OOP Spring 2005
Lecture 2
Instance creation, enums, and garbage collection

Kasper Østerbye
IT University Copenhagen

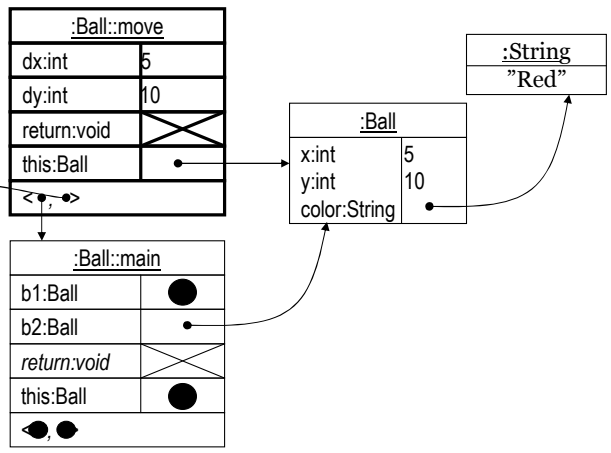
Today's schedule

- Creating objects
  - initialization
    - objects
    - arrays
  - constructors
  - inheritance
  - cloning
  - singular objects
  - enums and singular objects
  - static methods and fields
- Garbage collection
  - The problem
  - The techniques
    - reference counting
    - mark and sweep
  - finilize() method in class Object
  - method calls

Method call


```
class Ball {
    private int x,y;
    private String color;
    public Ball(String color){
        this.color = color;
        x=0; y=0;
    }
    public void move(int dx, dy){
        x+=dx;
        y+=dy;
    }
    …
}
public static void main(String[] args){
    Ball b1;
    Ball b2 = new Ball("Red");
    b2.move(5,10);´
…}
```

Follow the third statement in main



Comments:

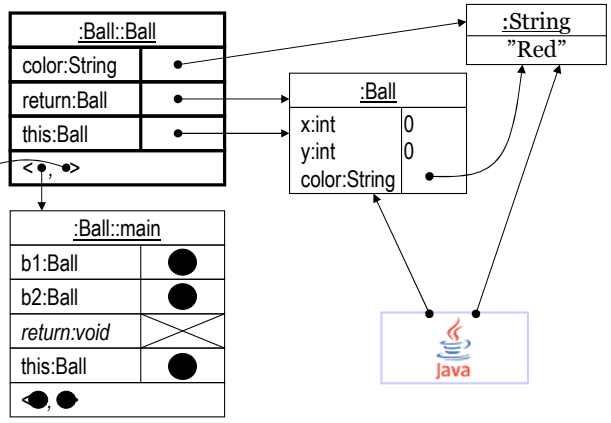
1) The return information in the main call cannot be shown. After the program has finished the java virtual machine stops running.
2) main is a static method. Static methods have no this reference. Perhaps it would be better to draw static method without a this reference. Here I have drawn it with a null this.
3) Similarly, one can say that void methods do not have a return value field. Here I have marked it with a cross. In real virtual machines, there will not be a return field.
4) Notice the type of the this reference is the name of the class in which the method occurs.

5) The update in move, where x is changed to 5 and y to 10, is worth noticing. There is no local variable named x in the method move. The compiler sees this, and looks in the enclosing class. Here it finds an x. The compiler then knows that x is really this.x.

Constructor call


```
class Ball {
    private int x,y;
    private String color;
    public Ball(String color){
        this.color = color;
        x=0; y=0;
    }
    public void move(int dx, dy){
        x+=dx;
        y+=dy;
    }
    …
}
public static void main(String[] args){
    Ball b1;
    Ball b2 = new Ball("Red");
    b2.move(5,10);´
…}
```

Follow the first two statements in main



This slide is a copy of a slide from lecture 1. Find the comments there.

Pay attention to the order in which the objects are created.

1) First the String object red is created, this is implicit because "Red" appears in the argument to the constructor.
2) Then the Ball object is created, with default values for x,y and color. The default values are 0 for integers, and null for references (color is a String reference).
3) Then the constructor is called, which creates a method call with the newly created Ball object as this, and the red String object as parameter.
4) The constructor copies the parameter color to the field color in the Ball object.
5) The result of the constructor is *always* the newly created object.

Examining object initialization

Object creation is done as

new ASuperClass()

When this expression is executed, the following happens:

1. An object of class ASuperClass is allocated from the Java heap.
2. Its fields are initialized to standard values. All numbers become 0, booleans are false, and references are null.
3. The fields that have an initializer, and *initializer blocks* are executed in the order they appear in the class
4. The constructor corresponding to the new expression is called.
5. The object is returned as result of the new ASuperClass() expression

```
public class ASuperClass {
   private TellTale field1;
   private TellTale field2 = new TellTale("A");

   ASuperClass(){
      System.out.println
            ("Starting construtor in ASuperClass");
      field1 = new TellTale
            ("In super constructor");
      System.out.println
            ("Ending construtor in ASuperClass");
   }

   {System.out.println
         ("Initializer block in ASuperClass");
   }
   private TellTale field3 = new TellTale("C");
}
```

Note: The class named ASuperClass serves no purpose other than to illustrate the order in which initialization takes place.

The notion of a Java heap will be explained later in the slides. For now, it is just a place in the computers memory. Where the object ends up in memory cannot be controlled by the Java programmer. In the C# programming language, it is possible for the programmer to decide that a given object must be located a specific place in memory. This is normally completely unimportant where an object is located, but in order to communicate with the hardware (platform dependence), it is sometimes a necessity.

As an exercise regarding point 2 above, write a class with public fields of primitive different types (int, double, char, boolean), and print out what their initial value is. This is done in a main method that creates an object, and reads and prints the value of each field.

So far, you have probably never encountered an initializer block. But they exist in Java. No one really uses them, but I just want to point out their existence, so you can recognize the syntax when you see it.

Examining how Java works

An investigation program is a program one writes to find out how Java works, or how a class works.

```
public class ASubClass extends ASuperClass{
    private TellTale field4;
    private TellTale field5  = new TellTale("AA");
    {   System.out.println
        ("Initializer block in ASubClass");}

    ASubClass(){
        super();
        System.out.println
            ("Starting construtor in ASubClass");
        field4 = new TellTale
             ("In subclass constructor");
        System.out.println
            ("Starting construtor in ASubClass");
    }
    private TellTale field6 = new TellTale("CC");
}
```

```
public class TellTale {
    TellTale(String s){
        System.out.println("[TellTale] " + s);
    }
}
public class InitializationOrderTest{
    public static void main(String[] args){
        System.out.println(
            "--- Making new ASuperClass ---");
        ASuperClass a = new ASuperClass();
        System.out.println(
            "--- Done making new ASuperClass ---");
        System.out.println(
            "--- Making new ASubClass ---");
        ASubClass aa = new ASubClass();
        System.out.println(
            "--- Done making new ASubClass ---")
    }
}
```

The classes ASuperClass, ASubClass, TellTale and InitializationOrderTest are all part of an investigation into how initialization takes place.

TellTale is the utility I have created for this, it has one constructor, which prints out its parameter so we can see when exactly the object was created.

When the main method is executed, one should get the following result:

```
--- Making new ASuperClass ---
[TellTale] A
Initializer block in ASuperClass
[TellTale] C
Starting construtor in ASuperClass
[TellTale] In super constructor
Ending construtor in ASuperClass
--- Done making new ASuperClass ---

--- Making new ASubClass ---
[TellTale] A
Initializer block in ASuperClass
[TellTale] C
Starting construtor in ASuperClass
[TellTale] In super constructor
Ending construtor in ASuperClass
[TellTale] AA
Initializer block in ASubClass
[TellTale] CC
Starting construtor in ASubClass
[TellTale] In subclass constructor
Starting construtor in ASubClass
--- Done making new ASubClass ---
```

Notice, when we create ASubClass, all initialization (initializers and constructors) of the super class is done before any initialization of the sub class. This is always the case.

Constructors and inheritance


```
class A {
    public String field1;
    A(String f){field1 = f;}
    A(){this("Hello");}
}
class B extends A {
    public int field2;
    B(String f, int i){
        super(f);
        field2=i;
    }
}
class C extends A{}
```

These are the important rules for constructors in Java:

1. All classes have at least one constructor. If you do not define any constructor, the Java compiler creates a default constructor with no arguments.
    1. The default constructor for class C is C(){super();}
    2. If you define a constructor yourself, no default constructor is created.
2. One *does not* inherit constructors, there is no constructor for C that takes a string argument
3. The first statement of a constructor must be a call to a super constructor, and there can be no call to a super constructor later.
4. If no super call is placed at the top, an implicit call to the parameterless super constructor is inserted before any other statement in the constructor.

Notice the this("Hello") statement in the parameter-less constructor of class A. It means that the A() constructor calls the A(String) constructor to do the initialization. This is good programming practice. If a class has more than one constructor, it is a good idea to have only one constructor that actually sets the fields, and the other constructors call it. That way there is only one constructor to change if the objects need a different initialization.

At the lecture, I (hoped to) show the following two classes online. They break the rules about constructors in Java:

```
class D extends B{} // notice – extends B, not A
```

This class is erroneous, as a default constructor D(){super();} is created, but B does not have a constructor without arguments, hence there is no match for the super call.

```
class E extends B{// notice – extends B, not A
        E(){super("Hello");}
}
```

This class is also erroneous, as the class B does not inherit the constructor that only takes a String as parameter.

Array initialization

Remember the following things:

An array is an object.

An array can not change its size once it is created

All java types has a corresponding array type, including arrays (known as multiple dimension arrays).

An instance of an array is created using new T[10], which makes an array where the elements are of type T, and are indexed from 0 to 9.

There are several ways in which one can initialize an array.

1. The expression new T[10] creates an array of 10 places, each place initialized with the default value of T.
2. The expression {x,y,z} can be used in array initializers such as

   ```
   T[] field1= {x,y,z}
   ```

   or

   ```
   field1 = new T[]{x,y,z}
   ```

   x, y, and z must have type T.
3. Two dimensional arrays are declared as T[][]. (Three dimensions as T[][][])
4. The expression new T[2][3] creates a new array object with two places. Each place is initialized with an array of three places, capable of containing a T value.

It is important to remember that arrays are really objects. The important thing is that a variable students, declared as: ”Person[] students” is a variable containing a reference to an array. Thus if we have an other array variable participants, declared as Person[] participants, they both have type ”Person[]”.

Hence, we can assign one to the other as ”students = participants”. An assignment such as ”students[8] = new Person(”Lars”);” will change the array object at index 8. But as both variables refers to the same array, reading the 8th index via the participants variable, will give the Person named Lars we just created.

Notice the difference in how array-initializers look, depending on whether the variable is declared in the statement or not. In the case it is, we can omit the ”new” operator.

Please observe that a two dimensional array is really an array of arrays. In most other programming languages (including C++, C#, Pascal, Visual Basic), there exist real multiple dimensional arrays. In Java, int[][] is an array of int[].

This mean that we can declare an array like:

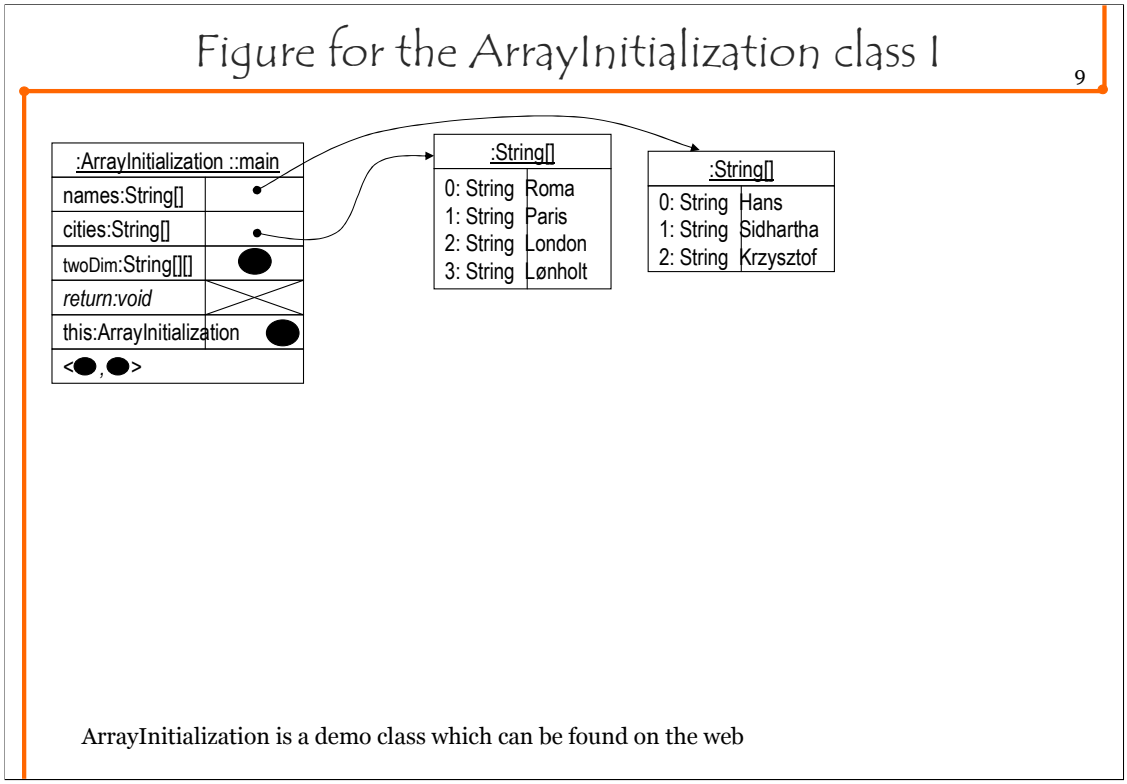
```
int[][] jagged = { {1}, {2,3}, {4,5,6}, {7,8}, {9}};
```

where there is a different size to each array in the second dimension array.

If you want to loop through each element, this should be done as:

```
for (int i=0;i<jagged.length;i++)
    for (int j=0; j<jagged[i].length;j++)
        sum += jagged[i][j];
```

Notice that we get a different length for each i in jagged[i].



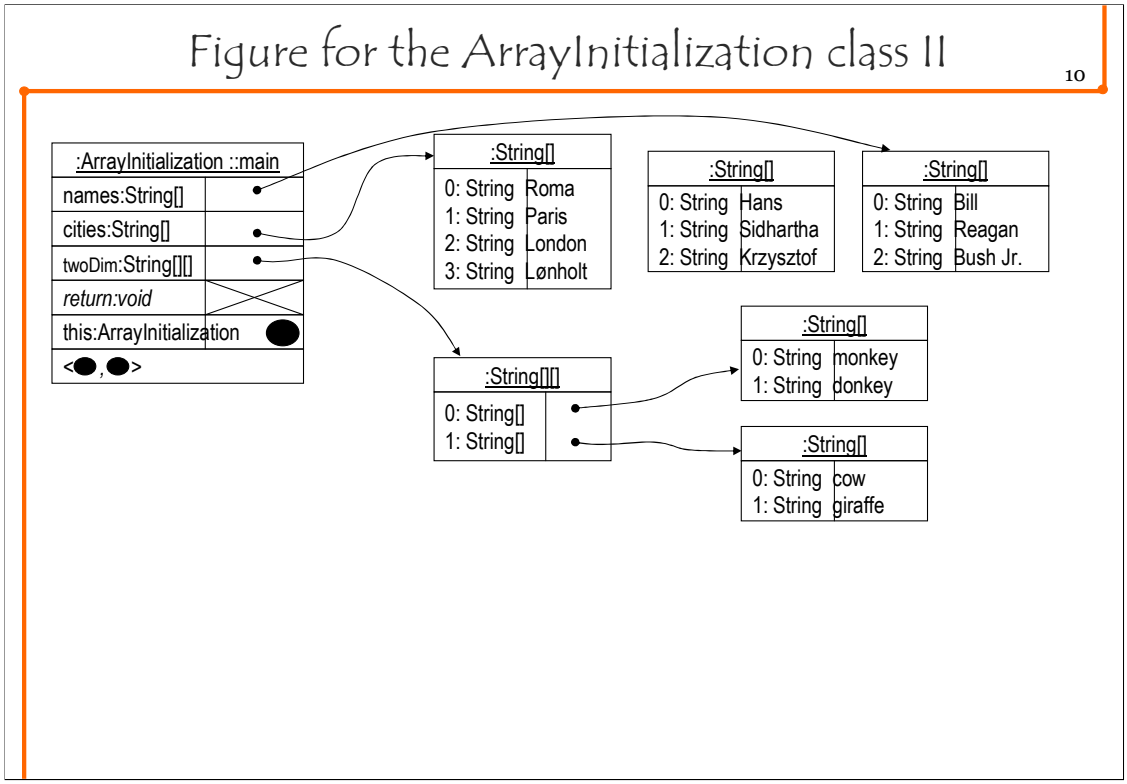
I have here used an abbreviated syntax for the Strings in the figure. In reality, Strings are objects, which internally contain an array of char. But that is too big a figure to draw.

The code is found in the class ArrayInitialization, and in the code it is shown which snapshots corresponds to where in the code.

Remember that the main method is static, hence the this reference is null. I have omitted the args parameter from the main method call, as I do not use it for anything.

The code below is what is shown sofar:

```
public static void main(String[] args){
    String[] names = null;
    String[] cities = {"Roma", "Paris", "London", "Lønholt"};
    names = new String[3];
    names[0] = "Hans";
    names[1] = "Sidhartha";
    names[2] = "Krzysztof";
    // State shown in slide 1
    …
}
```

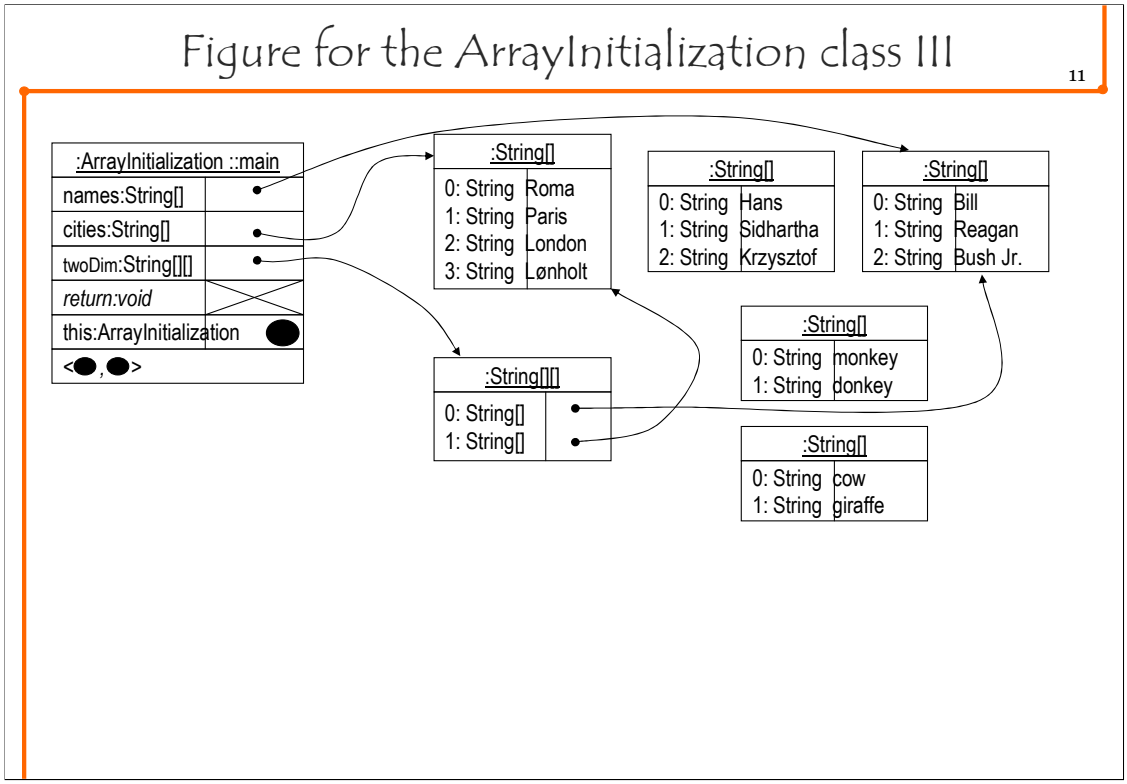


In this figure the following two statements have been executed as well:

```
names = new String[]{"Bill","Reagan","Bush Jr"};
String[][] twoDim = { {"monkey","donkey"}, {"cow", "giraffe"}};
```

Observe that there is no longer any reference to the array with the names Hans, Siddhatha, and Krzyztof. This array will be garbage collected when Java needs space for new objects.

Also, the two dimensional array is really an array of arrays, here drawn as such.



Notice, the twoDim array is declared as String[][], but when instantiated, it is really an array with element type String[]. Therefore it is possible to make the first place in the String[][] object refer to the String[] refered to by names, and the second to the String[] refered to by cities.

The above figure shows the situation just after the last two statements in the main method:

```
twoDim[0] = names;
twoDim[1] = cities;
```

Observe how the first array in twoDim refers to an array with length 4, and the second refers to one of length 3 – A jagged array.

We will take one more look at arrays when we next look at cloning

An other way to get a new object, cloning

Rather than building an object from scratch each time, sometimes one would like a copy of an existing object.

In Java, the class Object offers a method clone(), which returns a copy of the object. The clone method copies all fields, private as well as protected.

There are two different approaches to clone

- Deep clone, in which all objects refered to are clones as well.
- Shallow clone, in which the original and the clone refer to the same objects from their field variables of reference type.

more on deep vs. shallow cloning on next slide

Typical properties of a clone in Java

1. clone != original
2. clone.getClass() == original.getClass()
3. clone.equals(original) is true
4. If a class should be clonable, it must implement the Cloneable interface.
5. If one tries to clone an object that does not implement the Clonable interface, an exception is thrown.

Most object oriented programming languages support cloning, one way or the other. No language I am aware of supports deep cloning in the language itself, deep cloning must always be implemented by the programmer.

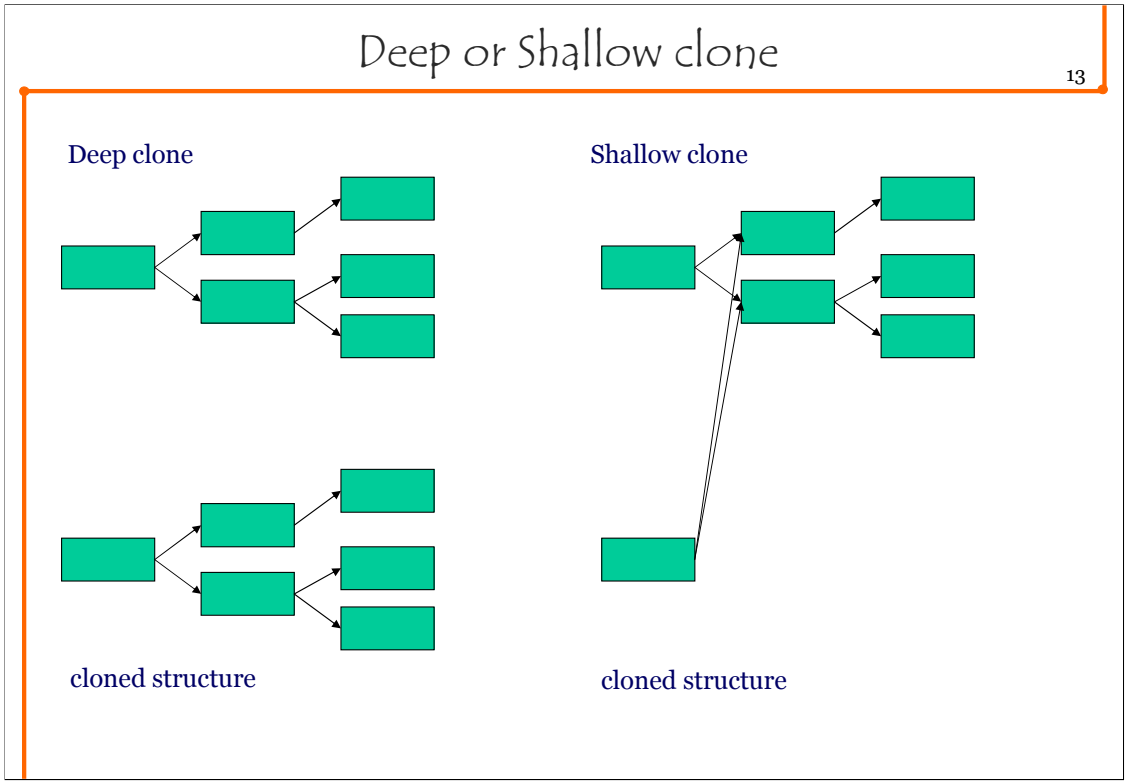
Notice the first three properties of a clone.

1) This states that the clone and the original are different objects.
2) This states that the clone and the original are instances of the same class.
3) This states that the clone is equal to its original. In Java, it is your responsibility to program clone and equals in such a manner that this rule is kept, it is not something enforced by Java.

Java has done cloning particulary complicated.

1. There is a clone method in class Object, which shallow copies all fields in the object. That method is protected, which means that one can not in general clone an object as myObj.clone().
2. If one wants to equip a class C with a clone method, one must override the clone method from Object. The overridden clone method should then be declared public.
3. If one wants to be able to use a clone method in a class C, C should implement the *marker interface* Cloneable. A marker interface is an interface without any methods, that the Java Virtual machine uses for its internal behaviour. In the case of cloning, it throws an exception *CloneNotSupportedException*, if one tries to clone on an object which do not implement Cloneable.
4. Unfortunately, this means that one should always be prepared to catch a *CloneNotSupportedException* when you clone an object.
5. Finally, clone is defined in class Object to return something of type Object. One cannot define clone in class C to return something of type C, but only of class Object.

Cloning is not used much in Java!



In the deep clone version, when one clones an object, one also deep clones all the objects refered to by the object (notice the recursive definition).

In the shallow copy, one only clones the top object (the message receiver). Therefore all reference fields in the clone refers to the same objects as the original.

All primitive fields are copied in both versions.

Java clone is shallow clone (sometimes also called copy). But one can override the clone method to do the kind of clone one wants.

In particluar notice that if one clones an array, the elements are not cloned. That is, arrays are not different from other objects, and are shallowly cloned.

If one declares a twodimentsional array:

```
int[][] myTable = { {1,2,3}, {4,5,6}, {7,8,9}};
```

and creates a clone of it

```
int[][] clonedTable = (int[][])myTable.clone();
```

and changes the element at 1,1 in myTable

```
myTable[1][1]=77;
```

that also changes the element at 1,1 in the cloned array

```
System.out.println(clonedTable[1][1]); // prints 77
```

This is because a two dimentional array is really an array of arrays, and only the outermost array is cloned. Try to draw myTable as an array of arrays, and shallow clone it.

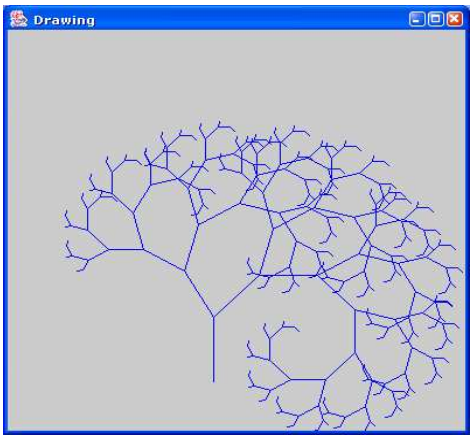
A clone example

A tree can be drawn by first drawing a line, then drawing two smaller trees at the end of that line. Very small trees are without any branches.

By tradition we often use a pen object when drawing.

When drawing a tree, we need to be able to find back to the end of the first line when we start to draw the second smaller tree.

```
private static void tree(Pen p, int height){
    if (height < 2 )   return;
    p.move(height);
    Pen leftTreePen = (Pen)p.clone();
    leftTree.turn(30);
    Pen rightTreePen = (Pen)p.clone();
    rightTree.turn(-45);
    tree(leftTreePen,height-10);
    tree(rightTreePen, height-5);
}
```



The example code is in the file Drawing.java.

It consists of three classes. Drawing, which has the main method, which sets up a Frame for drawing upon. If you have not seen this before, just ignore the main method. The Drawing class also has the above tree method.

A MyFrame class is the specific Frame used for drawing.

There is a clue in that class, namely the method makePen. It makes sure that the new pen has a frame to draw upon, by passing this as its second parameter. This means that in the rest of the code, the pen knows what paper to draw upon.

The Pen class is the one that implements the simple methods for drawing. A pen knows where it is (x,y), and what direction it is pointing in (heading). The move method takes a parameter *length*, and when called, draws a line of that length, starting at its current position, drawing in the pen's direction, and after the line has been drawn, the (x,y) is updated to the new position. The moveTo method draws from the current position to the new position given as parameter. The move methods do not change the heading. The jump methods work the same way, only they do not draw any line, but just update the position. The turn method changes the direction of the pen. The clone method creates a clone of the pen, with the same position and heading.

Singular object (Singleton)

- In Java, C#, C++ and several other languages, all objects must be instances of a named Class.
- Sometimes only one object can or should exist of the given class. E.g. a class representing the screen.
- How can we ensure that only one instance is created, and how can we make that instance accessible throughout the system.

- The simple, and OK solution, is shown to the right, and has two key points
  - A static method to get the Santa, as Santa.theOneAndOnly();
  - A **private** constructor, preventing new Santa() from outside the class, or from any subclass.

```
public class Santa {
    private static Santa santa;

    public static Santa theOneAndOnly(){
        if (santa == null) { // lazy instantiation
            santa = new Santa();
        }
        return santa;
    }

    private Child[] theGoodOnes;
    private Child[] theBadOnes;

    private Santa(){
        theGoodOnes=new Child[1000000000];
        theBadOnes =new Child[1]; // you :-)
    }

    public String toString(){
        return "Ho Ho Ho";
    }

    // more Santa stuff
}
```

The trick mentioned in the static method theOneAndOnly called lazy instantiation covers an *ideom* (fixed phrase), in which one do not initialize a field in the constructor, but in the get method. If the reference is null, noone has asked for the field before, and we create a new object, and stores the reference to that object in the field.

Next time we ask for the object, the field is not null, and we return the object directly.

In general, lazy instantiation is useful for initializing variables one does not know if we will need, thus we only make an object if anyone is interested. It is called lazy, because is is done at the latest possible time.

For the principle to work in this context, it is important to program all the methods in Santa in such a way that the variable Santa is not given a new value inside the class. In particular one must not set it to null, as that causes a new instance to be created next time theOneAndOnly() method is called.

Alternatively, we could have written: `public static final Santa santa = new Santa();`

This alternative initializes the santa variable *eagerly* (at the earliest possible time). Also notice that the final keyword ensures that no assignment is made to this reference later. Hence, it is OK to make it public, noone can assign it any new value, it can only be read.

The principle with the private constructor is worth noticing. It prevents the usage of new from outside the class.

Enumeration types

16

- Sometimes we need not one, but a few of a specific type.
  - Weekdays: Monday, Tuesday, …
  - Months: January, February, …
  - Playing cards: Spades, Clubs,…
  - Traffic light: Red, Yellow, Green
- Common for this situation is that:
  - It will often not make any sense to add yet a value.
  - Each of the values do not really have much state if its own.
  - Sometimes we are interested in their order

- enum types are just for this:

```
public enum Weekday
{
   monday, tuesday, wednesday, thursday,
   friday, saturday, Sunday
}
```

- This is really very similar to:

```
public class Weekday
{  private Weekday(){};
   public static final monday = new Weekday();
   public static finale tuesday = new Weekday();
   …
}
```

- This is know as the *Typed Enum Pattern*

Enumeration types have a long history: You might encounter them in Pascal/Delphi, and C/C++ and C#.

In some languages (C and C++), enum TrafficLight{ Red, Yellow, Green} becomes three integer constants. This means that:

TrafficLight.Red+1 == TraficLight.Yellow

TrafficLight.Green + 100 which is not a valid color

In other words, the enumerations are not type-safe, you can compare Monday and Spades.

In Delphi/Pascal they are implemented as integer constants, but are not type compatible with integer. Thus, one cannot add two weekdays, one cannot compare a weekday with a playing card etc.

C# is very similar to Pascal/Dephi.

Static fields and methods – class objects

- As an alternative to the singular object implementation, one can declare everything in Santa as static.
- A static field exists exactly once in the system, hence we have what corresponds to one object.
- Santa methods can be accessed as in Santa.tellStory(”Peters christmas”);
- Initialization can be done in static initializers, which are executed when the class is loaded into the virtual machine.
- In general, one uses the previous solution.

```
public class Santa {
    private Santa(){throw Exception("Do not!");}

    private static Child[] theGoodOnes;
    private static Child[] theBadOnes;

    static { // a static initializer block
        theGoodOnes=new Child[1000000000];
        theBadOnes =new Child[1]; // you :-)
    }

    public static String toString(){
        return "Ho Ho Ho";
    }

    public static String tellStory(String storyName){
        … return …
    }
    // more Santa stuff
}
```

One should in general not use the class to represent a singular object. There are several things one cannot do with objects represented as classes. If we had defined Santa as a subclass of Person, a person variable cannot refer to Santa as a class.

Also, we cannot pass Santa as a parameter this way.

And should we later decide to disregard the singular restriction (E.g. putting two screens on one computer), it is less cumbersome to change the code.

The story below is not required reading. Ask in a break if you are interested.

In some programming languages this notion of the class as an object is pursued a long way. In the language Java, all classes are instances of one class named Class (one always names a class after what kind of instances it has, which is another reason not to let Santa be represented using static only fields).

In the language Smalltalk, which was originally defined in 1980, all classes are instances of each their own class. So the class Person is an instance of PersonClass. Classes whose instances are classes, are called meta-classes. Therefore PersonClass is an instance of MetaClass. MetaClass is then again instance of the class MetaClassClass. The story ought to go on, but it does not, MetaClassClass is, like PersonCass, considered to be a MetaClass. Hence MetaClassClass is an instance of MetaClass. This is truly strange, MetaClass is an instance of MetaClassClass, which is an instance of MetaClass. This is of course not possible, it is a chicken and egg situation. The virtual machine has these classes predefined, so the Smalltalk world solves the problem by just having both from the beginning.

The language C++ solves the problem by classes not being objects. Boring, but consistent. Java says all classes are instances of Class, even though they all have different static methods and fields.

Garbage collection/Memory recycling

- What is the problem
  - Every time the program creates a new object, it has to do it somewhere in memory.
  - If all objects require a new place in memory, we cannot keep on allocating memory for new objects.
- Therefore
  - If the system needs to create more objects than can fit in memory, we should be able to recycle the memory of objects which are no longer in use.
- No recycling is needed when
  - The program only needs to make a limited number of objects

```
class Garbage {
    public static void main(String[] args){
        Person p1 = new Person(”Hans”);
        Person p2 = new Person(”Olga”);
        p1 = new Person(”Lars”);
        // what happens to the
        //”Hans” person object.
    }
```

So, the assumption is that from now on

- we actually need more objects than can fit in memory, and
- some objects are no longer needed by the program – for example the person object with name ”Hans” above.

Memory recycling is about automatic reuse of the memory of dead objects. A dead object is an object which can no longer be used in the program. An object is certainly dead if no variable in the system refers to it.

Memory recycling (also called garbage collection) is about finding objects which are no longer refered to by any variable.

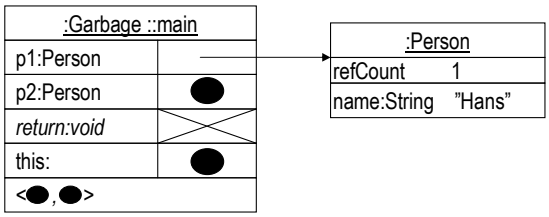
On the next slides we shall examine three different approaches to this:

- reference counting
- Mark and sweap
- Copying collectors

Reference counting and mark and sweap leads to a thing called fragmentation.

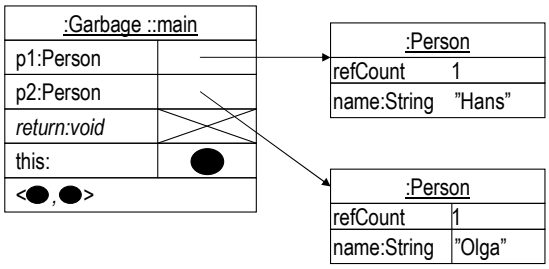
We shall not dive deep into these algorithms, but just get an overview of the different approaches. Today virtual machines like JVM uses variations of all three, they each have different properties, and can be combined for better performance.

Reference counting



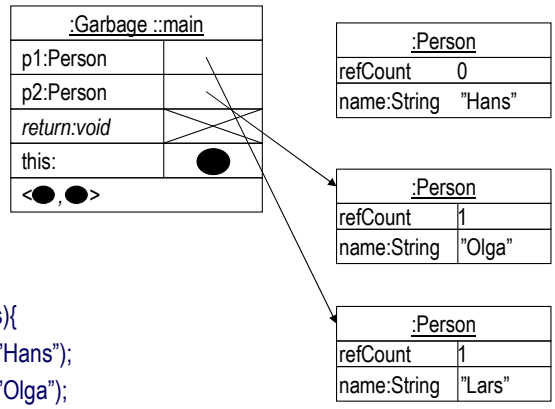

```
class Garbage {
   public static void main(String[] args){
            Person p1 = new Person("Hans");
            Person p2 = new Person("Olga");
            p1 = new Person("Lars");
            // what happens to the
            //"Hans" person object.
   }
```

Reference counting



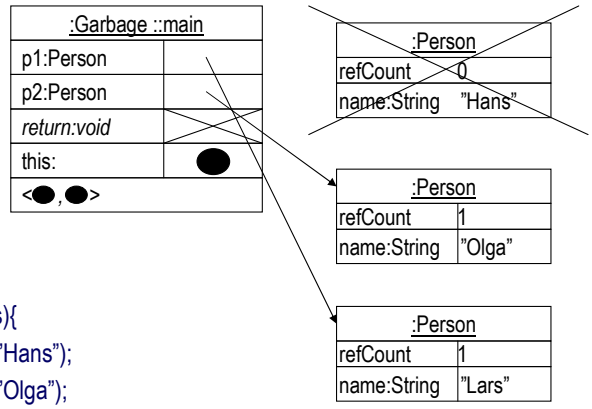

```
class Garbage {
   public static void main(String[] args){
            Person p1 = new Person("Hans");
            Person p2 = new Person("Olga");
          p1 = new Person("Lars");
            // what happens to the
            //"Hans" person object.
   }
```

Reference counting




```
class Garbage {
   public static void main(String[] args){
            Person p1 = new Person("Hans");
            Person p2 = new Person("Olga");
            p1 = new Person("Lars");
          ●// what happens to the
            //"Hans" person object.
   }
```

Reference counting



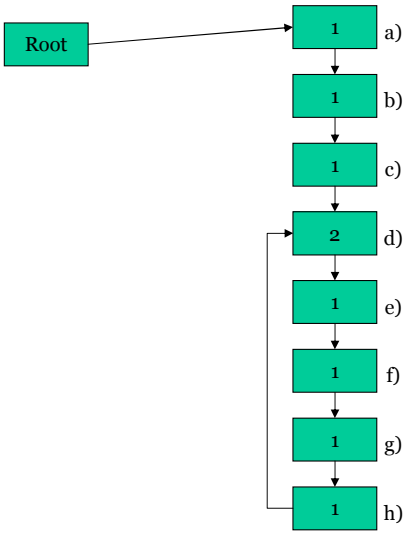

```
class Garbage {
   public static void main(String[] args){
            Person p1 = new Person("Hans");
            Person p2 = new Person("Olga");
            p1 = new Person("Lars");
            // what happens to the
            //"Hans" person object.
   }
```

Reference counting

The key problem with reference counting is that the approach can not handle cyclic structures.

There is no known good solution to this.

But it is useful for objects that themselves do not have references to other objects, like strings, pictures, sound. It can also be used if we are absolutely sure that no cycles can occur in the data.



When we remove the reference from the root to the top object (a), its reference count becomes zero. When the a-object is removed, the reference to the b object is removed, and b's reference count becomes zero. When the c-object is removed, d's reference count is decreased to 1, not to zero, because h refers to d. Hence, the cycle of d,e,f,g,h remains, and cannot be reclaimed.

Memory fragmentation

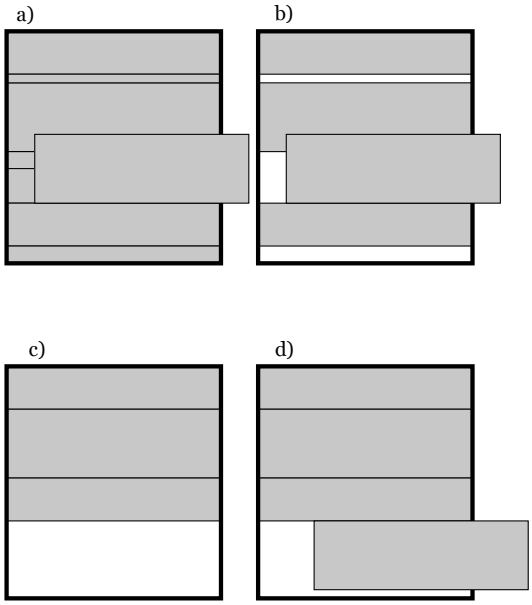
What is the problem

a) We want a new large object, but memory is filled, and we must collect garbage.

b) Garbage has been collected, which leaves us empty space for new objects. But there is no space big enough for the new large object

c) We compact the memory, which means that we move all objects close together to make sure all vacant space is all in one place.

d) Now there is room for our large new object.



Memory fragmentation is another problem which is a consequence of the reference counting principle. The copying garbage collector which we shall look at later, does not have this problem.

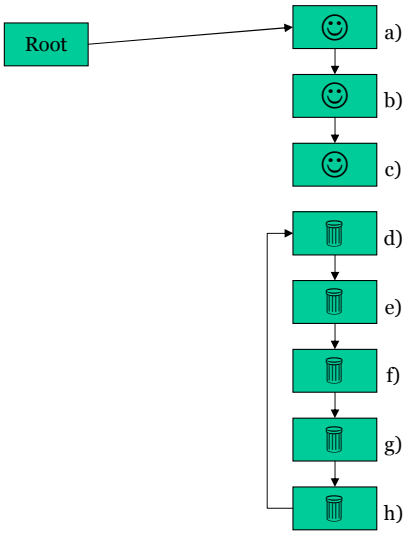
Mark and Sweep

Assume we removed the reference from c to d.

1) Mark all objects as garbage (🗑)
2) Mark all objects that can be reached from the root as live (☺).
3) Remove all objects marked as garbage (sweep)

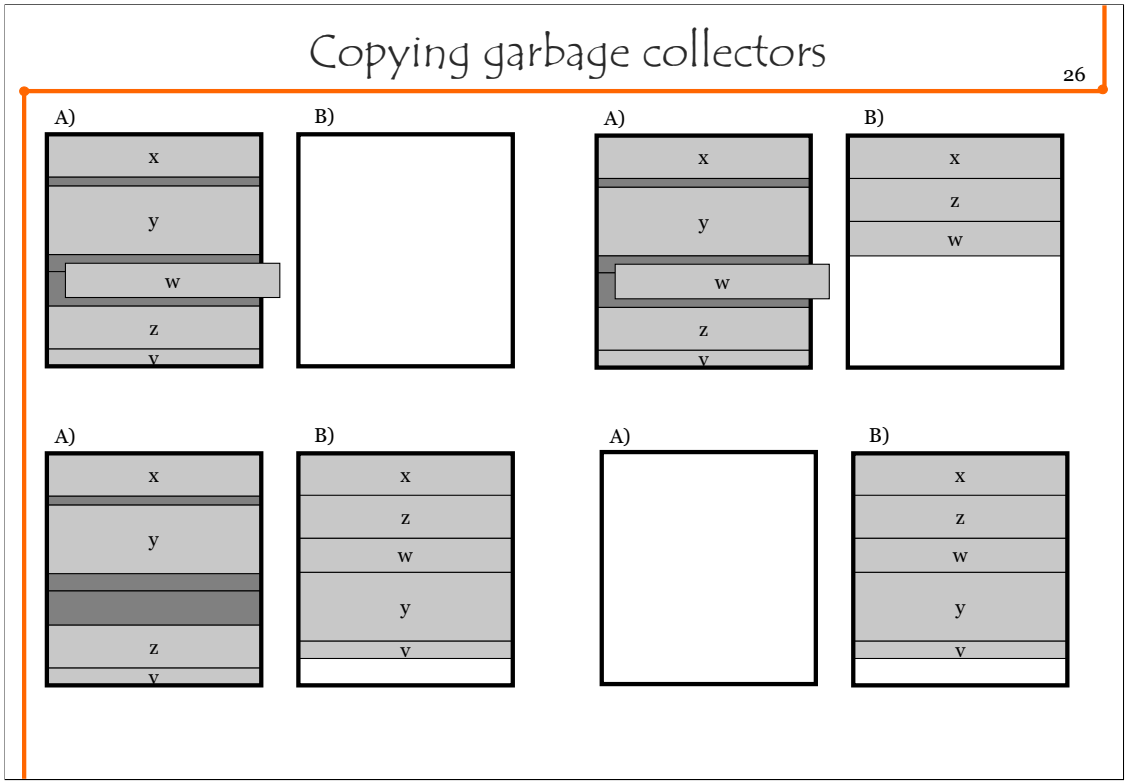
This collects also all cycles.

But it takes long time to do this, one must examine all objects twice, and nothing can happen while it goes on.



The mark and sweep principle has two phases, a mark phase and a sweep phase.

If objects are marked as garbage from the beginning, and in the sweep phase they are still marked as garbage, then they will be removed.



A copy based garbage collector has two rooms for allocating objects, A and B.

Only one of the rooms are used for new objects (at any given time).

Top left

Assume room A is filled and we want to add a new object w. The B-room is empty.

Top right

We start from the root (not shown), and copy objects to the new area (B). Objects which cannot be reached, are not copied.

At any point in time, we can stop copying, and just add new objects. After having copied x and z, we insert object w.

Bottom left

When all objects have been copied, the old room (A) is full of objects which are either garbage or has been copied.

Bottom right

The A room can be cleared.

The advantage of this strategy is that it automatically compacts data, and that one do not have to copy all objects at a time.

The disadvantage is that at any given time, the same object might sit in both rooms, thus a lot of space is wasted. Another disadvantage is that the more the room is full of live data, the more often copying will be done, but at the same time, the less space will be regained.

finalize method

One cannot change the way Java does garbage collection. But we have some possibilities for control:

1) We can call the garbage collector directly: System.gc();
2) We can do ”one last act” using the finalize() method.

The finalize method is good for objects that have references to external resources like files, databases, network ports.

The simple Garbage program allows us to follow what goes on:

Observe that:

1) Garbage collection occurs occasionally
2) Different amount of garbage is collected each time the GC is executed
3) It behaves different for each run.

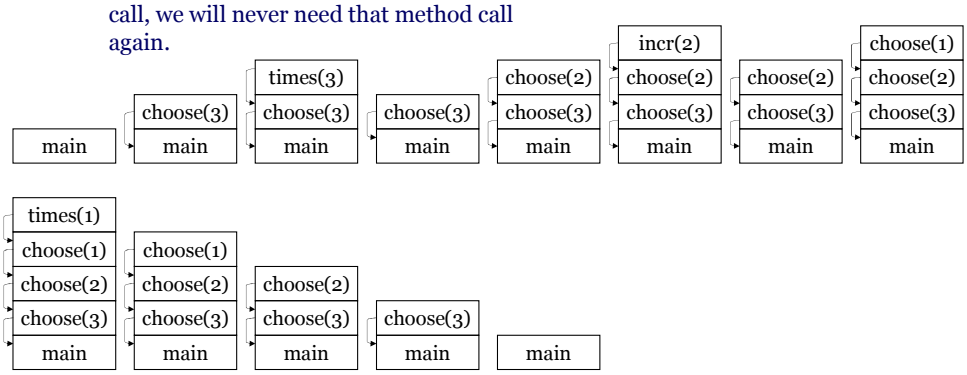
Conclusion: do not depend on when the GC is done.

One aspect which is useful to know of, but which is not part of the course is so-called weak-references. That is references that the garbage collector ignores. It is useful for some kinds of database systems, and for large web-server programs.

recycling and method invocations

- Each time a method is called, a method call instance is created.
  - How do we recycle these?
- This can be done very efficiently.
  1. We can never get a reference to a method call instance which can be stored in a variable.
  2. Once we have returned from a method call, we will never need that method call again.

- The idea is to place all method invocations on a stack.
  - When a method is called, a new method call instance is placed on the top
  - When we return from a method call instance, we remove the top element
  - the top element represents the currently executing method call



The figures illustrates how the method call instances are allocated on a stack. The figure starts to the left, with just the main method being called. From main, choose is called, with parameter 3. The small arrow indicates which method call instance to return to, when the method call from which the arrow starts returns. In the figures, we have ommitted the reference to where in the code we return to, we have ommitted the this reference. Nor is return value, nor local variables indicated on this figure.

What is worth remembering, is that the place in memory which was used for the method call instance for the call ”times(3)”, is shortly after reused for the all ”choose(2)”, and similarly, the memory used for the call ”incr(2)” is reused for the call ”choose(1)”.

Notice, in the figures above, all method call instances seem to have the same size. This is not necessarily the case in practice. But that is not a problem. Try to draw the same diagram, with choose method call instances being only half the size of times and incr. The stack idea still works.

In some languages, we can get a reference to a method invocation instance, and that reference can be stored in a variable. This has the effect that we cannot know if a method invocation can be used again, and therefore we cannot assume that the method call is really finished and can be thrown away.