3.2 Objects and classes in Java

Classes and data types, objects and variables
Object creation: dynamic memory, *garbage collection*
Class *members*:

- instance variables (attributes): visibility and access
- instance methods: invoked on an object
- class variables (**`static`** shared by class instances)
- class methods (**`static`** invoked on the class)

Overloading and parameter passing
Constructors and the special variable **`this`**
Classic abstract data types: *getter* & *setter* methods
Generic classes (using types as parameters)
Enumeration types

Classes: data structures + operations
Define Data Types

```
class BankAccount {
    long number;
    String owner;
    long balance = 0;

    void deposit(long amount) {
        balance += amount;
    }
    void withdraw(long amount) {
        if (amount > balance)
            System.out.println("insufficient balance");
        else balance = balance - amount;
    }
}   // end of class BankAccount
```

Instance Variables (attributes)

Instance Methods

Instance attributes

- Components of the data structure (similar to C)

```
class BankAccount {
    long number;
    String owner;
    long balance;
}
```

```
typedef struct {
    long number;
    char *owner;
    long balance;
} BANK_ACCOUNT;
```

- They can refer to objects of any class including itself (as in C)

```
class BankAccount {
    long number = -1;
    Client owner;
    long balance;
}
```

```
class Client {
    String name;
    long id;
}
```

Instance variables can be initialized explicitly when declared
(by way of an expression that does not raise exceptions)

- By default initialized to **`0`**, **`null`** or **`false`**

Object creation: class instantiation

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

```
BankAccount account1, account2;
```

  and which includes operations to manipulate those variables.

- `account1` and `account2` are ***instances*** of `BankAccount`

- Also, they are variables of the ***reference type*** defined by the class

- Objects **are created** by using the operator **`new`**

```
BankAccount account1 = new BankAccount();
BankAccount account2;      // not yet created
```

- Object creation allocates memory space for instance variables (and more) and returns a *reference* to the new object

Data types vs. Classes

- A variable (or constant) declaration states its type:

  `balance` and **`amount`** are declared of type **`long`**

  `account1` is declared of type **`BankAccount`**

- The type of an expression can be inferred by the compiler

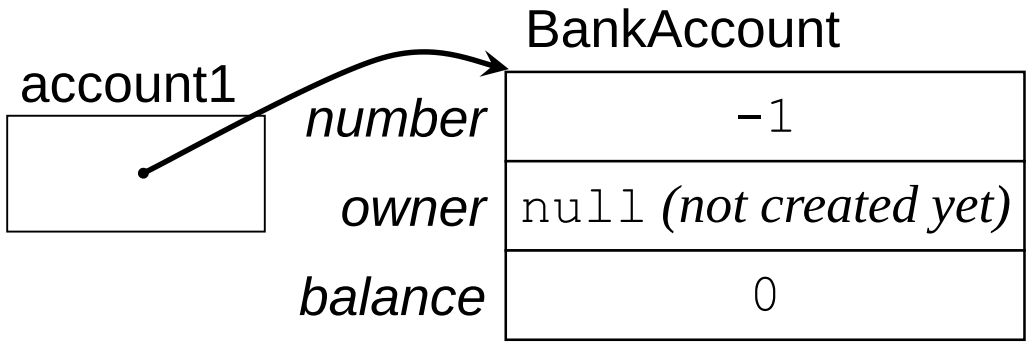
  expression **`balance - amount`** is of type **`long`**

- The data type restricts the values that a variable can hold (or an expression can return) at runtime
- In Java, if a variable type is a reference type (not a primitive one), then it will hold a reference to an object, *whose class* must be *compatible* with the type used in the variable declaration
- We will analyse such compatibility in detail when introducing subclasses

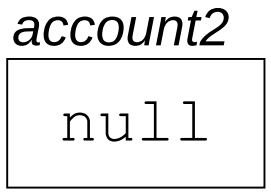
Object creation (instantiation)

Objects always use dynamic memory

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

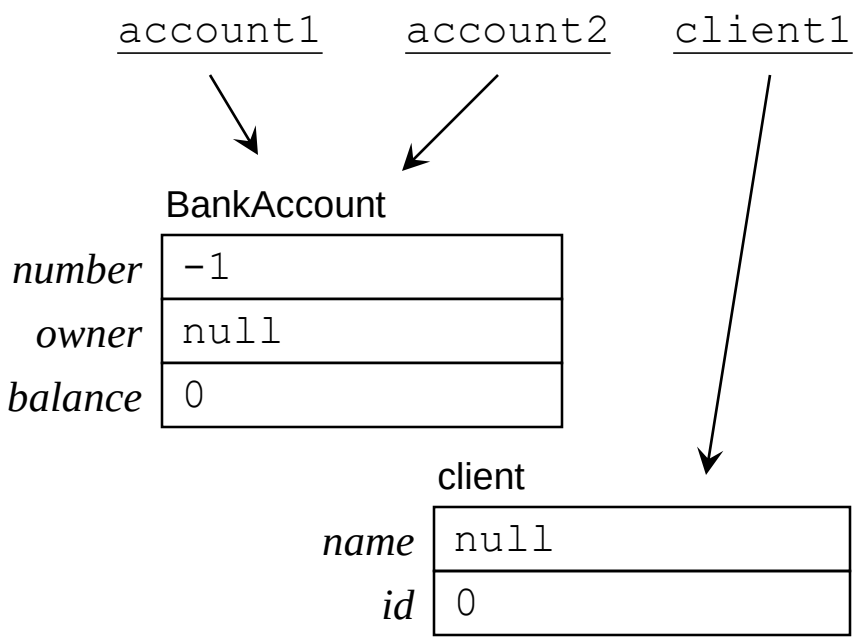



```
BankAccount account2;      // not created yet
```



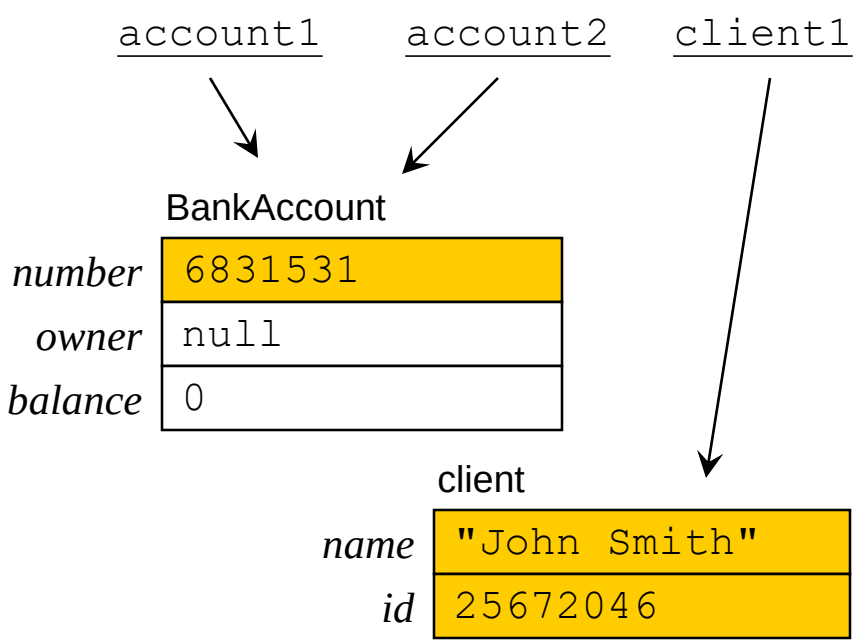
Object creation and assignment

```
BankAccount account1, account2;

account1 = new BankAccount();
account2 = account1;

Client client1 = new Client();
```



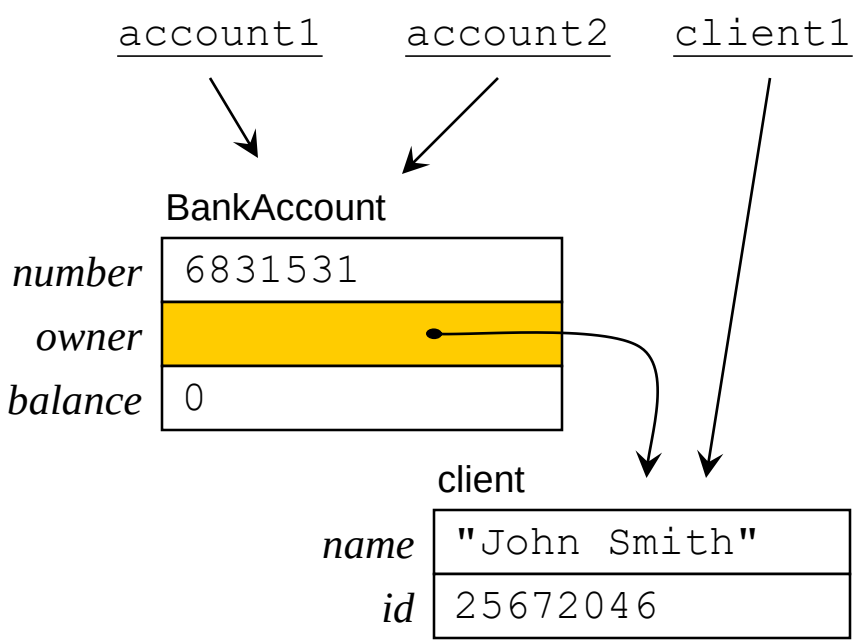
Direct access to instance variables (`public`)

```
BankAccount account1, account2;

account1 = new BankAccount();
account2 = account1;

Client client1 = new Client();

client1.name = "John Smith";
client1.id = 25672046;
account1.number = 6831531;
```



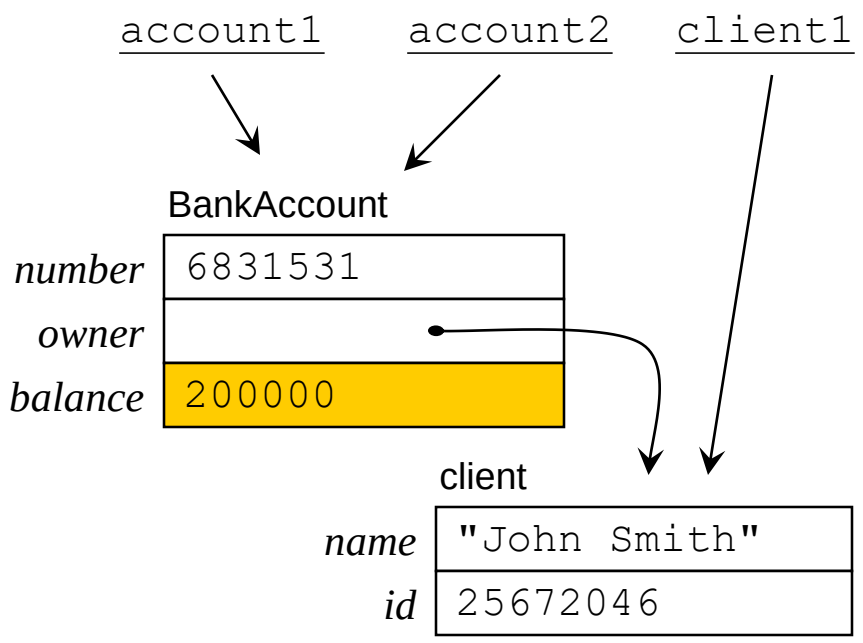
Direct access to instance variables

```
BankAccount account1, account2;

account1 = new BankAccount();
account2 = account1;

Client client1 = new Client();

client1.name = "John Smith";
client1.id = 25672046;
account1.number = 6831531;
account1.owner = client1;
```



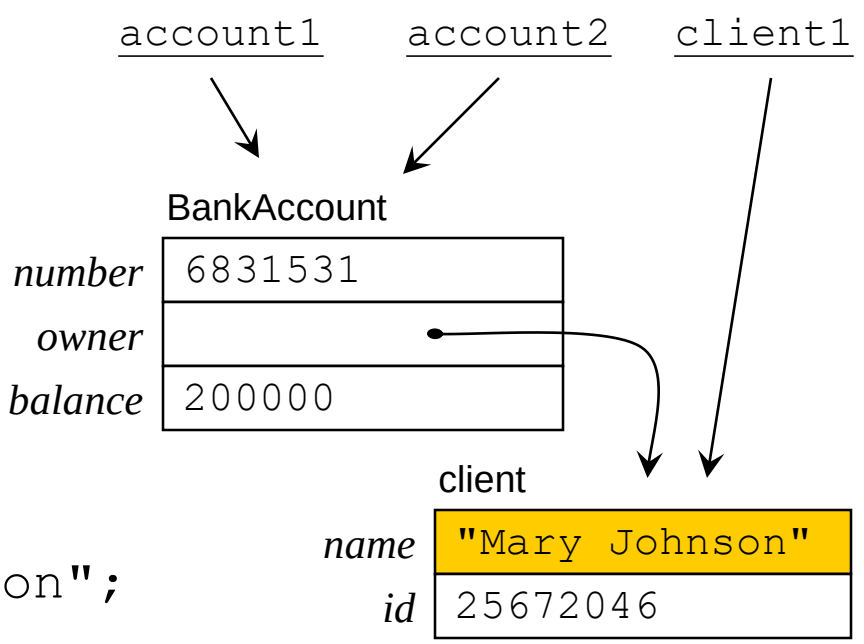
Instance variables: may refer to other objects

```
BankAccount account1, account2;

account1 = new BankAccount();
account2 = account1;

Client client1 = new Client();

client1.name = "John Smith";
client1.id = 25672046;
account1.number = 6831531;
account1.owner = client1;
account1.balance = 100000;
account2.balance = 200000;
```



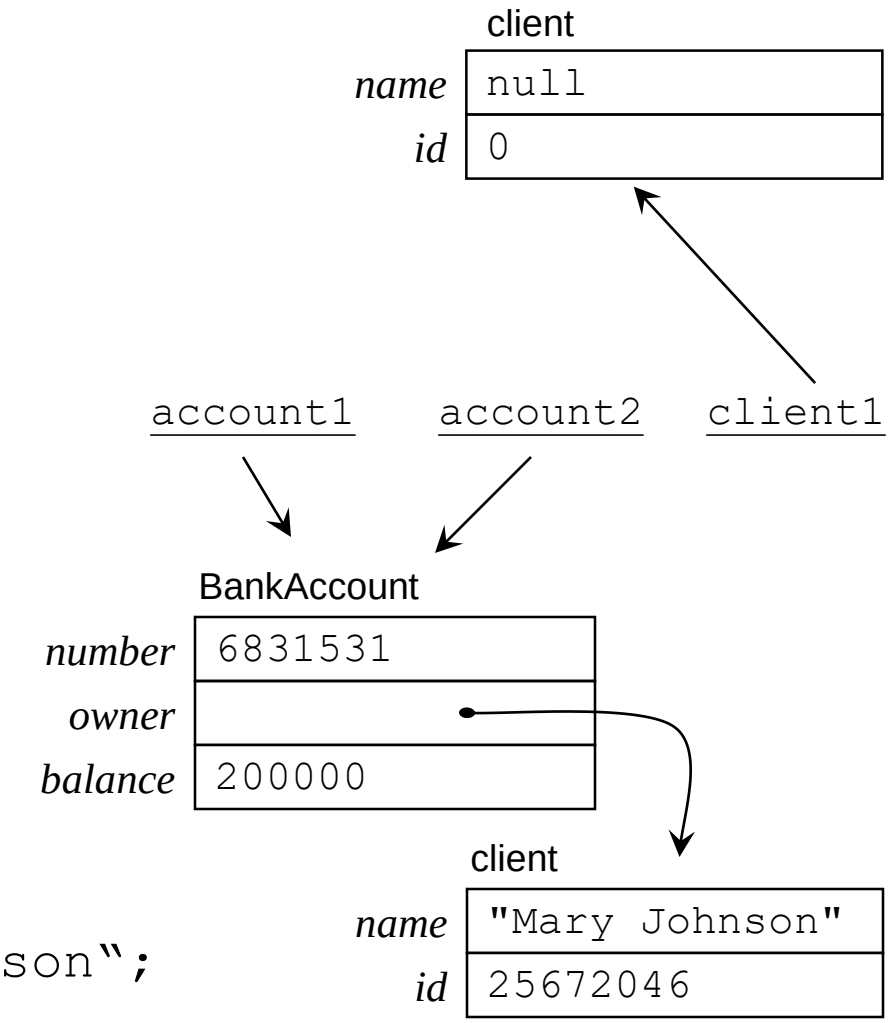
Access through other objects

```
BankAccount account1, account2;

account1 = new BankAccount();
account2 = account1;

Client client1 = new Client();

client1.name = "John Smith";
client1.id = 25672046;
account1.number = 6831531;
account1.owner = client1;
account1.balance = 100000;
account2.balance = 200000;
account2.owner.name = "Mary Johnson";
```



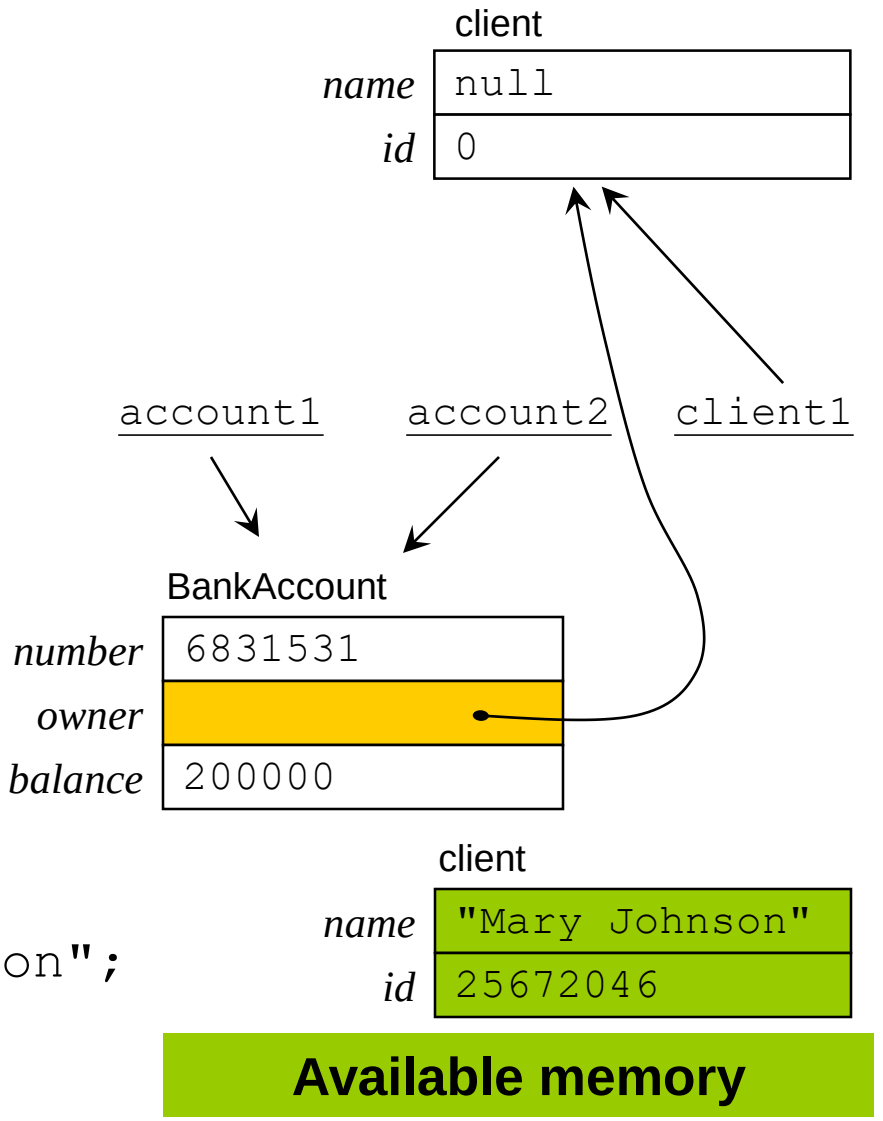
Reassigning references to objects

```
BankAccount account1, account2;

account1 = new BankAccount();
account2 = account1;

Client client1 = new Client();

client1.name = "John Smith";
client1.id = 25672046;
account1.number = 6831531;
account1.owner = client1;
account1.balance = 100000;
account2.balance = 200000;
account2.owner.name = "Mary Johnson";
client1 = new Client();
```



Gargabe collection
(no explicit freeing of memory)

```
BankAccount account1, account2;

account1 = new BankAccount();
account2 = account1;

Client client1 = new Client();

client1.name = "John Smith";
client1.id = 25672046;
account1.number = 6831531;
account1.owner = client1;
account1.balance = 100000;
account2.balance = 200000;
account2.owner.name = "Mary Johnson";
client1 = new Client();
account1.owner = client1;
```



Methods

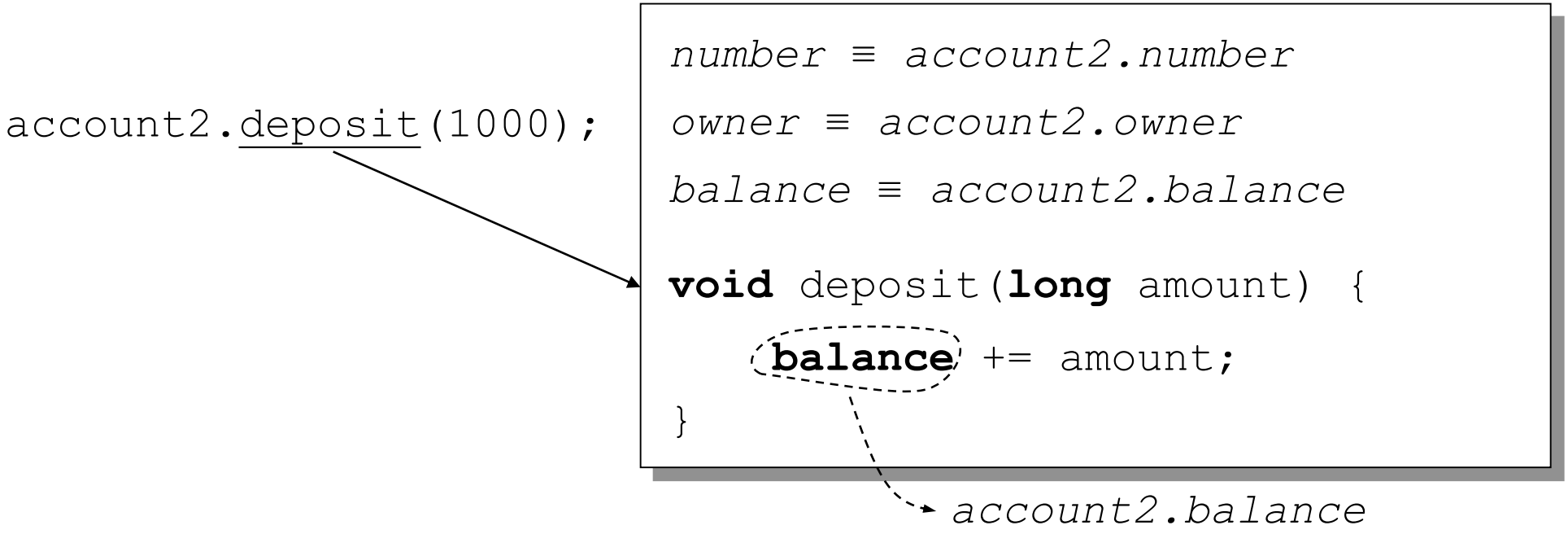
- Procedures or functions defined within a class declaration
- Part of the structure (data type) defined by the class
  (similar to storing pointers to a function within a data structure `struct` in C)
- Methods can access variables and methods defined in the same class
- Two kinds: **instance methods** and **class methods**
- Instance methods are associated with each object (instance)
- Instance methods must be invoked on a particular instance of the class that defines the method

```
account2.deposit(1000);
```

- `account2` is similar to an implicit additional parameter

Invoking instance methods

- Instance methods are invoked on an instance of the class they belong to
- When an instance method is executing, its references to instance variables are regarded as references to the variables within the instance on which the method was invoked



Which one of these two versions is to be preferred?

```
public class Person {
    private String    name;
    private int       age;

    public Person (String n, int e) {
      this.name = n; this.age = e;
    }

    public String toString() {
        return this.name + "\nAge: " + this.age;
    }

    public String getName() { return this.name; }
    public int getAge() { return this.age;}
}
```

Which one of these two versions is to be preferred?

```
public class Person {
    private String    name;
    private int       age;

    public Person (String n, int e) {
      this.name = n; this.age = e;
    }

    public String toString() {
        return this.getName() + "\nAge: " +
               this.getAge();
    }

    public String getName() { return this.name; }
    public int getAge() { return this.age;}
}
```

Invoking methods from other methods

- Instance methods may directly invoke other methods in the same class
- When an instance method is executing, calls to other methods in that class are regarded as if methods were invoked to that same object unless they are explicitly invoked on a different object

```
BankAccount account3 = new BankAccount();
account2.orderTransfer(account3, 1000);
```

```
class BankAccount {  . . .
 void orderTransfer(BankAccount toAcct, long amount) {
        if (amount <= balance ) {
           withdraw(amount);
            toAcct.deposit(amount);
        }
    }
 } // orderTransfer
```

account2.withdraw (amount)

Instance methods execute within the context of the object on which they are invoked

- From an instance method we may access to
  1. Variables and methods of executing object (implicitly)
  2. Objects stored in local variables or passed as parameters
  3. Objects stored in ***class variables***

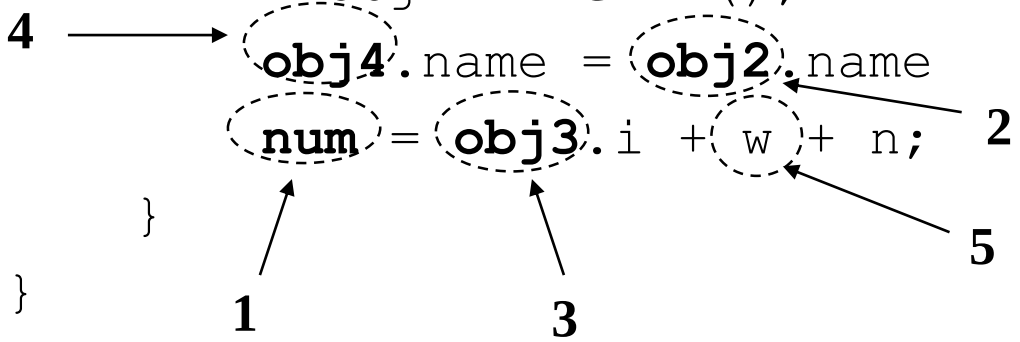
- In C, the invocation object would be an additional parameter
- In OOP, the invocation object plays a central role:

  the instance method invoked is part of the object itself
- Within an instance method, the invocation object is accesible explicitly using the predefined variable "**`this`**"

Data accessible from a method

```
ClassA obj1 = new ClassA();     obj1.f(7, new Y());
```

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



1. *Invocation object´s variable* ***obj1***
2. *Object stored in the method´s local variable*
3. *Object passed as a parameter to the method*
4. *Object stored in an instance variable*
5. *Class variable from the class of object* ***obj1***

The invocation object is ***obj1***
It is not accesible directly like the other objects (2, 3, 4) but is ***implictily accesible****: method* `f` *implicitly access to* ***obj1*** *variables (such as* ***num****) and also through* ***this***

More on the *this* variable

```
public class Point {
   private long x, y;    // coordinates of each point

   public Point(long x, long y) {
      this.x = x;
      this.y = y;

      }

      …
}
```

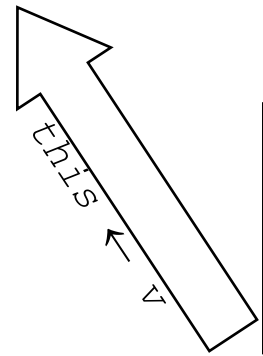
The reserved word **this** is a reference to the invocation object, but is also used, within a constructor, to refer the object being created. It can be used to access its components, and also to be passed as a parameter to other methods.

It can only be used within instance methods and constructors

Using the variable `this` as parameter

An instance method can pass its invocation object as parameter to other methods

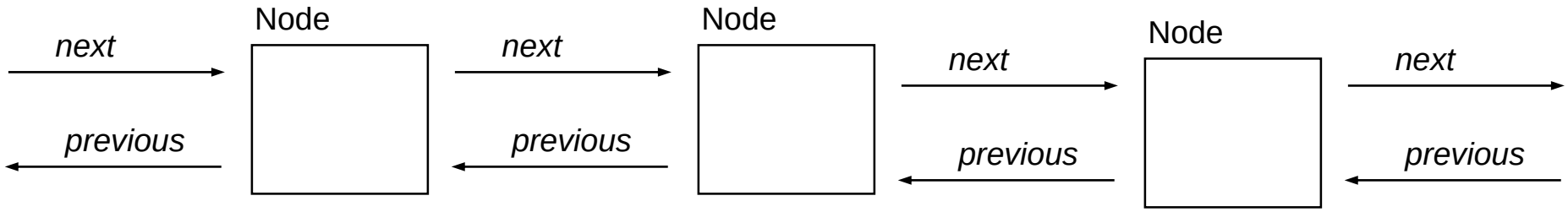
```
public class Vector3D {
    private double x, y, z;
    ...
    public double dotProduct(Vector3D u) {
        return x * u.x + y * u.y + z * u.z;
    }
    public double modulo() {
        return  Math.sqrt(dotProduct(this));
    }           //Math.sqrt(this.dotProduct(this));
}
```




```
// main method
Vector3D v = new Vector3D(2, -2, 1);
v.modulo();
```

Creating reverse links with `this`

```
public class Node {
    private Node previous;
    private Node next;
    …
    public void linkToNode(Node nextNode) {
        next = nextNode;
        nextNode.previous = this;
    }
}
```



Class variables (`static`)

```
public class Point {
    private long x, y;    // coordinates of each Point
    private static long nmrPoints = 0; // class variable

    public Point(long x, long y) {
        this.x = x;
        this.y = y;
        nmrPoints++;  // count each point that is created
    }
    …
}
```

`static` "not belonging to instances of the class but to the class itself"

If it was not declared **`private`** it would be accesible also from outside the class in two ways: **`Point.nmrPoints`** `//` preferred and **`p.nmrPoints`** `//` assuming `Point p;`

Class methods (`static`)

```
public class Math {   // predefined class in java.lang

   // class variable with a given final value, i.e., constant
   public static final double PI = 3.141592653589793;

   static long round(double a) {   // class method
      …
   }
   static double sin(double a) { … }
   …
}
```

`static` "not belonging to instances of the class but to the class itself"

In fact, **`Math`** has not instance variable/methods nor constructor,

it only has class variables/methods **`Math.sin(Math.PI / 2)`**

Exercise

Write a Java program that allows creating Parts (pieces) with a certain weight. Each part will be automatically assigned a unique identifier. The Part class must contain a static method that returns the heaviest part created so far.

Parameter passing: always by value (in Java)

```
class MainClass {
    public static void main(String[] args) {
        int n = 5;
        System.out.println("Before:   " + n);
        f(n);
        System.out.println("After: " + n);
    }
    static void f(int i) {
        i = i + 1;
        System.out.println ("Inside:  " + i);
    }
} // generates the following output
```

Parameter passing: always by value (in Java)

```
class MainClass {
    public static void main(String[] args) {
        int n = 5;
        System.out.println("Before:   " + n);
        f(n);
        System.out.println("After: " + n);
    }
    static void f(int i) {
        i = i + 1;
        System.out.println ("Inside:  " + i);
    }
} // generates the following output
//Before:   5
//Inside:  6
//After: 5
```

Parameter by value: recall *reference types* (I)

```
class MainClass {
  public static void main(String[] args) {
     BankAccount account = new BankAccount();
     account.balance = 100000;
     System.out.println("balance now: " + account.balance);
     empty(account);
     System.out.println("balance after:" + account.balance);
  } // end main
  static void empty(BankAccount acct) {
     acct.balance = 0;
     acct = null; //¿?
  }
} // output is:
//balance now: 100000
//balance after: 0
```

Parameter by value: recall *reference types* (II)

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

Method overloading: Example

```
public class Plane3D {
    private double a, b, c, d;
    // Plane equation:  a*x + b*y + c*z + d = 0
    public Plane3D (double aa, double bb,
                    double cc, double dd) {
        a = aa; b = bb; c = cc; d = dd;
    }
    public boolean isParallelTo(Plane3D p) {
        Vector3D u = new Vector3D(a, b, c);
        Vector3D v = new Vector3D(p.a, p.b, p.c);
        return u.isParallelTo(v);
    }
    public boolean isParallelTo(Line3D r) {
        Vector3D u = new Vector3D(a, b, c);
        return u.isPerpendicularTo(r.getVector());
    }
}
```

Same name, different *signature*

Calling overloaded methods: Example

```
public class Line3D {
    private Point3D point;
    private Vector3D vector;
    public Line3D (Point3D p, Vector3D v) {
        point = p; vector = v;
    }
    public Vector3D getVector() { return vector; }
}
```

```
// in main method
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.isParallelTo(p2);
p1.isParallelTo(r);
```

Method overload ambiguity

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

executes the most specific of the compatible definitions

```
// In main method
A a = new A();
byte   b = 3;
long   l = 3;
double d = 3;
a.f(l);
a.f(b);
a.f(d); //ERROR: explicit casting
        // is needed
```

Constructors

- They are not methods (but they are declared in a similarl way)
- They are not invoked on an object, but with **`new`** `AnyClass(…)`
- They are not components of the object (like instance methods)
- They are not directly accesible from methods, but they can be (implicitly and explicitly) invoked from other constructors (**`this`**, **`super`**).

- They are necessary to create objects:

  allocate their memory and initialize their components

- They are declared like methods, but without return type,

  sometimes with parameters, and always with the same name of their class. Often they are **`public`** but they don't need to be.

- There may be more than one constructor in a class,

  but always with different (number or types of) parameters.

Public constructors: examples

```
public class Client {
    private String name;
    private long id;
    public Client(String str, long num) {
        name = str; id = num;
    }
    // … methods
}
```

```
class BankAccount {
    private long number;
    private Client owner;
    private long balance;
    public BankAccount(long num, Client tit) {
        number = num; owner = tit; balance = 0;
    }
    // … methods
}
```

Public constructors: examples

```
class BankAccount {
   private long number;
   private Client owner;
   private long balance;

   public BankAccount(long num, Client tit) {
     number = num; owner = tit; balance = 0;
   }

   public BankAccount(long num, Client tit, long s) {
     number = num; owner = tit; balance = s;
   }

   // … methods
}
```

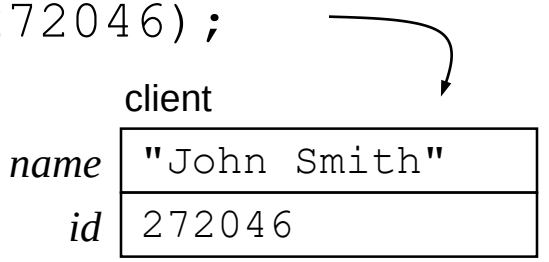
Public constructors: examples

```
class BankAccount {
    private long number;
    private Client owner;
    private long balance;

    public BankAccount(long num, Client tit) {
      this(num, tit, 0); // best!! Reusing code
    }

    public BankAccount(long num, Client tit, long s) {
      number = num; owner = tit; balance = s;
    }

    // … methods
}
```

Using constructors to create objects

Constructors are executed to create objects using operator **new**

```
Client client1 = new Client("John Smith", 272046);
```




```
BankAccount account1 =
    new BankAccount(683531, client1);

BankAccount account2 =
    new BankAccount(835284,
                    new Client("Mary Johnson", 151442),
                    2000);
```

Default constructor (without parameters)

- If no constructor is defined, Java provides a default constructor

```
class ClassX {
    // implicitly defines constructor ClassX() { }
    // allows creating objects by ClassX x = new ClassX();
}
```

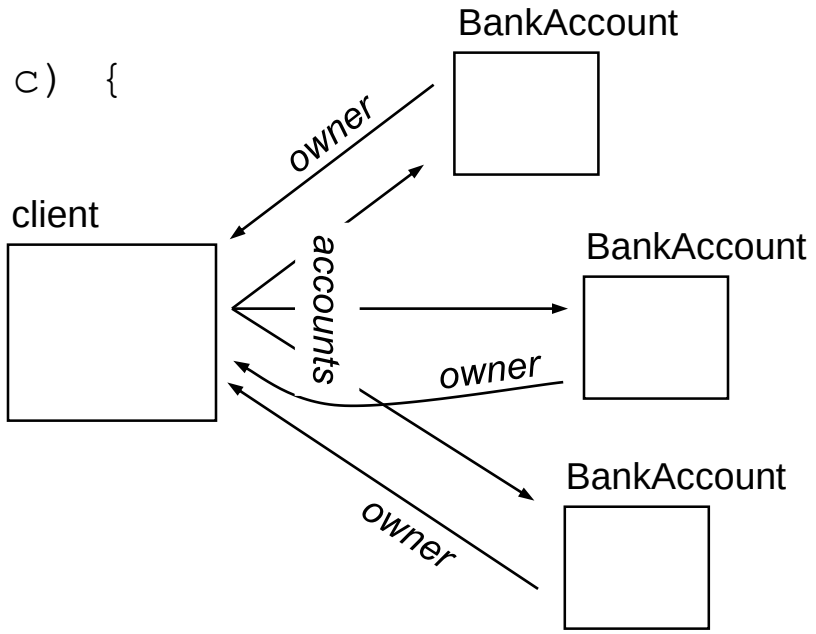
- If a constructor is defined, the default constructor is not created

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

```
// in main method
Client client1 = new Client();
// Error: No constructor matching Client() found in client
```

Using variable `this` inside constructors

```
public class BankAccount {
    private long number;
    private Client owner;
    private long balance = 0;
    public BankAccount(long num, Client c) {
        number = num; owner = c;
        c.addAccount(this);
        // owner.addAccount(this);
    }
}
```



Similar to creating reverse links

```
public class Client {
    // … name, id, … constructor …

    public static final int MAX_ACCT = 20;
    private BankAccount accounts[] = new BankAccount[MAX_ACCT];
    int nAccounts = 0;
    void addAccount(BankAccount account) {
        if (nAccounts < MAX_ACCT) accounts[nAccounts++]=account;
    }
}
```

Overloading constructors: different signatures

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

public class Vector3D {
    private double x, y, z;

    // Vector defined by origin at(0,0,0)
    // and vertex at the given coordinates
    public Vector3D(double xx, double yy, double zz) {
        x = xx; y = yy; z = zz;
    }

    // Vector defined by p and q points, origin and vertex
    public Vector3D(Point3D p, Point3D q) {
        x = q.x - p.x; y = q.y - p.y; z = q.z - p.z;
    }
    …
}
```

Constructors with diferent *signatures*

Private Constructors: Singleton Pattern

```
public class PrintQueue {
  private static PrintQueue INSTANCE;
  …
  // Private constructor prevents any instantation from other clases
  private PrintQueue() { }

  public static PrintQueue getInstance() {
    if (INSTANCE==null) INSTANCE = new PrintQueue();
    return INSTANCE;
  }

  public void addJob (Job j) {
    …
  }
}
```

```
public class Application {
 …
   public static void main(String [] args) {
      PrintQueue queue = PrintQueue.getInstance(); // There will be only
        …                                          // one PrintQueue object
   }
}
```

Quizzes

Classic Abstract Data Types (ADT)

- Internal data structure will be hidden from the outside (encapsulation) except by means of methods explicitly offered to that end.

- Basic elements and features of an ADT in Java:

  Public class

  Private instance variables (and class variables)

  Public constructor(s)

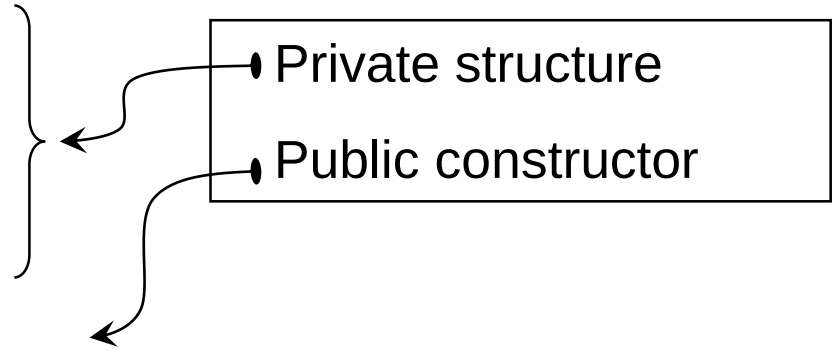
  Methods to get values of object components (*getters*)

  Methods to set values of object components (*setters*)

  Other methods to complete functionality proper to each ADT

Classic Abstract Data Types (ADT)

```
public class LEDSign {
  private String text;
  private int width;
  private double speed;
  private int posX, posY;
  private boolean visible;

  public Marquee(String t, int w, int x, int y) {
     text = t;  width = a; posX = x;  posY = y;
     visible = false;  speed = 1.0;
  }
```




```
} // end class LEDSign
```

Classic Abstract Data Types (ADT)

```
public class LEDSign {
  private String text;
  private int width;
  private double speed;
  private int posX, posY;
  private boolean visible;

  // constructor(s)



  // getters
  public String getText() { return text; }
  public int getWidth() { return width; }
  public int getX() { return posX; }
  public int getY() { return posY; }
  public boolean getVisible() { return visible; }



} // end class LEDSign
```

getter methods: return the value of object components, to allow programming using those values (but without creating a dependecy with the internal structure or implementation details of the object).

Classic Abstract Data Types (ADT)

```
public class LEDSign {
  private String text;
  private int width;
  private double speed;
  private int posX, posY;
  private boolean visible;

  // constructor(s)
  // getters

  // setters
  public void setText(String t) { this.text = t; }
  public void setWidth(int w) { this.width = w; }
  public void setPosition(int x, int y) {
     posX = x; posY = y;
  }
  public void setVisible() { visible = true; }
  public void setInvisible() { visible = false; }

} // end class LEDSign
```

setters methods: only when required, allow client code to change the value of object components, but in a controlled manner and independently of object internal details.

Classic Abstract Data Types (ADT)

```
public class LEDSign {
  private String text;
  private int width;
  private double speed;
  private int posX, posY;
  private boolean visible;

  // constructor(s)
  // getters
  // setters
  // other methods
  public void revUp() { speed *= 1.25; }
  public void slowDown() {speed *= 0.8; }
  public void startScroll() {
     …
  }
  public void stopScroll() {
     …
  }
} // end class LEDSign
```

Other methods complete the range of operations allowed on objects of this class

Generic data types (generic classes)

- They are parameterized by a *base type*
- When used for declaring a variable the base type is made concrete

```
Vector<Point> myVectorOfPoints;
```

Advantages:

- Maximize code reusability (facilitate maintenance)
- The compiler does no longer require certains explicit castings
- More errors detected at compile time, rather than during execution
- Java provides many generic classes, particularly for *collections*: vector, stack, list, direct-access list (`ArrayList`)
- We can also define our own generic classes

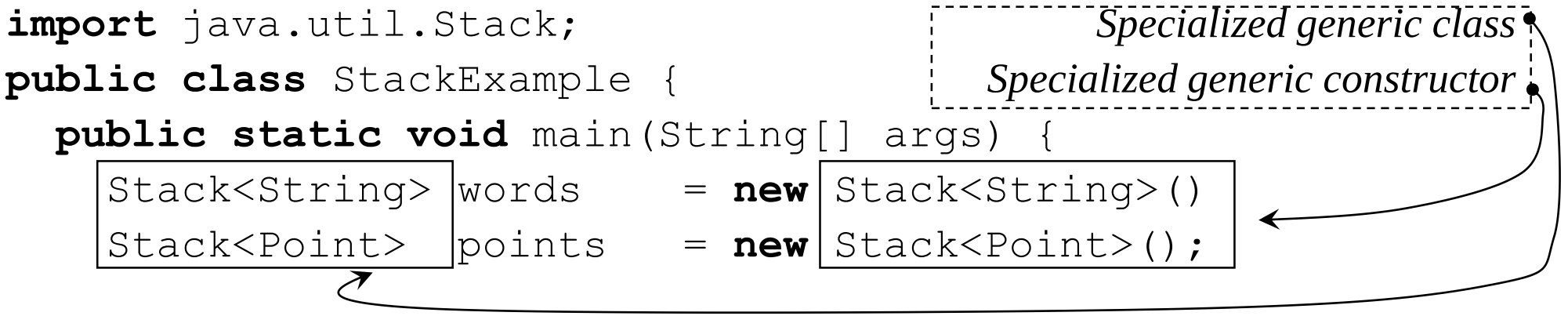
Old Java Class: non-generic stack

Hard to use a stack without knowing the type of elements it stores

```
import java.util.Stack;
public class StackExample {
  public static void main(String[] args) {
    Stack st = new Stack(); //  stack of unknown-type objects

    st.push("book");
    st.push(new Point(1,2));

    // this would be a compilation error: incompatible types
    // Point p = st.pop();
    Point p1 = (Point) st.pop();  // casting mandatory

    // and now an identical pop raises an execution error
    Point p2 = (Point) st.pop();  // why is this an error?
}
```

Generic Class: parameterized Stack

Better to use the generic Stack class to create specialized stacks




```
import java.util.Stack;
public class StackExample {
  public static void main(String[] args) {
    Stack<String> words  = new Stack<String>()
    Stack<Point>  points = new Stack<Point>();

    words.push("book");          words.push("block");
    points.push(new Point(1,2)); points.push(new Point(3,0));

    Point p1 = points.pop();  // no casting needed
    String s2 = words.pop();

    System.out.println("Word: " + words.pop());
    System.out.println("Point: " + points.pop());
} // prints  Word: book  and  Point: Point@1bc4459  What!?
```

Class Point with a conversion to String

In the previous example the Point object would have been displayed better defining a public method named `toString()`

```
public class Point {
  private int x, y;
  // constructor
  public Point(int x, int y) { this.x = x;  this.y = y; }
  // getters
  public int getX() {return x;}
  public int getY() {return y;}
  // setters
  public void setX(int x) {this.x = x;}
  public void setY(int x) {this.y = y;}
  // conversion to String
  public String toString() {return "(" + x + "," + y + ")";}
}
```

// now the previous example prints `Word: book` and `Point: (1,2)`

Old class **`Stack`** *was* documented in its API

`download.oracle.com/javase/1.4.2/docs/api/`

Constructor Summary

```
Stack() // Creates an empty Stack.
```

Method Summary

```
boolean empty() // Tests if this stack is empty.
Object peek() // Looks at the object at the top of this
              // stack without removing it from the stack
Object pop() // Removes the object at the top of this
             // stack and returns that object as the
             // value of this function.
Object push(Object item) // Pushes an item onto the top
                         // of this stack.
int    search(Object o) // Returns the 1-based position
                        // where an object is on this stack.
```

Other predefined generic class: `ArrayList`

- A list that provides fast access to elements by position index
- Implemented using resizable-arrays that allow lists to grow
- Adding *n* elements requires only time *O(n)*
- Operations more efficient than those on linked lists

- Example: to compute a table of frequencies for each word stored in a file (one word per each line), we can use two **`ArrayList`** objects with positionally related elements:

  one will store the words and the other the frequency of the word that holds the same position as the former

Example: using **ArrayList**

```
import java.io.*;
import java.util.*;
public class Frequencies {
  public static void main(String[] args) throws IOException {
    BufferedReader buffer =
                new BufferedReader(
                     new InputStreamReader(
                          new FileInputStream("words") ) );
    ArrayList<String> words = new ArrayList<String>();
    ArrayList<Integer> freq = new ArrayList<Integer>();
    String w;
    while ((w = buffer.readLine()) != null) {
       if (words.contains(p)) {
            int i = words.indexOf(w);
            freq.set(i, freq.get(i) + 1);
       } else { words.add(w);  freq.add(1); }
    }
    for (int k=0; k < words.size(); k++)
      System.out.println(words.get(k)+": "+freq.get(k));
    buffer.close();
  }
} // end class Frequencies
```

Better with a HashMap

Class **`ArrayList`** is documented in its API

`docs.oracle.com/javase/8/docs/api/`

Constructor Summary (here we show only two selected constructors)

```
ArrayList() // Constructs empty list, initial capacity 10
ArrayList(int initialCapacity)
```

Method Summary (solo métodos seleccionados)

```
void add(int index, Object element) // insert at index
boolean add(Object o) // Appends element to the end
void clear() // Removes all of elements from this list
boolean contains(Object elem)
Object get(int index) // Returns element at position index
int indexOf(Object elem) // Searches for first occurence
Object set(int index, Object element)
boolean isEmpty()                    | int lastIndexOf(Object elem)
Object remove(int index)             | int size()
```

Enumeration Data Types

- More sophisticated than in other languages (e.g., Pascal)
- They are defined as **`enum`** classes
- Each value of the enumeration is similar to a constant object
- Each value of the enumeration will print as its **`String`** name
- Class method **`values()`** returns an array with all values of the enumeration in the order they were declared
- The internal representation of each value may be hidden,

  or can be controlled explicitly
- Let us see some useful examples,

  but some details of the enum classes will be better understood later

Enumeration: a simple example

Enumeration with hidden internal values. Enumeration objects can be printed and are accesible with predefined method **`values()`**

```
public class CourseEnumeration{
  enum  Course {              // enumeration objects
      ALGEBRA, CALCULUS, PHYSICS, PROGRAMMING, WORKSHOP;
  }
  public static void main(String[] args) {
    Course c = Course.PHYSICS;
    System.out.println("is: " + c); //prints is: PHYSICS
    Course[] courses = Course.values();
    for (Course x : courses) System.out.println(x);
  }
}
```

```
ALGEBRA
CALCULUS
PHYSICS
PROGRAMMING
WORKSHOP
```

Enumeration types: setting internal value

Private internal value specified when invoking the private constructor

```
public class EnumDay {
  enum Day {
    MONDAY(1), TUESDAY(2), WEDNESDAY(3), THURSDAY(4),
    FRIDAY(5), SATURDAY(6), SUNDAY(0);

    private Day(int d) { value = d; } // private constructor
    private final int value;    // controlled internal value

    public int valor() { return value; }
  }

  public static void main(String[] args) {
    Day dia = Day.FRIDAY;
    …
    Day[] week = new Day[Day.values().length];
    for (Day d : Day.values()) { week[d.value()] = d; }
    for (Day d : Day.values())
      System.out.println("week["+d.value()+"] is " + d);
  }
}
```

```
week[1] is MODAY
week[2] is TUESDAY
week[3] is WEDNESDAY
week[4] is THRUSDAY
week[5] is FRIDAY
week[6] is SATURDAY
week[0] is SUNDAY
```

Enumerations: with internal values and with additional methods

```
public enum Planet {                    // file Planet.java
  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);

  Planet(double m, double r) { mass = m; radius = r; }

  private final double mass;   // kg
  private final double radius;  // meters

  private double mass() { return mass; }
  private double radius() { return radius; }

  // Gravitational constant
  public static final double G = 6.673E-11;
  // additional methods
  double surfaceGravity(){return G * mass/(radius*radius);}
  double weight(double mass){return mass*surfaceGravity();}
}
```

Enumerations as components of other objects

```
public class Card { // Cards in the classic Spanish deck

    public enum Rank { ACE, TWO, THREE, FOUR, FIVE,
        SIX, SEVEN, JACK, KNIGHT, KING}

    public enum Suit { OROS, ESPADAS, COPAS, BASTOS }

    // private components of each Card object
    private final Rank rank;
    private final Suit suit;

    // public constructor
    public Card(Rank r, Suit s) { rank = r; suit = s; }

    public Rank rank() { return rank; }
    public Suit suit() { return suit; }

    public String toString() {return rank + " of " + suit;}

}
```