

Classes and Objects in Java

Class Definition: Data Structure + Functions

```

class BankAccount {
    long number;
    String holder;
    long balance;
}

void deposit (long amount) {
    balance += amount;
}

void withdraw (long amount) {
    if (amount > balance)
        System.out.println ("Saldo insuficiente");
    else balance -= amount;
}
    
```

} Variables

} Methods

2

Variables

- They define a data structure (like a C struct)


```

class BankAccount {
    long number;
    String holder;
    long balance;
}

struct BankAccount {
    long number;
    char *holder;
    long balance;
};
            
```
- May store objects


```

class BankAccount {
    long number;
    Customer holder;
    long balance = 0;
}

class Customer {
    String name;
    long ssn;
}
            
```

Variables can be initialized directly in the class (with any arbitrary expression that does not throw checked exceptions)
- Initialized to 0 or null

3

Object Creation

- A class defines a data type that can be used to declare variables


```
BankAccount account1, account2;
```
- To declare an object is to declare a reference to an object
- Objects are created with the new operator


```
account1 = new BankAccount ();
```

number	-- undefined --
holder	-- undefined --
balance	-- undefined --
- To create an object means to allocate memory space for its variables
- new allocates memory for an object and returns a reference to the object

Objects always use dynamic memory

4

Object Creation and Access to Variables

```

BankAccount account1, account2;
account1 = new BankAccount ();
account2 = account1;
Customer customer1 = new Customer ();

customer1.name = "John Smith";
customer1.ssn = 25672046;
account1.number = 6831531;
account1.holder = customer1;
account1.balance = 100000;
account2.balance = 200000;

account2.holder.name = "Lois Lane";
    
```

5

Methods

- Methods are functions defined within a class (Similar to a pointer to a function in a struct in C)
- Methods can directly reference the variables of the class
- Methods can only be invoked on objects of the class to which they belong
- During the execution of a method, invoked upon an object of class A, the variables in the class take the value they have in the object

```

account2.deposit(1000);
    
```

```

number ← account2.number
holder ← account2.holder
balance ← account2.balance

void deposit (long amount) {
    balance += amount;
}
            
```

account2.balance

6

Method Calls from within a Method

- Methods can directly invoke any other method of the same class
- When executing a method invoked on an object of class A, calls to other methods of class A are executed on the same object, unless invoked explicitly on a different object

```

account3 = new BankAccount ();
account2.transfer (account3, 1000);

class BankAccount {
    ...
    void transfer (BankAccount target, long amount) {
        if (amount <= balance ) {
            withdraw (amount);
            target.deposit (amount);
        }
    }
}
account2.withdraw (amount)
    
```

7

Methods are Executed in the Context of an Object

- Objects a method can access:
 - Object of the invocation: implicitly, by directly accessing its variables and methods
 - Object defined in a local variable
 - Object passed as an argument
 - Object stored in a variable of the class
- In C, the object of the invocation would be one more parameter
- In OOP, the invocation object plays a different role: the invoked method belongs to the object and not vice versa
- In a method's code, the invocation object is not explicitly visible: its variables and methods are accessed

8

Objects a Method can Access

```

class X { String name; }
class Y { int i; }
class Z { String name; }
    
```

- A variable of the invocation object
- Object in local variable
- Object passed as an argument
- Object stored in a variable of the class

```

class A {
    int num;
    X obj4;
    void f (int n, Y obj3) {
        Z obj2 = new Z ();
        obj4.name = obj2.name;
        num = obj3.i + n;
    }
}
    
```

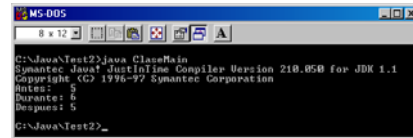
The invocation object (1) is not seen like the other objects (2, 3, 4) but is implicit: the method f accesses its variables

9

Passing Arguments: Always by Value

```

class MainClass {
    public static void main (String args[] ) {
        int n = 5;
        System.out.println ("Antes: " + n);
        f (n);
        System.out.println ("Despues: " + n);
    }
    static void f (int i) { System.out.println ("Durante: " + ++i); }
}
    
```



10

Passing Arguments: References (I)

```

class MainClass {
    public static void main (String args[] ) {
        ...
        f (a);
        System.out.println ("Despues: " + a[3]);
    }
    static void f (int x[]) {
        x[3] = 0;
        x = new int[8];
        x[3] = 5;
    }
}
    
```

11

Passing Arguments: References (II)

```

class MainClass {
    public static void main (String args[] ) {
        int a[] = {5, 4, 3, 2, 1};
        System.out.println ("Antes: " + a[3]);
        f (a);
        System.out.println ("Despues: " + a[3]);
    }
    static void f (int x[]) {
        x[3] = 0;
        x = new int[8];
        x[3] = 5;
    }
}
    
```

12

Constructors

- "Methods" that are executed automatically when the objects of a class are created (i.e. when allocating memory for the objects, i.e. when calling new)
- Typical purpose
 - Initial values for the object variables
 - Other initialization operations
- Advantages
 - Syntax simplification
 - Encapsulation of object variables: avoid external access

13

Constructors: Example

```
class Customer {
    String name;
    long ssn;
    Customer (String str, long num) {
        name = str; ssn = num;
    }
}

class BankAccount {
    long number;
    Customer holder;
    long balance;
    BankAccount (long num, Customer clt, long s) {
        number = num; holder = clt; balance = s;
    }
}
```

14

Object Creation with Constructors

Constructors are executed automatically when creating objects

```
Customer customer1 = new Customer ("John Smith", 25672046);
```

Customer	
name	"John Smith"
ssn	25672046

```
BankAccount account1 =
    new BankAccount (6831531, customer1, 100000);
```

```
BankAccount account2 =
    new BankAccount (8350284,
        new Customer ("Lois Lane", 15165442),
        200000);
```

15

Default Constructors

- If no constructors are defined, Java defines one by default
- If a constructor is explicitly defined, the default constructor is not defined

```
class A {
    A () { }
}

class Customer {
    ...
    Customer (String str, long num) { ... }
}
```

```
// Main block
Customer customer1 = new Customer ();
// Error: No constructor matching Customer() found in Customer
```

16

The this Variable

- Implicitly defined in body of all methods
- Reference to the object on which the method is invoked

```
class Vector3D {
    double x, y, z;
    ...
    double dotProduct (Vector3D u) {
        return x * u.x + y * u.y + z * u.z;
        // return this.x * u.x + this.y * u.y + this.z * u.z;
    }
    double module () {
        return (double) Math.sqrt (dotProduct (this));
    }
}
```

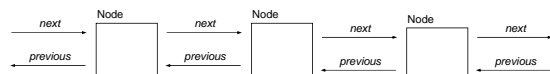
```
// Main block
Vector3D v = new Vector3D (2, -2, 1);
v.module ();
```



17

The this Variable: Inverse Relations

```
class Node {
    Node previous;
    Node next;
    void connect (Node z) {
        next = z;
        z.previous = this;
    }
}
```



18

3. Classes and Objects in Java

this in Constructors

```

class Customer {
    String name; long ssn;
    Customer (String str, long num) { name = str; ssn = num; }
    BankAccount accounts[] = new BankAccount [20];
    int naccounts = 0;
    void newAccount (BankAccount account) {
        accounts[naccounts++] = account;
    }
}

class BankAccount {
    long number;
    Customer holder;
    long balance;
    BankAccount (long num, Customer clt, long s) {
        number = num; holder = clt; balance = s;
        clt.newAccount (this);
    }
}
    
```

19

Method Overloading

```

class Plane3D {
    double a, b, c, d;
    Plane3D (double aa, double bb, double cc, double dd) {
        a = aa; b = bb; c = cc; d = dd;
    }
    boolean parallel (Plane3D p) {
        Vector3D u = new Vector3D (a, b, c);
        Vector3D v = new Vector3D (p.a, p.b, p.c);
        return u.parallel (v);
    }
    boolean parallel (Line3D r) {
        Vector3D u = new Vector3D (a, b, c);
        return u.perpendicular (r.vector);
    }
}
    
```

20

Constructor Overloading

```

class Point3D {
    double x, y, z;
    Point3D (double xx, double yy, double zz) {
        x = xx; y = yy; z = zz;
    }
}

class Vector3D {
    double x, y, z;
    Vector3D (double xx, double yy, double zz) {
        x = xx; y = yy; z = zz;
    }
    Vector3D (Point3D p, Point3D q) {
        x = q.x - p.x; y = q.y - p.y; z = q.z - p.z;
    }
    boolean parallel (Vector3D u) {
        return (x * u.y == y * u.x) && (x * u.z == z * u.x);
    }
    boolean perpendicular (Vector3D u) {
        return dotProduct (u) == 0;
    }
}
    
```

21

Call to Overloaded Methods

```

class Line3D {
    Point3D point;
    Vector3D vector;
    Line3D (Point3D p, Vector3D v) {
        point = p; vector = v;
    }
}

// Main block
Plane3D p1 = new Plane3D (2, 4, 3, 1);
Plane3D p2 = new Plane3D (1, 0, -2, 1);
Line3D r = new Line3D (new Point3D (1, 0, 1),
                    new Vector3D (1, 1, -1));

p1.parallel (p2);
p1.parallel (r);
    
```

22

Method Overloading: Ambiguity

```

class A {
    void f (int n) {
        System.out.println ("Type int");
    }
    void f (float x) {
        System.out.println ("Type float");
    }
}

// Main block
A a = new A();
byte b = 3;
long l = 3;
double d = 3;
a.f(l);
a.f(b);
a.f(d); // ERROR: needs explicit cast
    
```

The most specific compatible definition is executed

23

Object Destruction (I)

- The memory allocated for objects is freed by Garbage Collection
- GC frees the memory of all objects that are nowhere referenced

```

BankAccount account1 =
    new BankAccount (8350284,
        new Customer ("John Smith", 15165442),
        200000);
account1.holder = new Customer ("Lois Lane", 25672046);
    
```

GC

Object Destruction (II)

- Release a reference explicitly
`account1 = null;`

- An object can be freed only when all references to it are released
`BankAccount account2 = account1;`
`account1 = null; // The object is not yet freed`
- It is possible to request a garbage collection: `System.gc ();`

25

Destructors

```
class BankAccount {
    ...
    protected void finalize () {
        // Destructor code here
    }
}
```

- Automatically invoked before the object memory is freed
- Free resources associated to the object: open files, sockets, etc.
- Ensure a consistent program state after an object disappears
- Limitation: the programmer does not control when the memory is freed

26

Access Control Public vs. Private Class Members

```
class A {
    public int x;
    private int y;
    public void f () { ... }
    private void g () { ... }
    void h () {
        x = 2;
        y = 6;
        f ();
        g ();
        A a = new A ();
        a.x = 2;
        a.y = 6;
        a.f ();
        a.g ();
    }
}

class B {
    void h () {
        A a = new A ();
        a.x = 2;
        a.y = 6; // Error
        a.f ();
        a.g (); // Error
    }
}
```

27

Other Access Control Levels

Variable, method, and constructor hiding modalities

	Class	Package	Subclass	Any
<i>private</i>	X			
(by default) <i>package</i>	X	X		
<i>protected</i>	X	X	X	
<i>public</i>	X	X	X	X

Class hiding modalities: *public* or *package*

28

Variable and Method Hiding within Classes

```
class A {
    int w; // package
    private int x;
    protected int y;
    public int z;
    private void f () { ... }
    void h () {
        w = 2;
        x = 6;
        y = 8;
        z = 3;
        f ();
    }
}

class B {
    void h () {
        A a = new A ();
        a.w = 2;
        a.x = 6; // Error
        a.y = 8;
        a.z = 3;
        a.f (); // Error
    }
}
```

29

Class Hiding within Packages

```
A.java
package p1;
// Public class
public class A {
    ...
}

// Package class
class B {
    ...
}

package p2;
class C {
    void f () {
        p1.A a;
        p1.B b; // Error
    }
}
```

30

Packages

- Set of classes defined in a folder
- Avoid symbol conflicts
- Each class belongs to a package
- If no package is defined for a class, java includes it in the DefaultPackage

31

Define Packages

graphics\Circle.java

```
package graphics;

public class Circle {
    public void paint () {
        ...
    }
    ...
}
```

graphics\Rectangle.java

```
package graphics;

public class Rectangle {
    public void paint () {
        ...
    }
    ...
}
```

32

Using Classes of a Different Package

```
...
graphics.Circle c = new graphics.Circle ();
c.paint ();
...
```

```
import graphics.Circle;
...
Circle c = new Circle ();
c.paint ();
...
```

Import class

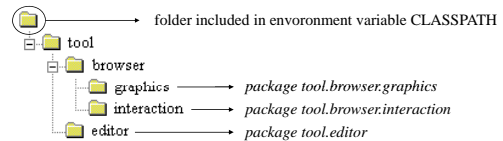
```
import graphics.*;
...
Circle c = new Circle ();
Rectangle r = new Rectangle ();
c.paint (); r.paint ();
...
```

Import all classes of package

33

Packages and Folders

- Package name → folder structure
- CLASSPATH: list of folders where java looks for packages



- Automatically imported packages:
 - java.lang
 - DefaultPackage
 - Current package

```
package tool.editor;

public class WorkArea {
    ...
}
```

34

Predefined Packages in java 1.1

java.applet	java.rmi
java.awt	java.rmi.dgc
java.awt.datatransfer	java.rmi.registry
java.awt.event	java.rmi.server
java.awt.image	java.security
java.beans	java.security.acl
java.io	java.security.interfaces
java.lang	java.sql
java.lang.reflect	java.text
java.math	java.util
java.net	java.util.zip

35

Hiding within Classes and Packages

```
package p1;

public class A {
    int w; // package
    private int x;
    protected int y;
    public int z;
}
```

```
package p2;

class C {
    void h () {
        pl.A a = new pl.A ();
        a.w = 2; // Error
        a.x = 6; // Error
        a.y = 8; // Error
        a.z = 3;
    }
}

class D extends pl.A {
    void h () {
        pl.A a = new pl.A ();
        w = 2; a.w = 2; // Error
        x = 2; a.x = 6; // Error
        z = 3; a.z = 3;
        a.y = 8; // Error
        y = 8;
        D d = new D ();
        d.y = 8;
    }
}
```

Static Variables (I)

```
class CheckingAccount {  
    static double interest;  
}
```

- Similar to global variables
 - Class variables (i.e. static) vs. instance variables (by default)
 - Static variables belong to the class, not to the objects
 - Access to static variables: from class or from objects
- ```
CheckingAccount account = new CheckingAccount ();
account.interest = 0.3;
CheckingAccount.interest = 0.2;
```

37

## Static Variables (II)

- Static variables are shared by all objects of the class (unlike instance variables, a copy per object is not created)
- ```
CheckingAccount account1 = new CheckingAccount ();  
CheckingAccount account2 = new CheckingAccount ();  
account1.interest = 0.03;  
account2.interest = 0.04;  
System.out.println (account1.interest);  $\longrightarrow$  0.04
```
- Memory for static variables is allocated when the class is loaded in the Java interpreter

38

Static Methods

```
class CheckingAccount {  
    long number;  
    static long naccounts;  
    static long generateNumber () {  
        return naccounts++;  
    }  
}
```

- Similar to global functions
 - Can be invoked from the class or from objects
- ```
CheckingAccount account = new CheckingAccount ();
account.number = account.generateNumber ();
account.number = CheckingAccount.generateNumber ();
```

39

## Restrictions for Static Methods

- They can access static variables and methods of the same class
  - They cannot access this
  - They cannot access instance variables or methods in the class
- ```
static long generateNumber () {  
    number = numaccounts++; // Error  
    return naccounts;  
}
```
- Instance methods can access static variables and methods
- ```
void payInterest () {
 balance += balance * interest;
}
```
- ```
BankAccount () { // Constructor  
    number = generateNumber ();  
}
```
- A constructor cannot be static

40

Static Initialization Blocks

```
class A {  
    static double num;  
    String x;  
    static {  
        num = 5.2;  
        x = "Hello"; // Error  
    }  
}
```

```
class A {  
    static double num = 5.2;  
    String x = "Hello";  
}
```

- They are executed only once: when the class is loaded
- An arbitrary number of static blocks are allowed within a class
- They cannot reference instance variables or methods
- Useful to initialize static variables with error handling (exceptions)
- Whenever possible, better initialize in variable declaration or in constructors

41

final Variables

- Similar to constants, their value cannot be changed
 - Initialization is mandatory
- ```
class Circle {
 final double PI = 3.141592653589793;
 void f () {
 PI = 3.0; // Error
 }
}
```
- More efficient: static final
  - A constructor cannot be final

42

## Fundamental Class Libraries

### Character Strings: The java.lang.String class

- String encapsulates arrays of characters and their manipulation
- String ≠ char[]
- The characters in a String cannot be changed
- Constructors

```
String (char[])
String (String)
```

- String creation

```
char[] chars = {'a','b','c'};
String s = new String (chars);
String s1 = "Hello";
String s2 = new String (s1);
```

- Literals: Java creates objects of type String for literals

44

### Methods of the String Class (I)

- Access (the str[n] notation does not exist for String)

```
String str = "abcdabc";
str.length (); // → 7
str.charAt (4); // → 'a'
str.substring (3, 5); // → "da"
str.indexOf ("bc"); // → 1
str.lastIndexOf ("bc"); // → 5
```

- Manipulation (a String is returned)

```
str.concat ("xyz"); // → "abcdabcxyz"
str.toUpperCase (); // → "ABCDABC"
str.replace ('b', 'x'); // → "axcdaxc"
" How are you ".trim (); // → "How are you"
```

45

### Methods of the String Class (II)

- Conversion to String: static String valueOf (<any type>)

```
String.valueOf (2.34567); // → "2.34567"
String.valueOf (34); // → "34"
String.valueOf (new Plane3D ()); // → "Plane3D@1d07be"
```

- Comparison

```
String s1 = "abc", s2 = new String ("abc"), s3 = "abx";
s1 == s2; // → false
s1.equals (s2); // → true
s1.compareTo (s3); // → -21 < 0
```

46

### Character Strings: The java.lang.StringBuffer Class

- The characters within a StringBuffer can be modified
- The objects of type StringBuffer manage their capacity automatically
  - They take an initial capacity
  - They increase it when necessary

- Constructors

```
StringBuffer() // Initial capacity: 16 characters
StringBuffer(int c) // Initial capacity: c characters
StringBuffer(String s) // Initial capacity:
// s.length() + 16 characters
```

47

### StringBuffer Methods (I)

- Access (same as for String): length (), charAt (int), ...
- Conversion to String: toString ()

- Modification of characters:

```
StringBuffer str = new StringBuffer ("abcdef");
str.setCharAt (2, 'q'); // str = "abqdef"
str.append ("ghi"); // str = "abqdefghi"
str.insert (3, "xyz"); // str = "abqxyzdefghi"
str.insert (6, 1.23); // str = "abqxyz1.23defghi"
str.delete (2, 10); // str = "abdefghi" (versión 1.2)
str.reverse (); // str = "ihgfedba"
```

48

## StringBuffer Methods (II)

- Manipulating length and capacity  
`length ()`, `capacity ()`  
`setLength (int)`, `ensureCapacity (int)`
- Concatenation operator: a `StringBuffer` is used internally

```
{
"ssn de " + customer.name + ": " + customer.ssn
new StringBuffer().append("ssn de ")
 .append(customer.name).append(": ")
 .append(customer.ssn).toString()
}
```

49

## Standard I/O: The java.lang.System Class

### System.out, System.err

- Objects of the `java.io.PrintStream` class
- Methods: `print (<any type>)`, `println (<any type>)`, `flush ()`

### System.in

- Object of the `java.io.InputStream` class
- Methods: `read ()`, `read (byte [])`, `read (byte [], int, int)`
- The methods throw `java.io.IOException`

### Redirecting standard I/O:

- `System.setOut(PrintStream)`, `System.setErr(PrintStream)`
- `System.setIn(InputStream)`

50

## Standard Input with java.io.BufferedReader

`InputStream` → `InputStreamReader` → `BufferedReader` → `String`  
(`System.in`)

```
import java.io.*;

public static void main (String args[]) throws IOException {
 BufferedReader reader =
 new BufferedReader (new InputStreamReader (System.in));
 String str = reader.readLine ();
}
```

51

## Writing to a File in Text Mode with java.io.PrintStream

- Allows writing in text mode (like with `System.out`)

```
PrintStream printer =
 new PrintStream (new FileOutputStream ("abc.txt"));
printer.print ("Two + " + 2);
printer.println (" = " + (2+2));
...
printer.close ();
```

- Throws `java.io.IOException`

52

## Reading from a File in Text Mode with java.io.BufferedReader

- Allows reading in text mode (like with `System.in`)

```
BufferedReader reader =
 new BufferedReader (new FileReader ("abc.txt"));
String str = reader.readLine ();
...
reader.close ();
```

- Throws `java.io.IOException`

53

## I/O in Binary Mode: The java.io.RandomAccessFile Class

- Allows reading and writing primitive data types (not objects) in binary form

```
Open a file
// Modalities: "r", "rw"
RandomAccessFile f = new RandomAccessFile("abc.txt", "r");
f.close();
```

- Methods:

```
readInt () → int, readBoolean () → boolean, etc.
writeInt (int), writeBoolean (boolean), etc.
getFilePointer (), seek (long), length (), setLength ()
```

- The methods throw `java.io.IOException`

54

## 3. Classes and Objects in Java

### Math Functions: The java.lang.Math Class

- Constants: Math.PI, Math.E
- Methods: sqrt (double), pow (double, double), random (), abs (double), max (double, double), round (double), cos (double), sin(double), tan (double), acos (double), exp (double), log (double), etc.  
(float, long, and int versions exist for abs, max, min, round)

55

### Classes for Numeric Types (I) (java.lang package)

- Byte, Short, Integer, Long, Float, Double
- Encapsulation of values for generic handling of different types of values  
Integer m = new Integer (54);  
Integer n = new Integer ("23");  
int i = n.intValue ();
- Conversion to String and conversion between numeric types  
m = Integer.valueOf ("91"); // String → Integer  
String s1 = m.toString (); // Integer → String  
(String s2 = String.valueOf (m); // Integer → String )  
float f = m.floatValue (); // Integer → float  
short c = new Float (f) .shortValue (); // Float → short  
// etc.

56

### Classes for Numeric Types (II)

- Class variables
- ```
Integer.MIN_VALUE → -2147483648
Integer.MAX_VALUE → 2147483647
Float.MIN_VALUE → 1.4E-45
Float.MAX_VALUE → 3.4028235E38
// etc.
Float.NEGATIVE_INFINITY
Float.POSITIVE_INFINITY
Float.NaN
```
- } public
static
final

57

Classes for other Primitive Types

- java.lang.Character class
Character c = new Character ('a');
char ch = c.charValue ();
Character.isDigit ('2'); // → true
Character.isLetter ('a'); // → true
Character.isLowerCase ('a'); // → true
ch = Character.toUpperCase ('a'); // ch = 'A'
- java.lang.Boolean class
Boolean b1 = new Boolean (true);
Boolean b2 = new Boolean ("false");
boolean b = b1.booleanValue ();
b1 = Boolean.valueOf ("true"); // String → Boolean
String s1 = b1.toString (); // Boolean → String

58

The java.util.ArrayList Class

- Variable set of any type of objects
- Similar to an array, but its capacity grows or shrinks dynamically
- Length, capacity, increment
- Constructors
ArrayList v = new ArrayList (); // Initial capacity: 10
ArrayList v = new ArrayList (100); // Initial capacity: 100
- Adjust capacity: ensureCapacity (), trimToSize ()

59

ArrayList Methods (I)

- ```
ArrayList list = new ArrayList (); // list = { }
```
- Insertion  
list.add (2); // Error  
list.add (new Integer (2)); // list = { 2 }  
list.add (new Float (4.5)); // list = { 2, 4.5 }  
list.add (1, "Hello"); // list = { 2, "Hello", 4.5 }  
list.set (0, new Character ('b')); // list = { 'b', "Hello", 4.5 }
  - Remove elements:  
list.remove (new Float (4.5)); // list = { 'b', "Hello" }  
list.remove (0); // list = { "Hello" }

60

## ArrayList Methods (II)

- Access

```
// Assuming list = { 'b', "Hello", 4.5 }
list.get (1); // → "Hello"
list.indexOf ("Hello"); // → 1 (-1 if not found)
```
- Length and capacity

```
size (), isEmpty (), ensureCapacity (), trimToSize ()
```
- Iteration: `iterator () → Iterator`

61

## Iteration over an ArrayList: Interface `java.util.Iterator`

- Methods of the `Iterator` interface 

|                         |                        |
|-------------------------|------------------------|
| <code>hasNext ()</code> | → <code>boolean</code> |
| <code>next ()</code>    | → <code>Object</code>  |
- Example (assuming the `Point3D` class has a `print()` method)

```
ArrayList list = new ArrayList ();
list.add (new Point3D (0, 1));
list.add (new Point3D (2, -3));
list.add (new Point3D (-1, 1));
System.out.println ("The elements of the list are: ");

Iterator iter = list.iterator ();
while (iter.hasNext())
 ((Point3D) iter.next ()) .print ();

for (int i = 0; i < list.size (); i++)
 ((Point3D) list.get (i)) .print ();
```

62

## Other Classes

- `java.util` package: `LinkedList`, `HashMap`, `Set`, `Collections`, `Date`, `StringTokenizer`...
- `java.text` package: `DateFormat`, `DecimalFormat`
- `java.math` package: `BigDecimal`, `BigInteger` (arbitrary precision and capacity)
- The `java.lang.System` class: `in`, `out`, `exit(int)`
- The `java.lang.Runtime` class: `getRuntime()`, `exec(String)`, `exit(int)`

63