiti++;}  
  
    void decPrestiti() {nPrestiti--;}  
  
    boolean prestitoAmmissibile () {return nPrestiti < maxVolumi;}  
}
```

```
package biblioteca;
```

```
public class Libro {
```

```
    private String titolo; private int nPrestiti = 0;
```

```
    private Volume[] volumi;
```

```
    Libro(String titolo, int n){
```

```
        this.titolo = titolo; volumi = new Volume[n];
```

```
        for (int i = 0; i < volumi.length; i++)
```

```
            volumi[i] = new Volume(this);}
```

```
    String getTitolo() {return titolo;}
```

```
    int getNPrestiti() {return nPrestiti;}
```

```
    Volume getVolume() {
```

```
        for (int i = 0; i < volumi.length; i++){
```

```
            Volume v = volumi[i];
```

```
            if (v.getDisp()) {v.setDisp(false); nPrestiti++; return v;}
```

```
        }
```

```
        return null;}
```

```
    public String toString() {return titolo + ":" + nPrestiti;}}
```

# *Libro*



# *Volume*

```
package biblioteca;

class Volume {

    private static int nuovoCodice = 100;
    private int codice; private boolean disp = true;
    private Libro libro;

    Volume(Libro l){libro = l; codice = nuovoCodice++;}
    boolean getDisp () {return disp;}
    void setDisp(boolean val) {disp = val;}
    int getCodice() {return codice;}
    Libro getLibro() {return libro;}
}
```

# *Prestito*

```
package biblioteca;
class Prestito implements Comparable<Prestito>{
    private Utente utente;
    private Volume volume;
Prestito(Utente utente, Volume volume)
{this.utente = utente; this.volume = volume;}
Utente getUtente() {return utente;}
Volume getVolume() {return volume;}
public int compareTo(Prestito p) {
    int r = this.volume.getLibro().getTitolo().compareTo(p.volume.getLibro().getTitolo());
    if (r == 0)
        if (this.volume.getCodice() > p.volume.getCodice()) r = 1;
        else r = -1;
    return r;}
public String toString() {return volume.getLibro().getTitolo() + ":" + volume.getCodice() + "(" +
utente.getNome() + ")";}
```

# *Biblioteca*

```
package biblioteca;
import java.util.*;

public class Biblioteca {
    private Map<String,Libro> libri = new HashMap<String,Libro>();
    private Map<String,Utente> utenti = new HashMap<String,Utente>();
    private Map<Integer,Prestito> prestiti = new HashMap<Integer,Prestito>();public Libro
newLibro(String titolo, int nVolumi) throws BiblioEccezione {
        if (titolo == null || nVolumi <= 0) throw new BiblioEccezione
            ("newLibro: dati errati - " + titolo + " " + nVolumi);
        if (libri.containsKey(titolo)) throw new BiblioEccezione
            ("newLibro: libro duplicato - " + titolo);
        Libro l = new Libro(titolo, nVolumi);  libri.put(titolo, l); return l;}
    public Utente newUtente(String nome, int maxVolumi) throws BiblioEccezione {
        if (nome == null || maxVolumi <= 0) throw new BiblioEccezione
            ("newUtente: dati errati - " + nome + " " + maxVolumi);
        if (utenti.containsKey(nome)) throw new BiblioEccezione
            ("newUtente: utente duplicato - " + nome);
        Utente u = new Utente(nome, maxVolumi);utenti.put(nome, u); return u;}
```

```
public int newPrestito(String utente, String titolo) throws BiblioEccezione {  
    if (titolo == null || utente == null)        throw new BiblioEccezione  
        ("newPrestito: dati errati - " + titolo + " " + utente);  
    Utente u = utenti.get(utente);  
    if (u == null) throw new BiblioEccezione  
        ("newPrestito: utente non definito - " + utente);  
    if (!u.prestitoAmmissibile()) throw new BiblioEccezione  
        ("newPrestito: max prestiti per " + utente);  
    Libro l = libri.get(titolo);  
    if (l == null) throw new BiblioEccezione  
        ("newPrestito: dati errati - " + titolo);  
    Volume v = l.getVolume();  
    if (v == null) throw new BiblioEccezione  
        ("newPrestito: volume non disponibile - " + titolo);  
    Prestito p = new Prestito(u, v);  
    int c = v.getCodice(); prestiti.put(c, p);  
    u.incPrestiti(); return c;}  
}
```

# *Biblioteca*

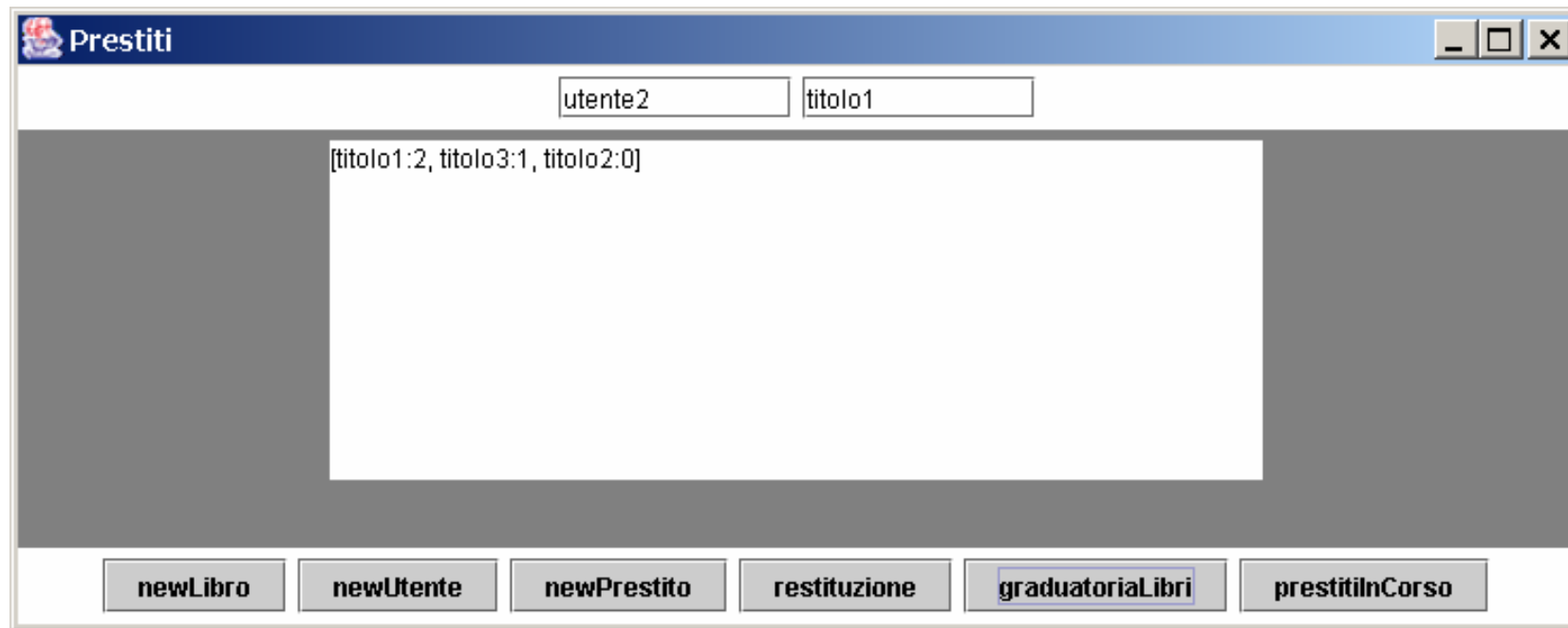
```
public void restituzione(String utente, int codice) throws BiblioEccezione {  
    Utente u = utenti.get(utente);  
    if (u == null) throw new BiblioEccezione  
        ("restituzione: dati errati - " + utente);  
    Prestito p = prestiti.get(codice);  
    if (p == null || u != p.getUtente()) throw new BiblioEccezione  
        ("restituzione: dati errati");  
    u.decPrestiti(); p.getVolume().setDisp(true);  
    prestiti.remove(codice);  
}
```

*Biblioteca*

```
public String graduatoriaLibri() {  
    Comparator<Libro> ordine = new Comparator<Libro>() {  
        public int compare(Libro l1, Libro l2) {  
            if (l2.getNPrestiti() == l1.getNPrestiti())  
                return l1.getTitolo().compareTo(l2.getTitolo());  
            if (l2.getNPrestiti() > l1.getNPrestiti()) return 1;  
            else return -1;  
        };  
    }  
    List<Libro> list1 = new ArrayList<Libro>(libri.values());  
    Collections.sort(list1, ordine); return list1.toString();  
public String prestitiInCorso(){  
    List<Prestito> list1 = new ArrayList<Prestito>(prestiti.values());  
    Collections.sort(list1); return list1.toString();  
}
```

# *BiblioEccezione*

```
package biblioteca;  
public class BiblioEccezione extends Exception {  
    BiblioEccezione(String s) {super(s);}}}
```



*GUI*

Si progetti un'interfaccia grafica (**attributi** e metodo **actionPerformed** della classe GUI) che consenta l'uso del package di cui al punto 1. I pulsanti corrispondono alle operazioni definite al punto 1.

Ad esempio, detti t1 e t2 i valori inseriti nelle due caselle di testo, **newLibro** inserisce il libro con titolo t1 e con n. volumi pari a t2.

Eventuali eccezioni sono mostrate nell'area di testo. L'area di testo mostra la conferma "done" nel caso di newLibro, newUtente e restituzione se l'operazione è andata a buon fine, il codice del prestito nel caso di newPrestito, il risultato della query negli altri casi. La figura precedente presenta il risultato di graduatoriaLibri.



```
private JTextArea ta;  
private JTextField tf1; private JTextField tf2;  
private Biblioteca bib = new Biblioteca();  
  
public static final String LIBRO = "newLibro";  
public static final String UTENTE = "newUtente";  
public static final String PRESTITO = "newPrestito";  
public static final String RESTITUZIONE = "restituzione";  
public static final String INCORSO = "prestitiInCorso";  
public static final String GRAD = "graduatoriaLibri";
```

```
public void actionPerformed(ActionEvent e)    {  
    String actionCommand = e.getActionCommand();  
    ta.setText(""); String s1 = tf1.getText(); String s2 = tf2.getText();  
    try{ if (actionCommand.equals(LIBRO)){  
        bib.newLibro(s1, Integer.parseInt(s2)); ta.setText("done");}  
    else if (actionCommand.equals(UTENTE)) {  
        bib.newUtente(s1, Integer.parseInt(s2)); ta.setText("done");}  
    else if (actionCommand.equals(PRESTITO)) {  
        ta.setText(bib.newPrestito(s1, s2) + "");}  
    else if (actionCommand.equals(RESTITUZIONE)) {  
        bib.restituzione(s1, Integer.parseInt(s2));  
        ta.setText("done"); }  
    else if (actionCommand.equals(GRAD)) {  
        ta.setText(bib.graduatoriaLibri());}  
    else if (actionCommand.equals(INCORSO)) {  
        ta.setText(bib.prestitiInCorso());}  
    }catch(BiblioEccezione ex){ta.setText(ex.getMessage());}  
    catch(NumberFormatException ex){ta.setText(ex.getMessage());}}
```

## *Tema n. 3*

```
import voli.*;

public class TestVoli {

    public static void main(String[] args) {

        try{
            Voli vi = new Voli();
            Aereo a1 = vi.addAereo("a1", "aereo a1", 100);
            Aereo a2 = vi.addAereo("a2", "aereo a2", 150);
            vi.addVolo("v1", "a1", "L1", 8, "L2", 10);
            vi.addVolo("v10", "a1", "L23", 6, "L34", 7);
            vi.addVolo("v2", "a2", "L3", 12, "L4", 14);
            //vi.addVolo("v20", "a1", "L5", 9, "L6", 11); volo sovrapposto
            vi.addPrenotazione("v1", "pass5", 1);
            vi.addPrenotazione("v1", "pass2", 1);
            //vi.addPrenotazione("v1", "pass2", 1); prenotazione duplicata
            VoloGiornaliero vg = vi.getVoloGiornaliero("v1", 1);
            System.out.println(vg); // 1 v1 da L1:8 a L2:10[pass2, pass5]
            System.out.println(a1);

                // a1 aereo a1 100 [v10 da L23:6 a L34:7, v1 da L1:8 a L2:10]

        }catch(VoliEx e){System.out.println(e);}}}
```

Il programma riguarda la gestione delle prenotazioni di voli giornalieri nell'ambito di una settimana.

**addAereo** codiceAereo, descrizione, numeroPosti

genera un aereo e produce un'eccezione se codiceAereo è null o ripetuto o se numeroPosti < 0.

**addVolo** codiceVolo, codiceAereo, luogoPartenza, oraPartenza, luogoArrivo, oraArrivo

produce un'eccezione se codiceVolo è null o ripetuto o se l'aereo non esiste, se i luoghi sono nulli e le ore non sono comprese tra 1 e 24 e oraArrivo non è maggiore di oraPartenza; inoltre produce un'eccezione se tale volo si sovrappone ad un altro volo per lo stesso aereo (detti v1 e v2 due voli per lo stesso aereo, non si ha sovrapposizione se

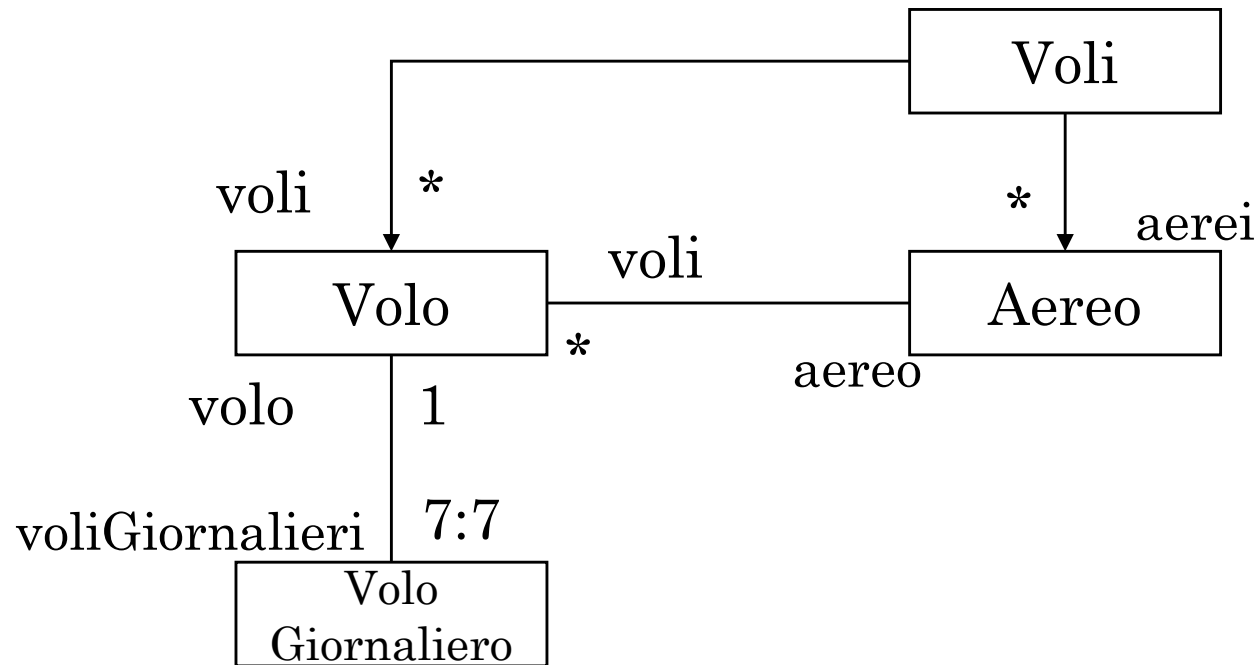
$v1.oraArrivo < v2.oraPartenza$  o  $v1.oraPartenza > v2.oraArrivo$ .

Genera il volo in base al suggerimento seguente: conviene che un oggetto volo, che stabilisce l'orario di un volo, contenga un array di 7 voli giornalieri, perché le prenotazioni si riferiscono ai voli giornalieri.

**addPrenotazione** codiceVolo, passeggero, giorno

produce un'eccezione se il volo non esiste, passeggero è nullo o giorno non è compreso tra 1 e 7 e inoltre se non vi sono posti disponibili sul volo giornaliero nel giorno indicato o la prenotazione è duplicata; aggiunge il passeggero alle prenotazioni del volo giornaliero nel giorno indicato.

# *Progetto*



## Attributi

Aereo: String codiceAereo, String descrizione, int numeroPosti.

Volo: String codiceVolo, String luogoPartenza, String luogoArrivo, int oraPartenza, int oraArrivo.

VoloGiornaliero: int giorno.

cardinalità 1 di  
default

# *Aereo*

```
package voli;
import java.util.*;
public class Aereo {
    private String codiceAereo; private String descrizione;
    private int numeroPosti; private TreeSet voli = new TreeSet();
    Aereo(String codiceAereo, String descrizione, int numeroPosti){
        this.codiceAereo = codiceAereo;
        this.descrizione = descrizione;
        this.numeroPosti = numeroPosti;}
    int getNumeroPosti() { return numeroPosti;}
    void addVolo(Volo volo) throws VoliEx{
        if (!voli.add(volo)) throw new
            VoliEx("sovrapposizione voli:" + volo.getCodiceVolo());
    }
    public String toString(){
        return codiceAereo + " " + descrizione + " " + numeroPosti + " " + voli;}
}
```

# *Volo*

```
package voli;

public class Volo implements Comparable {

    private String codiceVolo;

    private String luogoPartenza; private String luogoArrivo;

    private int oraPartenza; private int oraArrivo;

    private Aereo aereo; private VoloGiornaliero[] voliGiornalieri;

    String getCodiceVolo() {return codiceVolo;}

    Aereo getAereo() {return aereo;}

    Volo(String codiceVolo, Aereo aereo, String luogoPartenza, int oraPartenza, String luogoArrivo,
    int oraArrivo){

        this.codiceVolo = codiceVolo;

        this.aereo = aereo;

        this.luogoPartenza = luogoPartenza; this.oraPartenza = oraPartenza;

        this.luogoArrivo = luogoArrivo; this.oraArrivo = oraArrivo;

        voliGiornalieri = new VoloGiornaliero[7];

        for (int g = 0; g < 7; g++)

            voliGiornalieri[g] = new VoloGiornaliero(this, g);

    }
}
```

# *Volo*

```
public int compareTo(Object o){
    Volo v2 = (Volo) o;
    if (this.oraArrivo < v2.oraPartenza) return -1;
    else if (this.oraPartenza > v2.oraArrivo) return 1;
    else return 0;
}

VoloGiornaliero getVoloGiornaliero(int giorno){
    return voliGiornalieri[giorno];
}

public String toString() {return codiceVolo + " da "
    + luogoPartenza + ":" + oraPartenza + " a " + luogoArrivo
    + ":" + oraArrivo;}}
```



```
package voli;
import java.util.*;
public class VoloGiornaliero {
    private TreeSet prenotazioni; private Volo volo;
    private int giorno;
    VoloGiornaliero(Volo volo, int giorno){
        this.volo = volo; this.giorno = giorno;
        prenotazioni = new TreeSet();}
    public String toString(){
        return giorno + " " + volo.toString() + " " + prenotazioni;}
    void addPrenotazione(String passeggero) throws VoliEx{
        if (prenotazioni.size() == volo.getAereo().getNumeroPosti())
            throw new VoliEx("volo completo:" + volo.getCodiceVolo()
                + ":" + giorno);
        if (!prenotazioni.add(passeggero))
            throw new VoliEx("prenotazione duplicata:" +
                volo.getCodiceVolo() + ":" + giorno + " " + passeggero);
    }
}
```

## *VoloGiornaliero*

# *Voli*

```
package voli;  
import java.util.*;  
public class Voli {  
    private Map aerei = new HashMap(); private Map voli = new HashMap();  
    public Aereo addAereo(String codiceAereo, String descrizione, int numeroPosti) throws VoliEx{  
        if (codiceAereo == null || numeroPosti <= 0  
            || aerei.containsKey(codiceAereo))  
            throw new VoliEx("dati errati in addAereo: " + codiceAereo);  
        Aereo a1 = new Aereo(codiceAereo, descrizione, numeroPosti);  
        aerei.put(codiceAereo, a1); return a1;}  
}
```

# *Voli*

```
public void addVolo(String codiceVolo, String codiceAereo,  
                    String luogoPartenza, int oraPartenza,  
                    String luogoArrivo, int oraArrivo) throws VoliEx {  
    if (codiceVolo == null || voli.containsKey(codiceVolo)  
        || luogoPartenza == null || luogoArrivo == null  
        || oraPartenza < 1 || oraPartenza > 24 || oraArrivo < 1  
        || oraArrivo > 24 || oraPartenza >= oraArrivo  
        || !aerei.containsKey(codiceAereo))  
        throw new VoliEx("dati errati in addVolo: " + codiceVolo);  
    Aereo a = (Aereo)aerei.get(codiceAereo);  
    Volo v1 = new Volo(codiceVolo, a, luogoPartenza, oraPartenza,  
                      luogoArrivo, oraArrivo);  
    a.addVolo(v1);    //verifica sovrapposizione voli su aereo  
    voli.put(codiceVolo, v1);  
}
```

```
public void addPrenotazione(String codiceVolo, String passeggero, int giorno) throws  
VoliEx {  
    getVoloGiornaliero(codiceVolo, giorno).addPrenotazione(passeggero);  
}
```

```
public VoloGiornaliero getVoloGiornaliero(String codiceVolo, int giorno) throws  
VoliEx {  
    if (codiceVolo == null || giorno < 1 || giorno > 7  
        || !voli.containsKey(codiceVolo)) throw new  
        VoliEx("dati errati in getVoloGiornaliero: " + codiceVolo);  
    Volo v1 = (Volo) voli.get(codiceVolo);  
    return v1.getVoloGiornaliero(giorno);  
}}
```

# *VoliEx*

```
package voli;  
public class VoliEx extends Exception {  
    public VoliEx(String s) {super(s);}  
}
```

# *Estensioni della versione 5*

Dalla documentazione sul sito sun

# *Enhanced for Loop*

The Iterator class is used heavily by the Collections API. It provides the mechanism to navigate sequentially through a Collection. The new enhanced for loop can replace the iterator when simply traversing through a Collection as follows. The compiler generates the looping code necessary and with generic types no additional casting is required.

## **Before**

```
ArrayList<Integer> list = new ArrayList<Integer>();  
for (Iterator i = list.iterator(); i.hasNext();) {  
    Integer value=(Integer)i.next();  
}
```

## **After**

```
ArrayList<Integer> list = new ArrayList<Integer>();  
for (Integer i : list) { ... }
```

## *Enhanced for Loop*

```
for (Iterator i = suits.iterator(); i.hasNext(); )  
    for (Iterator j = ranks.iterator(); j.hasNext(); )  
        sortedDeck.add(new Card(i.next(), j.next()));
```

Can you spot the bug? Don't feel bad if you can't. Many expert programmers have made this mistake at one time or another. The problem is that the next method is being called too many times on the “outer” collection (suits). It is being called in the inner loop for both the outer and inner collections, which is wrong. In order to fix it, you have to add a variable in the scope of the outer loop to hold the suit:

```
// Fixed, though a bit ugly  
for (Iterator i = suits.iterator(); i.hasNext(); ) {  
    Suit suit = (Suit) i.next();  
    for (Iterator j = ranks.iterator(); j.hasNext(); )  
        sortedDeck.add(new Card(suit, j.next()));  
}
```

So what does all this have to do with the for-each construct? It is tailor-made for nested iteration! Feast your eyes:

```
for (Suit suit : suits)  
    for (Rank rank : ranks)  
        sortedDeck.add(new Card(suit, rank));
```



# *Varargs*

The varargs functionality allows multiple arguments to be passed as parameters to methods. It requires the simple ... notation for the method that accepts the argument list and is used to implement the flexible number of arguments required for printf.

```
void argtest(Object ... args) {  
  for (int i=0; i <args.length; i++) {  
  }  
}
```

```
argtest("test", "data");
```

# *Autoboxing and Auto-Unboxing of Primitive Types*

Converting between primitive types, like `int`, `boolean`, and their equivalent Object-based counterparts like `Integer` and `Boolean`, can require unnecessary amounts of extra coding, especially if the conversion is only needed for a method call to the Collections API, for example.

The autoboxing and auto-unboxing of Java primitives produces code that is more concise and easier to follow. In the next example an `int` is being stored and then retrieved from an `ArrayList`. The 5.0 version leaves the conversion required to transition to an `Integer` and back to the compiler.

## **Before**

```
ArrayList<Integer> list = new ArrayList<Integer>();  
list.add(0, new Integer(42));  
int total = (list.get(0)).intValue();
```

## **After**

```
ArrayList<Integer> list = new ArrayList<Integer>();  
list.add(0, 42);  
int total = list.get(0);
```

# *Autoboxing*

As you know, the Java programming language has a "split type system": some types are primitives and others are object references. You can't put primitives into collections, so you end up converting back and forth between primitives (such as `int`) and "wrapper types" (such as `Integer`) when you need to store them in collections. Anyone who has done this can attest that it isn't pretty.

For example, take a look at this program, which generates a frequency table of the words found on the command line. It uses a `Map` whose keys are the words and whose values are the number of times that each word occurs on the line:

```
public class Freq {  
    private static final Integer ONE = new Integer(1);  
    public static void main(String args[]) { // Maps word (String) to frequency (Integer)  
        Map m = new TreeMap();  
        for (int i=0; i<args.length; i++) {  
            Integer freq = (Integer) m.get(args[i]);  
            m.put(args[i], (freq==null ? ONE : new Integer(freq.intValue() + 1)));  
        }  
        System.out.println(m);  
    }  
}
```

Notice how messy the inner-loop code that increments the count looks? Now take a look at the same program rewritten with autoboxing, generics, and an enhanced for loop:

# *Autoboxing*

```
public class Freq {  
    public static void main(String args[]) {  
        Map<String, Integer> m = new TreeMap<String, Integer>();  
        for (String word : args) {  
            Integer freq = m.get(word); m.put(word, (freq == null ? 1 : freq +  
1));  
        }  
        System.out.println(m);  
    }  
}
```

## *Static import*

In order to access static members, it is necessary to qualify references with the class they came from. For example, one must say:

```
double r = Math.cos(Math.PI * theta);
```

The static import construct allows unqualified access to static members without inheriting from the type containing the static members. Instead, the program imports the members, either individually:

```
import static java.lang.Math.PI;
```

or en masse:

```
import static java.lang.Math.*;
```

Once the static members have been imported, they may be used without qualification:

```
double r = cos(PI * theta);
```

# Generics

```
List myIntList = new LinkedList(); // 1  
myIntList.add(new Integer(0)); // 2  
Integer x = (Integer) myIntList.iterator().next(); // 3
```

The cast on line 3 is slightly annoying. Typically, the programmer knows what kind of data has been placed into a particular list. However, the cast is essential. The compiler can only guarantee that an Object will be returned by the iterator. To ensure the assignment to a variable of type Integer is type safe, the cast is required.

# Generics

Generics allow you to abstract over types. The most common examples are container types, such as those in the Collections hierarchy.

What if programmers could actually express their intent, and mark a list as being restricted to contain a particular data type? This is the core idea behind generics. Here is a version of the program fragment given above using generics:

```
List<Integer> myIntList = new LinkedList<Integer>(); // 1'  
myIntList.add(new Integer(0)); // 2'  
Integer x = myIntList.iterator().next(); // 3'
```

Notice the type declaration for the variable `myIntList`. It specifies that this is not just an arbitrary `List`, but a `List of Integer`, written `List<Integer>`. We say that `List` is a generic interface that takes a *type parameter*--in this case, `Integer`. We also specify a type parameter when creating the list object.

# Generics

Here is a small excerpt from the definitions of the interfaces List and Iterator in package java.util:

```
public interface List<E>{  
    void add(E x);  
    Iterator<E> iterator();  
}  
  
public interface Iterator<E>{  
    E next();  
    boolean hasNext();  
}
```

This code should all be familiar, except for the stuff in angle brackets. Those are the declarations of the *formal type parameters* of the interfaces List and Iterator.



# *Generics*

```
// Removes the 4-letter words from c
static void expurgate(Collection<String> c) {
    for (Iterator<String> i = c.iterator(); i.hasNext(); )
        if (i.next().length() == 4)
            i.remove();
}
```

# *Generics and Subtyping*

Let's test your understanding of generics. Is the following code snippet legal?

```
List<String> ls = new ArrayList<String>(); // 1
```

```
List<Object> lo = ls; // 2
```

Line 1 is certainly legal. The trickier part of the question is line 2. This boils down to the question: is a List of String a List of Object. Most people instinctively answer, "Sure!"

Well, take a look at the next few lines:

```
lo.add(new Object()); // 3
```

```
String s = ls.get(0); // 4: Attempts to assign an Object to a String!
```

In general, if Foo is a subtype (subclass or subinterface) of Bar, and G is some generic type declaration, it is **not** the case that G<Foo> is a subtype of G<Bar>. This is probably the hardest thing you need to learn about generics, because it goes against our deeply held intuitions.

## *Wild cards*

Consider the problem of writing a routine that prints out all the elements in a collection. Here's how you might write it in an older version of the language (i.e., a pre-5.0 release):

```
void printCollection(Collection c) {  
    Iterator i = c.iterator();  
    for (k = 0; k < c.size(); k++) {System.out.println(i.next());}  
}
```

And here is a naive attempt at writing it using generics (and the new for loop syntax):

```
void printCollection(Collection<Object> c) {  
    for (Object e : c) {System.out.println(e);}  
}
```

The problem is that this new version is much less useful than the old one. Whereas the old code could be called with any kind of collection as a parameter, the new code only takes `Collection<Object>`, which, as we've just demonstrated, is **not** a supertype of all kinds of collections!

## *Wild cards*

So what **is** the supertype of all kinds of collections? It's written `Collection<?>` (pronounced "collection of unknown"), that is, a collection whose element type matches anything. It's called a **wildcard type** for obvious reasons. We can write:

```
void printCollection(Collection<?> c) {  
  for (Object e : c) {System.out.println(e);}  
}
```

# *Bounded Wildcards*

Consider a simple drawing application that can draw shapes such as rectangles and circles. To represent these shapes within the program, you could define a class hierarchy such as this:

```
public abstract class Shape {public abstract void draw(Canvas c);}
public class Circle extends Shape {private int x, y, radius;
    public void draw(Canvas c) {    ...    }}
public class Rectangle extends Shape {private int x, y, width, height;
    public void draw(Canvas c) {    ...    }}
```

These classes can be drawn on a canvas:

```
public class Canvas {
    public void draw(Shape s) {s.draw(this);}}
```

# *Bounded Wildcards*

Any drawing will typically contain a number of shapes. Assuming that they are represented as a list, it would be convenient to have a method in Canvas that draws them all:

```
public void drawAll(List<Shape> shapes) {  
    for (Shape s: shapes) {s.draw(this);}
```

Now, the type rules say that drawAll() can only be called on lists of exactly Shape: it cannot, for instance, be called on a List<Circle>. That is unfortunate, since all the method does is read shapes from the list, so it could just as well be called on a List<Circle>. What we really want is for the method to accept a list of **any** kind of shape:

```
public void drawAll(List<? extends Shape> shapes) {...}
```

There is a small but very important difference here: we have replaced the type List<Shape> with List<? **extends** Shape>. Now drawAll() will accept lists of any subclass of Shape, so we can now call it on a List<Circle> if we want.

# *Bounded Wildcards*

Returning to our shape drawing problem, suppose we want to keep a history of drawing requests. We can maintain the history in a static variable inside class Shape, and have drawAll() store its incoming argument into the history field.

```
static List<List<? extends Shape>> history =  
new ArrayList<List<? extends Shape>>();  
  
public void drawAll(List<? extends Shape> shapes) {  
    history.addLast(shapes);  
    for (Shape s: shapes) { s.draw(this);}  
}
```

## **A Generic Class is Shared by All Its Invocations**

What does the following code fragment print?

```
List<String> l1 = new ArrayList<String>();
```

```
List<Integer> l2 = new ArrayList<Integer>();
```

```
System.out.println(l1.getClass() == l2.getClass());
```

You might be tempted to say false, but you'd be wrong. It prints true, because all instances of a generic class have the same run-time class, regardless of their actual type parameters.

Indeed, what makes a class generic is the fact that it has the same behavior for all of its possible type parameters; the same class can be viewed as having many different types.



## Casts and InstanceOf

Another implication of the fact that a generic class is shared among all its instances, is that it usually makes no sense to ask an instance if it is an instance of a particular invocation of a generic type:

```
Collection cs = new ArrayList<String>();
```

```
if (cs instanceof Collection<String>) { ...} // Illegal
```

Similarly, a cast such as

```
Collection<String> cstr = (Collection<String>) cs; // Unchecked  
warning
```

gives an unchecked warning, since this isn't something the runtime system is going to check for you.

# *Type erasure*

When a generic type is instantiated, the compiler translates those types by a technique called *type erasure* — a process where the compiler removes all information related to type parameters and type arguments within a class or method. Type erasure means that Java applications that use generics maintain binary compatibility with Java libraries and applications created before generics.

For instance, `Iterator<String>` is translated to type `Iterator`, which is called the *raw type* — a raw type is a class without a type argument. This means that you can't find out what type of `Object` a generic class is using at runtime. The following operations are not possible:

```
public class MyClass<E> {  
    public static void myMethod(Object item) {  
        if (item instanceof E) { //Compiler error        ...    }  
        E item2 = new E(); //Compiler error  
        E iArray[] = new E[10]; //Compiler error  
        E obj = (E) new Object(); //Unchecked cast warning  
    }  
}
```

The compiler removes all information about the actual type argument (represented by the type param **E**) at compile-time.

# Generics

A `java.util.TreeSet<E>` represents a tree of elements of type `E` that are ordered. One way to construct a `TreeSet` is to pass a `Comparator` object to the constructor. That comparator will be used to sort the elements of the `TreeSet` according to a desired ordering.

```
TreeSet(Comparator<E> c)
```

The `Comparator` interface is essentially:

```
interface Comparator<T> {int compare(T fst, T snd);}
```

Suppose we want to create a `TreeSet<String>` and pass in a suitable comparator. We need to pass it a `Comparator` that can compare `Strings`. This can be done by a `Comparator<String>`, but a `Comparator<Object>` will do just as well. However, we won't be able to invoke the constructor given above on a `Comparator<Object>`. We can use a lower bounded wildcard to get the flexibility we want:

```
TreeSet(Comparator<? super E> c)
```

This code allows any applicable comparator to be used.

# *Type Parameter Conventions*

You have already seen the angle bracket and single letter notation used to represent a type parameter. By convention, a type parameter is a single, uppercase letter — this allows easy identification and distinguishes a type parameter from a class name. The most common type parameters you will see are:

<T> — Type

<S> — for Type, when T is already in use

<E> — Element (used extensively by the Java Collections Framework)

<K> — Key

<V> — Value

<N> — Number

You can invoke a generic type with any class, but *you cannot invoke a generic type with a primitive type, such as int.*

# *Generic Methods*

Not only types can be parameterized; methods can be parameterized too. A generic method defines one or more type parameters in the method signature, before the return type:

```
static <T> boolean myMethod(List<? extends T>, T obj)
```

A type parameter is used to express dependencies between:

- the types of the method's arguments

- the type of the method's argument and the method's return type

- both

Static methods, non-static methods, and constructors can all have type parameters.

# Generic Methods

For example, the fill method in the Collections class has a dependency between its two arguments:

static <T> void fill(List<? super T> list, T obj)

The type parameter is used to show that the second argument, of type T, is related to the first argument, a List that contains objects of type T or objects of a supertype of T. For more examples, the algorithms defined by the Collections class make abundant use of generic methods.

One difference between generic types and generic methods is that generic methods are usually invoked like regular methods. The type parameters are inferred from the invocation context, as in this example that calls the fill method:

```
public static void main(String[] args) {  
    List<String> list = new ArrayList<String>(10);  
    for (int i = 0; i < 10; i++) {list.add("");}  
    String filler = "1";  
    Collections.fill(list, filler);    ...}
```

# Generic Methods

Consider writing a method that takes an array of objects and a collection and puts all objects in the array into the collection. Here's a first attempt:

```
static void fromArrayToCollection(Object[] a, Collection<?> c) {  
  for (Object o : a) {c.add(o); // Compile time error  
  }  
}
```

By now, you will have learned to avoid the beginner's mistake of trying to use `Collection<Object>` as the type of the collection parameter. You may or may not have recognized that using `Collection<?>` isn't going to work either. Recall that you cannot just shove objects into a collection of unknown type.

## Generic Methods

The way to deal with these problems is to use *generic methods*. Just like type declarations, method declarations can be generic--that is, parameterized by one or more type parameters.

```
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {  
for (T o : a) { c.add(o); // Correct  
}
```

We can call this method with any kind of collection whose element type is a supertype of the element type of the array.

```
Object[] oa = new Object[100];  
Collection<Object> co = new ArrayList<Object>();  
fromArrayToCollection(oa, co); // T inferred to be Object  
String[] sa = new String[100];  
Collection<String> cs = new ArrayList<String>();  
fromArrayToCollection(sa, cs); // T inferred to be String  
fromArrayToCollection(sa, co); // T inferred to be Object
```



```
public class Library<E> {  
    private List resources = new ArrayList<E>();  
    public void addMedia(E x) {resources.add(x);}  
    public E retrieveLast() {  
        int size = resources.size();  
        if (size > 0) {return resources.get(size - 1);}  
        return null;  
    }  
}
```

Library is a generic type with a single type parameter E. When using a generic type, specify a type argument for each of its type parameters. A Library containing only Book objects, for example, would be written as Library<Book>. This is an example of a parameterized type. The code snippet looks like this:

```
Library<Book> myBooks = new Library<Book>();
```

```
...
```

```
Book lastBook = myBooks retrieveLast();
```

## *Generic types*

```
public class AnimalHouse<E> {  
    private E animal;  
    public void setAnimal(E x) {animal = x;}  
    public E getAnimal() {return animal;}}  
public class Animal{}  
public class Cat extends Animal {}  
public class Dog extends Animal {}
```

For the following code snippets, identify whether the code:

1. fails to compile, 2. compiles with a warning
  3. generates an error at runtime, 4. none of the above (compiles and runs)
- 
- a. `AnimalHouse<Animal> house = new AnimalHouse<Cat>();`
  - b. `AnimalHouse<Dog> house = new AnimalHouse<Animal>();`
  - c. `AnimalHouse<?> house = new AnimalHouse<Cat>();`  
`house.setAnimal(new Cat());`
  - d. `AnimalHouse house = new AnimalHouse();`  
`house.setAnimal(new Dog());`

# *Enum*

```
enum Season { WINTER, SPRING, SUMMER, FALL }
```

```
enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
```

Suppose you want to add data and behavior to an enum. For example consider the planets of the solar system. Each planet knows its mass and radius, and can calculate its surface gravity and the weight of an object on the planet. Here is how it looks:

# *Planet*

```
public enum Planet {  
    MERCURY (3.303e+23, 2.4397e6),  
    VENUS   (4.869e+24, 6.0518e6),  
    EARTH   (5.976e+24, 6.37814e6),  
    MARS    (6.421e+23, 3.3972e6),  
    JUPITER (1.9e+27,   7.1492e7),  
    SATURN  (5.688e+26, 6.0268e7),  
    URANUS  (8.686e+25, 2.5559e7),  
    NEPTUNE (1.024e+26, 2.4746e7),  
    PLUTO   (1.27e+22, 1.137e6);  
  
    private final double mass; // in kilograms  
    private final double radius; // in meters  
    Planet(double mass, double radius) {  
        this.mass = mass;  
        this.radius = radius;  
    }  
    public double mass() { return mass; }  
    public double radius() { return radius; }
```

# *Planet*

```
// universal gravitational constant (m3 kg-1 s-2)
public static final double G = 6.67300E-11;

public double surfaceGravity() {
    return G * mass / (radius * radius);
}

public double surfaceWeight(double otherMass) {
    return otherMass * surfaceGravity();
}
}
```

The enum type Planet contains a constructor, and each enum constant is declared with parameters to be passed to the constructor when it is created.

Here is a sample program that takes your weight on earth (in any unit) and calculates and prints your weight on all of the planets (in the same unit):

```
public static void main(String[] args) {  
    double earthWeight = Double.parseDouble(args[0]);  
    double mass = earthWeight/EARTH.surfaceGravity();  
    for (Planet p : Planet.values())  
        System.out.printf("Your weight on %s is %.2f%n",  
                           p, p.surfaceWeight(mass));  
}
```

Your weight on MERCURY is 26,44  
Your weight on VENUS is 63,35  
Your weight on EARTH is 70,00  
Your weight on MARS is 26,51  
Your weight on JUPITER is 177,14  
Your weight on SATURN is 74,62  
Your weight on URANUS is 63,36  
Your weight on NEPTUNE is 79,68  
Your weight on PLUTO is 4,68

The idea of adding behavior to enum constants can be taken one step further. You can give each enum constant a different behavior for some method. One way to do this by switching on the enumeration constant. Here is an example with an enum whose constants represent the four basic arithmetic operations, and whose eval method performs the operation:

## *Enum*

```
public enum Operation {  
    PLUS, MINUS, TIMES, DIVIDE;  
  
    // Do arithmetic op represented by this constant  
    double eval(double x, double y){  
        switch(this) {  
            case PLUS:  return x + y;  
            case MINUS: return x - y;  
            case TIMES: return x * y;  
            case DIVIDE: return x / y;  
        }  
        throw new AssertionError("Unknown op: " + this);  
    }  
}
```

This works fine, but it will not compile without the throw statement, which is not terribly pretty. Worse, you must remember to add a new case to the switch statement each time you add a new constant to Operation. If you forget, the eval method will fail, executing the aforementioned throw statement.

# *Enum*

There is another way give each enum constant a different behavior for some method that avoids these problems. You can declare the method abstract in the enum type and override it with a concrete method in each constant. Such methods are known as constant-specific methods. Here is the previous example redone using this technique:

```
public enum Operation {  
    PLUS { double eval(double x, double y) { return x + y; } },  
    MINUS { double eval(double x, double y) { return x - y; } },  
    TIMES { double eval(double x, double y) { return x * y; } },  
    DIVIDE { double eval(double x, double y) { return x / y; } };  
  
    // Do arithmetic op represented by this constant  
    abstract double eval(double x, double y);  
}
```



# *Enum*

Here is a sample program that exercises the Operation class. It takes two operands from the command line, iterates over all the operations, and for each operation, performs the operation and prints the resulting equation:

```
public static void main(String args[]) {  
    double x = Double.parseDouble(args[0]);  
    double y = Double.parseDouble(args[1]);  
    for (Operation op : Operation.values())  
        System.out.printf("%f %s %f = %f%n", x, op, y, op.eval(x, y));  
}
```

public final class Scanner extends Object implements Iterator<String>

A simple text scanner which can parse primitive types and strings using regular expressions.

A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various next methods.

For example, this code allows a user to read a number from System.in:

```
Scanner sc = new Scanner(System.in); int i = sc.nextInt();
```

As another example, this code allows long types to be assigned from entries in a file myNumbers:

```
Scanner sc = new Scanner(new File("myNumbers"));  
while (sc.hasNextLong()) {long aLong = sc.nextLong();}
```

The scanner can also use delimiters other than whitespace. This example reads several items in from a string:

```
String input = "1 fish 2 fish red fish blue fish";  
Scanner s = new Scanner(input).useDelimiter("\\s*f\\s*");  
System.out.println(s.nextInt()); System.out.println(s.nextInt());  
System.out.println(s.next()); System.out.println(s.next()); s.close();
```

prints the following output:

1

2

red

blue

*Scanner*