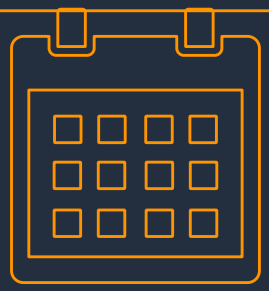


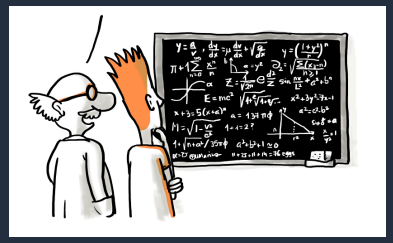
Ενότητα 2: **Διαχείριση Μνήμης** και Σχεδιασμός Κλάσεων

Μάθημα 11: 27/2/2024

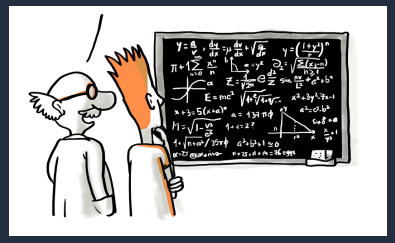
Κατασκευαστές
Υπερφόρτωση Μεθόδων
Παραδείγματα Διαχείρισης Μνήμης
Παραδείγματα Λειτουργίας Στοίβας και Σωρού
Αποκομιδή σκυβάλων

University of Cyprus
Department of Computer Science

Προηγούμενα



- Οργάνωση μνήμης Η/Υ
- Διαχείριση μνήμης από το Λ.Σ.
- Εικονική μνήμη (virtual memory), εικονικός χώρος διευθύνσεων (virtual address space)
- Οργάνωση και (δυναμική) διαχείριση εικονικής μνήμης: σωρός, στοίβα, εγγραφήματα δραστηριοποίησης
- Διεργασίες (process)
- Οργάνωση μνήμης JVM / περιοχές μνήμης
- Αρχικοποίηση και Κατασκευή αντικειμένων



Ενότητα 2: **Διαχείριση Μνήμης** και Σχεδιασμός Κλάσεων

Παράδειγμα διαχείρισης μνήμης

University of Cyprus
Department of Computer Science

Παράδειγμα Διαχείρισης Μνήμης

Class Car

java.lang.Object
    Car

```
public class Car
extends java.lang.Object
```

Constructor Summary

Constructors

Constructor and Description
`Car(int vin)` constructor

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`int`	`getOdometer()` reads and returns the Car's odometer reading
`void`	`moveCar(int dist)` increases the odometer's value by the driven distance

Constructor Detail

Car

```
public Car(int vin)
```

constructor

Parameters:

`vin - is the identification number given to the car at the factory`

Method Detail

moveCar

```
public void moveCar(int dist)
```

increases the odometer's value by the driven distance

Parameters:

`dist - is the distance driven by the car; positive if car moves forward, negative if car moves backwards`

getOdometer

```
public int getOdometer()
```

reads and returns the Car's odometer reading

Returns:

`odometer value`

```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
   return newC;
  }

  /* runs numTests on a given Car */
  private void testCar(Car testC) {
     testC.moveCar(10);
     testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```

Παράδειγμα διαχείρισης μνήμης

Η λειτουργία της στοίβας

University of Cyprus
Department of Computer Science

```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```

Thread Stack

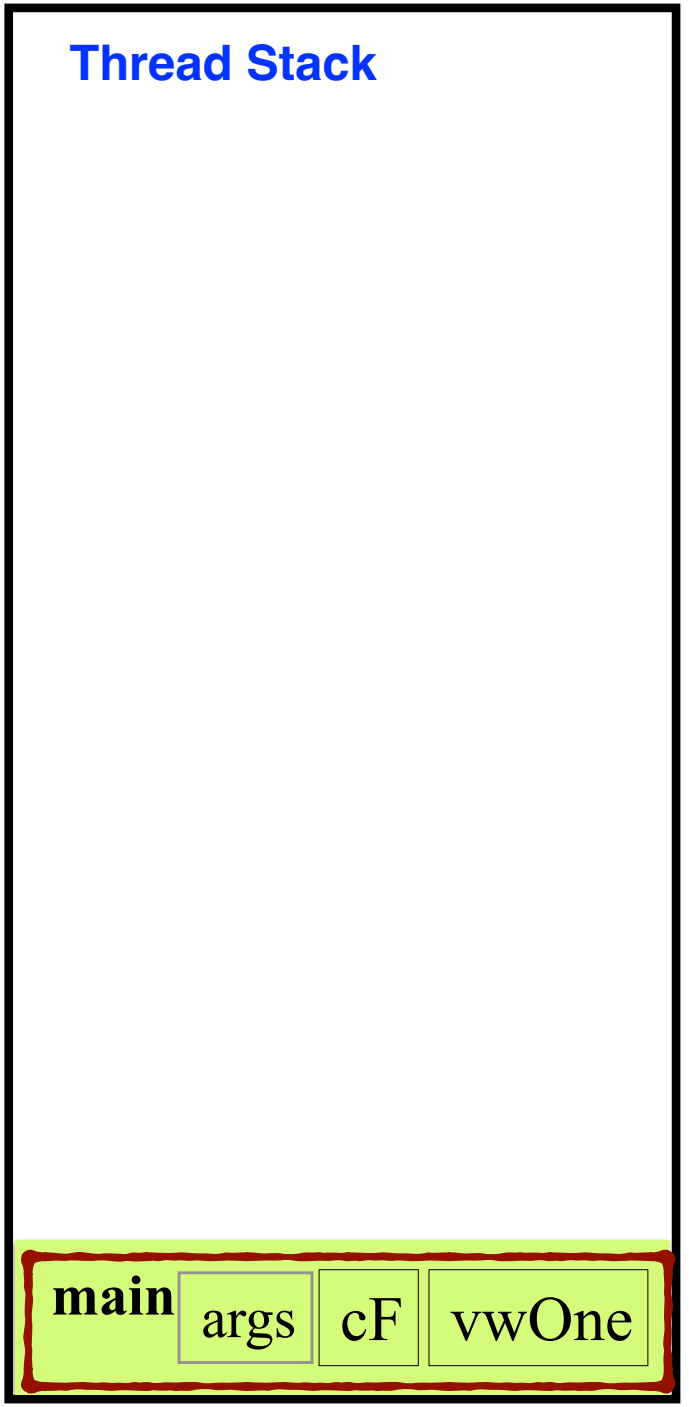
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



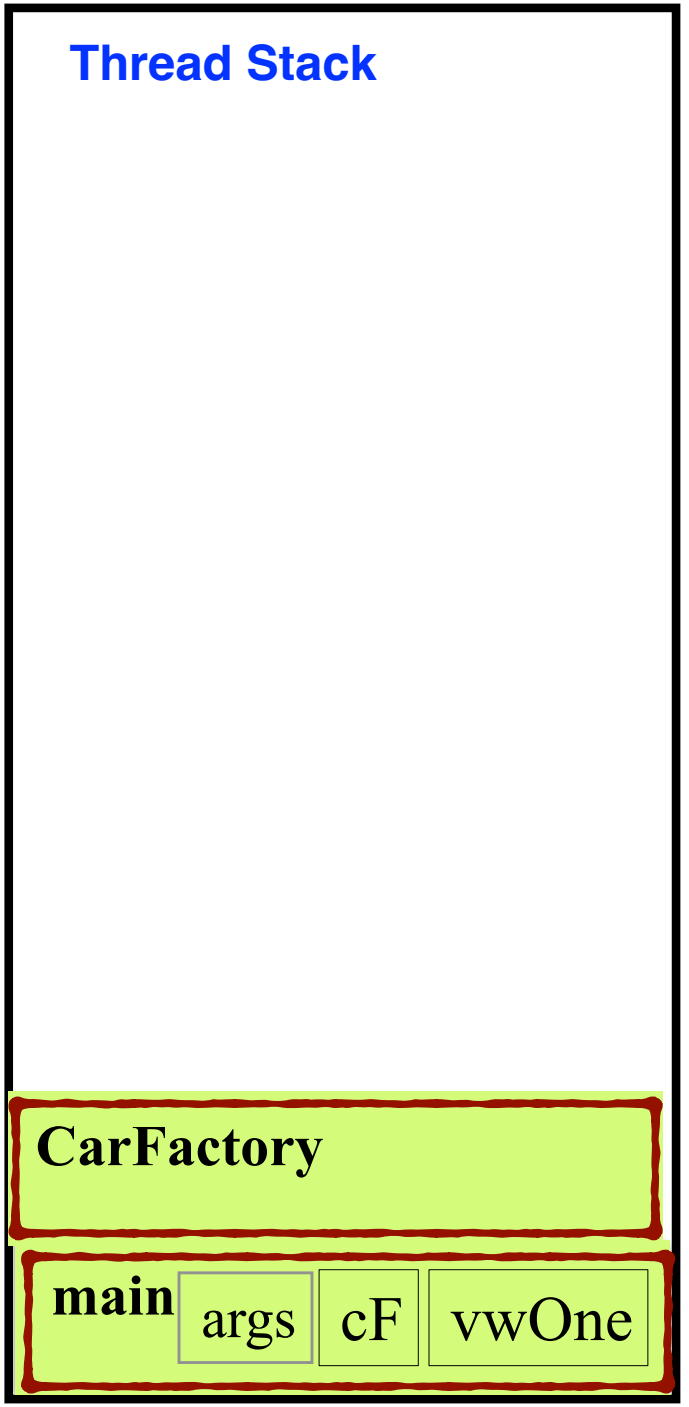
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



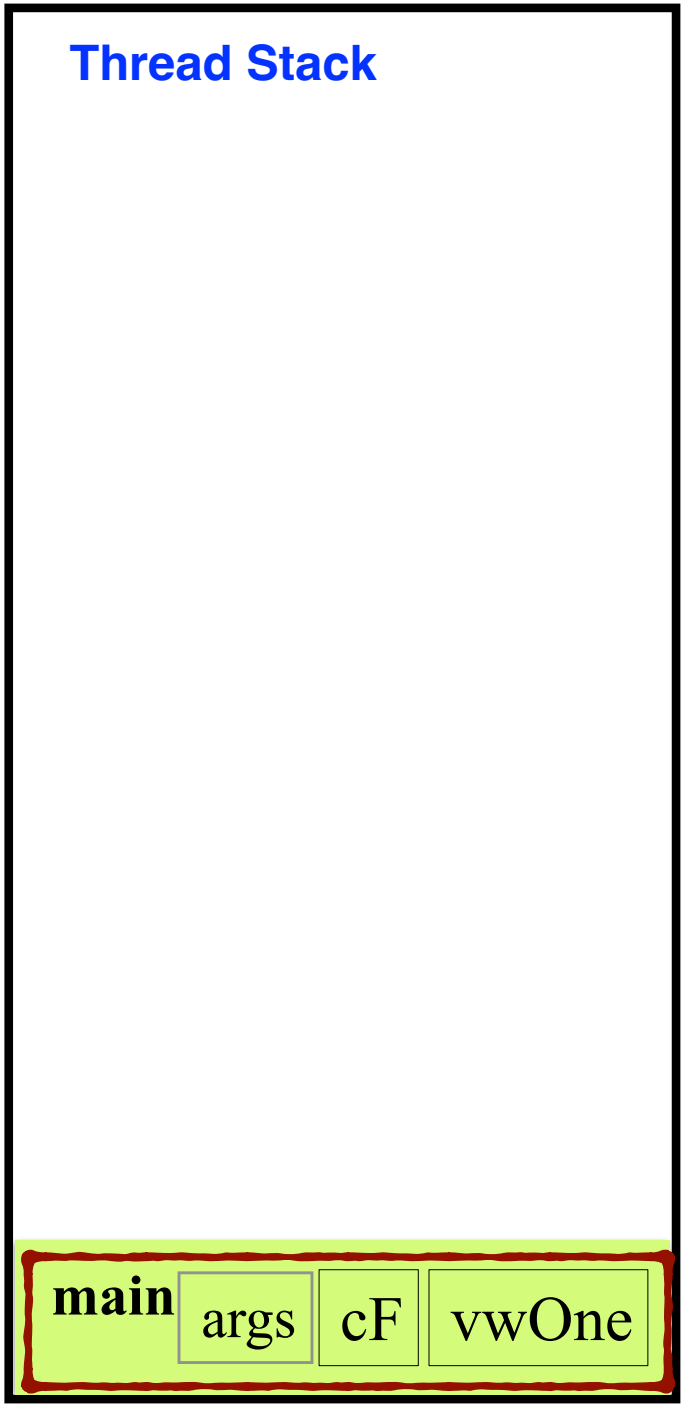
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



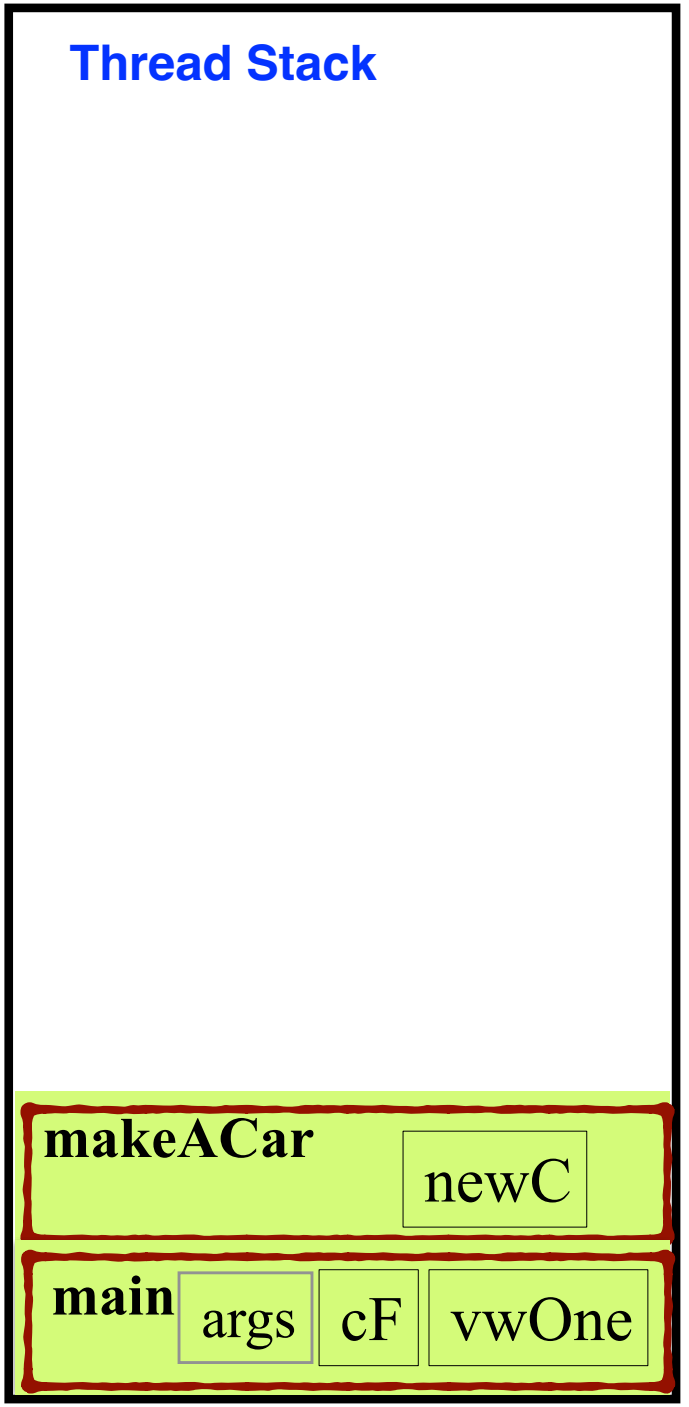
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



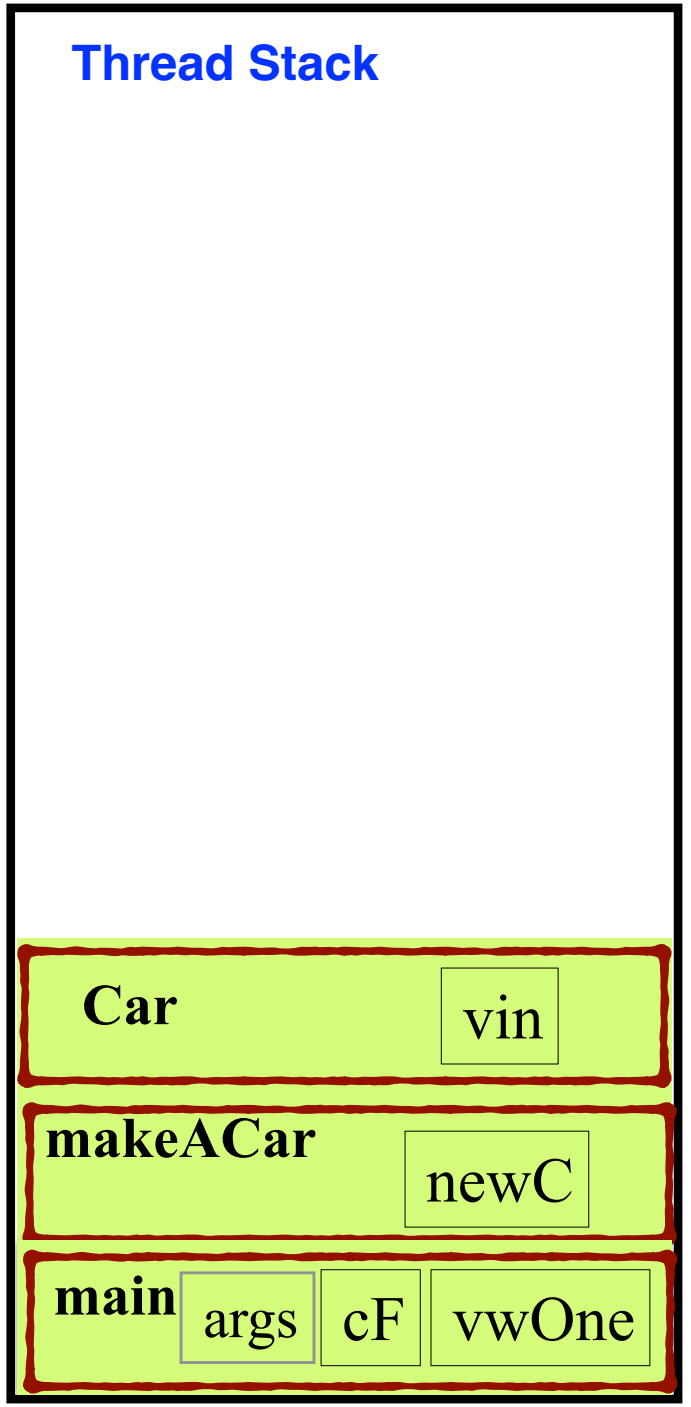
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



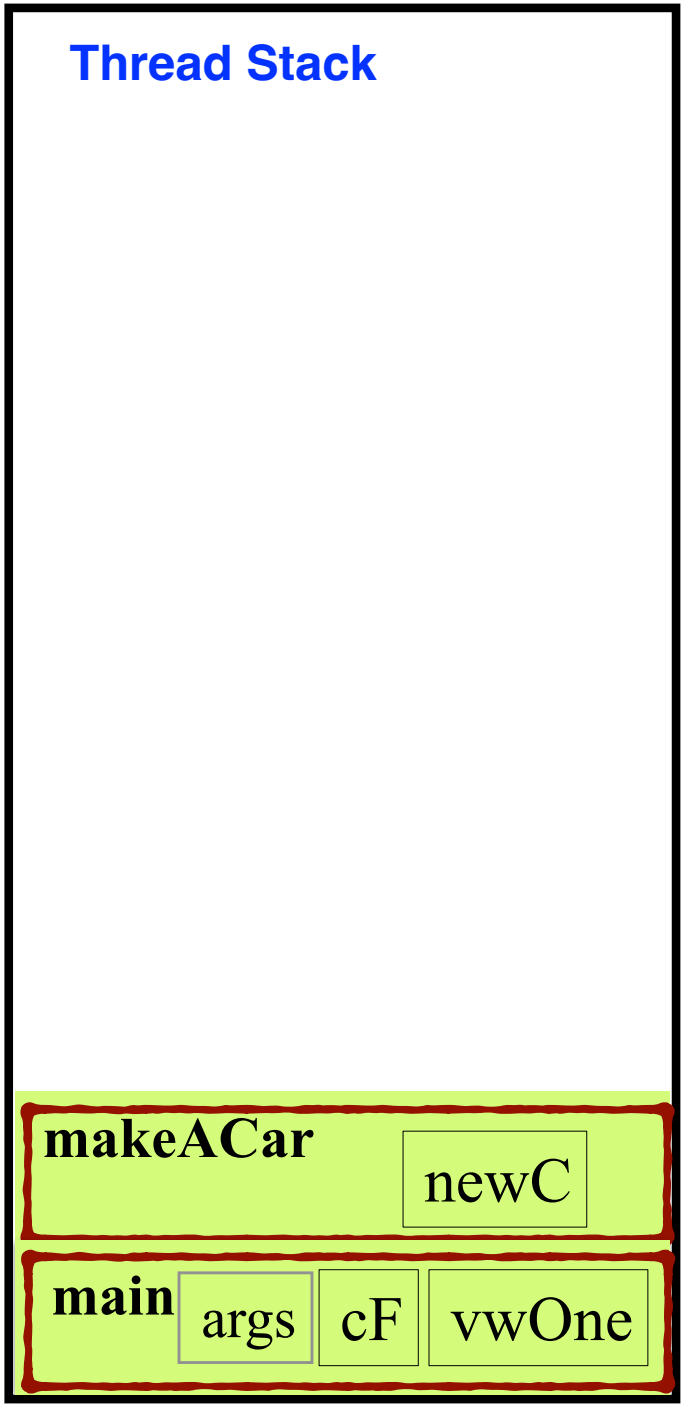
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



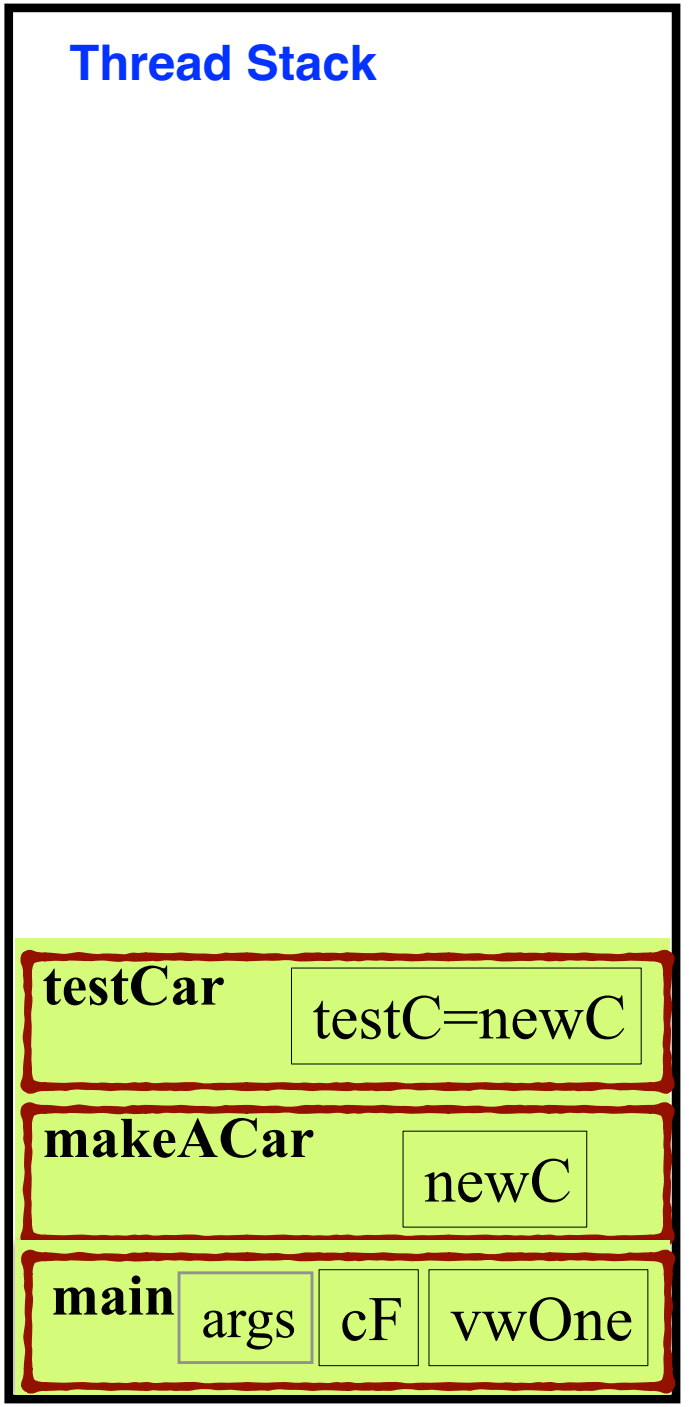
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



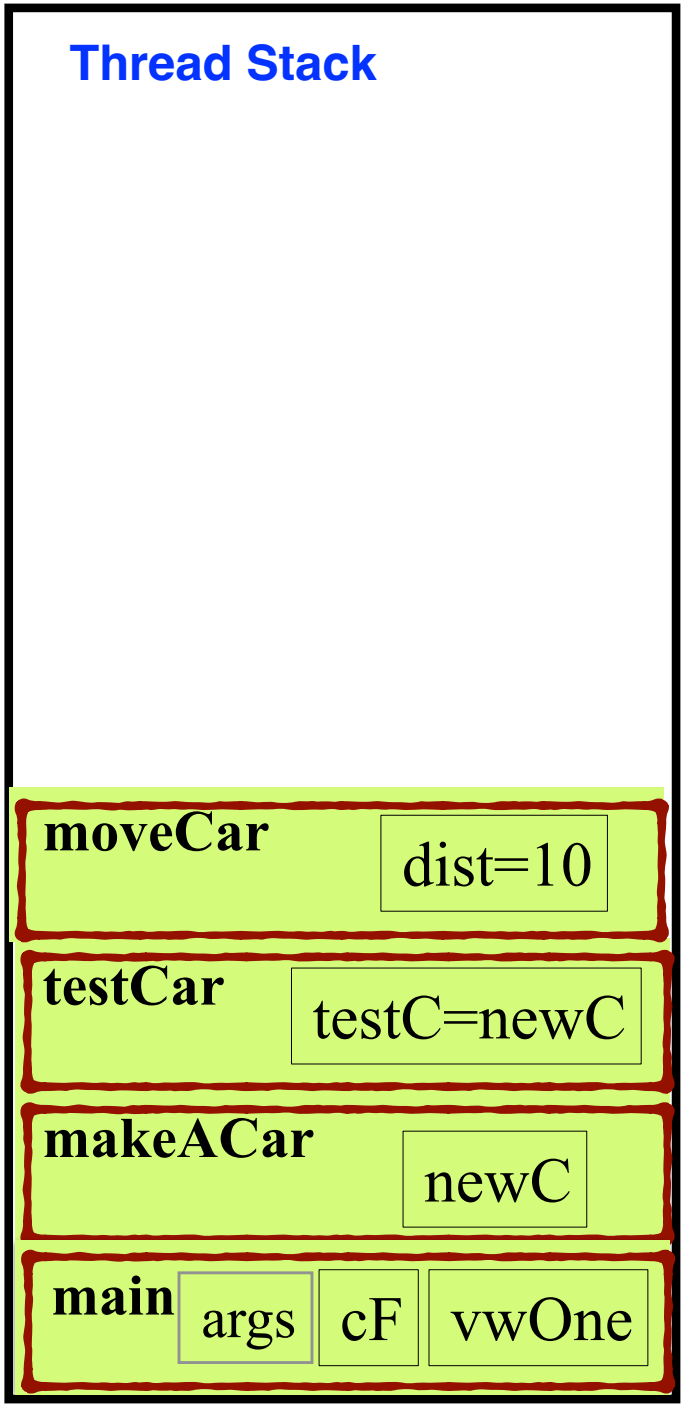
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



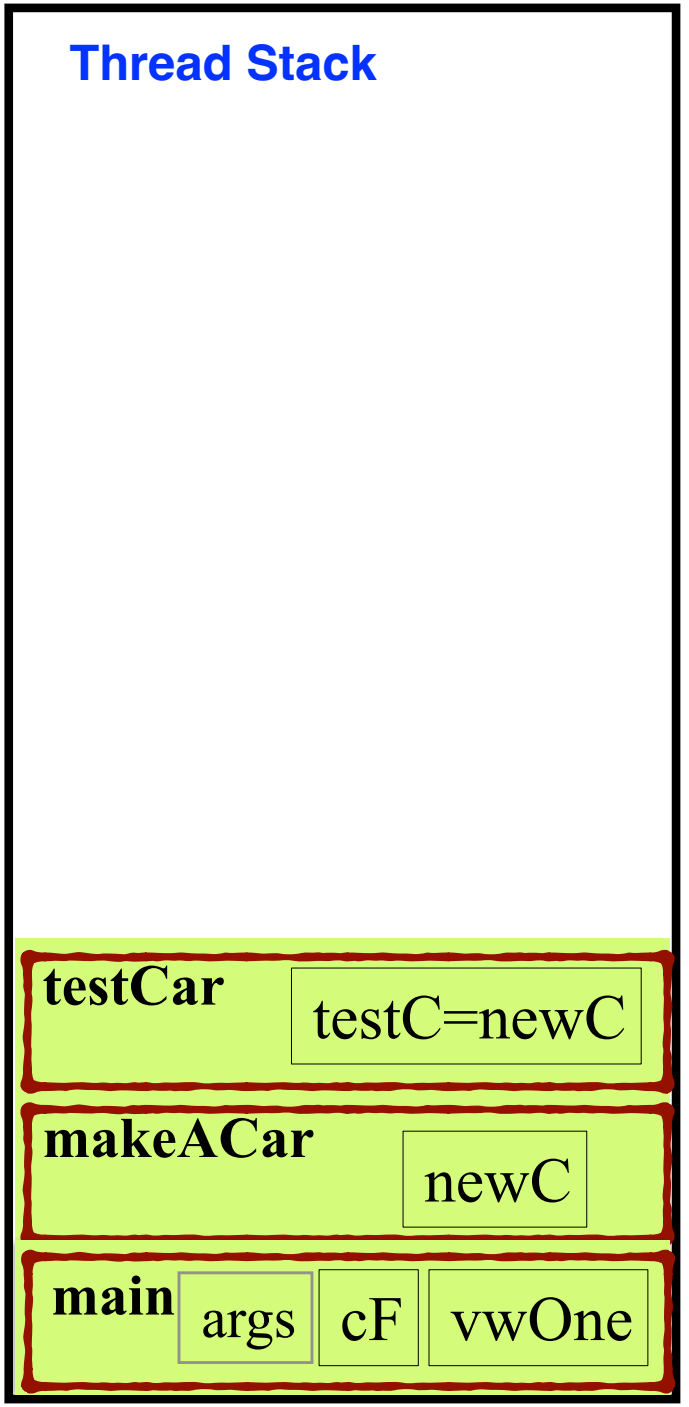
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



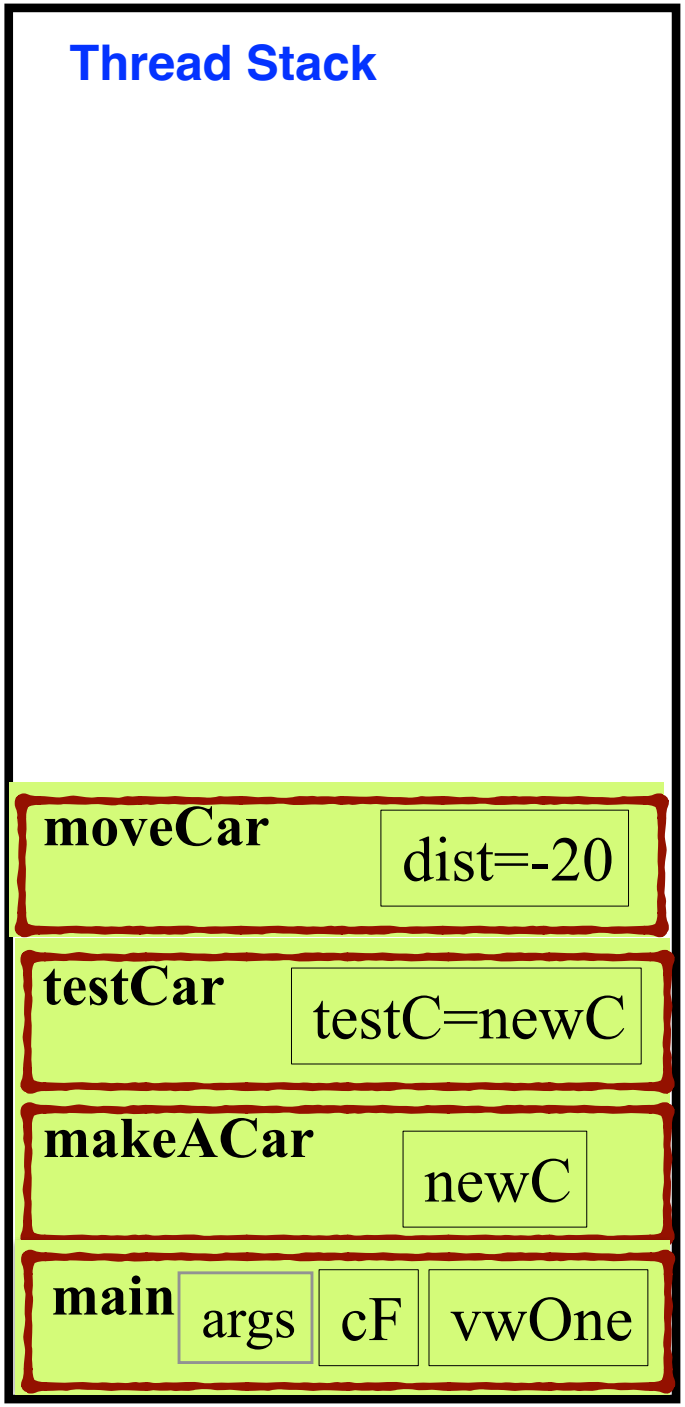
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



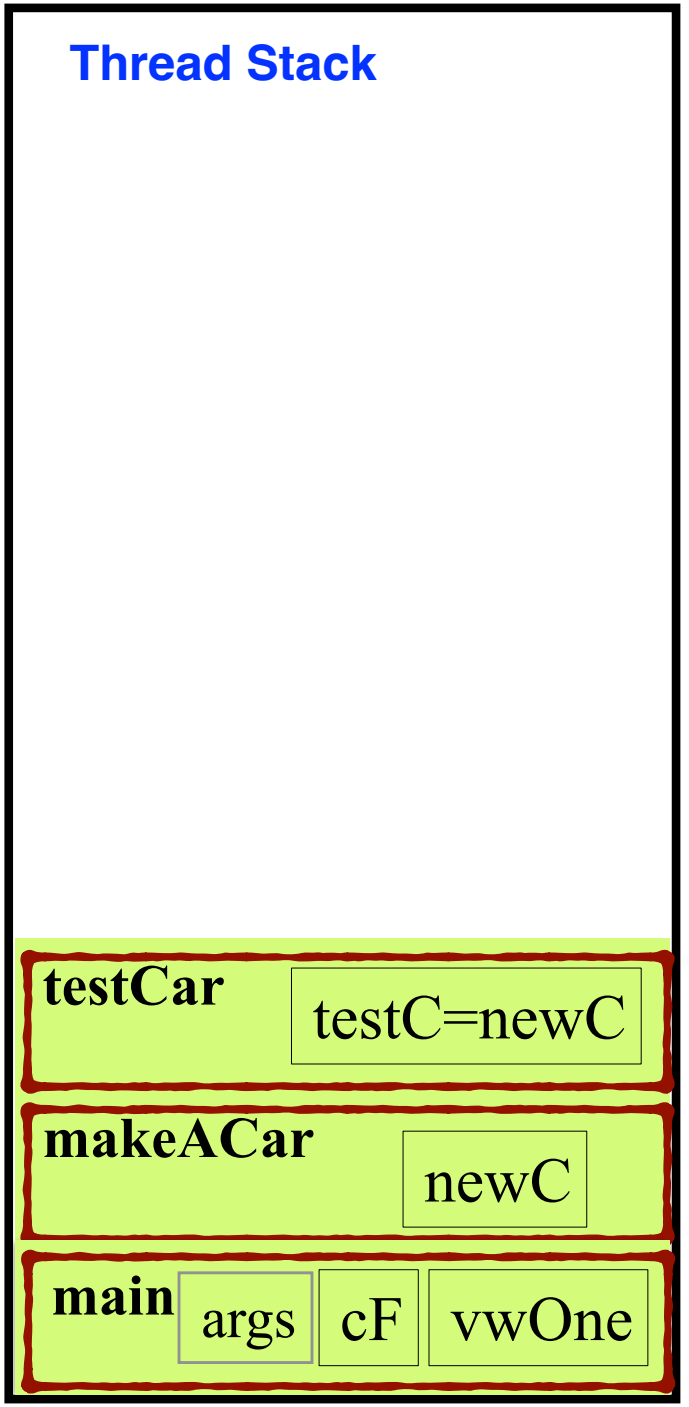
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



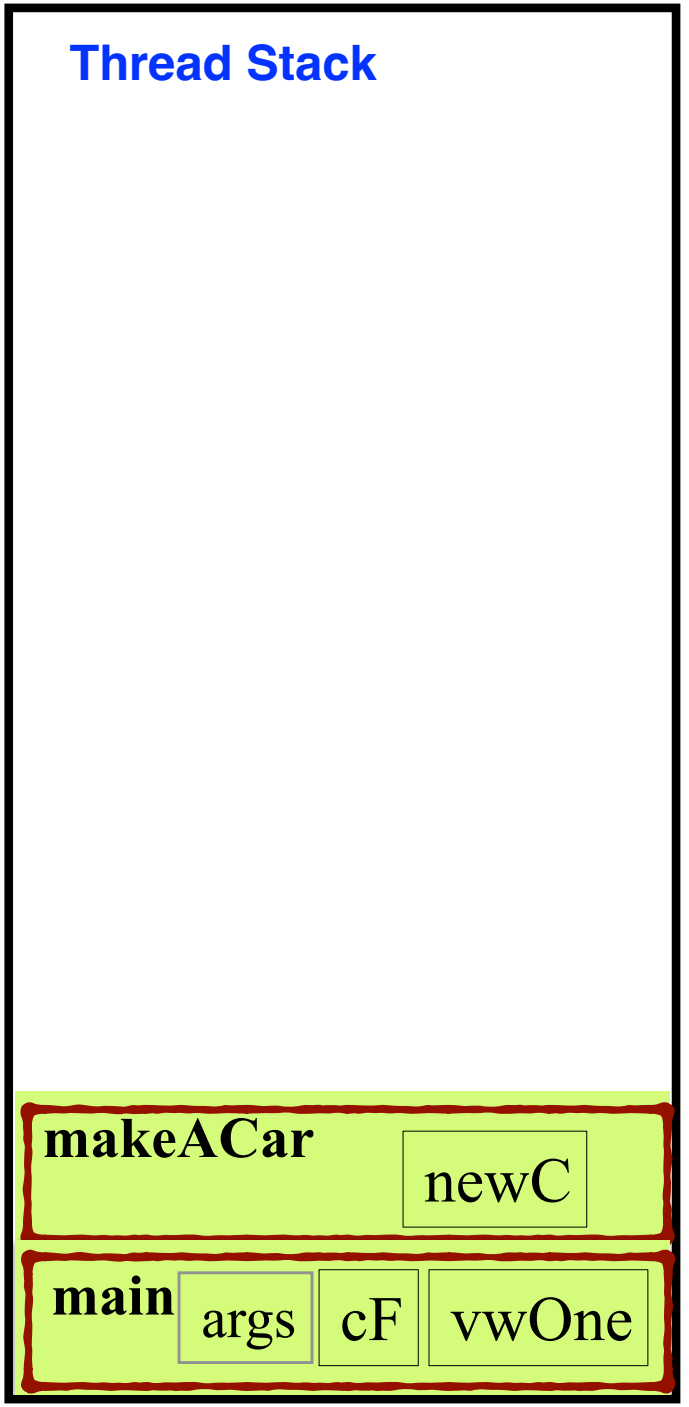
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```

Thread Stack

```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```

Παράδειγμα διαχείρισης μνήμης

Η λειτουργία του σωρού

University of Cyprus
Department of Computer Science

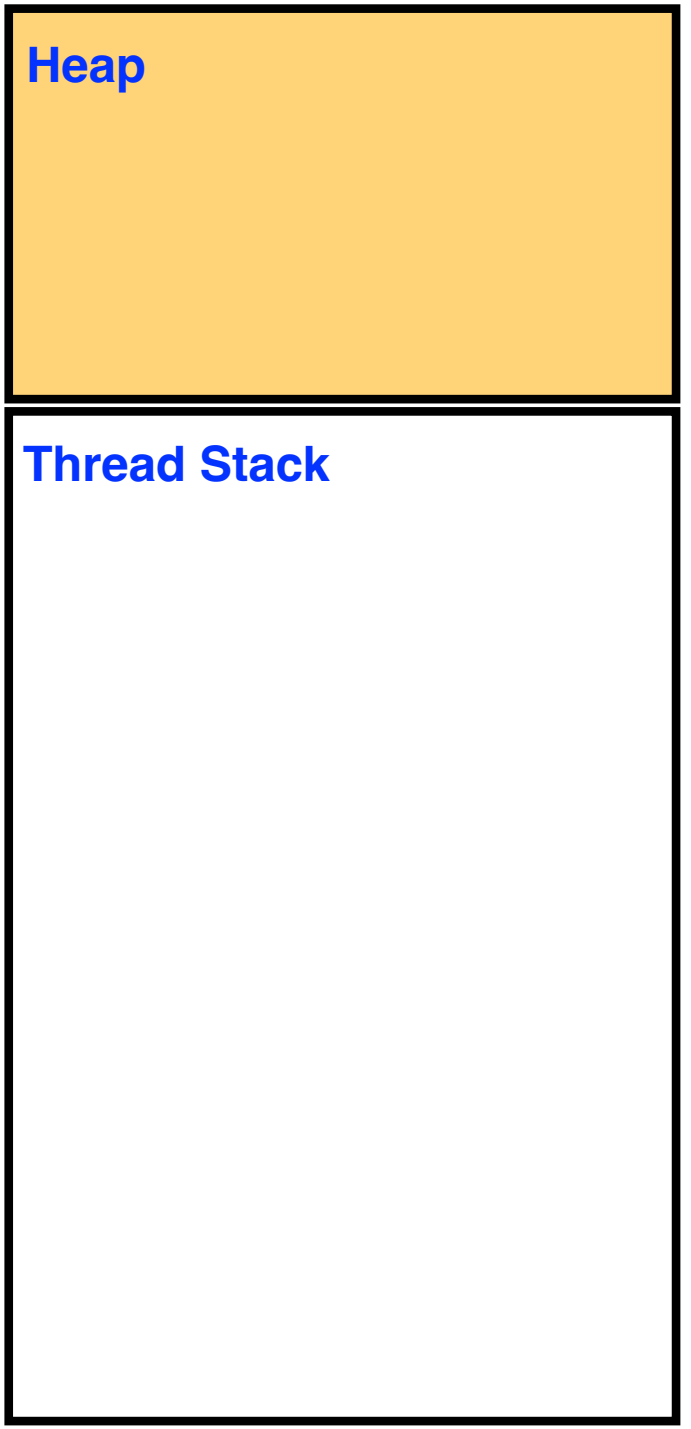
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
   return newC;
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



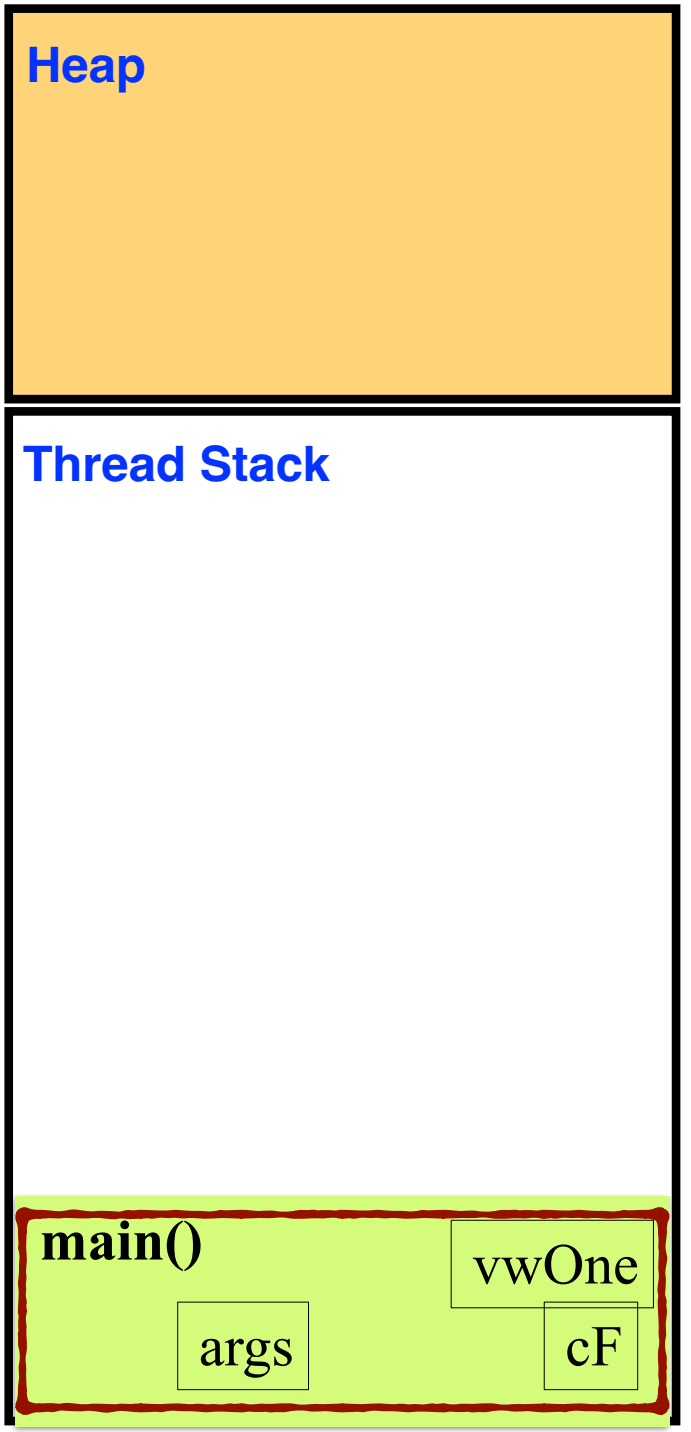
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
   return newC;
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



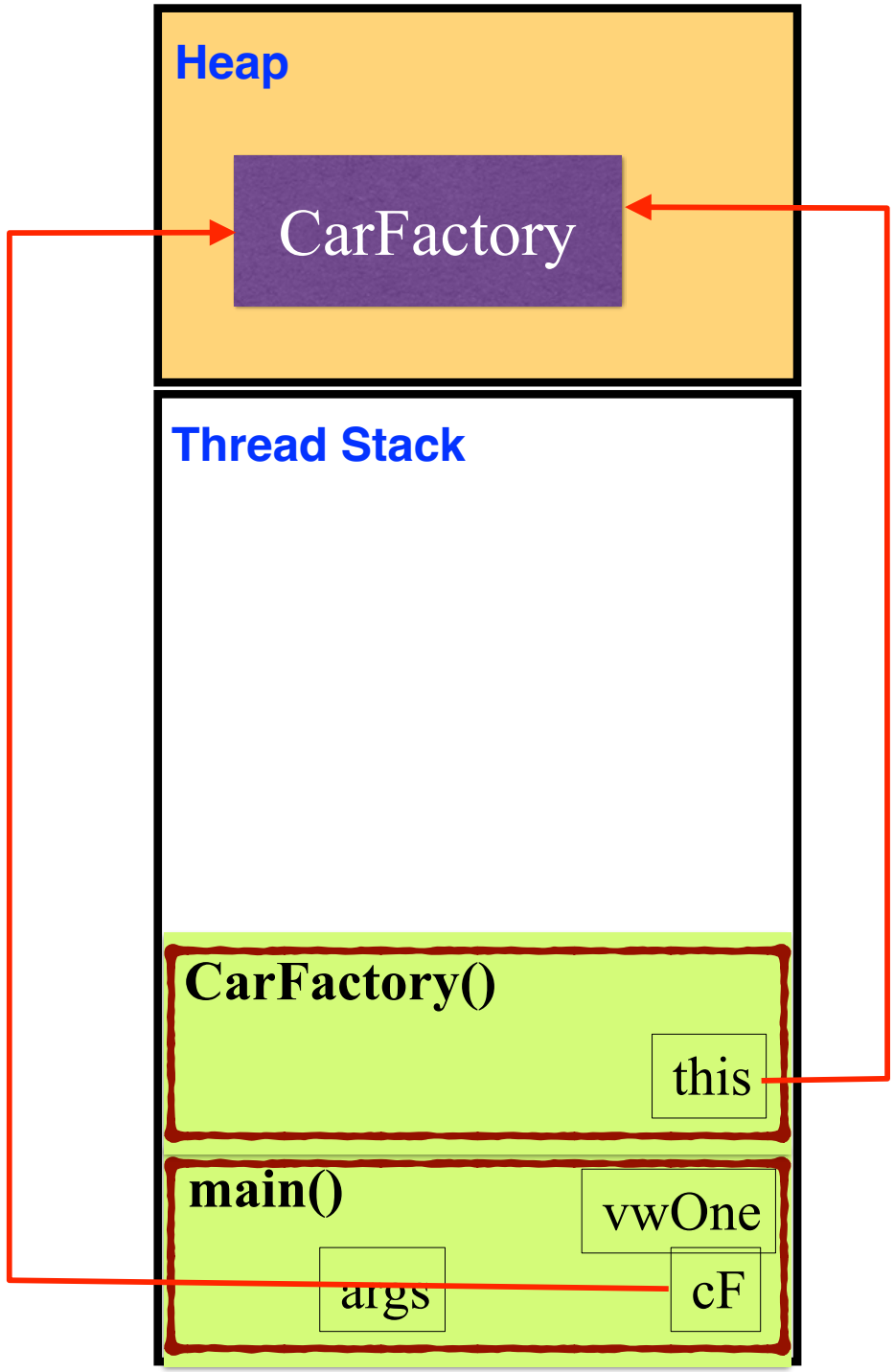
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
   return newC;
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



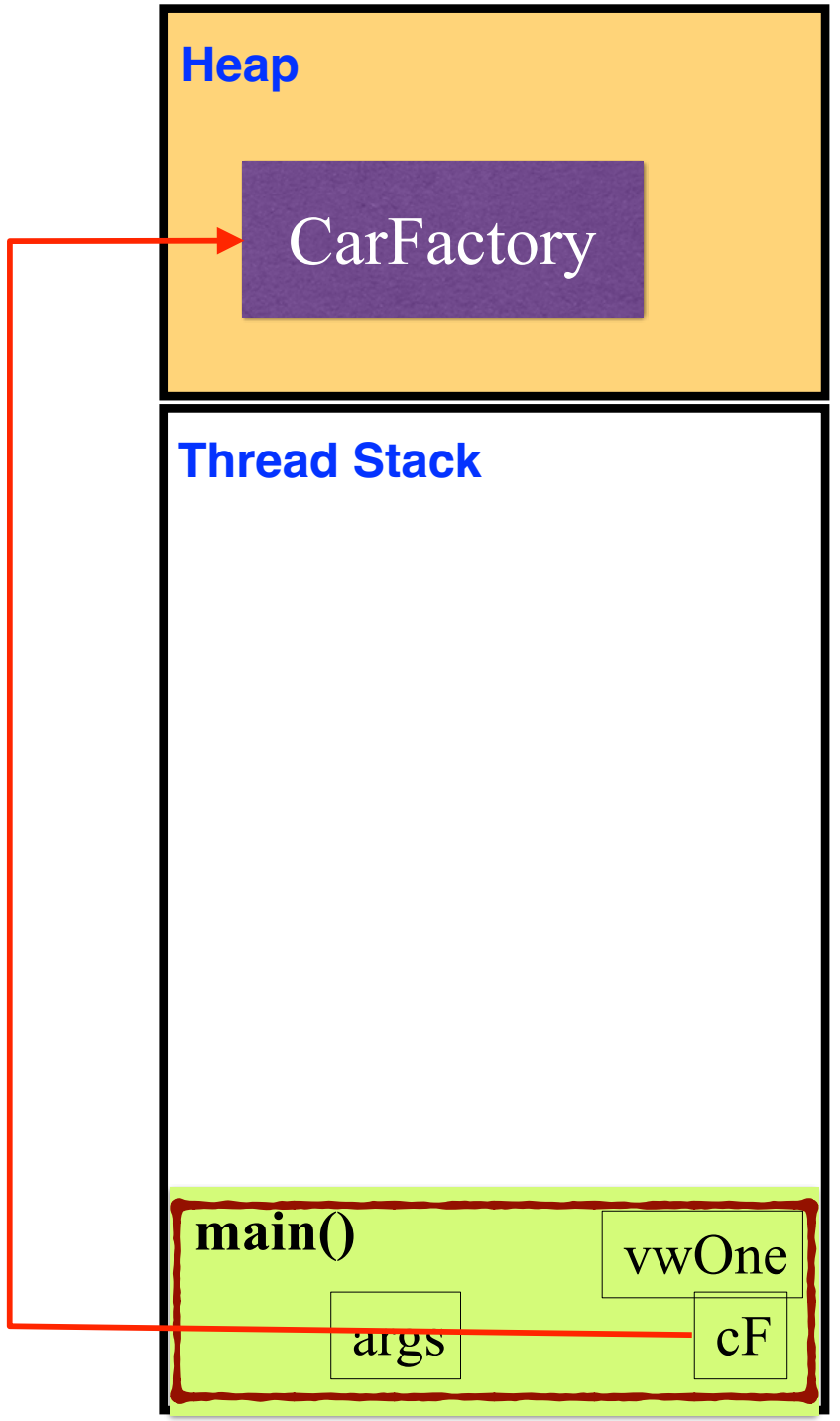
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
   return newC;
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



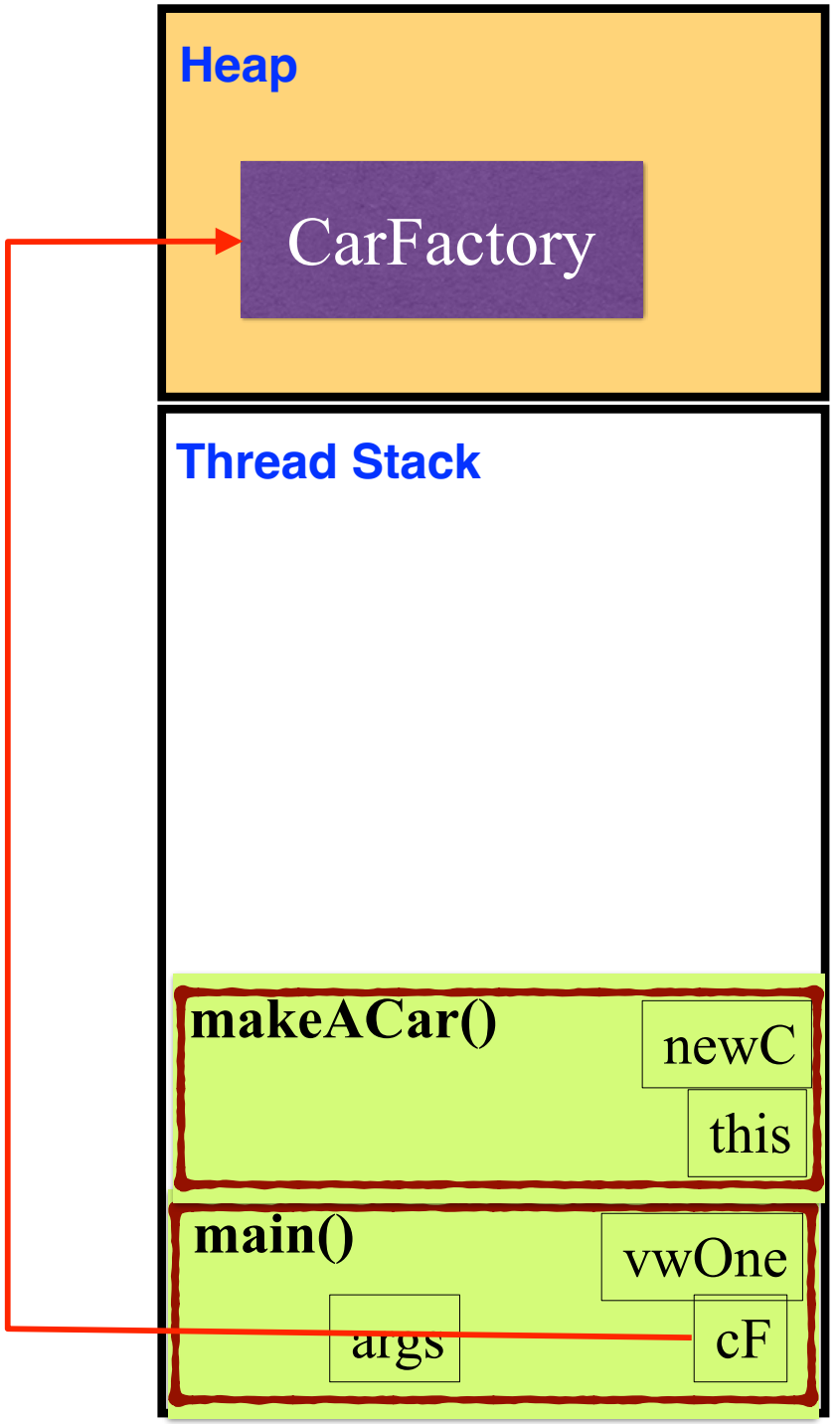
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
   return newC;
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



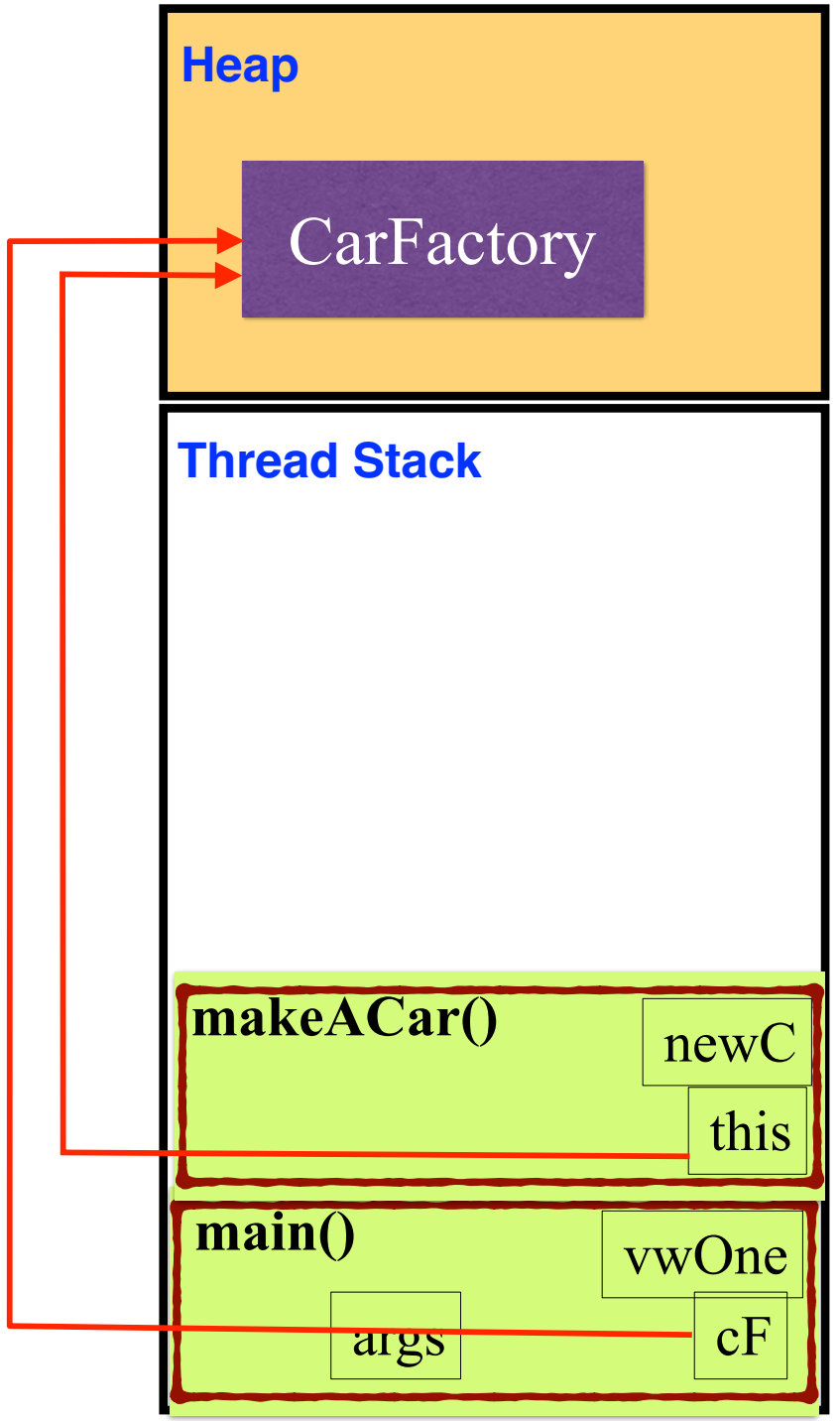
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
   return newC;
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



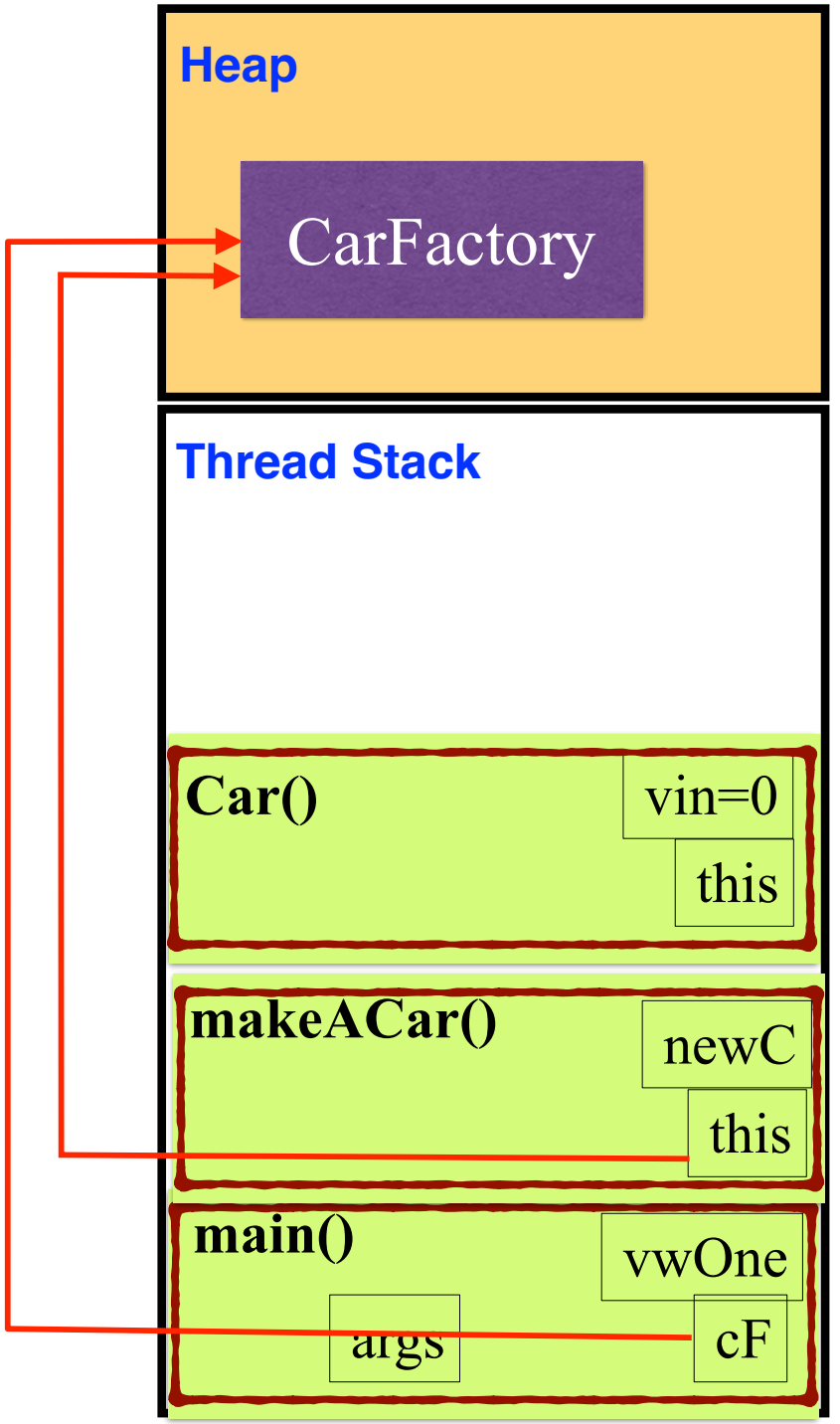
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
   return newC;
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



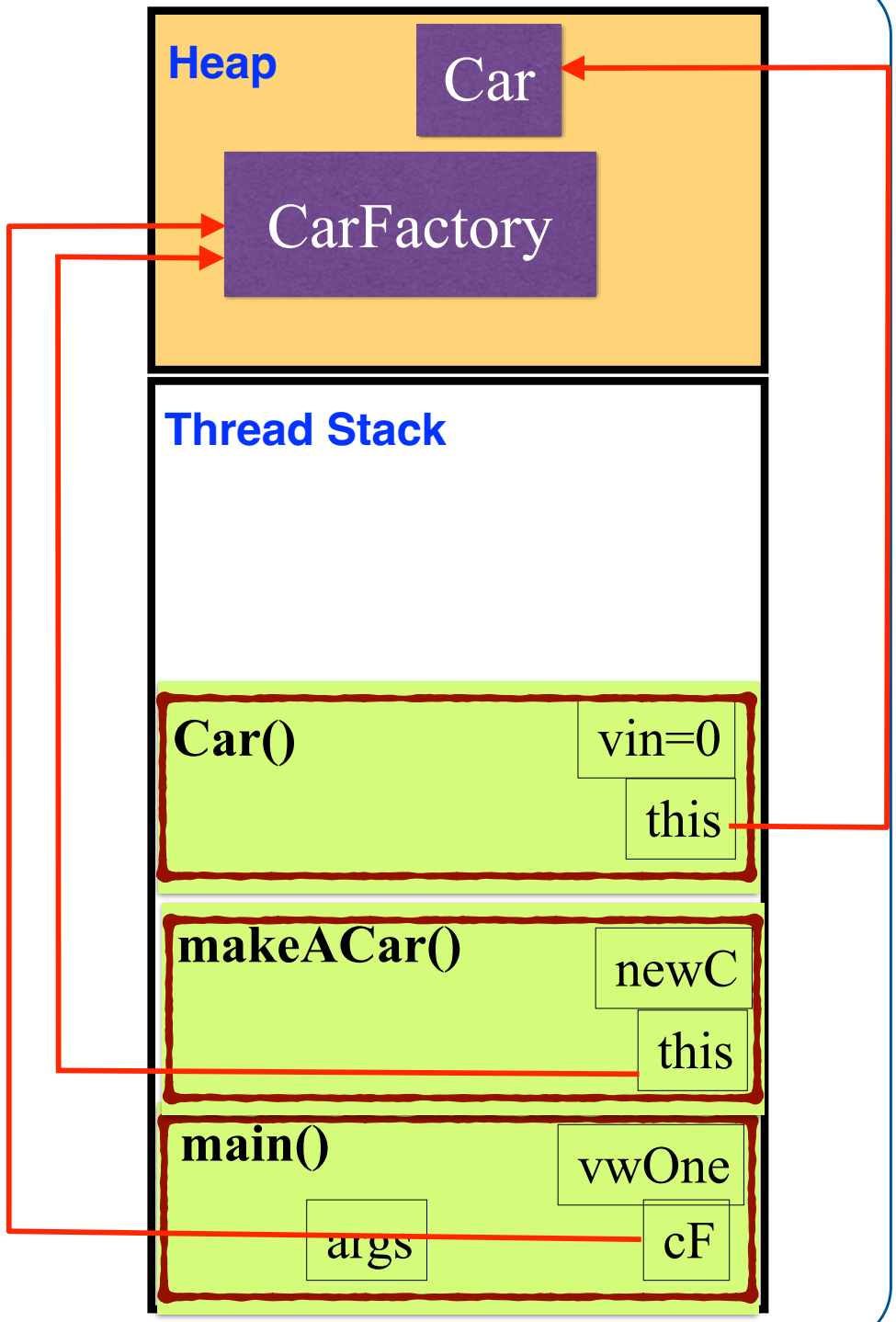
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
   return newC;
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
       testC.moveCar(10);
       testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



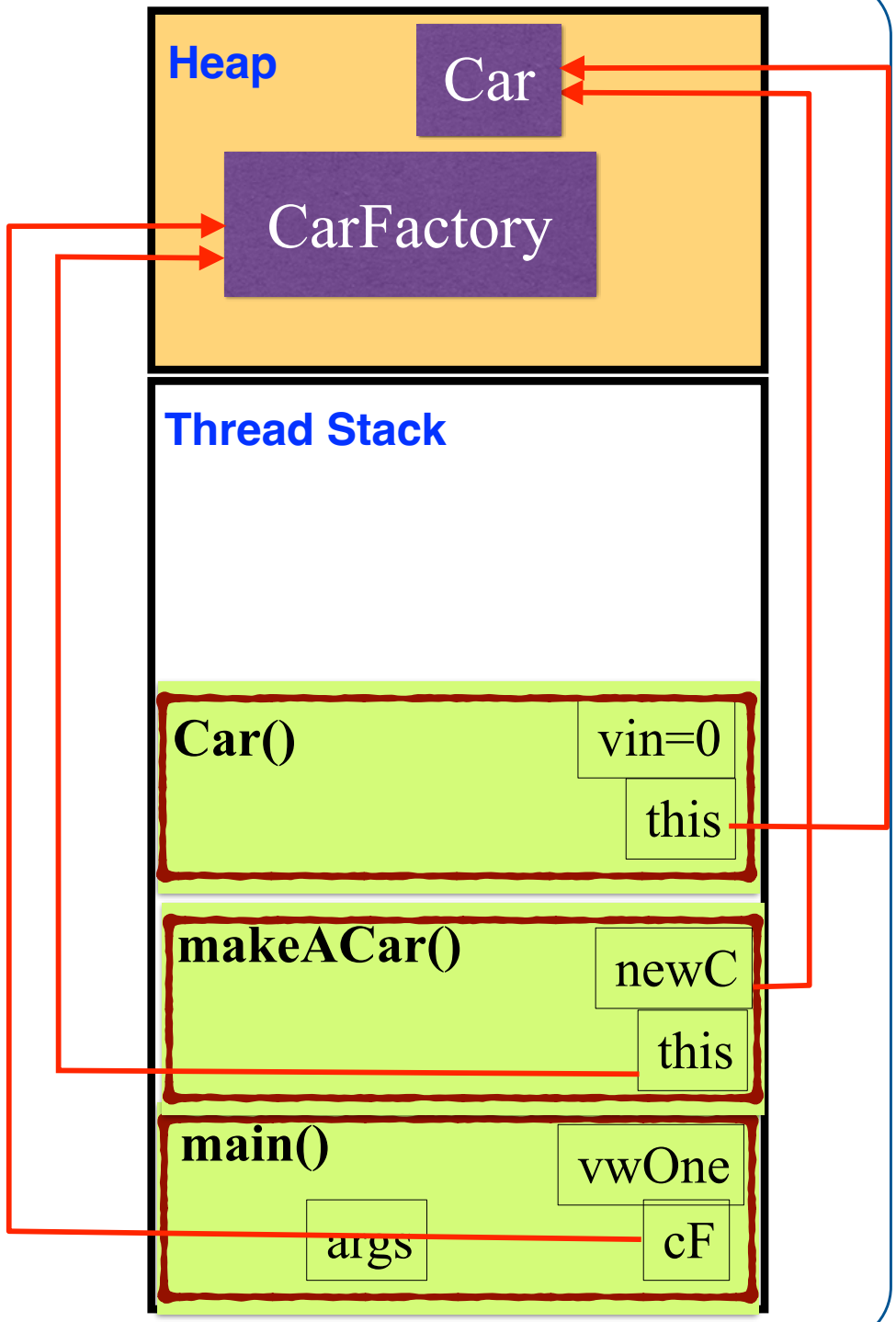
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
   return newC;
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



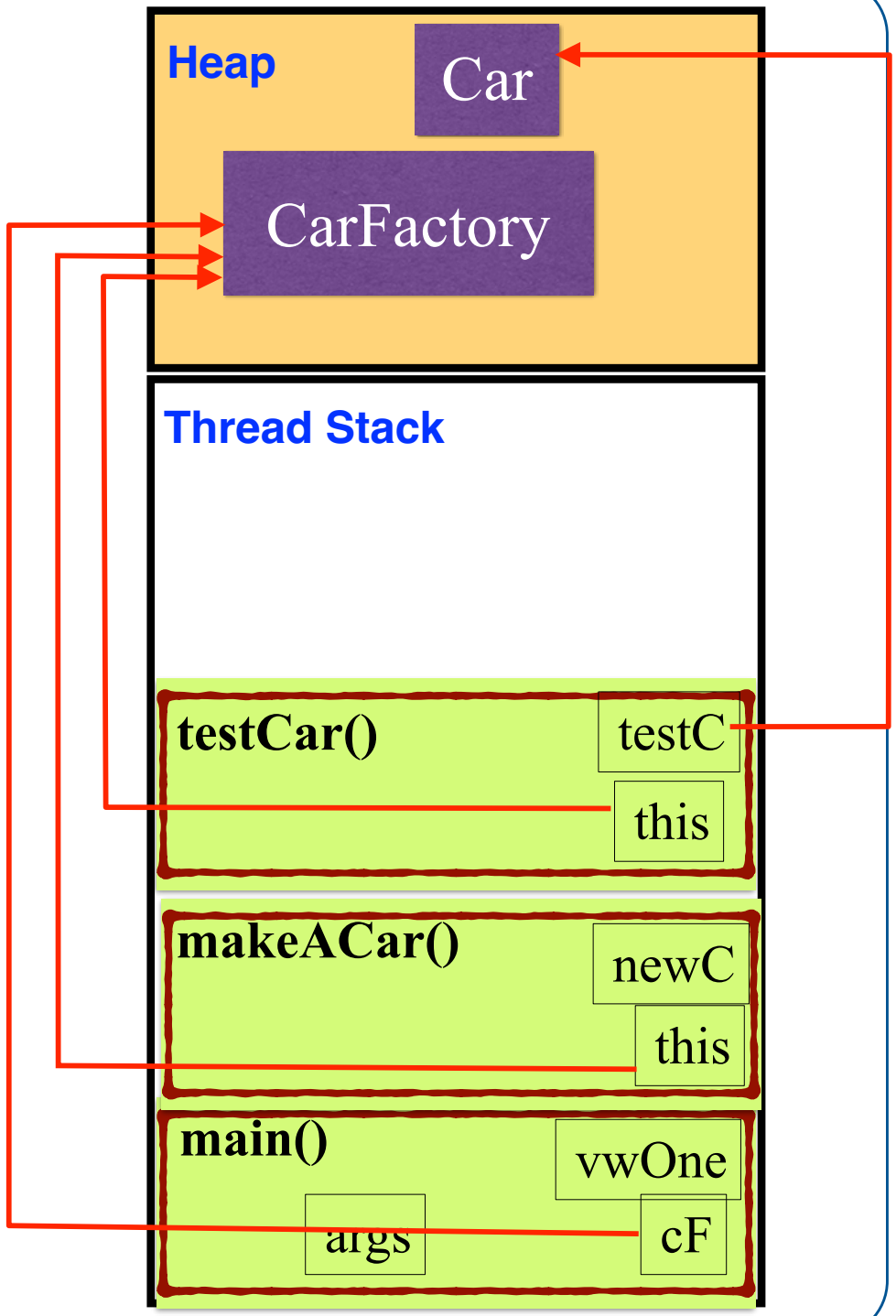
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
   return newC;
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



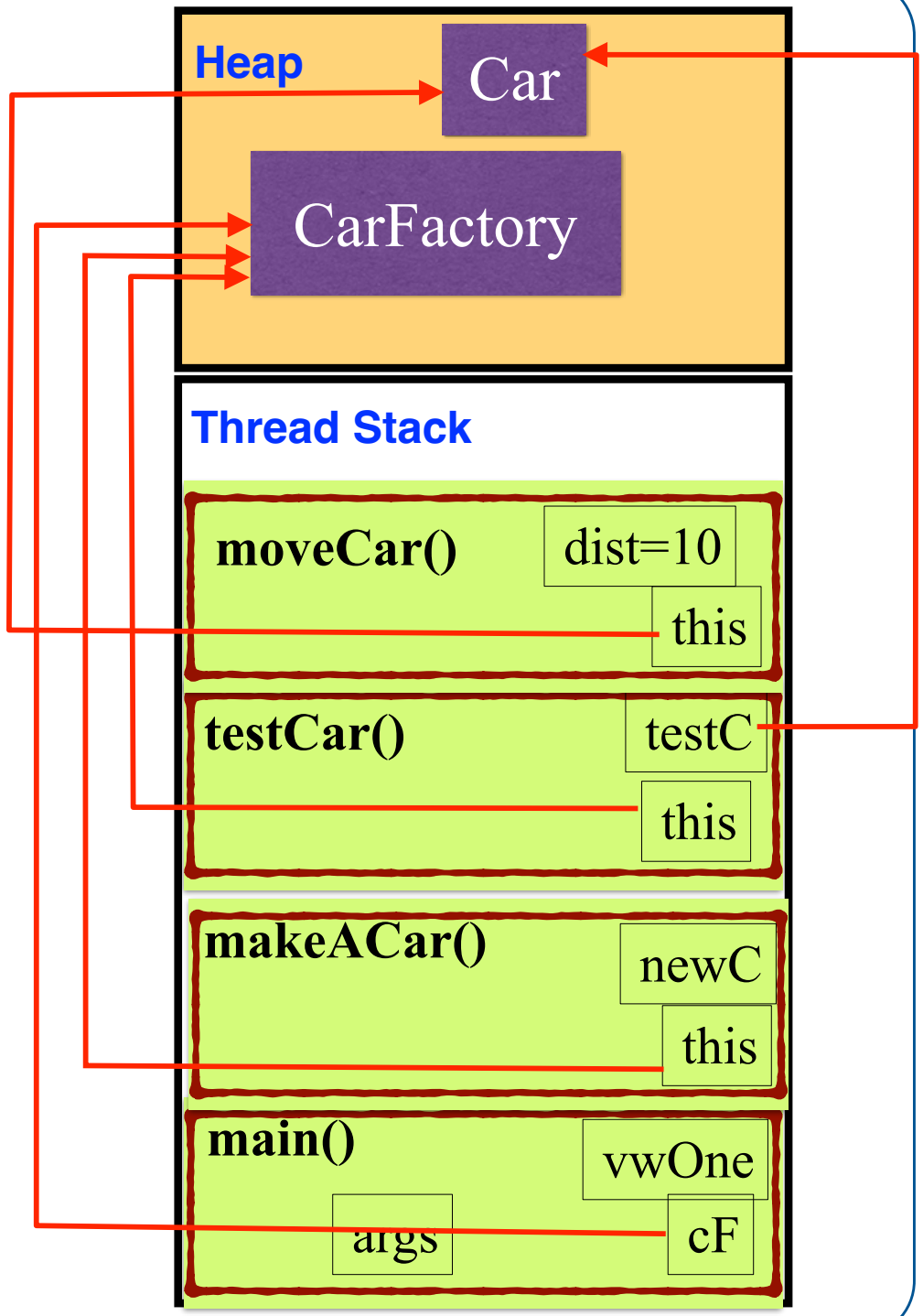
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
   return newC;
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



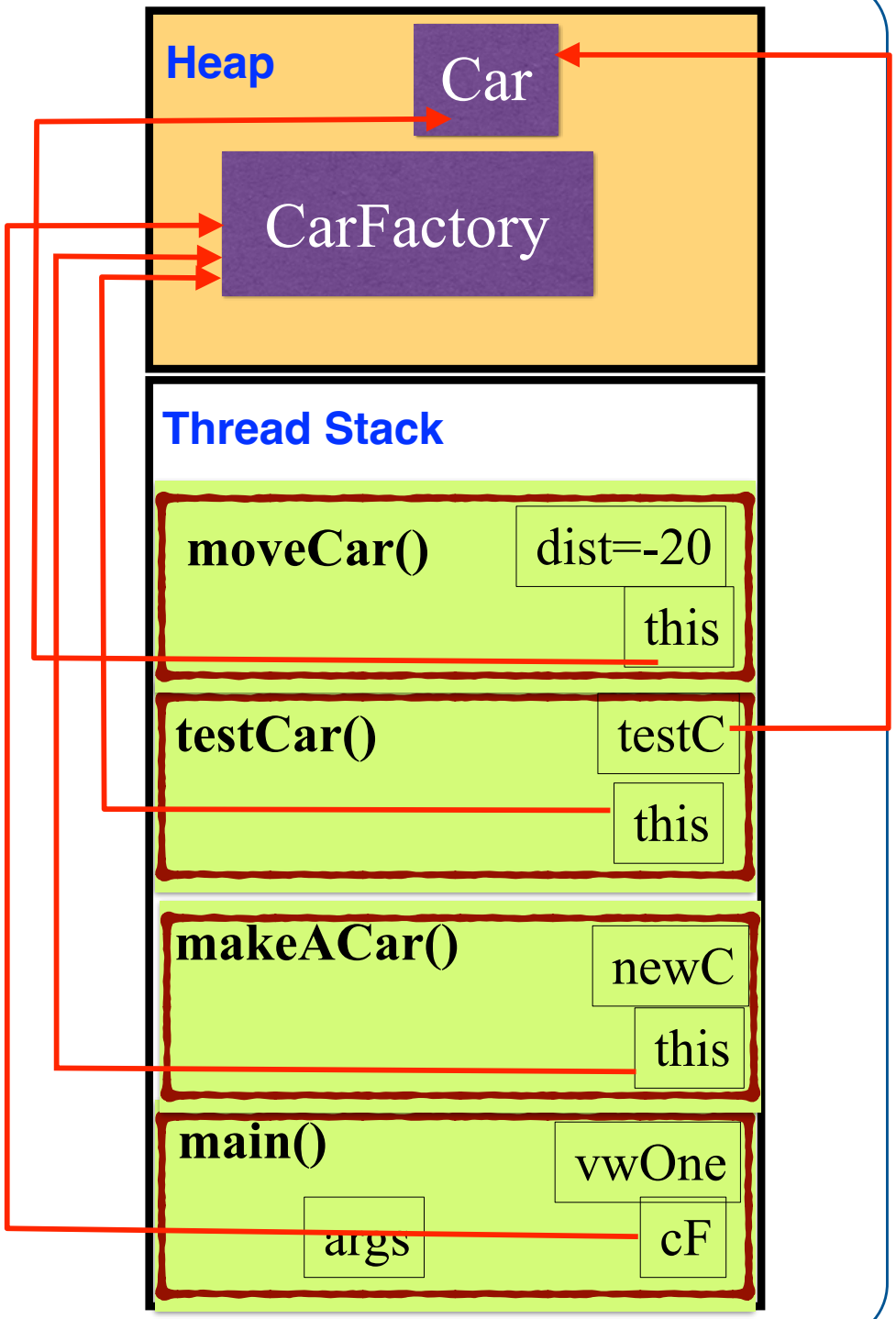
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
   return newC;
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



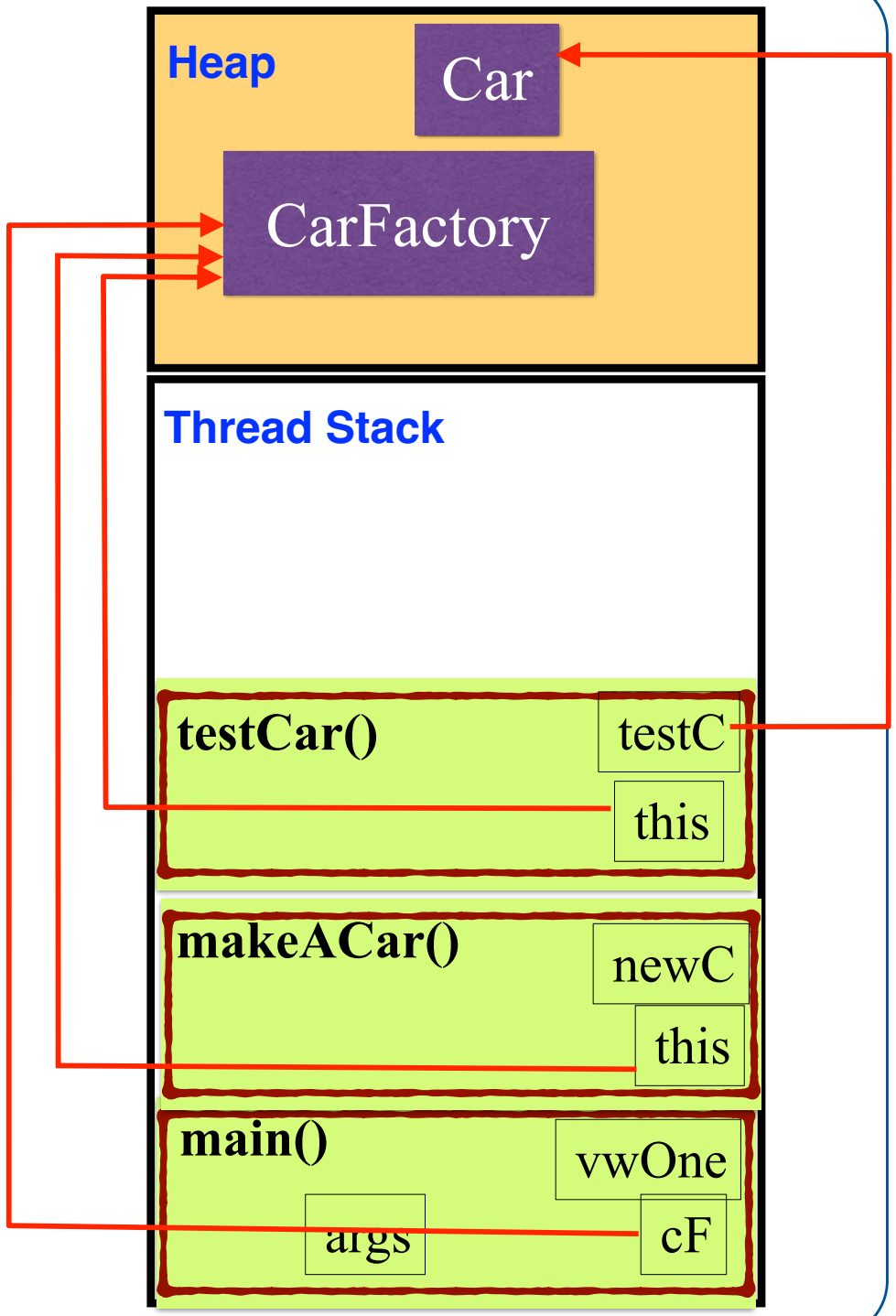
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
   return newC;
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



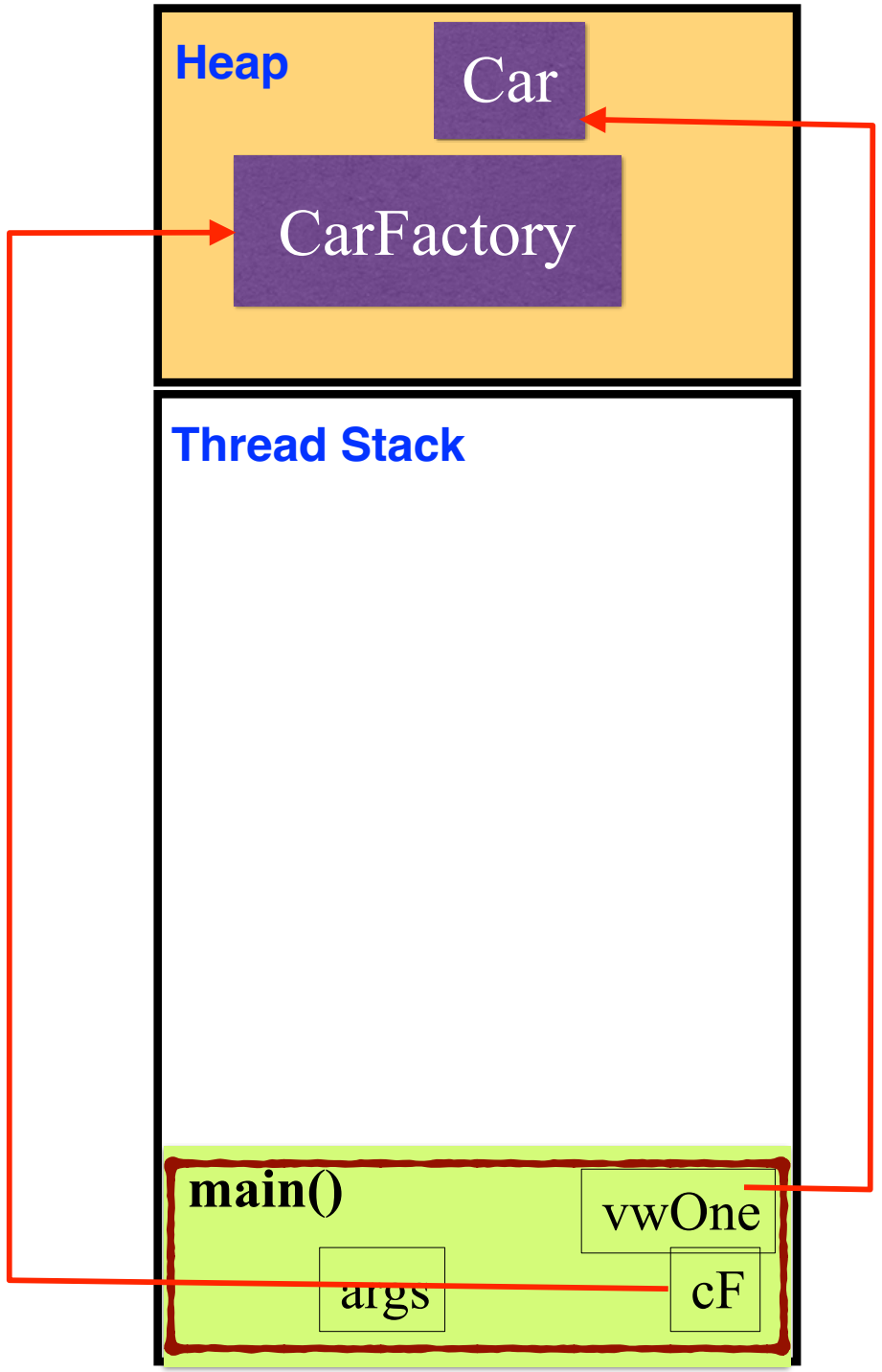
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
   return newC;
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



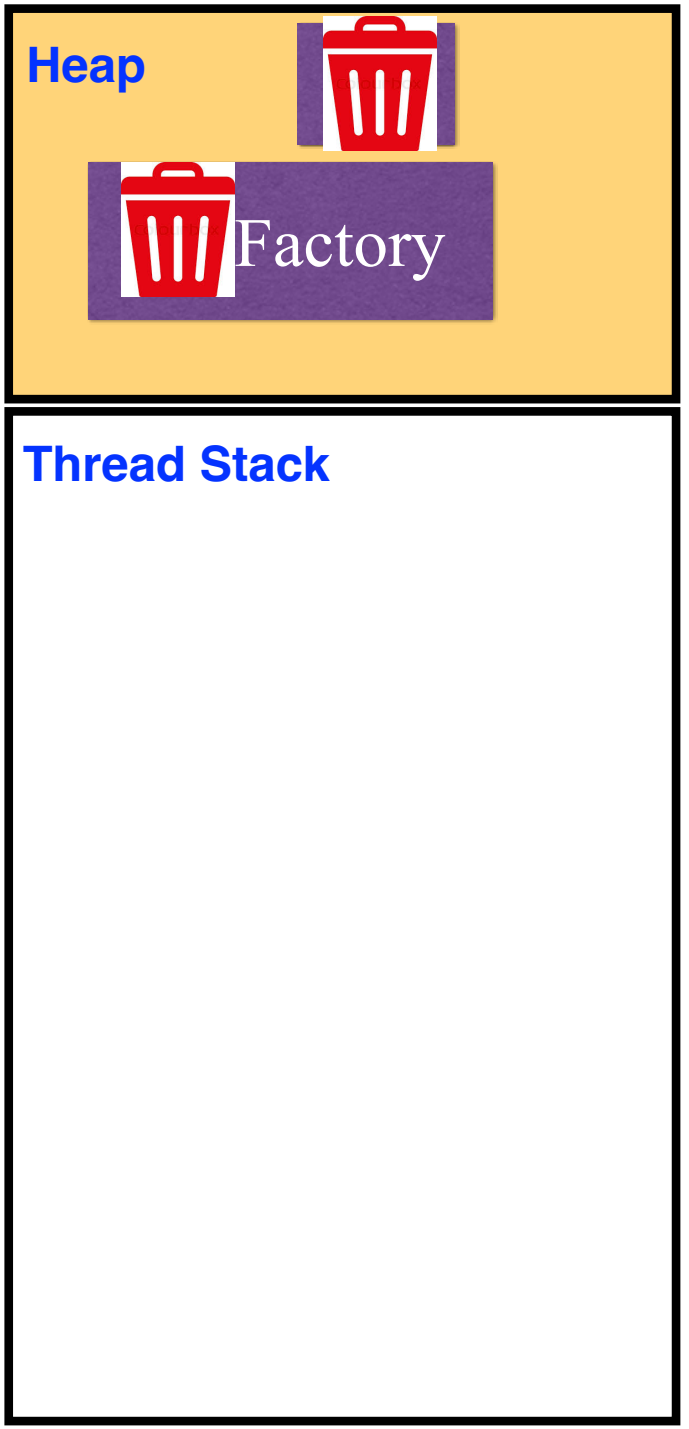
```
class CarFactory {
  static int count = 0;

  /** returns a tested new Car */
  public Car makeACar() {
   Car newC = new Car(count++);
   testCar(newC);
   return newC;
  }

  /* runs tests on a given Car */
  private void testCar(Car testC) {
      testC.moveCar(10);
      testC.moveCar(-20);
  }

  public static void main(String[] args) {

    CarFactory cF = new CarFactory();
    Car vwOne = cF.makeACar();
  }
```



Περίγραμμα

- Οργάνωση και Διαχείριση Μνήμης
- Διεργασίες και Εικονική Μνήμη
- Διαχείριση Μνήμης Java κ. JVM
- **Αρχικοποιήσεις και Κατασκευή Αντικειμένων**
- Σκύβαλα και Αποκομιδή
- Στατικές Μεταβλητές και Μέθοδοι
- Περιβάλλουσες Κλάσεις
- Παράμετροι Κλάσεων
- Έλεγχος ισότητας αντικειμένων
- Αναλλοίωτοι Περιορισμοί
- Παραβίαση ιδιωτικότητας και κατασκευαστές αντιγράφου

University of Cyprus
Department of Computer Science

Ενότητα 2: **Διαχείριση Μνήμης** και Σχεδιασμός Κλάσεων

Αρχικοποιήσεις και Κατασκευή Αντικειμένων

University of Cyprus
Department of Computer Science

Initializers

- Στην Java, πριν την χρήση μιας μεταβλητής αρχέγονου τύπου, η μεταβλητή αυτή θα πρέπει να έχει αρχικοποιηθεί .
- Διαφορετικά ο μεταφραστής διαμαρτύρεται και βγάζει μήνυμα λάθους. Π .χ.:

```
int i;
i = i++ ; // illegal
```

- Για την αποφυγή του προβλήματος αυτού , χρησιμοποιούμε μια ειδική ανάθεση κατά τη δήλωση μιας μεταβλητής, η οποία λέγεται *αρχικοποιητής* (initializer), και με την οποία αρχικοποιούμε μια μεταβλητή αρχέγονου τύπου την στιγμή της δήλωσής της:

```
int i = 1;
```

Αρχικοποίηση μεταβλητών στιγμιοτύπου

- Τα πεδία δεδομένων αντικειμένων (μεταβλητές στιγμιοτύπου - fields) **δεν** απαιτείται να αρχικοποιούνται από τον προγραμματιστή *(τι τιμές παίρνουν τότε* ;)
- Αν δεν κάνουμε εμείς ρητή αρχικοποίηση των μελών αρχέγονου τύπου μιας κλάσης, ο μεταγλωττιστής δίνει τις δικές του προκαθορισμένες τιμές, ως εξής:
  - Για boolean μεταβλητές: **false**
  - Για άλλους αρχέγονους τύπους το **0.**
  - Για χειριστήρια αντικειμένων: **null**
- Είναι καλή πρακτική να πραγματοποιούμε ρητά τις αρχικοποιήσεις μεταβλητών στιγμιοτύπου **μέσα στους κατασκευαστές**.

Παράδειγμα

```
class Measurement {
  boolean t; char c; byte b; short s;
  int i; long l; float f; double d;
  void print() {
   System.out.println("Data type Initial value\n" +
      "boolean " + t + "\n" +
      "char " + c + "\n" + "byte " + b + "\n" +
      "short " + s + "\n" +
      "int " + i + "\n" + "long " + l + "\n" +
      "float " + f + "\n" +"double " + d);
  }}

public class InitialValues {
   public static void main(String[] args){
      new Measurement().print;
} }
```

Εκροή Παραδείγματος

```
C:\users\epl233\eckel-code\c04>java InitialValues
Data type Initial value
boolean false
char
byte 0
short 0
int 0
long 0
float 0.0
double 0.0
```

- Το **char** αρχικοποιείται σε **null.**
- Επίσης σε **null** αρχικοποιούνται και όποια χειριστήρια αντικειμένων περιέχονται ως στοιχεία σε αντικείμενα .

Αρχικοποιήσεις

```
class Measurement {
      boolean t = false;
      char c = 'z';
      byte b = 9;
      short s = 0;
      int i = f(s);
      long l = 12;
      float f = 2.3;
      double d = 0.9;
      void print() { }
} … }
```

- Αρχικοποίηση μπορεί να γίνει με κλήση μεθόδου.
- Τα ορίσματα που δικαιούμαστε να περάσουμε στην μέθοδο αρχικοποίησης, πρέπει να έχουν ήδη αρχικοποιηθεί (αλλοιώς λαμβάνουμε **exception**).

Αρχικοποιήσεις και Κατασκευή Αντικειμένων

Κατασκευή αντικειμένων και Κατασκευαστές (Constructors)

University of Cyprus
Department of Computer Science

Κατασκευή αντικειμένων

- Για τη δημιουργία νέου αντικειμένου χρησιμοποιούμε τον τελεστή **new**:

```
BankAcct b1 = new BankAcct(21338, 0.0);
String s = new String("asdf");
    // String είναι ειδική περίπτωση τύπου
String s = ``asdf'';
```

- Το όνομα του κατασκευαστή (constructor) και η λίστα των (προαιρετικών) ορισμάτων του πρέπει υποχρεωτικά να ακολουθεί τον τελεστή **new**.
  - Αυτός είναι ο μοναδικός έγκυρος τρόπος για κλήση κατασκευαστή.
- Το νέο αντικείμενο μπορούμε να το αναθέσουμε σε ένα χειριστήριο (handle), ο τύπος του οποίου είναι ο ίδιος με την κλάση του αντικειμένου.

Κατασκευαστές (constructors)

Κατασκευαστές (constructors)

- Ο **constructor** είναι μια ειδική κατηγορία μεθόδου που χρησιμοποιείται για την αρχικοποίηση ενός αντικειμένου κατά την στιγμή της δημιουργίας του.

Κατασκευαστές (constructors)

- Ο **constructor** είναι μια ειδική κατηγορία μεθόδου που χρησιμοποιείται για την αρχικοποίηση ενός αντικειμένου κατά την στιγμή της δημιουργίας του.
- Οι κατασκευαστές δηλώνονται με το **ίδιο όνομα** με την κλάση στην οποία ανήκουν. Έτσι η ονομασία τους δεν χρειάζεται ιδιαίτερη διαχείριση, για αποφυγή τυχόν συγκρούσεων με άλλα ονόματα.

Κατασκευαστές (constructors)

- Ο **constructor** είναι μια ειδική κατηγορία μεθόδου που χρησιμοποιείται για την αρχικοποίηση ενός αντικειμένου κατά την στιγμή της δημιουργίας του.
- Οι κατασκευαστές δηλώνονται με το **ίδιο όνομα** με την κλάση στην οποία ανήκουν. Έτσι η ονομασία τους δεν χρειάζεται ιδιαίτερη διαχείριση, για αποφυγή τυχόν συγκρούσεων με άλλα ονόματα.
- Οι κατασκευαστές δεν επιστρέφουν τίποτε, χωρίς ωστόσο να δηλώνονται με τύπο επιστροφής **void.**

Κατασκευαστές (constructors)

- Ο **constructor** είναι μια ειδική κατηγορία μεθόδου που χρησιμοποιείται για την αρχικοποίηση ενός αντικειμένου κατά την στιγμή της δημιουργίας του.
- Οι κατασκευαστές δηλώνονται με το **ίδιο όνομα** με την κλάση στην οποία ανήκουν. Έτσι η ονομασία τους δεν χρειάζεται ιδιαίτερη διαχείριση, για αποφυγή τυχόν συγκρούσεων με άλλα ονόματα.
- Οι κατασκευαστές δεν επιστρέφουν τίποτε, χωρίς ωστόσο να δηλώνονται με τύπο επιστροφής **void.**
- Διευκολύνουν τον προγραμματισμό καθώς «ενοποιούν» ονοματολογικά την δήλωση και την αρχικοποίηση των κλάσεων και αντικειμένων.

Κλήση κατασκευαστών

Κλήση κατασκευαστών

- Ένας κατασκευαστής καλείται για τη δημιουργία αντικειμένου μιας κλάσης με χρήση της εντολής **new**

```
ClassName objectName = new ClassName(anyArgs);
```

Κλήση κατασκευαστών

- Ένας κατασκευαστής καλείται για τη δημιουργία αντικειμένου μιας κλάσης με χρήση της εντολής **new**

```
ClassName objectName = new ClassName(anyArgs);
```

- Εάν κληθεί ξανά ο κατασκευαστής (με χρήση **new**), δημιουργείται ένα νέο αντικείμενο.

Παράδειγμα constructor

```
class Rock {
  Rock() { // This is the constructor
      System.out.println("Creating Rock");
  }
}
public class SimpleConstructor {
  public static void main(String[] args) {
      for(int i = 0; i < 10; i++)
          new Rock();
  }
}
```

Περίγραμμα

- Οργάνωση και Διαχείριση Μνήμης
- Διεργασίες και Εικονική Μνήμη
- Διαχείριση Μνήμης Java κ. JVM
- **Αρχικοποιήσεις και Κατασκευή Αντικειμένων**
- Σκύβαλα και Αποκομιδή
- Στατικές Μεταβλητές και Μέθοδοι
- Περιβάλλουσες Κλάσεις
- Παράμετροι Κλάσεων
- Έλεγχος ισότητας αντικειμένων
- Αναλλοίωτοι Περιορισμοί
- Παραβίαση ιδιωτικότητας και κατασκευαστές αντιγράφου

University of Cyprus
Department of Computer Science

Λειτουργία κατασκευαστών

Λειτουργία κατασκευαστών

- Η πρώτη ενέργεια που γίνεται από έναν κατασκευαστή είναι να δημιουργήσει ένα αντικείμενο με μεταβλητές στιγμιοτύπου (instance variables), όπως καθορίζονται από την κλάση του αντικειμένου

Λειτουργία κατασκευαστών

- Η πρώτη ενέργεια που γίνεται από έναν κατασκευαστή είναι να δημιουργήσει ένα αντικείμενο με μεταβλητές στιγμιοτύπου (instance variables), όπως καθορίζονται από την κλάση του αντικειμένου
- Συνεπώς, είναι *έγκυρη η κλήση μιας άλλης μεθόδου εντός του σώματος του κατασκευαστή*, καθώς το νεο-δημιουργημένο αντικείμενο είναι το αντικείμενο κλήσης για τη μέθοδο

Λειτουργία κατασκευαστών

- Η πρώτη ενέργεια που γίνεται από έναν κατασκευαστή είναι να δημιουργήσει ένα αντικείμενο με μεταβλητές στιγμιοτύπου (instance variables), όπως καθορίζονται από την κλάση του αντικειμένου
- Συνεπώς, είναι *έγκυρη η κλήση μιας άλλης μεθόδου εντός του σώματος του κατασκευαστή*, καθώς το νεο-δημιουργημένο αντικείμενο είναι το αντικείμενο κλήσης για τη μέθοδο
  - Για παράδειγμα, μέθοδοι μεταλλαγής μπορεί να χρησιμοποιηθούν για τον ορισμό των τιμών των μεταβλητών στιγμιοτύπου

Λειτουργία κατασκευαστών

- Η πρώτη ενέργεια που γίνεται από έναν κατασκευαστή είναι να δημιουργήσει ένα αντικείμενο με μεταβλητές στιγμιοτύπου (instance variables), όπως καθορίζονται από την κλάση του αντικειμένου
- Συνεπώς, είναι *έγκυρη η κλήση μιας άλλης μεθόδου εντός του σώματος του κατασκευαστή*, καθώς το νεο-δημιουργημένο αντικείμενο είναι το αντικείμενο κλήσης για τη μέθοδο
  - Για παράδειγμα, μέθοδοι μεταλλαγής μπορεί να χρησιμοποιηθούν για τον ορισμό των τιμών των μεταβλητών στιγμιοτύπου
  - Είναι επίσης δυνατό ένας κατασκευαστής να καλέσει έναν άλλο κατασκευαστή

Κατασκευαστές και η παράμετρος `this`

Κατασκευαστές και η παράμετρος `this`

- Η πρώτη ενέργεια που γίνεται από έναν κατασκευαστή είναι να δημιουργήσει αυτόματα ένα αντικείμενο, στο οποίο δεσμεύεται χώρος για τις μεταβλητές στιγμιότυπου που καθορίζει η κλάση του.

Κατασκευαστές και η παράμετρος `this`

- Η πρώτη ενέργεια που γίνεται από έναν κατασκευαστή είναι να δημιουργήσει αυτόματα ένα αντικείμενο, στο οποίο δεσμεύεται χώρος για τις μεταβλητές στιγμιότυπου που καθορίζει η κλάση του.
- Όπως οι (μη στατικές) μέθοδοι, έτσι και οι κατασκευαστές έχουν πρόσβαση στην παράμετρο **`this`**

Κατασκευαστές και η παράμετρος `this`

- Η πρώτη ενέργεια που γίνεται από έναν κατασκευαστή είναι να δημιουργήσει αυτόματα ένα αντικείμενο, στο οποίο δεσμεύεται χώρος για τις μεταβλητές στιγμιότυπου που καθορίζει η κλάση του.
- Όπως οι (μη στατικές) μέθοδοι, έτσι και οι κατασκευαστές έχουν πρόσβαση στην παράμετρο `this`
- Στο σώμα ενός κατασκευαστή, η παράμετρος `this` υφίσταται και:

Κατασκευαστές και η παράμετρος `this`

- Η πρώτη ενέργεια που γίνεται από έναν κατασκευαστή είναι να δημιουργήσει αυτόματα ένα αντικείμενο, στο οποίο δεσμεύεται χώρος για τις μεταβλητές στιγμιότυπου που καθορίζει η κλάση του.
- Όπως οι (μη στατικές) μέθοδοι, έτσι και οι κατασκευαστές έχουν πρόσβαση στην παράμετρο `this`
- Στο σώμα ενός κατασκευαστή, η παράμετρος `this` υφίσταται και:
  - Αναφέρεται στο αντικείμενο που μόλις δημιουργήθηκε.

Κατασκευαστές και η παράμετρος `this`

- Η πρώτη ενέργεια που γίνεται από έναν κατασκευαστή είναι να δημιουργήσει αυτόματα ένα αντικείμενο, στο οποίο δεσμεύεται χώρος για τις μεταβλητές στιγμιότυπου που καθορίζει η κλάση του.
- Όπως οι (μη στατικές) μέθοδοι, έτσι και οι κατασκευαστές έχουν πρόσβαση στην παράμετρο `this`
- Στο σώμα ενός κατασκευαστή, η παράμετρος `this` υφίσταται και:
  - Αναφέρεται στο αντικείμενο που μόλις δημιουργήθηκε.
  - Μπορεί να χρησιμοποιηθεί ρητά για να αποκτήσουμε πρόβαση σε μεταβλητές στιγμιότυπου, αλλά πιο συχνά υπονοείται.

Προκαθορισμένος κατασκευαστής (No-Argument Constructor)

Προκαθορισμένος κατασκευαστής (No-Argument Constructor)

- Εάν δεν συμπεριλάβετε κανέναν κατασκευαστή στην κλάση σας, η Java θα δημιουργήσει αυτόματα έναν δημόσιο, προεπιλεγμένο (προκαθορισμένο - *default*) κατασκευαστή, χωρίς παραμέτρους (*no-argument*)

Προκαθορισμένος κατασκευαστής (No-Argument Constructor)

- Εάν δεν συμπεριλάβετε κανέναν κατασκευαστή στην κλάση σας, η Java θα δημιουργήσει αυτόματα έναν δημόσιο, προεπιλεγμένο (προκαθορισμένο - *default*) κατασκευαστή, χωρίς παραμέτρους (*no-argument*)
- Ο προκαθορισμένος κατασκευαστής δεν δέχεται ορίσματα, δεν εκτελεί αρχικοποιήσεις, αλλά επιτρέπει τη δημιουργία του αντικειμένου

Προκαθορισμένος κατασκευαστής (No-Argument Constructor)

- Εάν δεν συμπεριλάβετε κανέναν κατασκευαστή στην κλάση σας, η Java θα δημιουργήσει αυτόματα έναν δημόσιο, προεπιλεγμένο (προκαθορισμένο - *default*) κατασκευαστή, χωρίς παραμέτρους (*no-argument*)
- Ο προκαθορισμένος κατασκευαστής δεν δέχεται ορίσματα, δεν εκτελεί αρχικοποιήσεις, αλλά επιτρέπει τη δημιουργία του αντικειμένου
- Εάν συμπεριλάβετε έστω και έναν κατασκευαστή στην κλάση σας, η Java δεν θα παρέχει τον προκαθορισμένο κατασκευαστή, οπότε

Προκαθορισμένος κατασκευαστής (No-Argument Constructor)

- Εάν δεν συμπεριλάβετε κανέναν κατασκευαστή στην κλάση σας, η Java θα δημιουργήσει αυτόματα έναν δημόσιο, προεπιλεγμένο (προκαθορισμένο - *default*) κατασκευαστή, χωρίς παραμέτρους (*no-argument*)
- Ο προκαθορισμένος κατασκευαστής δεν δέχεται ορίσματα, δεν εκτελεί αρχικοποιήσεις, αλλά επιτρέπει τη δημιουργία του αντικειμένου
- Εάν συμπεριλάβετε έστω και έναν κατασκευαστή στην κλάση σας, η Java δεν θα παρέχει τον προκαθορισμένο κατασκευαστή, οπότε
  - φροντίστε να παρέχετε τον δικό σας κατασκευαστή χωρίς παραμέτρους, εφόσον τον χρειάζεστε.

Προκαθορισμένος κατασκευαστής
(default constructor)

```
class Bird {

    int i;

}

public class DefaultConstructorExample {

        public static void main(String[] args) {
             Bird nc = new Bird(); // default!
             Bird nc = new Bird(1); // error!
        }
    }
```

Προκαθορισμένος κατασκευαστής
(default constructor)

```
class Bird {

    int i;

}

public class DefaultConstructorExample {

        public static void main(String[] args) {
             Bird nc = new Bird(); // default!
             Bird nc = new Bird(1); // error!
        }
      }
```

```
Bird() {
   i = 0;
}
```

Added by the compiler

Κατασκευαστές με παραμέτρους

- Ένας constructor μπορεί να δεχεται παραμέτρους, οι οποίες καθορίζουν περαιτέρω το πώς θα αρχικοποιηθεί το αντίστοιχο αντικείμενο.

```
class Rock2 {
      Rock2(int i) {
        System.out.println("Creating Rock
         number " + i);
       }
}
public class SimpleConstructor2 {
 public static void main(String[] args) {
       for(int i = 0; i < 10; i++)
             new Rock2(i);
  }
}
```

Αρχικοποιήσεις και Κατασκευή Αντικειμένων

Overloading (Υπερφόρτωση)

University of Cyprus
Department of Computer Science

Υπερφόρτωση Μεθόδων (overloading)

Υπερφόρτωση Μεθόδων (overloading)

- **Υπερφόρτωση** μεθόδων προκύπτει στην Java όταν δύο ή περισσότερες μέθοδοι ή κατασκευαστές *της ίδιας κλάσης*, έχουν *το ίδιο όνομα*.

Υπερφόρτωση Μεθόδων (overloading)

- **Υπερφόρτωση** μεθόδων προκύπτει στην Java όταν δύο ή περισσότερες μέθοδοι ή κατασκευαστές *της ίδιας κλάσης*, *έχουν το ίδιο όνομα.*
- Για να είναι έγκυρες, δηλώσεις μεθόδων με το ίδιο όνομα πρέπει να έχουν διαφορετικές ***υπογραφές*** (*signatures*).

Υπερφόρτωση Μεθόδων (overloading)

- **Υπερφόρτωση** μεθόδων προκύπτει στην Java όταν δύο ή περισσότερες μέθοδοι ή κατασκευαστές *της ίδιας κλάσης*, *έχουν το ίδιο όνομα*.
- Για να είναι έγκυρες, δηλώσεις μεθόδων με το ίδιο όνομα πρέπει να έχουν διαφορετικές ***υπογραφές*** (*signatures*).
- Η **υπογραφή** μιας μεθόδου αποτελείται από:

Υπερφόρτωση Μεθόδων (overloading)

- **Υπερφόρτωση** μεθόδων προκύπτει στην Java όταν δύο ή περισσότερες μέθοδοι ή κατασκευαστές *της ίδιας κλάσης*, *έχουν το ίδιο όνομα*.
- Για να είναι έγκυρες, δηλώσεις μεθόδων με το ίδιο όνομα πρέπει να έχουν διαφορετικές ***υπογραφές*** (*signatures*).
- Η **υπογραφή** μιας μεθόδου αποτελείται από:
  - το όνομα της μεθόδου μαζί με

Υπερφόρτωση Μεθόδων (overloading)

- **Υπερφόρτωση** μεθόδων προκύπτει στην Java όταν δύο ή περισσότερες μέθοδοι ή κατασκευαστές *της ίδιας κλάσης*, *έχουν το ίδιο όνομα*.
- Για να είναι έγκυρες, δηλώσεις μεθόδων με το ίδιο όνομα πρέπει να έχουν διαφορετικές ***υπογραφές*** (*signatures*).
- Η **υπογραφή** μιας μεθόδου αποτελείται από:
  - το όνομα της μεθόδου μαζί με
  - την λίστα των τυπικών παραμέτρων της (formal parameters).

Υπερφόρτωση Μεθόδων (overloading)

- **Υπερφόρτωση** μεθόδων προκύπτει στην Java όταν δύο ή περισσότερες μέθοδοι ή κατασκευαστές *της ίδιας κλάσης*, *έχουν το ίδιο όνομα*.
- Για να είναι έγκυρες, δηλώσεις μεθόδων με το ίδιο όνομα πρέπει να έχουν διαφορετικές ***υπογραφές*** (*signatures*).
- Η **υπογραφή** μιας μεθόδου αποτελείται από:
  - το όνομα της μεθόδου μαζί με
  - την λίστα των τυπικών παραμέτρων της (formal parameters).
- Διαφοροποιημένη υπογραφή δύο υπερφορτωμένων μεθόδων σημαίνει διαφορετικός αριθμός και/ή τύπος παραμέτρων.

Υπερφόρτωση

- Η υπερφόρτωση μεθόδων χρησιμοποιείται ευρύτατα στον Α/ΣΠ, **ιδιαίτερα** στους κατασκευαστές-constructors.

Υπερφόρτωση

- Η υπερφόρτωση μεθόδων χρησιμοποιείται ευρύτατα στον Α/ΣΠ, **ιδιαίτερα** στους κατασκευαστές-constructors.
- Αν δύο μέθοδοι (κατασκευαστές) έχουν το ίδιο όνομα, τότε πως γνωρίζει η Java ποιά μέθοδο θέλουμε να καλέσουμε;

Υπερφόρτωση

- Η υπερφόρτωση μεθόδων χρησιμοποιείται ευρύτατα στον Α/ΣΠ, **ιδιαίτερα** στους κατασκευαστές-constructors.
- Αν δύο μέθοδοι (κατασκευαστές) έχουν το ίδιο όνομα, τότε πως γνωρίζει η Java ποιά μέθοδο θέλουμε να καλέσουμε;
  - Κάθε υπερφορτωμένη μέθοδος πρέπει να δέχεται μια διαφορετική λίστα τύπων παραμέτρων.

Υπερφόρτωση

- Η υπερφόρτωση μεθόδων χρησιμοποιείται ευρύτατα στον Α/ΣΠ, **ιδιαίτερα** στους κατασκευαστές-constructors.
- Αν δύο μέθοδοι (κατασκευαστές) έχουν το ίδιο όνομα, τότε πως γνωρίζει η Java ποιά μέθοδο θέλουμε να καλέσουμε;
  - Κάθε υπερφορτωμένη μέθοδος πρέπει να δέχεται μια διαφορετική λίστα τύπων παραμέτρων.
  - Ακόμη και *διαφορές στην σειρά των παραμέτρων είναι αρκετές για να ξεχωρίσουν δυό μεθόδους*, αλλά αυτό δεν είναι «καλή» προγραμματιστική τακτική .

Παράδειγμα Υπερφόρτωσης

```
import java.util.*;
class Tree {
   int height;
   Tree() {
       prt("Planting a seedling"); height = 0;
   }
   Tree(int i) {
       prt("Creating new Tree that is "+ i +
           " feet tall"); height = i;
      }
   void info() {
       prt("Tree is "+ height + " feet tall");
   }
   void info(String s) {
       prt(s + ": Tree is " + height + " feet tall");
   }
   static void prt(String s) {
       System.out.println(s); }}
```

Παράδειγμα Υπερφόρτωσης (συνέχεια)

```
public class Overloading {

   public static void main(String[] args) {

        for(int i = 0; i < 5; i++) {

                Tree t = new Tree(i);

                t.info();

                t.info("overloaded method");

        }

   // Overloaded constructor:

   new Tree();

   }

}
```

Υπερφόρτωση και Αρχέγονοι τύποι

Υπερφόρτωση και Αρχέγονοι τύποι

- Αν ορίσουμε μια μέθοδο : **void f (double x) { /* */}** και την καλέσουμε σαν **f(5**), η παράμετρος «5» θα προαχθεί αυτόματα (promotion) σε 5.0.

Υπερφόρτωση και Αρχέγονοι τύποι

- Αν ορίσουμε μια μέθοδο : **void f (double x) { /* */}** και την καλέσουμε σαν **f(5**), η παράμετρος «5» θα προαχθεί αυτόματα (promotion) σε 5.0.
- Τι θα συμβεί αν η f() είναι υπερφορτωμένη ώστε να δέχεται και ακέραιη παράμετρο;

Υπερφόρτωση και Αρχέγονοι τύποι

- Αν ορίσουμε μια μέθοδο : **void f (double x) { /* */}** και την καλέσουμε σαν **f(5**), η παράμετρος «5» θα προαχθεί αυτόματα (promotion) σε 5.0.
- Τι θα συμβεί αν η f() είναι υπερφορτωμένη ώστε να δέχεται και ακέραιη παράμετρο;
  - *Θα εκτελεσθεί η έκδοση της f() που δέχεται ακέραιες παραμέτρους.*

Υπερφόρτωση και Αρχέγονοι τύποι

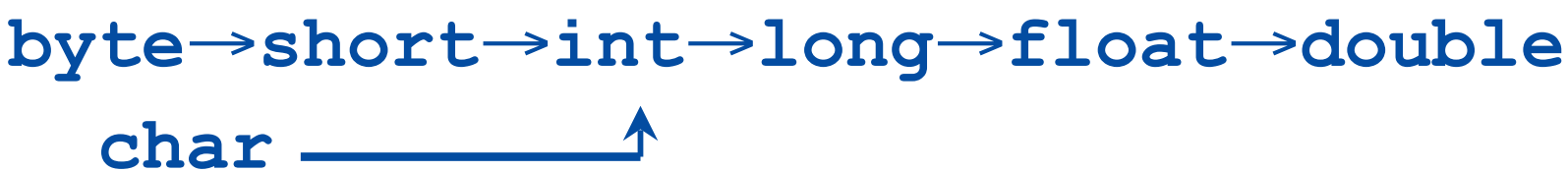
- Αν ορίσουμε μια μέθοδο : **void f (double x) { /* */}** και την καλέσουμε σαν **f(5**), η παράμετρος «5» θα προαχθεί αυτόματα (promotion) σε 5.0.
- Τι θα συμβεί αν η f() είναι υπερφορτωμένη ώστε να δέχεται και ακέραιη παράμετρο;
  - *Θα εκτελεσθεί η έκδοση της f() που δέχεται ακέραιες παραμέτρους.*
- Γενικότερα, αν η τιμή που περνάμε σε μια μέθοδο έχει τύπο *μικρότερο* από τον τύπο της δηλωμένης παραμέτρου, τότε η τιμή *προάγεται* στον τύπο της παραμέτρου.

Υπερφόρτωση και Αρχέγονοι τύποι

- Αν ορίσουμε μια μέθοδο : **void f (double x) { /* */}** και την καλέσουμε σαν **f(5**), η παράμετρος «5» θα προαχθεί αυτόματα (promotion) σε 5.0.
- Τι θα συμβεί αν η f() είναι υπερφορτωμένη ώστε να δέχεται και ακέραιη παράμετρο;
  - *Θα εκτελεσθεί η έκδοση της f() που δέχεται ακέραιες παραμέτρους.*
- Γενικότερα, αν η τιμή που περνάμε σε μια μέθοδο έχει τύπο *μικρότερο* από τον τύπο της δηλωμένης παραμέτρου, τότε η τιμή *προάγεται* στον τύπο της παραμέτρου.
- Αν υπάρχουν πολλοί τέτοιοι δυνατοί τύποι (πολλαπλών «υπερφορτώσεων»), τότε η προαγωγή γίνεται στον αμέσως μεγαλύτερο τύπο.

Υπερφόρτωση και Αρχέγονοι τύποι

- Εξαίρεση έχουμε στην περίπτωση που ο τύπος της δηλωμένης παραμέτρου είναι char κι εμείς περνάμε έναν ακέραιο, ο οποίος δεν αντιστοιχεί σε τιμή χαρακτήρα char, τότε η τιμή αυτή προάγεται σε ακέραιο (και όχι , π.χ. σε short).



Παράδειγμα

```
public class Foo {
  void f1(double x) {
     System.out.println("double f1 -->" + x); }
  void f1(int x) {
     System.out.println("int f1 -->" + x); }
  public static void main(String[] args) {
     Foo ff = new Foo();
     ff.f1(5);
  }}
```

Προσοχή: η υπερφόρτωση ονόματος μεθόδου δεν μπορεί να βασιστεί στον τύπο επιστροφής της μεθόδου

- Η υπογραφή (signature) μιας μεθόδου περιλαμβάνει μόνο το όνομα και τους τύπους των τυπικών παραμέτρων της μεθόδου:
  - **Δεν** περιλαμβάνει τον επιστρεφόμενο τύπο (type returned)
  - Η Java δεν επιτρέπει μεθόδους με την ίδια υπογραφή και διαφορετικούς τύπους επιστροφής μέσα στην ίδια κλάση
- Ambiguous method invocations will produce an error in Java

Προσοχή: η υπερφόρτωση ονόματος μεθόδου δεν μπορεί να βασιστεί στον τύπο επιστροφής της μεθόδου

- Η υπογραφή (signature) μιας μεθόδου περιλαμβάνει μόνο το όνομα και τους τύπους των τυπικών παραμέτρων της μεθόδου:
  - **Δεν** περιλαμβάνει τον επιστρεφόμενο τύπο (type returned)
  - Η Java δεν επιτρέπει μεθόδους με την ίδια υπογραφή και διαφορετικούς τύπους επιστροφής μέσα στην ίδια κλάση
- Ambiguous method invocations will produce an error in Java

Κλήση κατασκευαστών από κατασκευαστές

Κλήση κατασκευαστών από κατασκευαστές

- Μέσα σε έναν **constructor**, η δεσμευμένη λέξη **`this`** μπορεί να χρησιμοποιηθεί με σύνταξη κλήσης μεθόδου:

```
this();
```

Κλήση κατασκευαστών από κατασκευαστές

- Μέσα σε έναν **constructor**, η δεσμευμένη λέξη **`this`** μπορεί να χρησιμοποιηθεί με σύνταξη κλήσης μεθόδου: **`this();`**
- Σε αυτή την περίπτωση, η **`this();`** προκαλεί ρητή κλήση μιας διαφορετικής εκδοχής του constructor, η οποία εκδοχή έχει δηλωθεί με υπερφόρτωση.

Κλήση κατασκευαστών από κατασκευαστές

- Μέσα σε έναν **constructor,** η δεσμευμένη λέξη **`this`** μπορεί να χρησιμοποιηθεί με σύνταξη κλήσης μεθόδου: **`this();`**
- Σε αυτή την περίπτωση, η **`this();`** προκαλεί ρητή κλήση μιας διαφορετικής εκδοχής του constructor, η οποία εκδοχή έχει δηλωθεί με υπερφόρτωση.
- Η **`this();`** μπορεί να χρησιμοποιηθεί σαν κλήση **constructor:**

Κλήση κατασκευαστών από κατασκευαστές

- Μέσα σε έναν **constructor,** η δεσμευμένη λέξη **`this`** μπορεί να χρησιμοποιηθεί με σύνταξη κλήσης μεθόδου: **`this();`**
- Σε αυτή την περίπτωση, η **`this();`** προκαλεί ρητή κλήση μιας διαφορετικής εκδοχής του constructor, η οποία εκδοχή έχει δηλωθεί με υπερφόρτωση.
- Η **`this();`** μπορεί να χρησιμοποιηθεί σαν κλήση **constructor:**
  - μέσα σε κάποιον constructor,

Κλήση κατασκευαστών από κατασκευαστές

- Μέσα σε έναν **constructor,** η δεσμευμένη λέξη **`this`** μπορεί να χρησιμοποιηθεί με σύνταξη κλήσης μεθόδου: **`this();`**
- Σε αυτή την περίπτωση, η **`this();`** προκαλεί ρητή κλήση μιας διαφορετικής εκδοχής του constructor, η οποία εκδοχή έχει δηλωθεί με υπερφόρτωση.
- Η **`this();`** μπορεί να χρησιμοποιηθεί σαν κλήση **constructor:**
  - μέσα σε κάποιον constructor,
  - μόνο σαν ***πρώτη εντολή*** **του** ***constructor*** και

Κλήση κατασκευαστών από κατασκευαστές

- Μέσα σε έναν **constructor,** η δεσμευμένη λέξη **`this`** μπορεί να χρησιμοποιηθεί με σύνταξη κλήσης μεθόδου: **`this();`**
- Σε αυτή την περίπτωση, η **`this();`** προκαλεί ρητή κλήση μιας διαφορετικής εκδοχής του constructor, η οποία εκδοχή έχει δηλωθεί με υπερφόρτωση.
- Η **`this();`** μπορεί να χρησιμοποιηθεί σαν κλήση **constructor:**
  - μέσα σε κάποιον constructor,
  - μόνο σαν ***πρώτη εντολή*** **του** ***constructor*** και
  - μόνο για **μια φορά**.

Παράδειγμα

```
// Calling constructors with "this"
public class Flower {
  private int petalCount = 0;
  private String s = new String("null");
  Flower(int petals) {
          petalCount = petals;
  }
  Flower(String ss) {
          s = ss;
  }
```

Παράδειγμα (συνέχεια)

```
Flower(String s, int petals) {
  this(petals);
  this(s);  // Δεν επιτρέπεται δεύτερη κλήση του this
  this.s = s;
}
Flower() {
  this("hi", 47);
}
```

Παράδειγμα (συνέχεια)

```
void print() {

  this(11);  // Δεν επιτρέπεται η κλήση του this εκτός constructor

  System.out.println("petalCount = " + petalCount +
  " s = "+ s);

}

public static void main(String[] args) {

  Flower x = new Flower();

  x.print();

 } }
```

Περίγραμμα

- Οργάνωση και Διαχείριση Μνήμης
- Διεργασίες και Εικονική Μνήμη
- Διαχείριση Μνήμης Java κ. JVM
- Αρχικοποιήσεις και Κατασκευή Αντικειμένων
- **Σκύβαλα και Αποκομιδή**
- Στατικές Μεταβλητές και Μέθοδοι
- Περιβάλλουσες Κλάσεις
- Παράμετροι Κλάσεων
- Έλεγχος ισότητας αντικειμένων
- Αναλλοίωτοι Περιορισμοί
- Παραβίαση ιδιωτικότητας και κατασκευαστές αντιγράφου

University of Cyprus
Department of Computer Science

Ενότητα 2: **Διαχείριση Μνήμης** και Σχεδιασμός Κλάσεων

Συλλογή/Αποκομιδή Σκυβάλων

Garbage Collection

University of Cyprus
Department of Computer Science

Συλλογή/αποκομιδή Σκυβάλων

- Πως δημιουργούνται τα «σκουπίδια» στην Java και που είναι αποθηκευμένα;

Συλλογή/αποκομιδή Σκυβάλων

- Πως δημιουργούνται τα «σκουπίδια» στην Java και που είναι αποθηκευμένα;
- Ποιος είναι ο ρόλος του συλλέκτη σκυβάλων (**αποκομιστή**/σκουπιδιάρη);

Συλλογή/αποκομιδή Σκυβάλων

- Πως δημιουργούνται τα «σκουπίδια» στην Java και που είναι αποθηκευμένα;
- Ποιος είναι ο ρόλος του συλλέκτη σκυβάλων (**αποκομιστή**/σκουπιδιάρη);
- Να αποδεσμεύει μνήμη, η οποία έχει δεσμευθεί με τη **`new`** και να την επιστρέφει στο διαθέσιμο χώρο του σωρού (heap) για επαναχρησιμοποίηση.

Συλλογή δεν σημαίνει διαγραφή!

- Σημείωση: Η απελευθέρωση μνήμης **δεν** ισοδυναμεί με «εκκαθάριση» (διαγραφή του περιεχομένου) των αντικειμένων-σκυβάλων.

Συλλογή δεν σημαίνει διαγραφή!

- Σημείωση: Η απελευθέρωση μνήμης **δεν** ισοδυναμεί με «εκκαθάριση» (διαγραφή του περιεχομένου) των αντικειμένων-σκυβάλων.
- Η «εκκαθάριση» αντικειμένων δεν είναι απλή υπόθεση, την οποία να μπορεί να φέρει σε πέρας ο συλλέκτης:

Συλλογή δεν σημαίνει διαγραφή!

- Σημείωση: Η απελευθέρωση μνήμης **δεν** ισοδυναμεί με «εκκαθάριση» (διαγραφή του περιεχομένου) των αντικειμένων-σκυβάλων.
- Η «εκκαθάριση» αντικειμένων δεν είναι απλή υπόθεση, την οποία να μπορεί να φέρει σε πέρας ο συλλέκτης:
  - Π.χ., με τη δημιουργία κάποιων αντικειμένων, καλούνται βιβλιοθήκες οι οποίες δημιουργούν άλλα αντικείμενα ή γραφικά στην οθόνη του Η/Υ ή δεσμεύουν μνήμη χωρίς κλήση της new με επίκληση ιθαγενών-native μεθόδων ή δημιουργούν συνδέσεις μέσω δικτύου ή με βάσεις δεδομένων κοκ.

Συλλογή δεν σημαίνει διαγραφή!

- Σημείωση: Η απελευθέρωση μνήμης **δεν** ισοδυναμεί με «εκκαθάριση» (διαγραφή του περιεχομένου) των αντικειμένων-σκυβάλων.
- Η «εκκαθάριση» αντικειμένων δεν είναι απλή υπόθεση, την οποία να μπορεί να φέρει σε πέρας ο συλλέκτης:
  - Π.χ., με τη δημιουργία κάποιων αντικειμένων, καλούνται βιβλιοθήκες οι οποίες δημιουργούν άλλα αντικείμενα ή γραφικά στην οθόνη του Η/Υ ή δεσμεύουν μνήμη χωρίς κλήση της new με επίκληση ιθαγενών-native μεθόδων ή δημιουργούν συνδέσεις μέσω δικτύου ή με βάσεις δεδομένων κοκ.
  - Ο συλλέκτης σκυβάλων **δεν** διαγράφει από τη μνήμη τις σχετικές δομές και τα δεδομένα τους.

Γενικές Αρχές για την Συλλογή Σκυβάλων

- Τα αντικείμενα σας μπορεί να μην συλλεχθούν ποτέ από τον συλλέκτη, ακόμη κι αν καταστούν σκουπίδια.
- Η αποκομιδή σκυβάλων *δεν* ισοδυναμεί με *καταστροφή* - εκκαθάριση των περιεχομένων των αντικειμένων-σκουπιδιών.
- Η αποκομιδή σκυβάλων αφορά μόνο στην αποδέσμευση της μνήμης που αυτά καταλαμβάνουν και επιστροφή της στο σωρό.

Αποτελείωμα και Συλλογή Σκυβάλων

- Ο συλλέκτης σκυβάλων (αποκομιστής) - garbage collector - δεν λύνει το πρόβλημα της **εκκαθάρισης αντικειμένων** στη Java, διότι:

Αποτελείωμα και Συλλογή Σκυβάλων

- Ο συλλέκτης σκυβάλων (αποκομιστής) - garbage collector - δεν λύνει το πρόβλημα της **εκκαθάρισης αντικειμένων** στη Java, διότι:
  - «Απορρίματα» της Java μπορεί να μη συλλεχθούν από τον αποκομιστή σκυβάλων. Ο λόγος είναι ότι συχνά τα προγράμματα δεν ξεμένουν από μνήμη, οπότε δεν καλείται ο GC στο χρόνο ζωής τους.

Αποτελείωμα και Συλλογή Σκυβάλων

- Ο συλλέκτης σκυβάλων (αποκομιστής) - garbage collector - δεν λύνει το πρόβλημα της **εκκαθάρισης αντικειμένων** στη Java, διότι:
  - «Απορρίματα» της Java μπορεί να μη συλλεχθούν από τον αποκομιστή σκυβάλων. Ο λόγος είναι ότι συχνά τα προγράμματα δεν ξεμένουν από μνήμη, οπότε δεν καλείται ο GC στο χρόνο ζωής τους.
  - Ο αποκομιστής σκυβάλων γνωρίζει πως να αποδεσμεύει μνήμη που έχει κρατηθεί με την new, όχι όμως και τι θα κάνει με ιδιάζουσες περιπτώσεις κράτησης μνήμης από κάποιο αντικείμενο (π.χ. για γραφική απεικόνιση στην οθόνη, σύνδεση με δίκτυο ή ΒΔ).

Αποτελείωμα και Συλλογή Σκυβάλων

- Ο συλλέκτης σκυβάλων (αποκομιστής) - garbage collector - δεν λύνει το πρόβλημα της **εκκαθάρισης αντικειμένων** στη Java, διότι:
  - «Απορρίματα» της Java μπορεί να μη συλλεχθούν από τον αποκομιστή σκυβάλων. Ο λόγος είναι ότι συχνά τα προγράμματα δεν ξεμένουν από μνήμη, οπότε δεν καλείται ο GC στο χρόνο ζωής τους.
  - Ο αποκομιστής σκυβάλων γνωρίζει πως να αποδεσμεύει μνήμη που έχει κρατηθεί με την new, όχι όμως και τι θα κάνει με ιδιάζουσες περιπτώσεις κράτησης μνήμης από κάποιο αντικείμενο (π.χ. για γραφική απεικόνιση στην οθόνη, σύνδεση με δίκτυο ή ΒΔ).
  - Στη Java, η αποκομιδή σκυβάλων δεν ισοδυναμεί με καταστροφή των περιεχομένων των αντικειμένων (όπως στην C++). Αν υπάρχει κάποια δραστηριότητα που πρέπει να εκτελεσθεί πριν την ολοκλήρωση της χρήσης ενός αντικειμένου, τη δραστηριότητα αυτή πρέπει να την καθορίσει σαφώς ο προγραμματιστής.

finalize(): Καθαρισμός πριν την αποκομιδή

finalize(): Καθαρισμός πριν την αποκομιδή

- Σε κάθε κλάση της Java υπάρχει/μπορούμε να ορίσουμε μια μέθοδο **`finalize()`**, η οποία **μπορεί να κληθεί** από τον GC πριν την αποδέσμευση ενός αντικειμένου-σκουπιδιού από τη μνήμη.

finalize(): Καθαρισμός πριν την αποκομιδή

- Σε κάθε κλάση της Java υπάρχει/μπορούμε να ορίσουμε μια μέθοδο **`finalize()`**, η οποία **μπορεί να κληθεί** από τον GC πριν την αποδέσμευση ενός αντικειμένου-σκουπιδιού από τη μνήμη.
- Ορίζοντας μια κατάλληλη **`finalize()`** ο προγραμματιστής μπορεί να καθορίσει ρητά την εκκαθάριση πόρων και μνήμης, που δεν γίνεται αυτόματα από την Java, όπως π.χ.

finalize(): Καθαρισμός πριν την αποκομιδή

- Σε κάθε κλάση της Java υπάρχει/μπορούμε να ορίσουμε μια μέθοδο **`finalize()`**, η οποία **μπορεί να κληθεί** από τον GC πριν την αποδέσμευση ενός αντικειμένου-σκουπιδιού από τη μνήμη.
- Ορίζοντας μια κατάλληλη **`finalize()`** ο προγραμματιστής μπορεί να καθορίσει ρητά την εκκαθάριση πόρων και μνήμης, που δεν γίνεται αυτόματα από την Java, όπως π.χ.
  - Την διαγραφή περιεχομένου αντικειμένων που έγιναν σκουπίδια.

finalize(): Καθαρισμός πριν την αποκομιδή

- Σε κάθε κλάση της Java υπάρχει/μπορούμε να ορίσουμε μια μέθοδο **`finalize()`**, η οποία **μπορεί να κληθεί** από τον GC πριν την αποδέσμευση ενός αντικειμένου-σκουπιδιού από τη μνήμη.
- Ορίζοντας μια κατάλληλη **`finalize()`** ο προγραμματιστής μπορεί να καθορίσει ρητά την εκκαθάριση πόρων και μνήμης, που δεν γίνεται αυτόματα από την Java, όπως π.χ.
  - Την διαγραφή περιεχομένου αντικειμένων που έγιναν σκουπίδια.
  - Την απελευθέρωση πόρων που χρησιμοποιούνται από αντικείμενα-σκουπίδια (π.χ. παράθυρα σε γραφική διαπροσωπεία, συνδέσεις με βάσεις δεδομένων).

finalize(): Καθαρισμός πριν την αποκομιδή

- Σε κάθε κλάση της Java υπάρχει/μπορούμε να ορίσουμε μια μέθοδο **`finalize()`**, η οποία **μπορεί να κληθεί** από τον GC πριν την αποδέσμευση ενός αντικειμένου-σκουπιδιού από τη μνήμη.
- Ορίζοντας μια κατάλληλη **`finalize()`** ο προγραμματιστής μπορεί να καθορίσει ρητά την εκκαθάριση πόρων και μνήμης, που δεν γίνεται αυτόματα από την Java, όπως π.χ.
  - Την διαγραφή περιεχομένου αντικειμένων που έγιναν σκουπίδια.
  - Την απελευθέρωση πόρων που χρησιμοποιούνται από αντικείμενα-σκουπίδια (π.χ. παράθυρα σε γραφική διαπροσωπεία, συνδέσεις με βάσεις δεδομένων).
  - Την ελαχιστοποίηση διαρροών μνήμης.

finalize(): Καθαρισμός πριν την αποκομιδή

- Σε κάθε κλάση της Java υπάρχει/μπορούμε να ορίσουμε μια μέθοδο **`finalize()`**, η οποία **μπορεί να κληθεί** από τον GC πριν την αποδέσμευση ενός αντικειμένου-σκουπιδιού από τη μνήμη.
- Ορίζοντας μια κατάλληλη **`finalize()`** ο προγραμματιστής μπορεί να καθορίσει ρητά την εκκαθάριση πόρων και μνήμης, που δεν γίνεται αυτόματα από την Java, όπως π.χ.
  - Την διαγραφή περιεχομένου αντικειμένων που έγιναν σκουπίδια.
  - Την απελευθέρωση πόρων που χρησιμοποιούνται από αντικείμενα-σκουπίδια (π.χ. παράθυρα σε γραφική διαπροσωπεία, συνδέσεις με βάσεις δεδομένων).
  - Την ελαχιστοποίηση διαρροών μνήμης.
- Γενικά, η χρήση της finalize() δεν συνίσταται λόγω απρόβλεπτης συμπεριφοράς και ζητημάτων απόδοσης.

finalize()

- Στην περίπτωση που έχουμε ορίσει **`finalize`**, σε μια κλάση μας, όταν κληθεί ο συλλέκτης σκυβάλων (GC) και επιχειρήσει να απελευθερώσει τη μνήμη αντικειμένου της κλάσης ο GC:

finalize()

- Στην περίπτωση που έχουμε ορίσει **`finalize`**, σε μια κλάση μας, όταν κληθεί ο συλλέκτης σκυβάλων (GC) και επιχειρήσει να απελευθερώσει τη μνήμη αντικειμένου της κλάσης ο GC:
  - Ίσως καλέσει πρώτα την `finalize`

finalize()

- Στην περίπτωση που έχουμε ορίσει **`finalize`**, σε μια κλάση μας, όταν κληθεί ο συλλέκτης σκυβάλων (GC) και επιχειρήσει να απελευθερώσει τη μνήμη αντικειμένου της κλάσης ο GC:
  - Ίσως καλέσει πρώτα την `finalize`
  - στο επόμενο πέρασμα του θα απελευθερώσει τη μνήμη του αντικειμένου

finalize()

- Στην περίπτωση που έχουμε ορίσει **`finalize`**, σε μια κλάση μας, όταν κληθεί ο συλλέκτης σκυβάλων (GC) και επιχειρήσει να απελευθερώσει τη μνήμη αντικειμένου της κλάσης ο GC:
  - Ίσως καλέσει πρώτα την `finalize`
  - στο επόμενο πέρασμα του θα απελευθερώσει τη μνήμη του αντικειμένου
- Ο GC στοχεύει **στην απελευθέρωση μνήμης**. Αυτή θα πρέπει να είναι και η δραστηριότητα της `finalize`, όποτε χρησιμοποιείται.

finalize()

- Στην περίπτωση που έχουμε ορίσει **`finalize`**, σε μια κλάση μας, όταν κληθεί ο συλλέκτης σκυβάλων (GC) και επιχειρήσει να απελευθερώσει τη μνήμη αντικειμένου της κλάσης ο GC:
  - Ίσως καλέσει πρώτα την `finalize`
  - στο επόμενο πέρασμα του θα απελευθερώσει τη μνήμη του αντικειμένου
- Ο GC στοχεύει **στην απελευθέρωση μνήμης**. Αυτή θα πρέπει να είναι και η δραστηριότητα της `finalize`, όποτε χρησιμοποιείται.
- Η χρησιμότητα της finalize περιορίζεται κυρίως σε ειδικές περιπτώσεις («ιθαγενείς» μέθοδοι – native methods).

Ρητή Εκκαθάριση Αντικειμένων

- Στη Java τα αντικείμενα δημιουργούνται μόνο με χρήση της new.
- Δεν δημιουργούνται «τοπικά» αντικείμενα (στη στοίβα) και δεν υπάρχει μέθοδος delete για καταστροφή αντικειμένων.
- Αν θέλουμε να «εξαναγκάσουμε» την αποδέσμευση μη χρησιμοποιούμενων αντικειμένων, μπορούμε να καλέσουμε από το πρόγραμμά μας τον Συλλέκτη Σκυβάλων (GC), ακολουθούμενο από την μέθοδο runFinalization:

```
System.gc();
System.runFinalization();
```

- Πως λειτουργεί η **`System.runFinalization();`**

Παράδειγμα

```
// Demonstration of the garbage
// collector and finalization

class Chair {
  static boolean gcrun = false;
  static boolean f = false;
  static int created = 0;
  static int finalized = 0;
  int i;
  Chair() {
    i = ++created;
    if  (created == 47)
          System.out.println("Created 47");
  }
```

Παράδειγμα (συνέχεια)

```
public void finalize() {
   if (!gcrun) { // The first time finalize() is called:
          gcrun = true;
          System.out.println("Beginning to finalize after "
           + created  +" Chairs have been created");
   }
   if (i == 47) {
        System.out.println("Finalizing Chair #47, " +
                        "Setting flag to stop Chair creation");
        f = true;
   }
   finalized++;
   if  (finalized  >=  created)
        System.out.println(“All " + finalized + “finalized");
 }
}
```

Παράδειγμα (συνέχεια)

```
public class Garbage {
  public static void main(String[] args){
   while  (!Chair.f) {
         new Chair();
         new String("To take up space");
   }
  System.gc(); // forces execution of GC
  System.runFinalization(); // finalizes all
                            // unfinalized objects
  }
}
```

Τεχνικές Αποκομιδής Σκυβάλων

Τεχνικές Αποκομιδής Σκυβάλων

- Ο σωρός στην JAVA και το JVM λειτουργεί σαν ιμάντας (conveyor belt) – αντίθετα με την C++ όπου υπάρχει η δυνατότητα καταστροφής αντικειμένων.

Τεχνικές Αποκομιδής Σκυβάλων

- Ο σωρός στην JAVA και το JVM λειτουργεί σαν ιμάντας (conveyor belt) – αντίθετα με την C++ όπου υπάρχει η δυνατότητα καταστροφής αντικειμένων.
- Τεχνικές αποκομιδής σκυβάλων:

Τεχνικές Αποκομιδής Σκυβάλων

- Ο σωρός στην JAVA και το JVM λειτουργεί σαν ιμάντας (conveyor belt) – αντίθετα με την C++ όπου υπάρχει η δυνατότητα καταστροφής αντικειμένων.
- Τεχνικές αποκομιδής σκυβάλων:
- Reference counting (μέτρηση αναφορών) – τι γίνεται με αυτοαναφερόμενες κυκλικές δομές σκυβάλων;

Τεχνικές Αποκομιδής Σκυβάλων

- Ο σωρός στην JAVA και το JVM λειτουργεί σαν ιμάντας (conveyor belt) – αντίθετα με την C++ όπου υπάρχει η δυνατότητα καταστροφής αντικειμένων.
- Τεχνικές αποκομιδής σκυβάλων:
- Reference counting (μέτρηση αναφορών) – τι γίνεται με αυτοαναφερόμενες κυκλικές δομές σκυβάλων;
- Tracing live objects back to references living on the stack or static memory: ιχνηλασία σε ζώντα χειριστήρια.

Τεχνικές Αποκομιδής Σκυβάλων

- Ο σωρός στην JAVA και το JVM λειτουργεί σαν ιμάντας (conveyor belt) – αντίθετα με την C++ όπου υπάρχει η δυνατότητα καταστροφής αντικειμένων.
- Τεχνικές αποκομιδής σκυβάλων:
- Reference counting (μέτρηση αναφορών) – τι γίνεται με αυτοαναφερόμενες κυκλικές δομές σκυβάλων;
- Tracing live objects back to references living on the stack or static memory: ιχνηλασία σε ζώντα χειριστήρια.
- Προσέγγιση τού JVM: προσαρμοζόμενος ΑΣ (adaptive)

Τεχνικές Αποκομιδής Σκυβάλων

- Ο σωρός στην JAVA και το JVM λειτουργεί σαν ιμάντας (conveyor belt) – αντίθετα με την C++ όπου υπάρχει η δυνατότητα καταστροφής αντικειμένων.
- Τεχνικές αποκομιδής σκυβάλων:
- Reference counting (μέτρηση αναφορών) – τι γίνεται με αυτοαναφερόμενες κυκλικές δομές σκυβάλων;
- Tracing live objects back to references living on the stack or static memory: ιχνηλασία σε ζώντα χειριστήρια.
- Προσέγγιση τού JVM: προσαρμοζόμενος ΑΣ (adaptive)
  - Stop-and-copy: μεταφορά των ζώντων αντικειμένων από έναν σωρό σε κάποιον άλλο, αφού πρώτα σταματήσει η εκτέλεση του προγράμματος.

Τεχνικές Αποκομιδής Σκυβάλων

- Ο σωρός στην JAVA και το JVM λειτουργεί σαν ιμάντας (conveyor belt) – αντίθετα με την C++ όπου υπάρχει η δυνατότητα καταστροφής αντικειμένων.
- Τεχνικές αποκομιδής σκυβάλων:
- Reference counting (μέτρηση αναφορών) – τι γίνεται με αυτοαναφερόμενες κυκλικές δομές σκυβάλων;
- Tracing live objects back to references living on the stack or static memory: ιχνηλασία σε ζώντα χειριστήρια.
- Προσέγγιση τού JVM: προσαρμοζόμενος ΑΣ (adaptive)
  - Stop-and-copy: μεταφορά των ζώντων αντικειμένων από έναν σωρό σε κάποιον άλλο, αφού πρώτα σταματήσει η εκτέλεση του προγράμματος.
  - Mark-and-sweep: εντοπισμός και υποσημείωση των ζώντων αντικειμένων, ξεκινώντας από την στοίβα και την στατική μνήμη. Μετά το πέρας της υποσημείωσης, σάρωση του σωρού και εκκαθάριση των σκυβάλων.

Τεχνικές Αποκομιδής Σκυβάλων

- Ο σωρός στην JAVA και το JVM λειτουργεί σαν ιμάντας (conveyor belt) – αντίθετα με την C++ όπου υπάρχει η δυνατότητα καταστροφής αντικειμένων.
- Τεχνικές αποκομιδής σκυβάλων:
- Reference counting (μέτρηση αναφορών) – τι γίνεται με αυτοαναφερόμενες κυκλικές δομές σκυβάλων;
- Tracing live objects back to references living on the stack or static memory: ιχνηλασία σε ζώντα χειριστήρια.
- Προσέγγιση τού JVM: προσαρμοζόμενος ΑΣ (adaptive)
  - Stop-and-copy: μεταφορά των ζώντων αντικειμένων από έναν σωρό σε κάποιον άλλο, αφού πρώτα σταματήσει η εκτέλεση του προγράμματος.
  - Mark-and-sweep: εντοπισμός και υποσημείωση των ζώντων αντικειμένων, ξεκινώντας από την στοίβα και την στατική μνήμη. Μετά το πέρας της υποσημείωσης, σάρωση του σωρού και εκκαθάριση των σκυβάλων.
    - δουλεύει ικανοποιητικά όταν δεν υπάρχουν πολλά σκουπίδια