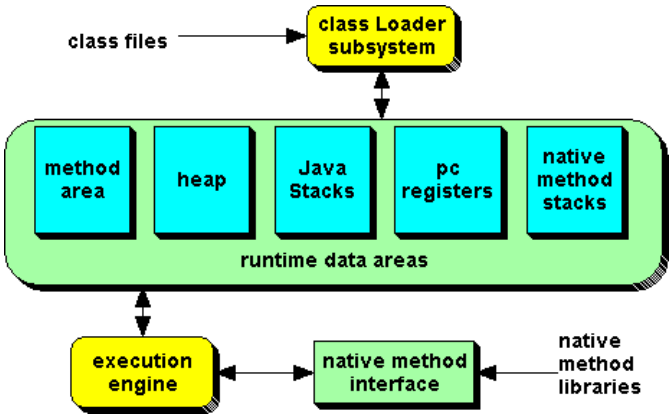
Java Virtual Machine (JVM)

JVM: una visione di insieme

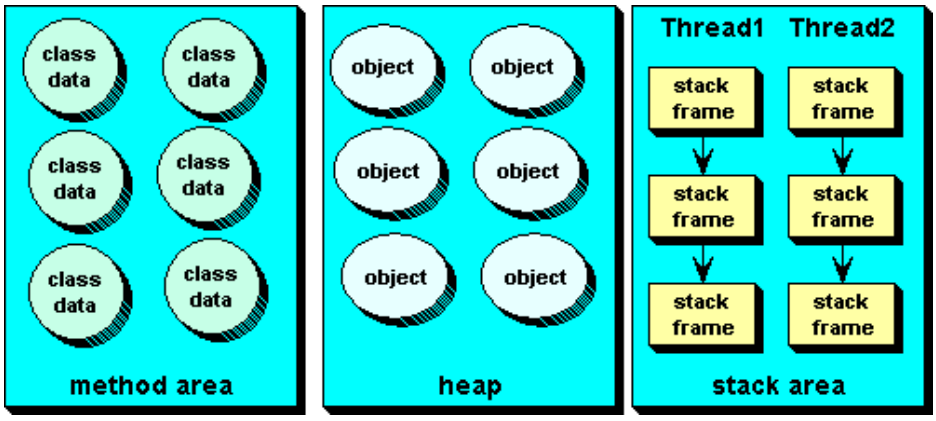
- JVM: interprete del bytecode



Struttura

- Ogni istanza della JVM e' caratterizzata
  - Area per memorizzare le tabelle dei metodi
  - Heap
  - Un insieme di stack (uno stack per ogni thread in esecuzione)
- Il class loading: quando l JVM carica un file di tipo class inserisce le informazioni nell'area delle tabelle.
- Gli oggetti sono allocati sullo heap (dinamicamente)
- Stack contiene i record di attivazione attivi

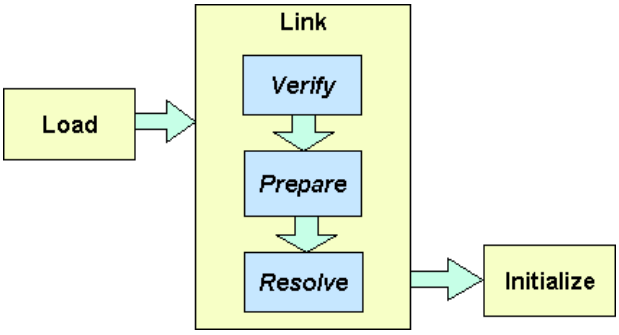
Struttura del run time



.

Il Class Loader

- Il class loader ha tre funzionalita' principale:
  - Loading, linking, initialization
- La fase di linking e' strutturata in:
  - verification, preparation, resolution



Class Loading

- Loading =
  - reading the class file for a type,
  - parsing it to get its information,
  - storing the information in the method area.

- Quali sono le informazioni memorizzate nella method area:
  - **The fully qualified name of the type**
  - **The fully qualified name of the type's direct superclass or if the type is an interface, a list of its direct super interfaces .**
  - **Whether the type is a class or an interface**
  - **The type's modifiers (public, abstract, final, etc)**
  - **Constant pool for the type: constants and symbolic references.**
  - **Field info : name, type and modifiers of variables (not constants)**
  - **Method info: name, return type, number & types of parameters, modifiers, bytecodes, size of stack frame and exception table.**

Class Loading

- Alla fine del processo di loading vengono create tutte le strutture per operare. In particolare vengono creati alcuni metodi della classe class java.lang.Clas che permettono la instrospection da programma

```
public String getName()
public Class getSupClass()
public boolean isInterface()
public Class[] getInterfaces()
public Method[] getMethods()
public Fields[] getFields()
public Constructor[] getConstructors()
```

Si deve prima invocare il metodo getClass() sull'istanza T per ottenere il riferimento all'istanzaT.

Class loading

```
import java.lang.reflect.Method; // Required!

//you must import your Circle class
public class TestClassClass{
  public static void main(String[] args) {
    String name = new String("Ahmed");
    Class nameClassInfo = name.getClass();
    System.out.println("Class name is : " + nameClassInfo.getName());
    System.out.println("Parent is : " + nameClassInfo.getSuperclass());
    Method[] methods = nameClassInfo.getMethods();
    System.out.println("\nMethods are: ");
    for(int i = 0; i < methods.length; i++)
      System.out.println(methods[i]);
  }
}
```

Cosa si vede

public methods are displayed ONLY!

Linking

- Linking: Verification, Preparation e Resolution
- Verification: Verifica se I binari hanno la struttura corretta
  - Manuali SUN: “The JVM has to make sure that a file it is asked to load was generated by a valid compiler and it is well formed”
  - Examples:
  - Every method is provided with a structurally correct signature
  - Every instruction obeys the type discipline of the Java language

Preparation

- Viene allocata la memoria per la classe (variabili static) e vengono definiti I valori iniziali.
- Sempre dai manuali “no java code is executed until initialization”.
- Valori di defauly:

Type	Initial Value
int	0
long	0L
short	(short) 0
char	'\u0000'
byte	(byte) 0
boolean	false
reference	null
float	0.0f
double	0.0d

Resolution

- Resolution: in questa fase vengono sostituiti I valori simbolici con I riferimenti attuali in memoria
- Esempio la classe seguente richiede: TestClassClass, String, System and Object.

```
public class TestClassClass{
   public static void main(String[] args){
      String name = new String(“Ahmed”);
      Class nameClassInfo = name.getClass();
      System.out.println("Parent is: “ + nameClassInfo.getSuperclass());
   }
}
```

- I nomi erano memorizzati nel constant pool di TestClassClass.
- La risoluzione rimpiazza I nomi con I riferimenti effettivi a run-time.

Class Initialization

- Inizializzare la classe come ha voluto il programmatore.

```
class Example1 {
      static double rate = 3.5;
      static int size = 3*(int)(Math.random()*5);
      ...
}
```

- Due fasi:
  - Inizializzazione delle superclassi (se esisono)
  - Eseguire l'inizializzazione
- Quale e' la prima classe inizializzata?
- static final variables (ostanti) sono compilate direttamente

```
class Example2 {
      static final int angle = 35;
      static final int length = angle * 2;
      ...
}
```

Initialization

- A questo punto una classe e pronta per l'uso.
- Quando una classe viene creata una sua istanza e' allocata sullo heap
- L'operazione viene fatta per le super class e tutte le classes della gerarchia.
- Le variabili di istanza vengono inizializzate ai valori di default.
- Viene invocato il metodo costruttore.
- Viene restituito il cammino di accesso all'oggettoi creato sullo heap.

ESEMPI

Object Aliasing example

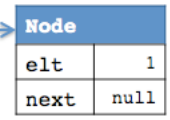
```
public class Node {
  public int elt;
  public Node next;
  public Node(int e0, Node n0) {
    elt  = e0;
    next = n0;
  }
}

public static void main(String[] args) {
    Node n1 = new Node(1,null);
    Node n2 = new Node(2,n1);
    Node n3 = n2;
    n3.next.next = n2;
    Node n4 = new Node(4,n1.next);
    n2.next.elt = 17;
}
```

Workspace Stack Heap

```
Node n1 = new Node(1,null);
Node n2 = new Node(2,n1);
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 17;
```

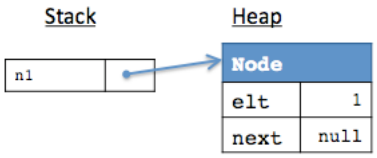
Workspace Stack Heap

```
Node n1 =
Node n2 = new Node(2,n1);
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 17;
```



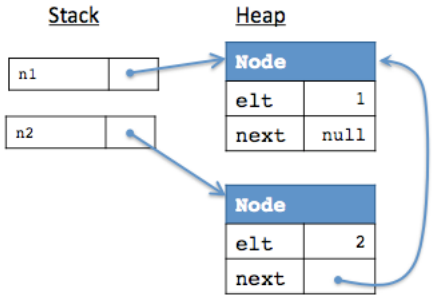
Workspace

```
Node n2 = new Node(2,n1);
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 17;
```



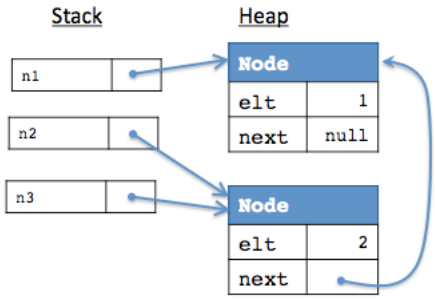
Workspace

```
Node n3 = n2;
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 17;
```



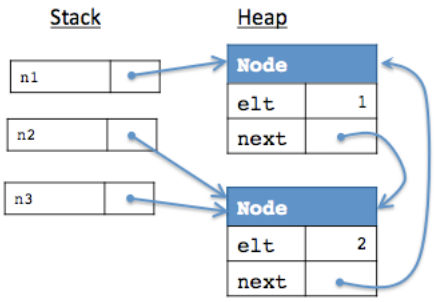
Workspace

```
n3.next.next = n2;
Node n4 = new Node(4,n1.next);
n2.next.elt = 17;
```



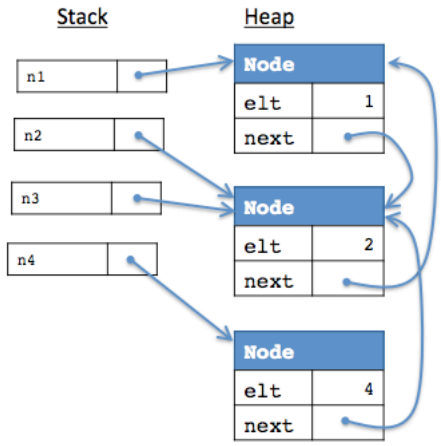
Workspace

```
Node n4 = new Node(4,n1.next);
n2.next.elt = 17;
```



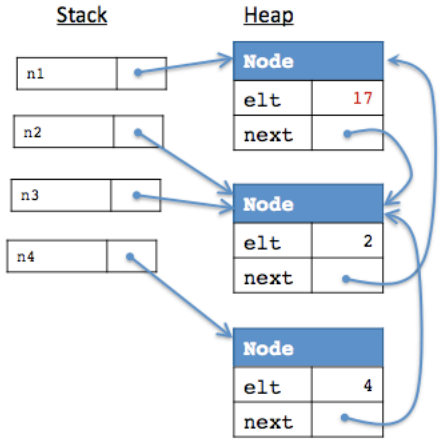
Workspace

```
n2.next.elt = 17;
```



Workspace

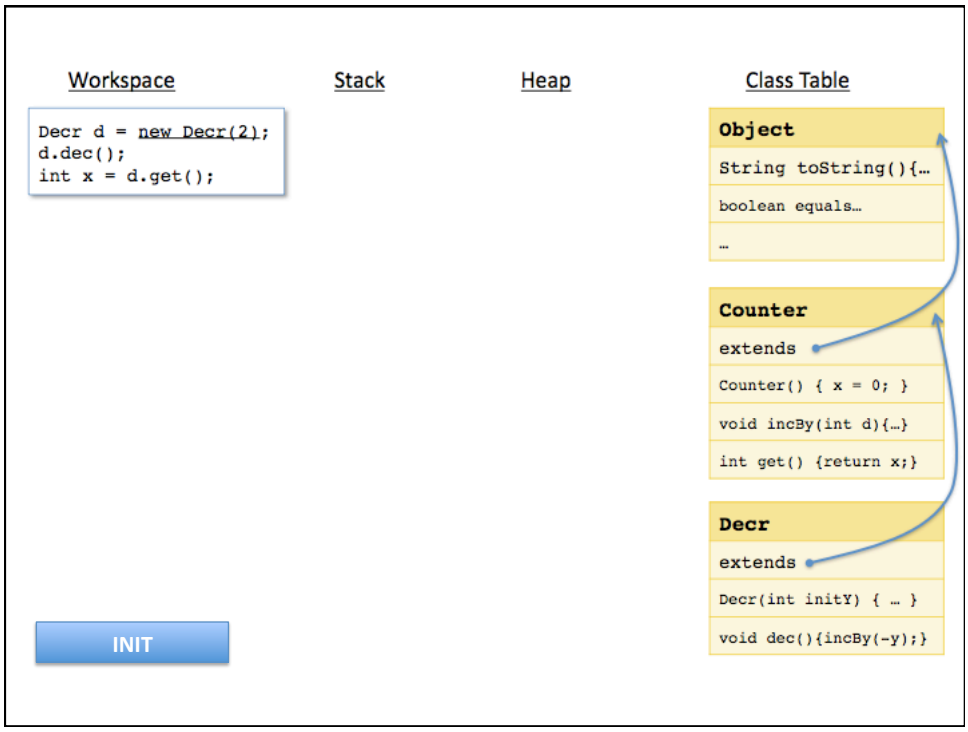
```
;
```

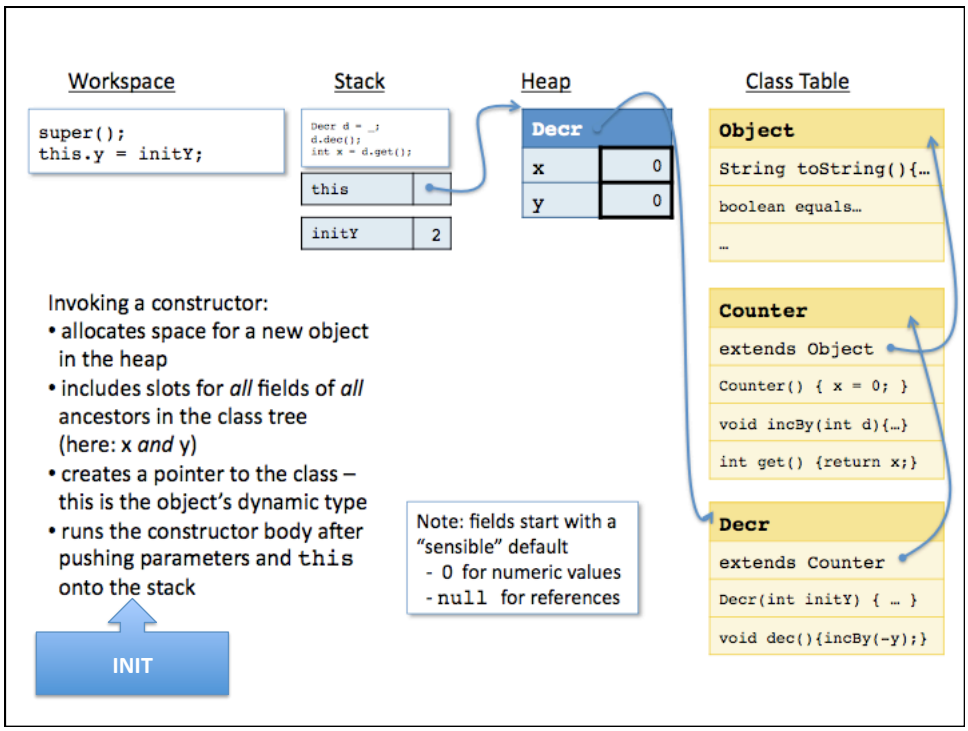


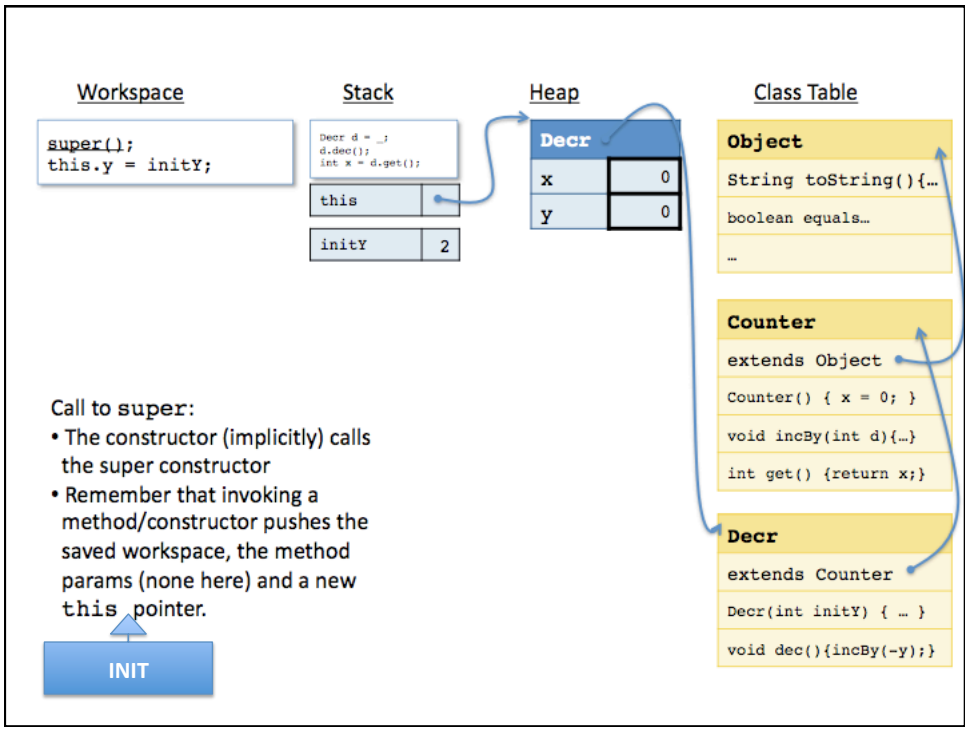
Oggetti e Heap

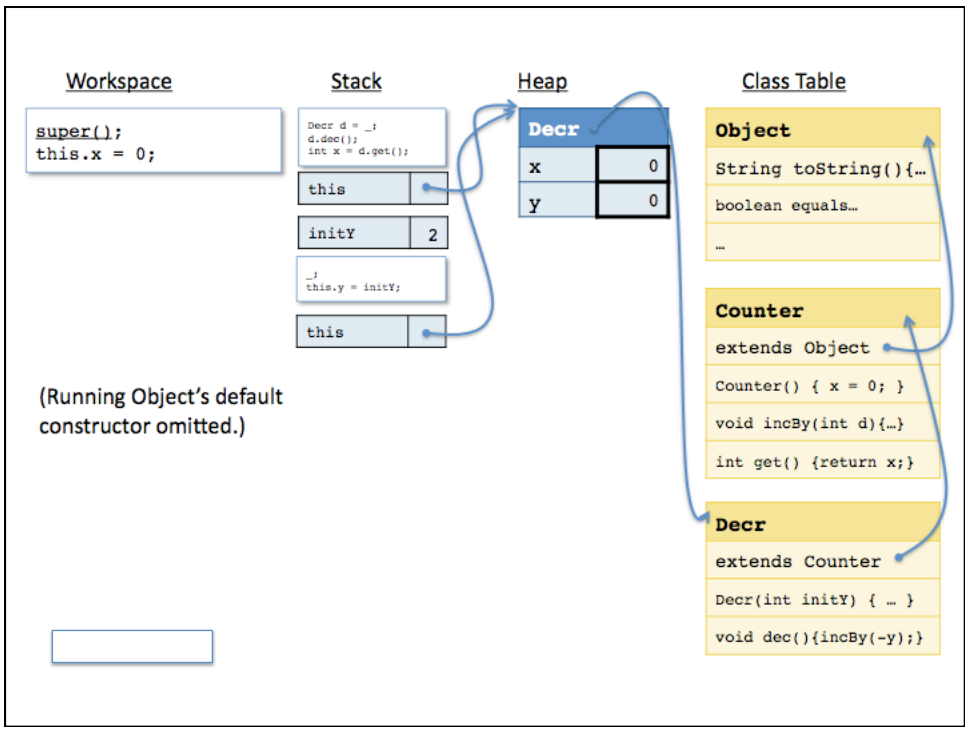
- Ogni oggetto allocato sullo heap contiene un puntatore alla tabella dei metodi della classe
- L'invocazione del metodo o.m() utilizza il puntatore alla tabella dei metodi per effettuare il "dispatch". Questo potrebbe comportare una operazione di ricerca nella gerarchia
- Metodi hanno un campo this che punta all'oggetto che ha operato la chiamata del metodo.

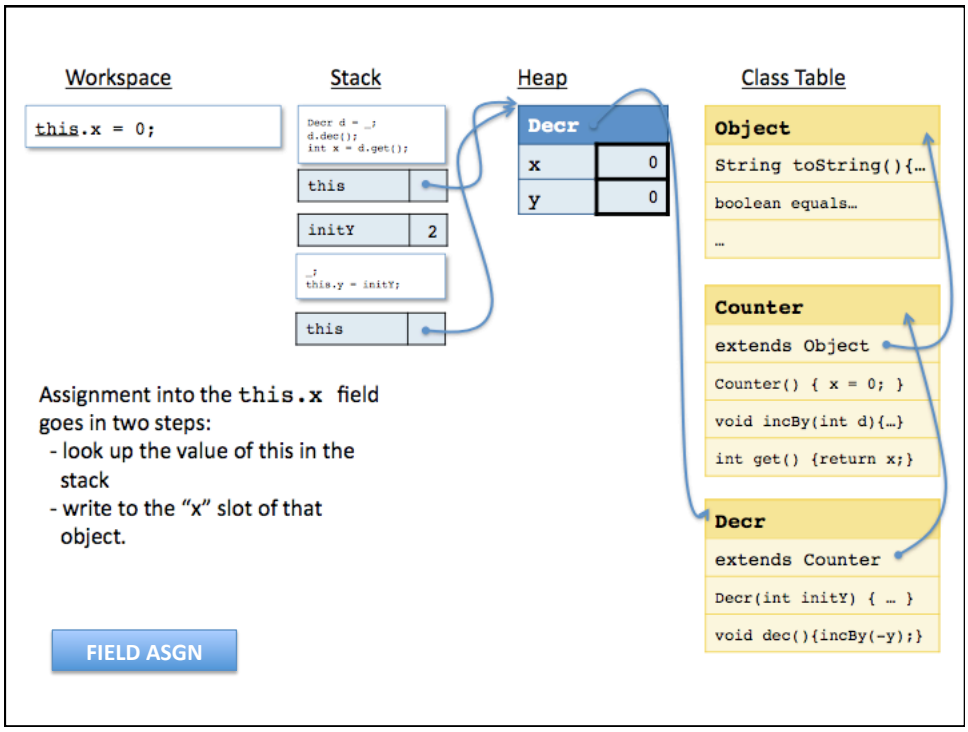
```
public class Counter extends Object {
  private int x;
  public Counter () { super(); this.x = 0; }
  public void incBy(int d) { this.x = this.x + d; }
  public int get() { return this.x; }
}

public class Decr extends Counter {
  private int y;
  public Decr (int initY) { super(); this.y = initY; }
  public void dec() { this.incBy(-this.y); }
}

// … somewhere in main:
Decr d = new Decr(2);
d.dec();
int x = d.get();
```

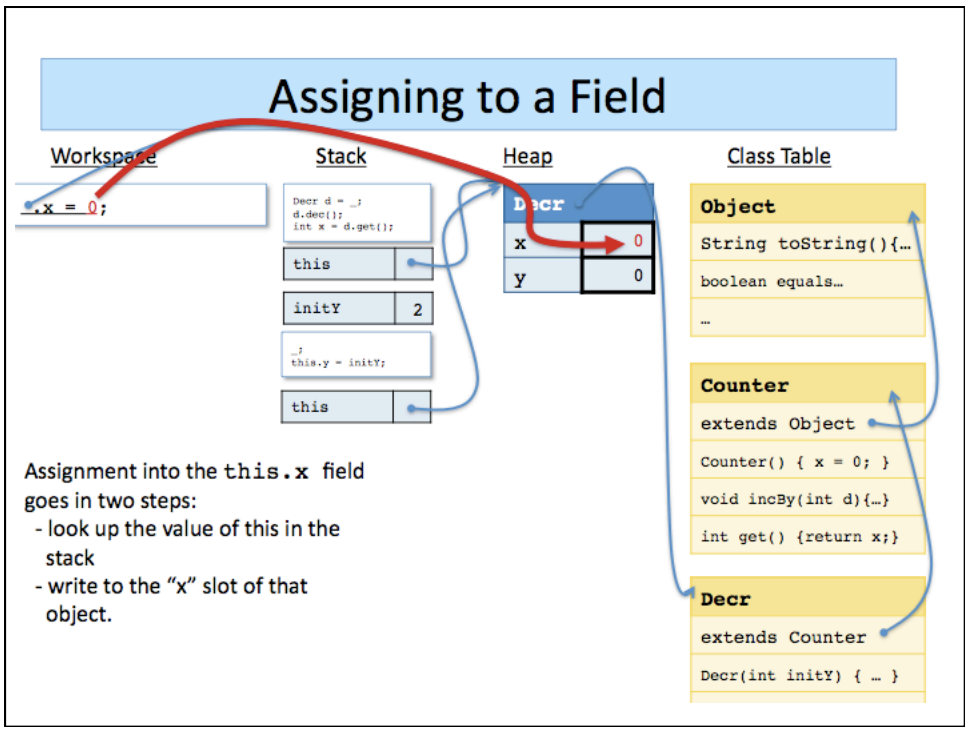

Workspace
Stack
Heap
Class Table
Decr d = new Decr(2);
d.dec();
int x = d.get();
Object
String toString(){…
boolean equals…
…
Counter
extends
Counter() { x = 0; }
void incBy(int d){…}
int get() {return x;}
Decr
extends
Decr(int initY) { … }
void dec(){incBy(-y);}
INIT

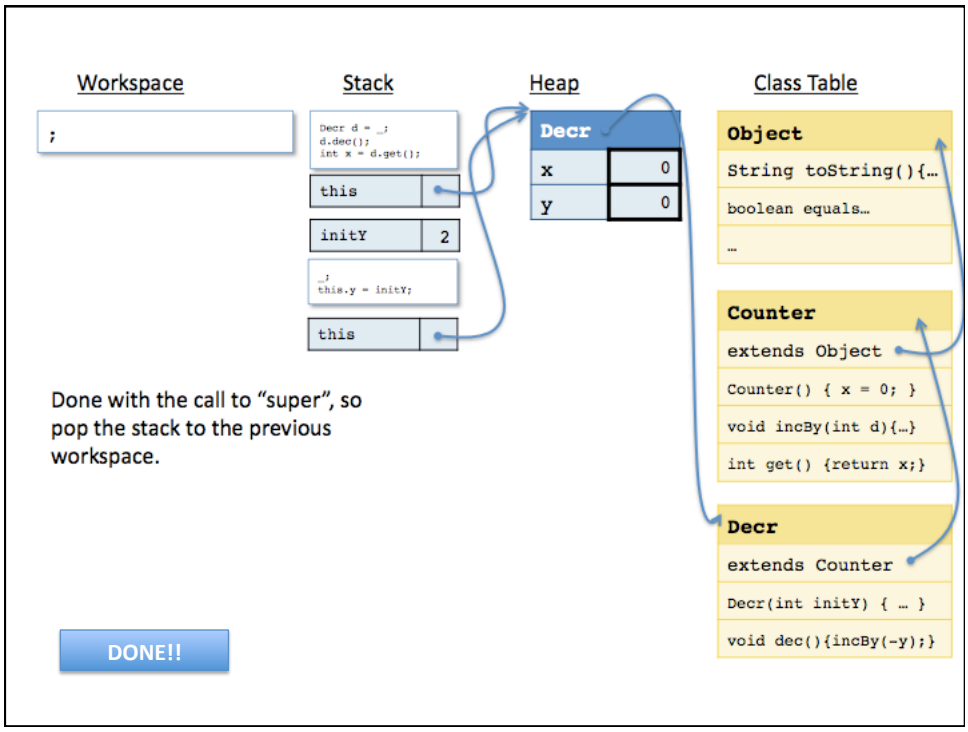

Workspace
Stack
Heap
Class Table
super();
this.y = initY;
this
initY
2
Decr
x
0
y
0
Object
String toString(){…
boolean equals…
…
Counter
extends Object
Counter() { x = 0; }
void incBy(int d){…}
int get() {return x;}
Decr
extends Counter
Decr(int initY) { … }
void dec(){incBy(-y);}
Invoking a constructor:
• allocates space for a new object in the heap
• includes slots for all fields of all ancestors in the class tree (here: x and y)
• creates a pointer to the class – this is the object's dynamic type
• runs the constructor body after pushing parameters and this onto the stack
Note: fields start with a "sensible" default
- 0 for numeric values
- null for references
INIT

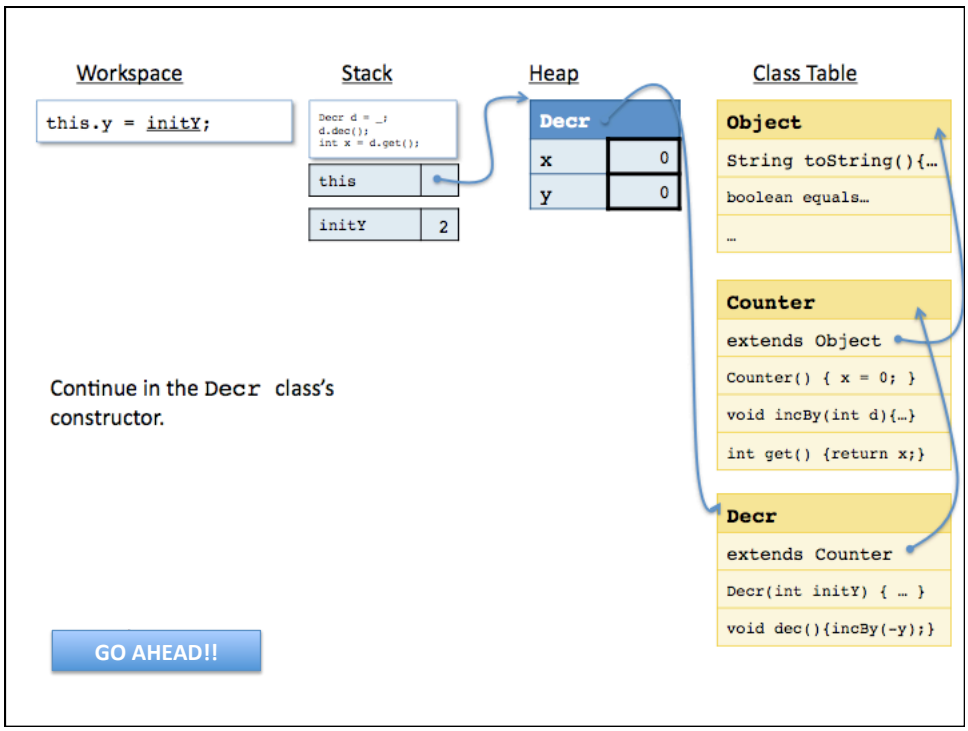

Workspace
Stack
Heap
Class Table
super();
this.y = initY;
Decr d = _;
d.dec();
int x = d.get();
this
initY
2
Decr
x
0
y
0
Object
String toString(){…
boolean equals…
…
Counter
extends Object
Counter() { x = 0; }
void incBy(int d){…}
int get() {return x;}
Decr
extends Counter
Decr(int initY) { … }
void dec(){incBy(-y);}
Call to super:
• The constructor (implicitly) calls the super constructor
• Remember that invoking a method/constructor pushes the saved workspace, the method params (none here) and a new this pointer.
INIT

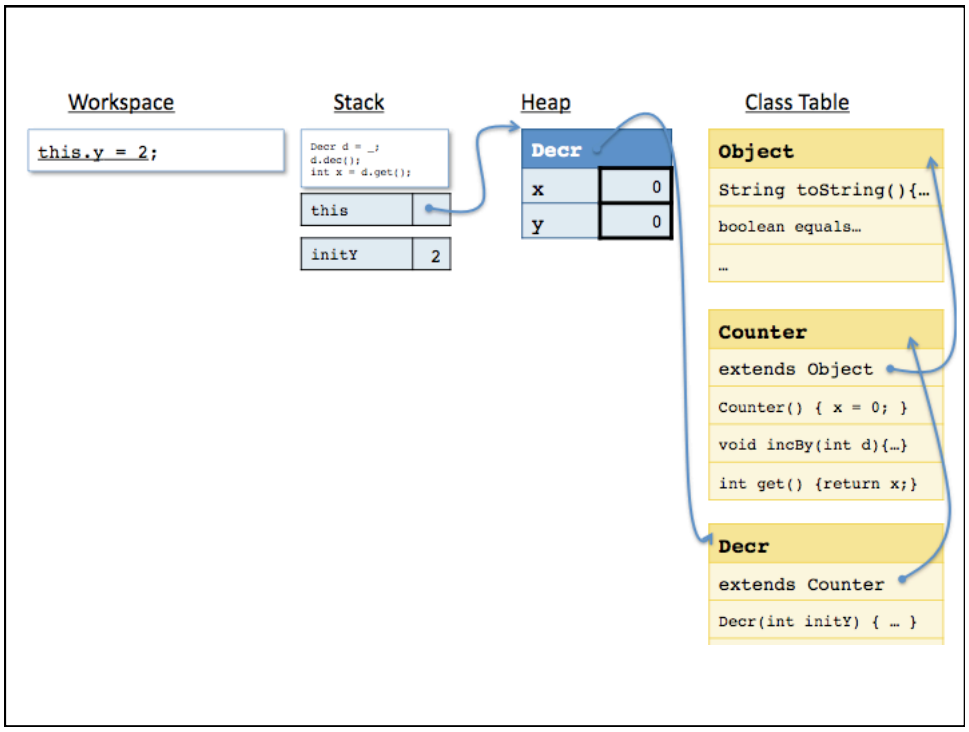

Workspace
Stack
Heap
Class Table
super();
this.x = 0;
Decr d = _;
d.dec();
int x = d.get();
this
initY
2
_;
this.y = initY;
this
Decr
x
0
y
0
Object
String toString(){…
boolean equals…
…
Counter
extends Object
Counter() { x = 0; }
void incBy(int d){…}
int get() {return x;}
Decr
extends Counter
Decr(int initY) { … }
void dec(){incBy(-y);}
(Running Object's default constructor omitted.)

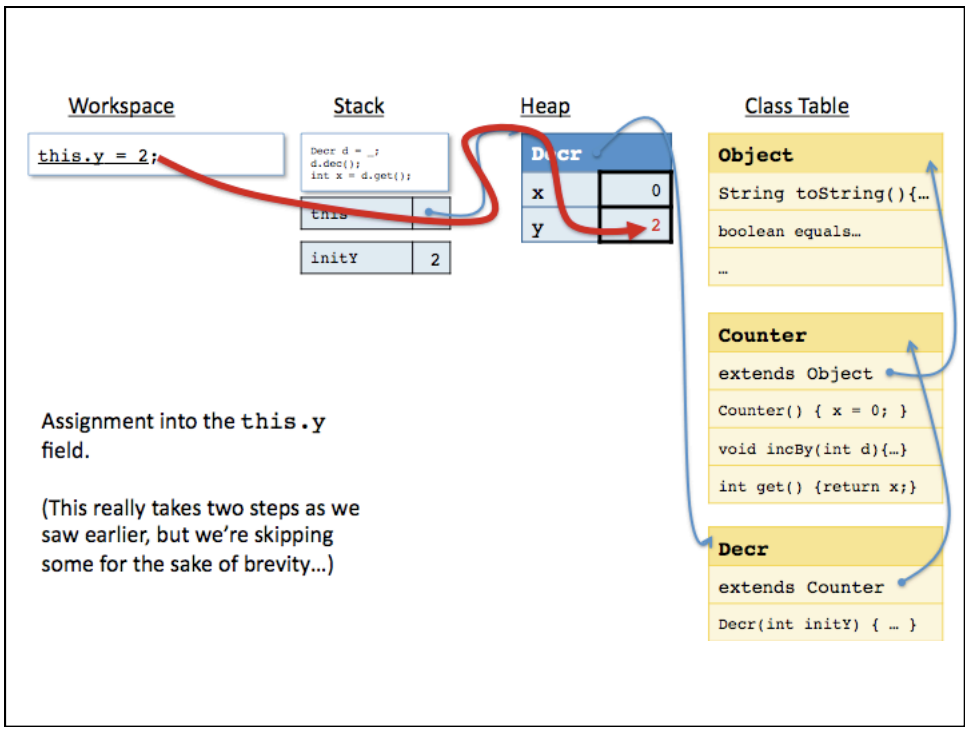

Workspace
Stack
Heap
Class Table
this.x = 0;
Decr d = _;
d.dec();
int x = d.get();
this
initY
2
_;
this.y = initY;
this
Decr
x
0
y
0
Object
String toString(){…
boolean equals…
…
Counter
extends Object
Counter() { x = 0; }
void incBy(int d){…}
int get() {return x;}
Decr
extends Counter
Decr(int initY) { … }
void dec(){incBy(-y);}
Assignment into the this.x field goes in two steps:
- look up the value of this in the stack
- write to the "x" slot of that object.
FIELD ASGN


Assigning to a Field
Workspace
Stack
Heap
Class Table
.x = 0;
Decr d = _;
d.dec();
int x = d.get();
this
initY
2
_;
this.y = initY;
this
x
0
y
0
Object
String toString(){…
boolean equals…
…
Counter
extends Object
Counter() { x = 0; }
void incBy(int d){…}
int get() {return x;}
Decr
extends Counter
Decr(int initY) { … }
Assignment into the this.x field goes in two steps:
- look up the value of this in the stack
- write to the "x" slot of that object.


Workspace
Stack
Heap
Class Table
;
Decr d = _;
d.dec();
int x = d.get();
this
initY
2
_;
this.y = initY;
this
Decr
x
0
y
0
Object
String toString(){…
boolean equals…
…
Counter
extends Object
Counter() { x = 0; }
void incBy(int d){…}
int get() {return x;}
Decr
extends Counter
Decr(int initY) { … }
void dec(){incBy(-y);}
Done with the call to "super", so pop the stack to the previous workspace.
DONE!!

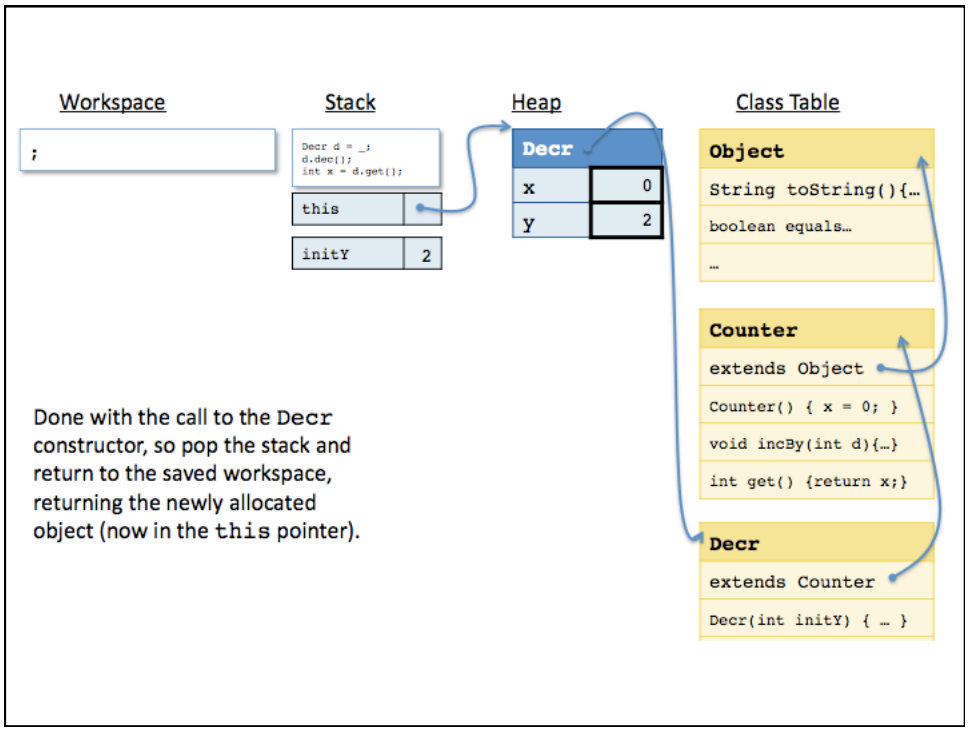

Workspace
Stack
Heap
Class Table
this.y = initY;
Decr d = _;
d.dec();
int x = d.get();
this
initY
2
Decr
x
0
y
0
Object
String toString(){…
boolean equals…
…
Counter
extends Object
Counter() { x = 0; }
void incBy(int d){…}
int get() {return x;}
Decr
extends Counter
Decr(int initY) { … }
void dec(){incBy(-y);}
Continue in the Decr class's constructor.
GO AHEAD!!

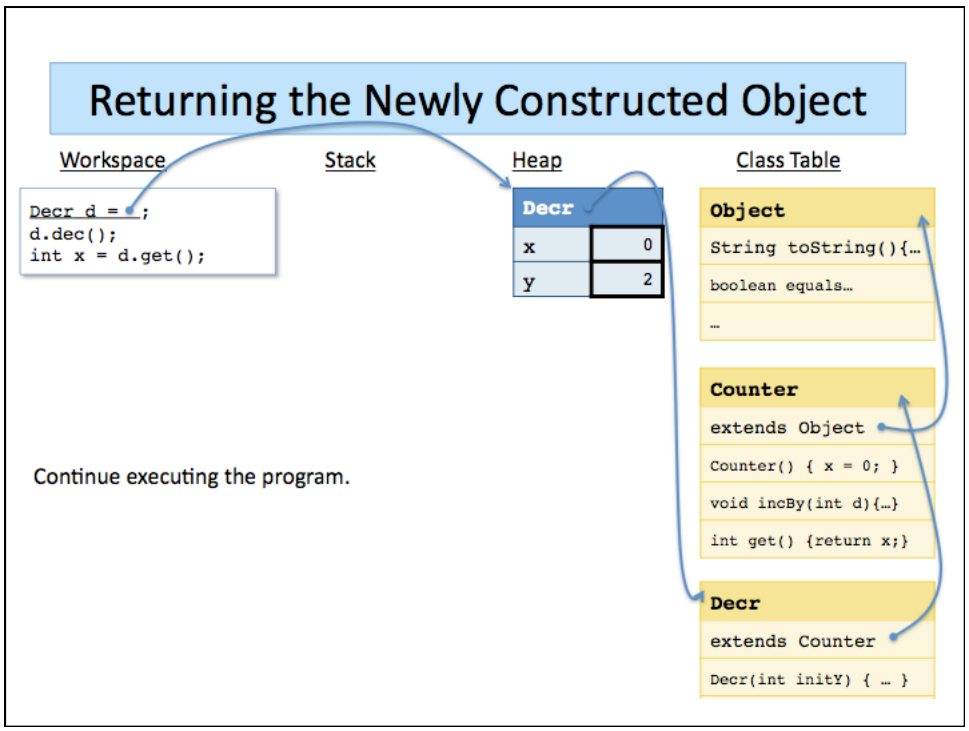


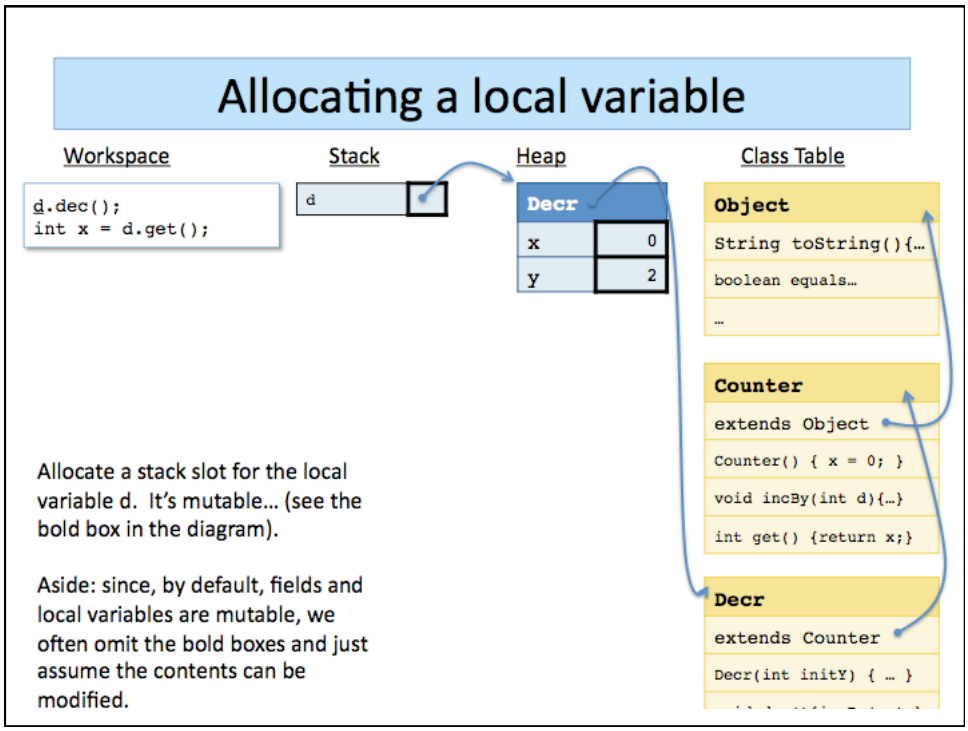


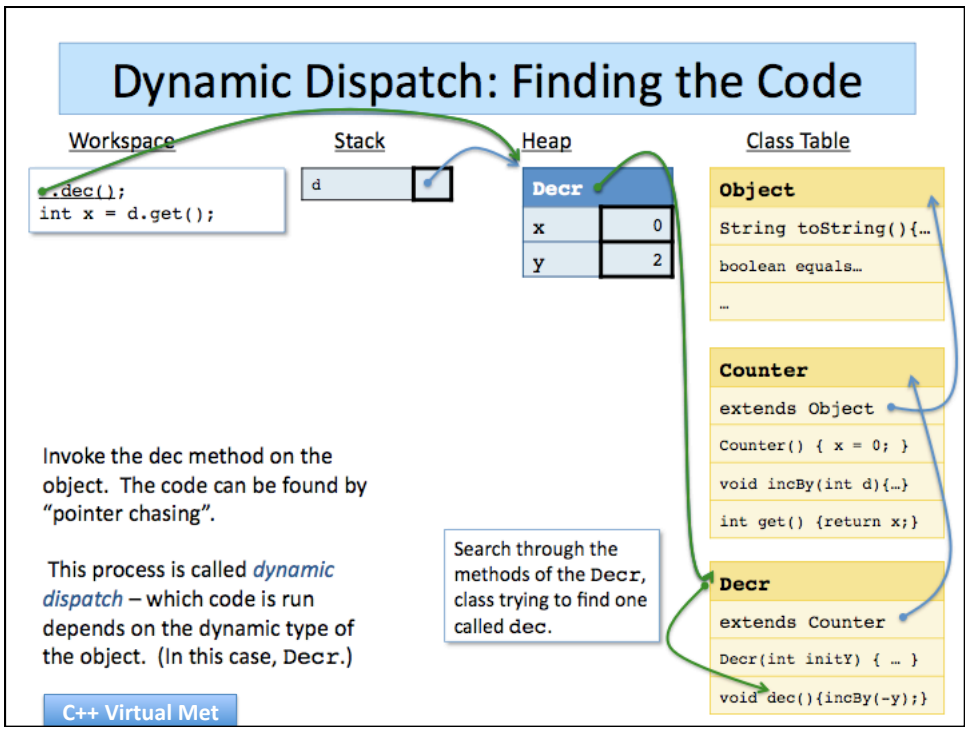
Assignment into the `this.y` field.

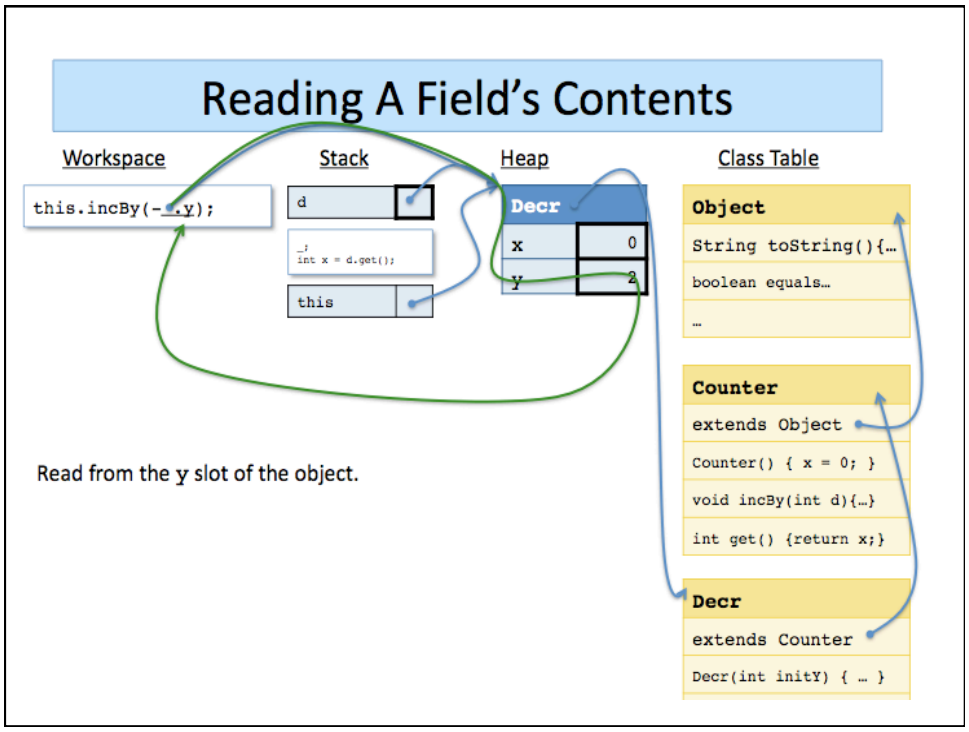
(This really takes two steps as we saw earlier, but we're skipping some for the sake of brevity…)

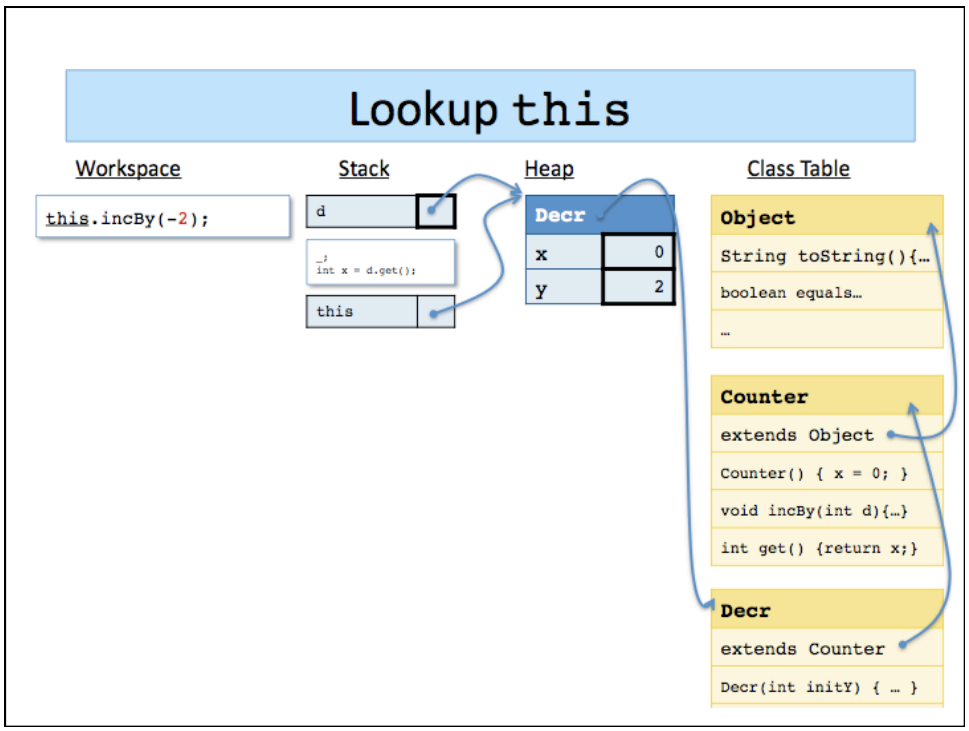

Workspace
Stack
Heap
Class Table
;
Decr d = _;
d.dec();
int x = d.get();
this
initY
2
Decr
x
0
y
2
Object
String toString(){…
boolean equals…
…
Counter
extends Object
Counter() { x = 0; }
void incBy(int d){…}
int get() {return x;}
Decr
extends Counter
Decr(int initY) { … }
Done with the call to the Decr constructor, so pop the stack and return to the saved workspace, returning the newly allocated object (now in the this pointer).

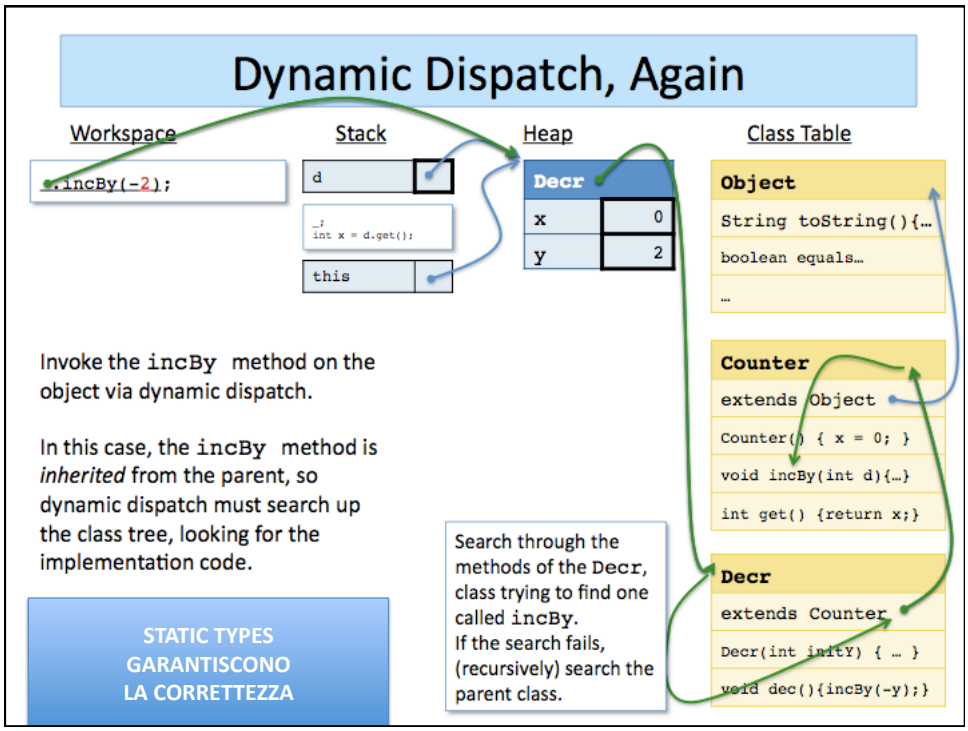

Returning the Newly Constructed Object
Workspace
Stack
Heap
Class Table
Decr d = ;
d.dec();
int x = d.get();
Decr
x
0
y
2
Object
String toString(){…
boolean equals…
…
Counter
extends Object
Counter() { x = 0; }
void incBy(int d){…}
int get() {return x;}
Decr
extends Counter
Decr(int initY) { … }
Continue executing the program.

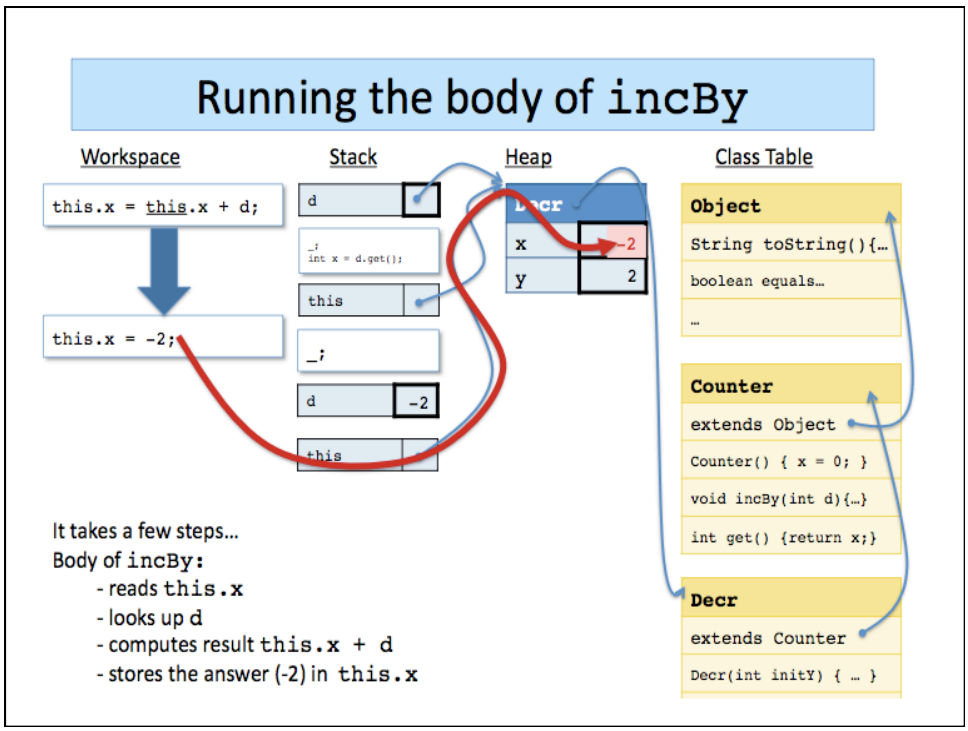

Allocating a local variable
Workspace
Stack
Heap
Class Table
d.dec();
int x = d.get();
d
Decr
x 0
y 2
Object
String toString(){…
boolean equals…
…
Counter
extends Object
Counter() { x = 0; }
void incBy(int d){…}
int get() {return x;}
Decr
extends Counter
Decr(int initY) { … }
Allocate a stack slot for the local variable d. It's mutable… (see the bold box in the diagram).
Aside: since, by default, fields and local variables are mutable, we often omit the bold boxes and just assume the contents can be modified.

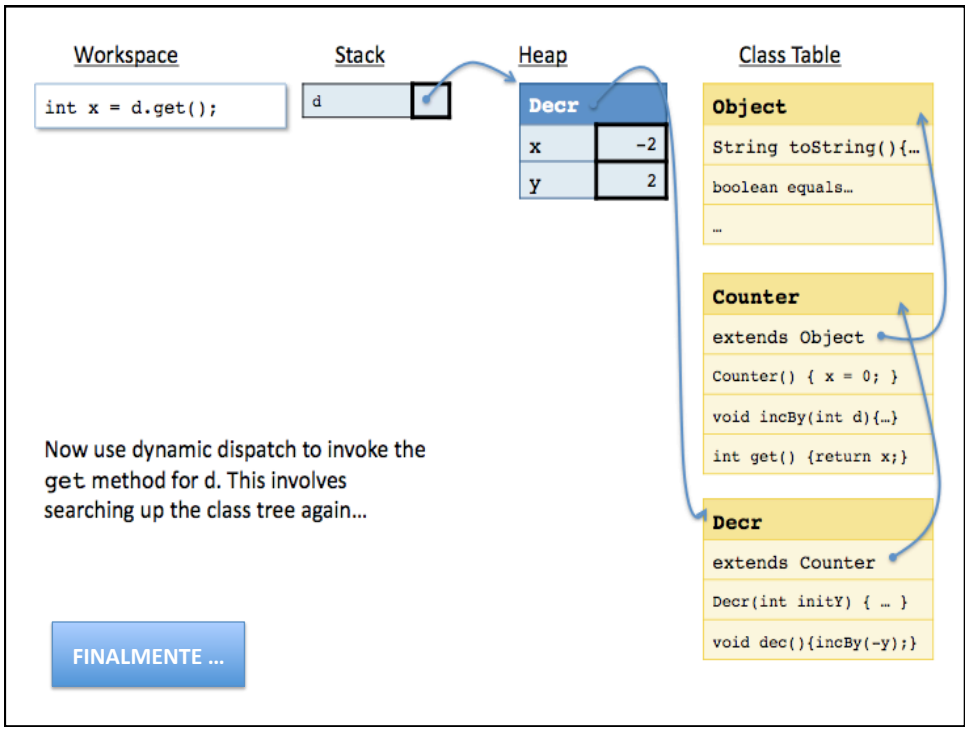

Dynamic Dispatch: Finding the Code
Workspace
Stack
Heap
Class Table
.dec();
int x = d.get();
d
Decr
x 0
y 2
Object
String toString(){…
boolean equals…
…
Counter
extends Object
Counter() { x = 0; }
void incBy(int d){…}
int get() {return x;}
Decr
extends Counter
Decr(int initY) { … }
void dec(){incBy(-y);}
Invoke the dec method on the object. The code can be found by "pointer chasing".
This process is called dynamic dispatch – which code is run depends on the dynamic type of the object. (In this case, Decr.)
Search through the methods of the Decr, class trying to find one called dec.
C++ Virtual Met

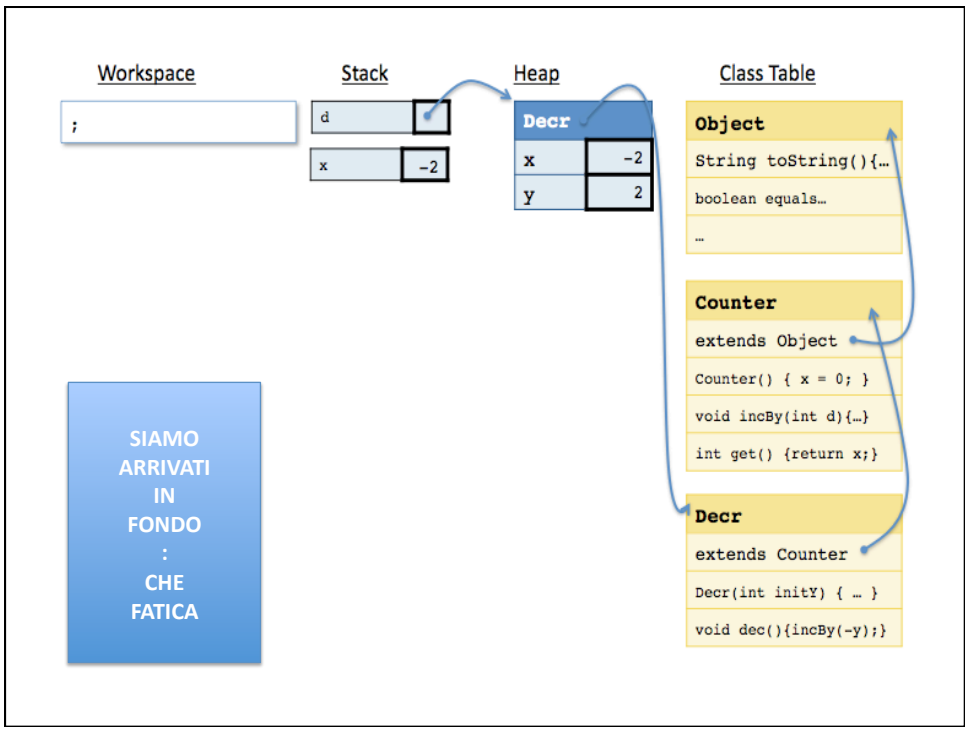

Reading A Field's Contents
Workspace
Stack
Heap
Class Table
this.incBy(-2.y);
d
_;
int x = d.get();
this
Decr
x
0
y
2
Object
String toString(){…
boolean equals…
…
Counter
extends Object
Counter() { x = 0; }
void incBy(int d){…}
int get() {return x;}
Decr
extends Counter
Decr(int initY) { … }
Read from the y slot of the object.


Lookup this
Workspace
Stack
Heap
Class Table
this.incBy(-2);
d
_;
int x = d.get();
this
Decr
x
0
y
2
Object
String toString(){…
boolean equals…
…
Counter
extends Object
Counter() { x = 0; }
void incBy(int d){…}
int get() {return x;}
Decr
extends Counter
Decr(int initY) { … }


Dynamic Dispatch, Again
Workspace
Stack
Heap
Class Table
.incBy(-2);
d
_;
int x = d.get();
this
Decr
x 0
y 2
Object
String toString(){…
boolean equals…
…
Counter
extends Object
Counter() { x = 0; }
void incBy(int d){…}
int get() {return x;}
Decr
extends Counter
Decr(int initY) { … }
void dec(){incBy(-y);}
Invoke the incBy method on the object via dynamic dispatch.
In this case, the incBy method is inherited from the parent, so dynamic dispatch must search up the class tree, looking for the implementation code.
STATIC TYPES GARANTISCONO LA CORRETTEZZA
Search through the methods of the Decr, class trying to find one called incBy. If the search fails, (recursively) search the parent class.


Running the body of incBy
Workspace
Stack
Heap
Class Table
this.x = this.x + d;
this.x = -2;
d
_;
int x = d.get();
this
_;
d -2
this
x -2
y 2
Object
String toString(){…
boolean equals…
…
Counter
extends Object
Counter() { x = 0; }
void incBy(int d){…}
int get() {return x;}
Decr
extends Counter
Decr(int initY) { … }
It takes a few steps…
Body of incBy:
- reads this.x
- looks up d
- computes result this.x + d
- stores the answer (-2) in this.x


Workspace
Stack
Heap
Class Table
int x = d.get();
d
Decr
x
-2
y
2
Object
String toString(){…
boolean equals…
…
Counter
extends Object
Counter() { x = 0; }
void incBy(int d){…}
int get() {return x;}
Decr
extends Counter
Decr(int initY) { … }
void dec(){incBy(-y);}
Now use dynamic dispatch to invoke the get method for d. This involves searching up the class tree again…
FINALMENTE …


Workspace
Stack
Heap
Class Table
;
d
x
-2
Decr
x
-2
y
2
Object
String toString(){…
boolean equals…
…
Counter
extends Object
Counter() { x = 0; }
void incBy(int d){…}
int get() {return x;}
Decr
extends Counter
Decr(int initY) { … }
void dec(){incBy(-y);}
SIAMO ARRIVATI IN FONDO : CHE FATICA

STATIC

- Classes in Java can also act as *containers* for code and data.
- The modifier `static` means that the field or method is associated with the class and *not* instances of the class.

You can do a static assignment to initialize a static field.

```
public class C {
  public static int x = 23;
  public static int someMethod(int y) { return C.x + y; }
  public static void main(String args[]) {
    …
  }
}

C.x = C.x + 1;
C.someMethod(17);
```

Access to the static member uses the class name `C.x` or `C.foo()`

- The class table entry for C has a field slot for `x`.
- Updates to `C.x` modify the contents of this slot: `C.x = 17;`

C
extends Object
static x 23
static int someMethod(int y) { return x + y; }
static void main(String args[]) {…}

- A static field is a *global* variable
  - There is only one heap location for it (in the class table)
  - Modifications to such a field are globally visible (if the field is public)
  - Generally not a good idea!

OVERRIDING

Dangers of Overriding

```
public class C {
  public void printTest() {
    if (onDate("April 1")) {
      System.out.println("as scheduled");
    } else { System.out.prinln("postponed"); }
  }
  public boolean onDate(String s) {
    return exam2.date().equals(s);
  }
}

public class D extends C {
  public boolean onDate(String s) {
    return final.date().equals(s);
  }
}

C c = new D();
c.printTest();   // what gets printed?
```

The C class might be in another package, or a library...

Whoever wrote D might not be aware of the implications of changing onDate.

Overriding the method can cause the behavior of `printTest` to change!

- Overriding can break invariants/abstractions relied upon by the superclass.