Programming languages Java

Data representation

Kozsik Tamás

ELTE
IK

Abstraction - type implementation

- Encapsulation
- Information hiding

Class, object, instantiation

Point.java

```java
public class Point {        // class definition
    int x, y;               // fields
}
```

Class, object, instantiation

Point.java

```
public class Point {        // class definition
    int x, y;               // fields
}
```

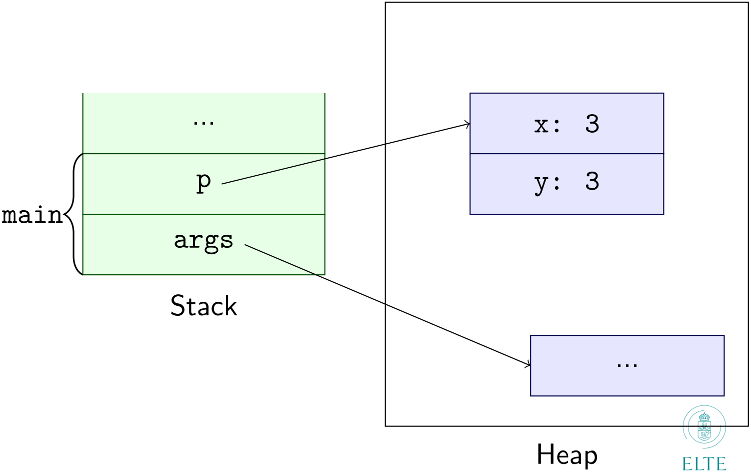
Main.java

```
public class Main {
    public static void main(String[] args) {
        Point p = new Point(); // instantiation (on heap)
        p.x = 3;               // changing object state
        p.y = 3;               // changing object state
    }
}
```

Compilation, execution

```
$ ls
Main.java  Point.java
$ javac *.java
$ ls
Main.class  Main.java  Point.class  Point.java
$ java Point
Error: Main method not found in class Point, please define
 the main method as:
   public static void main(String[] args)
$ java Main
$
```

Stack and heap



Fields: initialization

```
public class Point {
    int x = 3, y = 3;
}

public class Main {
    public static void main(String[] args) {
        Point p = new Point();
        System.out.println(p.x + " " + p.y);   // 3 3
    }
}
```

Fields: initialization with default value

A zero-like value is assigned to fields automatically.

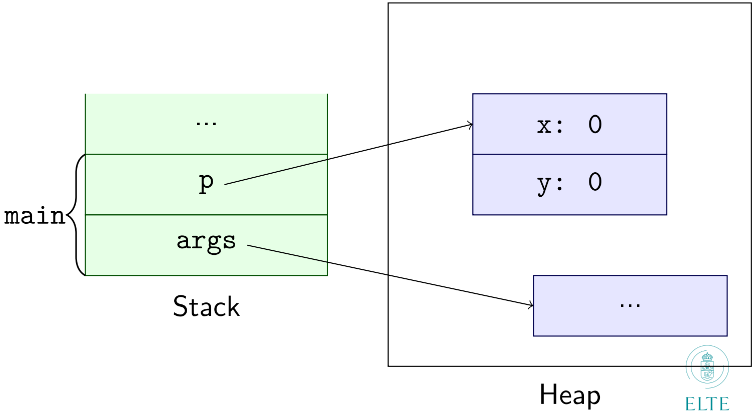
```
public class Point {
    int x, y = 3;
       // problem: it is difficult to see that x = 0,
       // better:  initialize all fields on separate lines
}

public class Main {
    public static void main(String[] args) {
        Point p = new Point();
        System.out.println(p.x + " " + p.y); // 0 3 (NOT 3 3)
    }
}
```

Method

```
public class Point {
    int x, y;   // 0 and 0
    void move(int dx, int dy) {   // implicit parameter: this
        this.x += dx;
        this.y += dy;
    }
}

public class Main {
    public static void main(String[] args) {
        Point p = new Point();
        p.move(3,3);   // p -> this, 3 -> dx, 3 -> dy
    }
}
```

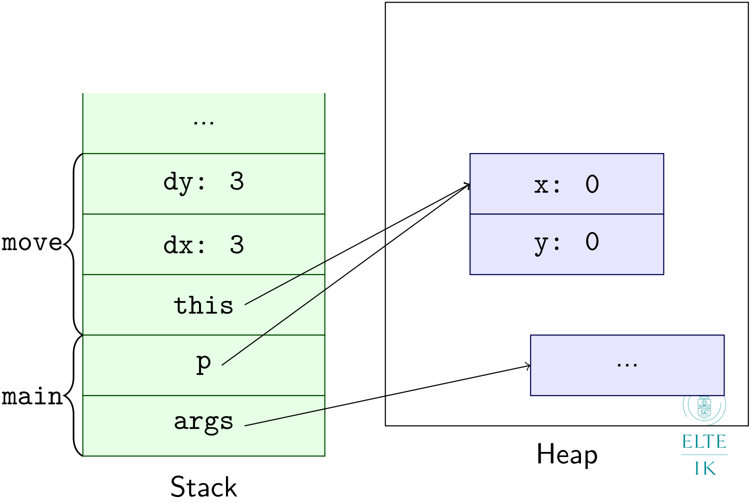
Method activation record – 1

```
Point p = new Point();
```



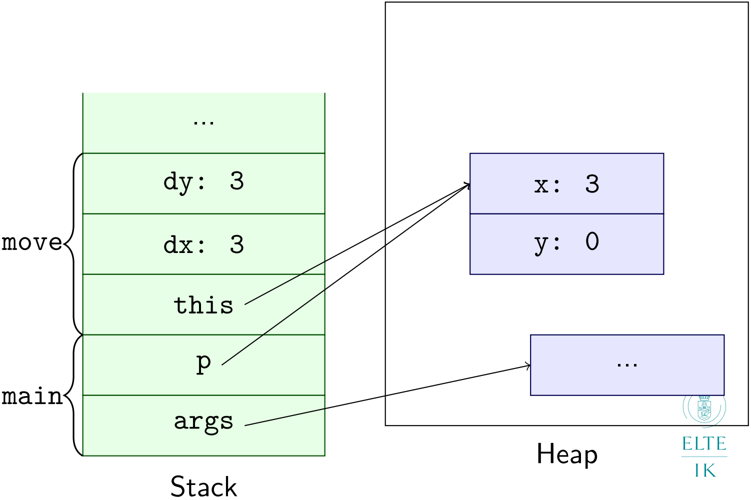
Method activation record – 2

```
p.move(3,3);
```



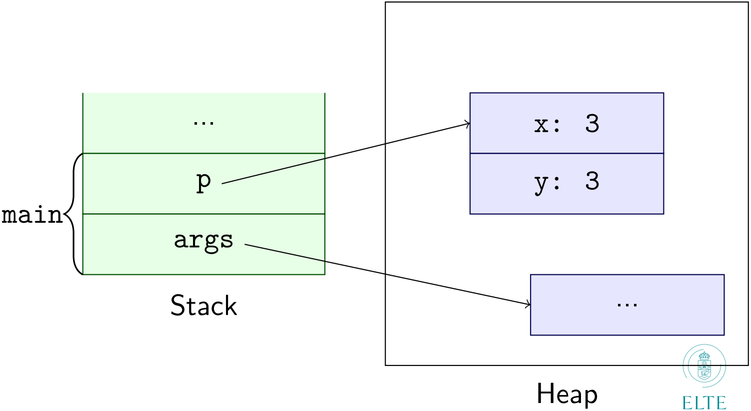
Method activation record – 3

```
this.x += dx;
```



Method activation record – 4

```
System.out.println(p.x + " " + p.y);
```



You may leave out this

Here, this is implicit.

```
public class Point {
    int x, y;   // 0, 0
    void move(int dx, int dy) {
        this.x += dx;
        y += dy;      // equivalent to this.y = dy
    }
}

public class Main {
    public static void main(String[] args) {
        Point p = new Point();
        p.move(3,3);
    }
}
```

Initialization using a constructor

```
public class Point {
    int x, y;
    Point(int initialX, int initialY) {
        this.x = initialX;
        this.y = initialY;
    }
}

public class Main {
    public static void main(String[] args) {
        Point p = new Point(0,3);
        System.out.println(p.x + " " + p.y);   // 0 3
    }
}
```

Initialization using a constructor without this

```
public class Point {
    int x, y;
    Point(int initialX, int initialY) {
        x = initialX;
        y = initialY;
    }
}

public class Main {
    public static void main(String[] args) {
        Point p = new Point(0,3);
        System.out.println(p.x + " " + p.y);   // 0 3
    }
}
```

Name reuse

```
public class Point {
  int x, y;
  public Point(int x, int y) {   // name hiding
    this.x = x; // field x is not visible: arg x hides it
                // this.x: qualified name
    this.y = y; // convention: give exactly the same name
                //             to related fields and args
  }
}

public class Main {
  public static void main(String[] args) {
    Point p = new Point(0,3);
    System.out.println(p.x + " " + p.y);   // 0 3
  }
}
```

No-arg constructor

This constructor takes no parameters.

```
public class Point {
    int x, y;
    Point() {}
}

public class Main {
    public static void main(String[] args) {
        Point p = new Point();
        System.out.println(p.x + " " + p.y);   // 0 0
    }
}
```

Default constructor

```
public class Point {
    int x, y;
}

public class Main {
    public static void main(String[] args) {
        Point p = new Point();
        System.out.println(p.x + " " + p.y);    // 0 0
    }
}
```

A no-arg constructor with an empty body is generated

```
Point() {}
```

Encapsulation

```java
public class Time {
  int hour;
  int min;
  Time(int hour, int min) {
    this.hour = hour;
    this.min = min;
  }
  void aMinPassed() {
    if (min < 59) {
      ++min;
    } else { ... }
  } // (C) Monty Python
}
```

```java
Time morning = new Time(6,10);
morning.aMinPassed();
int hour = morning.hour;
```

Type invariant

```
public class Time {
    int hour;                          // 0 <= hour < 24
    int min;                           // 0 <= min  < 60
    public Time(int hour, int min) {
        this.hour = hour;
        this.min = min;
    }
    void aMinPassed() {
        if (min < 59) {
            ++min;
        } else { ... }
    }
}
```

Creating a nonsense value

```
public class Time {
  int hour;
  int min;
  Time(int hour, int min) {
    this.hour = hour;
    this.min = min;
  }
  void aMinPassed() {
    if (min < 59) {
      ++min;
    } else { ... }
  }
}
```

```
Time morning = new Time(6,10);
morning.aMinPassed();
int hour = morning.hour;

morning.hour = -1;
morning = new Time(24,-1);
```

Ensure type invariant on creation

```
public class Time {
  int hour;                               // 0 <= hour < 24
  int min;                                // 0 <= min  < 60
  public Time(int hour, int min) {
    if (0 <= hour && hour < 24 && 0 <= min && min < 60) {
      this.hour = hour;
      this.min = min;
    }
  }
  void aMinPassed() {
    if (min < 59) {
      ++min;
    } else { ... }
  }
}
```

Avoid silent failure

```java
public class Time {
  int hour;                                 // 0 <= hour < 24
  int min;                                  // 0 <= min  < 60
  public Time(int hour, int min) {
    if (0 <= hour && hour < 24 && 0 <= min && min < 60) {
      this.hour = hour;
      this.min = min;
    } else {
      throw new IllegalArgumentException("Wrong time!");
    }
  }
  void aMinPassed() { ... }
}
```

Utility function

```java
public class Time {
  ...
  public Time(int hour, int min) {
    if (isBetween(hour, 0, 24) && isBetween(min, 0, 60)) {
      this.hour = hour;
      this.min = min;
    } else {
       throw new IllegalArgumentException("Wrong time!");
    }
  }
  // utility function: makes the code easier to understand
  private boolean isBetween(int value, int min, int max) {
    return min <= value && value < max;
  }
}
```

Early return

```
public class Time {
  public Time(int hour, int min) {
    // early return/throw: start by handling special cases
    if (!isBetween(hour, 0, 24) || !isBetween(min, 0, 60)) {
        throw new IllegalArgumentException("Wrong time!");
    }

    // "happy path": this is the common path
    this.hour = hour;
    this.min = min;
  }

  ...
}
```

Exception

- Occurs during runtime
- Indicates that execution continues in a non-usual way
  - ◇ Can signal some sort of problem
  - ◇ `throw` statement
- Possibly stops the program fully
- Can be handled in the program
  - ◇ `try–catch` statement

Runtime error

```
public class Main {
  public static void main(String[] args) {
    public Time morning = new Time(24,-1);
  }
}
```

```
$ javac Time.java
$ javac Main.java
$ java Main
Exception in thread "main" java.lang.IllegalArgumentException
Wrong time!
    at Time.<init>(Time.java:9)
    at Main.main(Main.java:3)
$
```

Fields can be changed directly

```java
public class Time {
  int hour;                      // 0 <= hour < 24
  int min;                       // 0 <= min  < 60
  ...
}
```

```java
public class Main {
  public static void main(String[] args) {
    public Time morning = new Time(6,10);
    morning.aMinPassed();

    morning.hour = -1;            // ouch!
  }
}
```

Hiding fields: private

```
public class Time {
  private int hour;                  // 0 <= hour < 24
  private int min;                   // 0 <= min  < 60
  ...
}
```

```
public class Main {
  public static void main(String[] args) {
    public Time morning = new Time(6,10);
    morning.aMinPassed();

    morning.hour = -1;                // compilation error
  }
}
```

Idiom: state is private, only changeable via methods

```
public class Time {
  private int hour;                // 0 <= hour < 24
  private int min;                 // 0 <= min  < 60
  public Time(int hour, int min) { ... }
  int getHour() { return hour; }
  int getMin()  { return min;  }
  void setHour(int hour) {
    if (0 <= hour && hour <= 23) {
      this.hour = hour;
    } else {
      throw new IllegalArgumentException("Wrong hour!");
    }
  }
  void setMin(int min) { ... }
  void aMinPassed()    { ... }
}
```

Convention: getters and setters

The field determines (almost) everything about getters/setters.

- name
- return type
- arguments
- body

```
public class Time {
  private int hour;              // 0 <= hour < 24
  public int getHour() { return hour; }
  public void setHour(int hour) {
    if (0 <= hour && hour <= 23) {
      this.hour = hour;
    } else {
      throw new IllegalArgumentException("Wrong hour!");
    }
  }
```

Can change the representation

```
public class Time {
  private short mins;
  public Time(int hour, int min) {
    if (...) throw new IllegalArgumentException("Wrong time!")
    mins = 60*hour + min;
  }
  int getHour() { return mins / 60; }
  int getMin() { return mins % 60; }
  void setHour(int hour) {
    if (...) throw new IllegalArgumentException("Wrong hour!")
    mins = 60 * hour + getMin();
  }
  void setmin(int min) { ... }
  void aMinPassed() { ... }
}
```

Information hiding

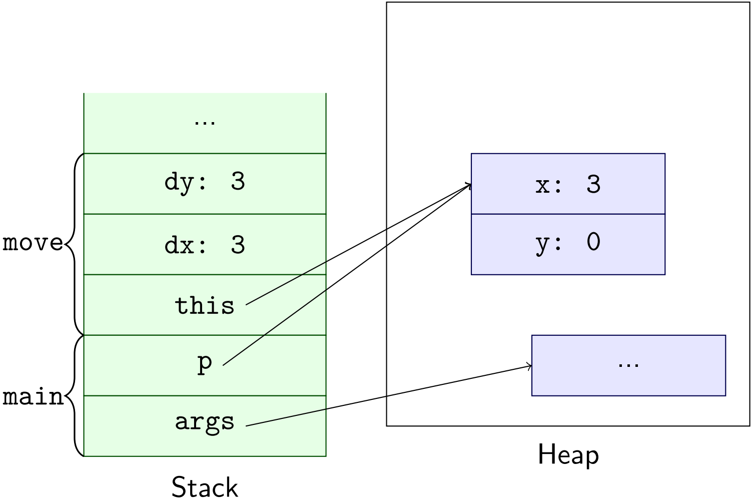
- Interface: make it lean
  - ◇ Allow only as much access to the object as necessary
  - ◇ This part is what is visible to the other classes
- Implementation details: make them inaccessible from the outside
  - ◇ Helper methods
  - ◇ Fields
- Advantages
  - ◇ Lets the class preserve its type invariant
  - ◇ Easier to evolve the code (representation change)
  - ◇ Loose coupling: other classes don't know about the internals
- Also desirable
  - ◇ Strong cohesion (the class has a single, well defined purpose)

Reference

```
Point p = new Point();
p.x = 3;
```

- Its type is a class (not a primitive)
- Refers to an object
- Stored on the heap
- Creation: `new`
- Dereference: `.`

Variables with different types in the memory



Primitive/reference types

Primitive types

- `byte`: $[-128..127]$
- `short`: $[-2^{15}..2^{15}-1]$
- `int`: $[-2^{31}..2^{31}-1]$
- `long`: 8 bytes long
- `float`: 4 bytes long
- `double`: 8 bytes long
- `char`: 2 bytes long
- `boolean`: $\{$`false`,`true`$\}$

Reference types

- Classes
- Array types
- …

Representation in the memory

Runtime stack

Local variables and parameters
(both primitives and references)

Heap

Objects and their fields
(both primitives and references)

Fields that belong to objects are called *instance variables*.

Scope and lifetime of variables

- Rules are similar to other languages such as C
- Lifetime of local variable: until the end of the scope
- Scope: from declaration to end of immediately containing block
- Hiding: only fields

```
public class Point {
    int x = 0, y = 0;
    void foo(int x) {       // OK
        int y = 3;          // OK
        {
            int z = y;
            int y = x;      // Compilation error
            ...
        }
    }
}
```

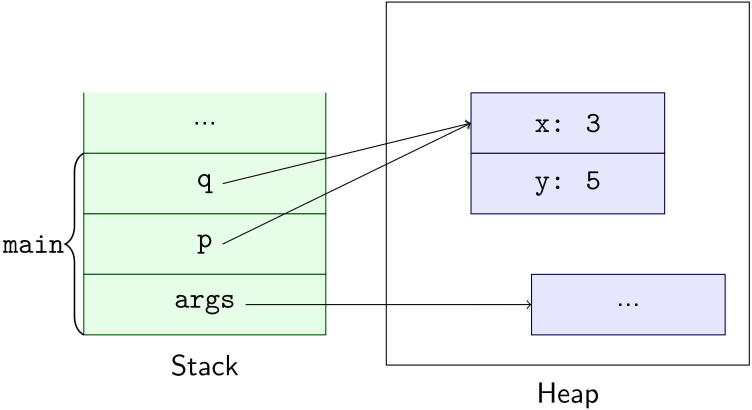
Lifetime of objects

- Creation + initialization
- Pointing references at the new objects
    - ◇ Aliasing
- Garbage collection

```
new Point(3,5)
Point p = new Point(3,5);
Point q = p;
p = q = null;
```

Aliasing

```
Point p = new Point(3,5), q = p;
q.x = 6;
```



Empty reference

```
Point p = null;
p = new Point(4,6);
if(p != null) {
    p = null;
}
p.x = 3;    // NullPointerException
```

Empty reference

```
Point p = null;
p = new Point(4,6);
if(p != null) {
    p = null;
}
p.x = 3;     // NullPointerExce
```

An error has occurred. See error log for more details.
java.lang.NullPointerException

An error has occurred. See error log for more details.
java.lang.NullPointerException

Initialization: fields

Automatically to a zero-like value

```
public class Point {
    int x = 0, y = 0;
}
```

```
public class Point {
    int x, y;
}
```

```
public class Point {
    int x, y = 0;
}
```

```
public class Point {
    int x, y = x;
}
```

Initialization: null reference

```
public class Hero {
    String name;          // == null
    Hero bestFriend;      // == null
}
```

```
Hero ironMan = new Hero();
ironMan.name = "Iron Man";
// ironMan.bestFriend == null
```

Initialization: local variables

- No automatic initialization
- Needs explicit assignment before it is used
  - ◇ Compilation error if missing (static semantic error)

```
public static void main(String[] args) {
    int i;
    Point p;
    p.x = i;    // compilation error for two reasons
}
```

Guaranteed assignment

- There has to be an assignment on all execution paths leading to the first use
- The Java compiler cannot check all corner cases (limited computability)

Example from JLS (Chapter 16, Definite Assignment)

```java
{
    int k;
    int n = 5;
    if (n > 2) {
        k = 3;
    }
    System.out.println(k); // k is not "definitely assigned"
                           // before this statement
}
```

Garbage collection

Releases (frees) objects that are not needed anymore

Correct

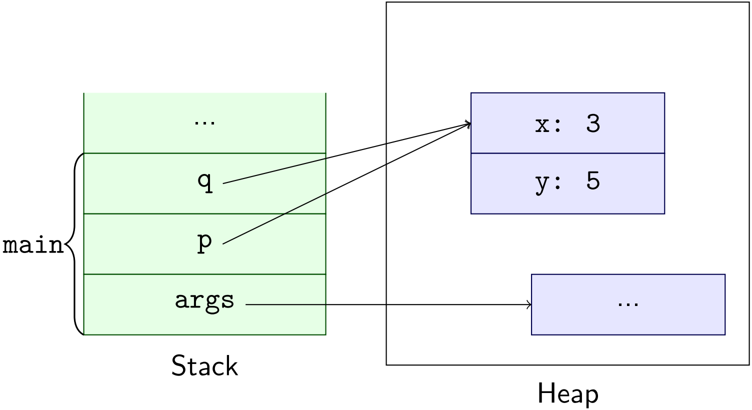
Only frees objects that are unreachable from the program

Complete

Frees all unreachable objects

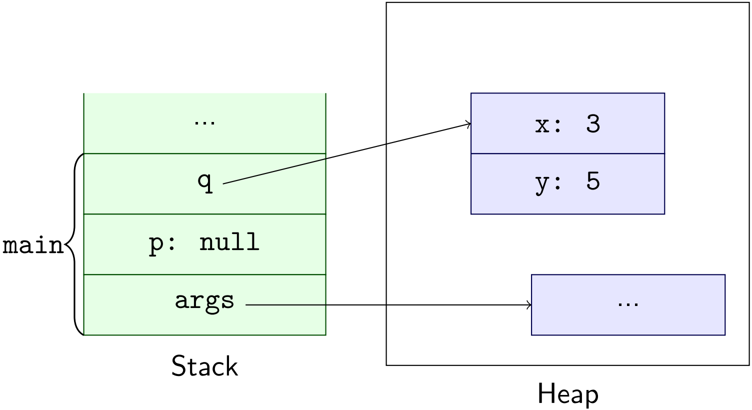
Cannot be released yet

```
Point p = new Point(3,5);
Point q = p;
```



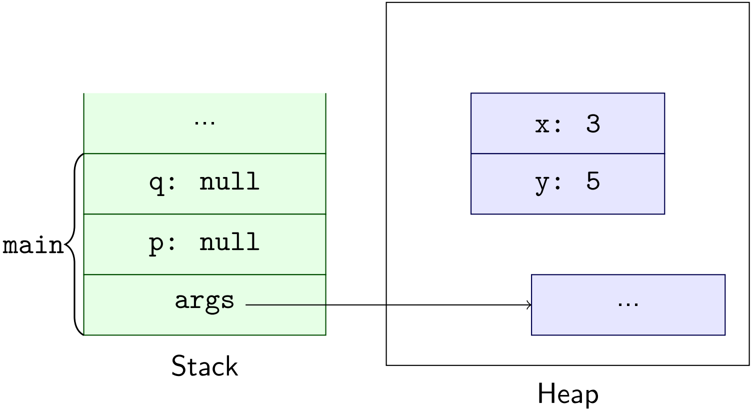
Still cannot be released

```
p = null;
```



Can be released now

```
q = null;
```

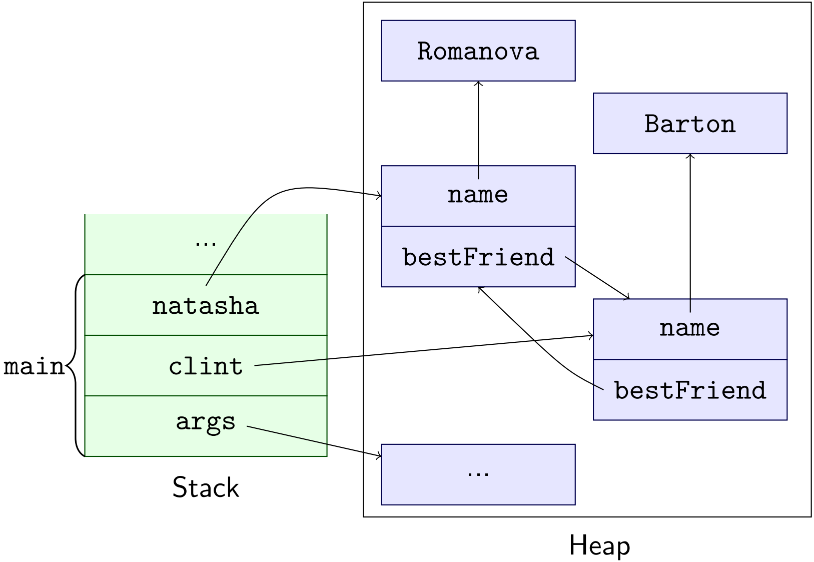


More complex example

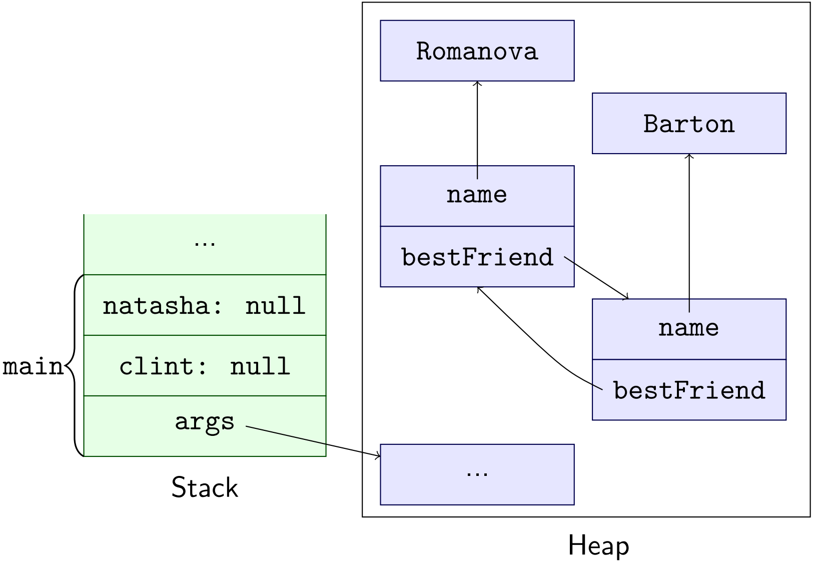
```java
public class Hero {
    String name;
    Hero bestFriend;
}
```

```java
Hero clint   = new Hero();
Hero natasha = new Hero();

clint.name         = "Barton";
natasha.name       = "Romanova";
clint.bestFriend   = natasha;
natasha.bestFriend = clint;
```

Heroes in the memory



```
natasha = clint = null;
```



Mark-and-Sweep garbage collection

- Mark phase
    - ◇ Starting point: references on the heap
    - ◇ Mark all objects reachable from there
    - ◇ Continue marking objects reachable from those until there are no more (transitive closure)
- Sweep phase
    - ◇ All unmarked objects can be freed now

Static fields

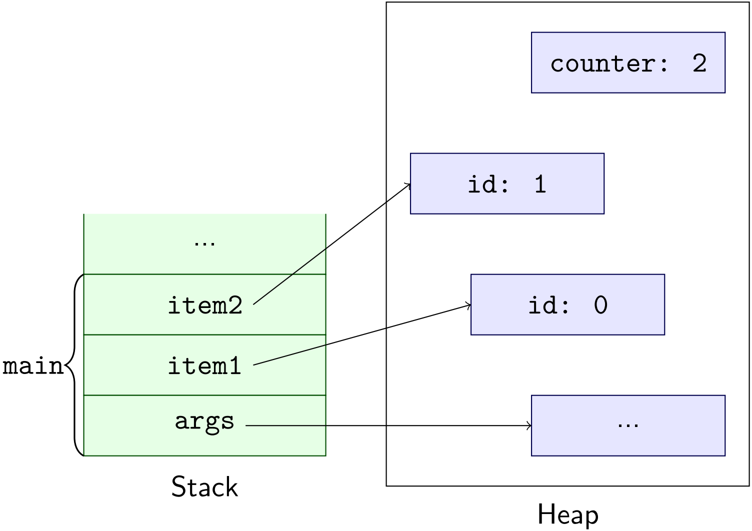
- Similar to global variables in C
- Only one instance exists
- Use the class name to access it
  - ◇ No need to have an instance
- Stored in *static storage*, not inside the objects themselves

```
public class Item {
    static int counter = 0;
}

public class Main {
    public static void main(String[] args) {
        System.out.println(Item.counter);
    }
}
```

Instance variables and class variables

```
public class Item {
    static int counter = 0;
    int id = counter++;          // meaning: id = Item.counter++
}

public class Main {
    public static void main(String[] args) {
        Item item1 = new Item(), item2 = new Item();
        System.out.println(item1.id);
        System.out.println(item2.id);

        // don't do this, even though it works, it's ugly
        // equivalent in meaning to   Item.counter
        System.out.println(item1.counter);
    }
}
```

```
Item item1 = new Item(), item2 = new Item();
```



Static methods

- Similar to global functions in C
- Can be called using the class
  - ◇ No need to have an instance
- Does not take the implicit parameter this
- Closely related to static fields

```
class Item {
    static int counter = 0;
    static void print(){
        System.out.println(counter);
    }
}
public class Main {
    public static void main(String[] args) {
        Item.print();
    }
}
```

Static methods lack `this`

```
class Item {
    static int counter = 0;
    int id = counter++;
    static void print(){
        System.out.println(counter);
        System.out.println(id); // cannot have a meaning
    }
}

public class Main {
    public static void main(String[] args) {
        Item.print();
    }
}
```

Array

- Data structure
- Elements one after the other in sequence
- Indexing in constant time
- First index is 0

Array types

```
String[] args
```

- `args` is a reference
- Arrays are objects
  - ◇ Stored on the heap
  - ◇ Creation: `new`
- Arrays know their size, e.g. `args.length`
- Indexing: runtime check
  - ◇ `ArrayIndexOutOfBoundsException`

Iterating arrays

```
public static void main(String[] args) {
    for (int i = 0; i < args.length; ++i) {
        System.out.println(args[i]);
    }
}
```

ArrayIndexOutOfBoundsException

```java
public static void main(String[] args) {
    for (int i = 0; i <= args.length; ++i) {
        System.out.println(args[i]);
    }
}
```

ELTE
IK

Enhanced for loop

```java
public static void main(String[] args) {
    for (String arg : args) {
        System.out.println(arg);
    }
}
```

Create, init, sort

```
class Sort {
    public static void main(String[] args) {
        int[] numbers = new int[args.length]; // all zeros

        for (int i = 0; i < args.length; ++i) {
            numbers[i] = Integer.parseInt(args[i]);
        }

        java.util.Arrays.sort(numbers);

        for (int n: numbers) { System.out.println(n); }
    }
}
```

Static import

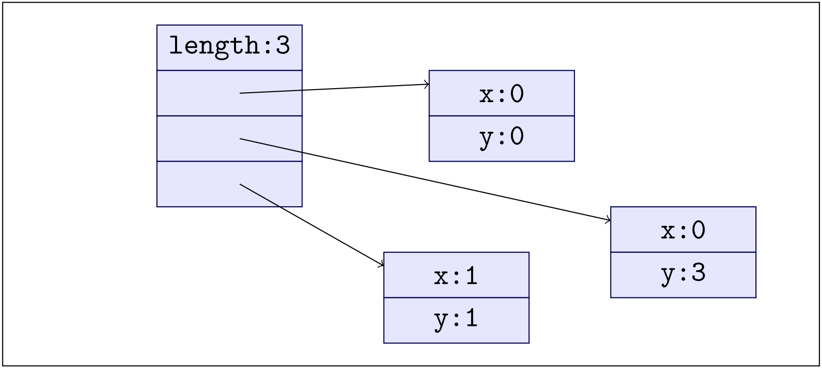
```
import static java.util.Arrays.sort;
class Sort {
    public static void main(String[] args) {

        int[] numbers = new int[args.length]; // all zeros

        for (int i = 0; i < args.length; ++i) {
            numbers[i] = Integer.parseInt(args[i]);
        }

        sort(numbers);

        for (int n: numbers) { System.out.println(i); }
    }
}
```

Array of references

```
Point[] triangle = { new Point(0,0),
                     new Point(0,3),
                     new Point(1,1) };
```

Array of references

```
Point[] triangle = { new Point(0,0),
                     new Point(0,3),
                     new Point(1,1) };
```



Heap

Step by step

```
static void walk(){
    Foot[] centipede;
    // error: The local variable centipede may not have been
    System.out.println(centipede.length);
```

Step by step

```
static void walk(){
    Foot[] centipede;
    // error: The local variable centipede may not have been
    System.out.println(centipede.length);

    centipede = null;
    System.out.println(centipede.length);
```

Step by step

```
static void walk(){
    Foot[] centipede;
    // error: The local variable centipede may not have been
    System.out.println(centipede.length);

    centipede = null;
    System.out.println(centipede.length);

    centipede = new Foot[100];
    System.out.println(centipede.length);
```

Step by step

```
static void walk(){
    Foot[] centipede;
    // error: The local variable centipede may not have been
    System.out.println(centipede.length);

    centipede = null;
    System.out.println(centipede.length);

    centipede = new Foot[100];
    System.out.println(centipede.length);

    for (int i = 0; i<100; i+=2) {
        centipede[i]   = new Foot("left");
        centipede[i+1] = new Foot("right");
    }
}
```

Matrix

```
double[][] id3 = { {1,0,0}, {0,1,0}, {0,0,1} };
```

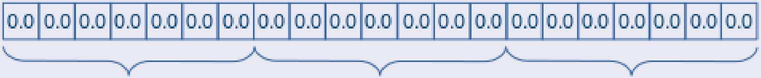
Matrix

```
double[][] id3 = { {1,0,0}, {0,1,0}, {0,0,1} };

static double[][] id(int n) {
    double[][] matrix = new double[n][n];
    for (int i=0; i<n; ++i) {
        matrix[i][i] = 1;
    }
    return matrix;
}
```

C versus Java

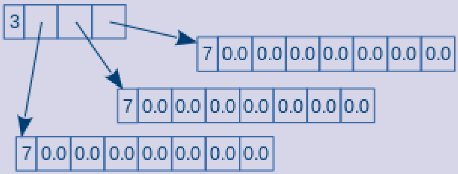
C: multidimensional array

```
double matrix[3][7];
for (int i=0; i<3; ++i)
    for (int j=0; j<7; ++j)
        matrix[i][j] = 0.0;
```



Java: array of arrays

```
double[][] matrix =
    new double[3][7];
```



Indexing

C: array with 3 dimensions

```
T t[L][M][N];
```

$$\mathrm{addr}(t_{i,j,k}) = \mathrm{addr}(t) + ((i \cdot M + j) \cdot N + k) \cdot \mathrm{sizeof}(T)$$

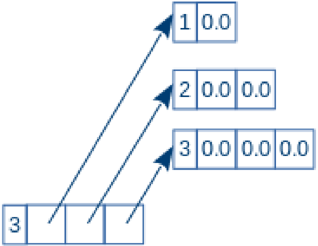
Java: array of arrays of arrays

```
T[][][] t = new T[L][M][N];
```

$$\mathrm{addr}(t_{i,j,k}) = \mathrm{val}_8\Big(\mathrm{val}_8(\mathrm{addr}(t)+4+i\cdot 8)+4+j\cdot 8\Big)+4+k\cdot\mathrm{sizeof}(T)$$

Lower triangular matrix

```
static double[][] zeroLowerTriangular(int n) {
    double[][] result = new double[n][];
    for (int i = 0; i<n; ++i) {
        result[i] = new double[i+1];
    }
    return result;
}
```



Command line arguments

- In C: `char *argv[]`
  - ◇ Java equivalent: `char[][] argv`
- In Java: `String[] args`

Reference types in Java

- `class`
  - ◇ enumeration types (`enum`)
- `interface`
- annotation types (`@interface`)

Enumeration type

```
enum Day { SUN, MON, TUE, WED, THU, FRI, SAT }
```

- It is a reference type
- Its values are objects, not simple `ints`

Enumeration type

```
enum Day { SUN, MON, TUE, WED, THU, FRI, SAT }
```

- It is a reference type
- Its values are objects, not simple `int`s

```
Day best = Day.SAT;        // also possible to "import static"
best = 3;                  // compilation error
int n = best;              // compilation error
int m = best.ordinal();    // 6
```

- The values are all created at the definition site
- No way to create further instances
  - ◇ The constructor cannot be called later: ~~`new Day()`~~

Constructors, members

```
enum Coin {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

    private final int centValue;

    Coin(int centValue) { this.centValue = centValue; }

    public int centValue() { return centValue; }

    public int percentageOf(Coin that) {
        return 100 * centValue / that.centValue();
    }
}   // Source: Java Community Process (modified)
```

In a switch

```
static int workingHours(Day day) {
    switch (day) {  // switch statement
        case SUN:
        case SAT: return 0;
        case FRI: return 6;
        default:  return 8;
    }
}
```

In a switch

```
static int workingHours(Day day) {
    switch (day) {  // switch statement
        case SUN:
        case SAT: return 0;
        case FRI: return 6;
        default:  return 8;
    }
}

static int workingHours(Day day) {
    return switch (day) {  // Java 12+: switch expression
        case SAT, SUN -> 0;
        case FRI -> 0;
        default -> 2;
    };
}
```