

Concepts of Object-Oriented Programming

Peter Müller

Chair of Programming Methodology

Autumn Semester 2017

ETH zürich

Meeting the Requirements

Cooperating Program Parts with Well-Defined Interfaces

- Objects (data + code)
- Interfaces
- Encapsulation

Classification and Specialization

- Classification, subtyping
- Polymorphism
- Substitution principle

Highly Dynamic Execution Model

- Active objects
- Message passing

Correctness

- Interfaces
- Encapsulation
- Simple, powerful concepts

My Billion Dollar Mistake



“I call it my billion-dollar mistake. It was the invention of the null reference in 1965. [...]

This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years. [...]

More recent programming languages like Spec# have introduced declarations for non-null references. This is the solution, which I rejected in 1965.” [Hoare, 2009]

7. Initialization

7.1 Simple Non-Null Types

7.2 Object Initialization

7.3 Initialization of Global Data

Main Usages of Null-References

```
class Map {  
    Map next;  
    Object key;  
    Object value;  
  
    Map( Object k, Object v ) {  
        key = k;  
        value = v;  
    }  
}
```

null terminates
recursion

All fields are
initialized to **null**

```
void add( Object k, Object v ) {  
    if( key.equals( k ) )  
        value = v;  
    else if( next == null )  
        next = new Map( k, v );  
    else next.add( k, v );  
}  
  
Object get( Object k ) {  
    if( key.equals( k ) ) return value;  
    if( next == null ) return null;  
    return next.get( k );  
}  
}
```

null indicates
absence of an
object

Main Usages of Null-References (cont'd)

```
class Map {  
    Map next;  
    Object key;  
    Object value;
```

Most variables
hold non-null
values

```
    Map( Object k, Object v ) {  
        key = k;  
        value = v;  
    }  
}
```

```
void add( Object k, Object v ) {  
    if( key.equals( k ) )  
        value = v;  
    else if( next == null )  
        next = new Map( k, v );  
    else next.add( k, v );  
}
```

```
Object get( Object k ) {  
    if( key.equals( k ) ) return value;  
    if( next == null ) return null;  
    return next.get( k );  
}  
}
```


Non-Null Types

- **Non-null type $T!$** consists of references to T -objects
- **Possibly-null type $T?$** consists of references to T -objects **plus null**
 - Corresponds to T in most languages
- A language designer would choose a default

```
class Map {  
    Map? next;  
    Object! key;  
    Object! value;  
  
    Map( Object! k, Object! v ) {  
        key = k;  
        value = v;  
    }  
  
    ...  
}
```


Type Safety

- (Simplified) type invariant:
If the static type of an expression *e* is a non-null type then *e*'s value at run time is different from **null**

- Goal: prevent null-dereferencing statically
 - Require non-null types for the receiver of each field access, array access, method call
 - Analogous to preventing “message not understood” errors with classical type systems

Subtyping and Casts

- The values of a type $T!$ are a proper subset of $T?$

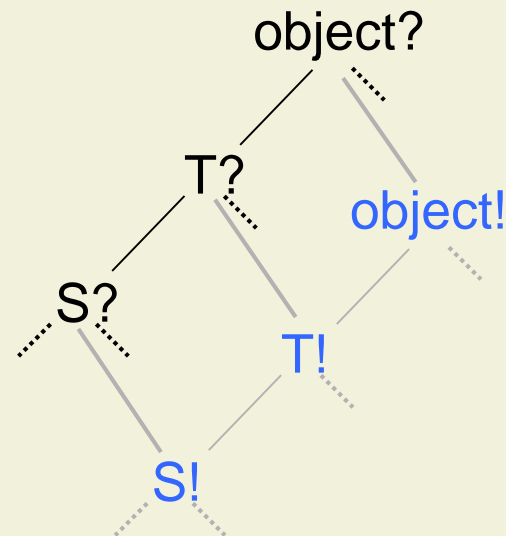
- $S! <: T!$
- $S? <: T?$
- $T! <: T?$

- **Downcasts** from possibly-null types to non-null types **require run-time checks**

```
nnT    = ( T! ) pnT;
nnT    = ( ! ) pnT;
```

```
class T { ... }
```

```
class S extends T { ... }
```



```
T! nnT = ...
T? pnT = ...
S! nnS = ...
```

```
nnT    = nnS;
pnT    = pnS;
pnT    = nnT;
```


Type Rules

- Most type rules of Java remain unchanged
- Additional requirement: expressions whose value gets dereferenced at run time must have a non-null type
 - Receiver of field access
 - Receiver of array access
 - Receiver of method call
 - Expression of a **throw** statement

```
T! nnT = ...  
T? pnT = ...  
S! nnS = ...
```

```
nnT.f = 5;  
nnS.foo( );
```

```
pnT.f = 5;  
pnT.foo( );
```

Compile-time error:
possible
null-dereferencing

Comparing against null

```
class Map {  
  Map? next;  
  
  ...  
  
  Object? get( Object! k ) {  
    ...  
    Map? n = next;  
    if( n == null ) return null;  
    return n.get( k );  
  }  
}
```

Compile-time error:
possible
null-dereferencing

```
class Map {  
  Map? next;  
  
  ...  
  
  Object? get( Object! k ) {  
    ...  
    Map? n = next;  
    if( n == null ) return null;  
    return ( ! ) n ).get( k );  
  }  
}
```

Shorthand for
cast to Map!

Dataflow Analysis

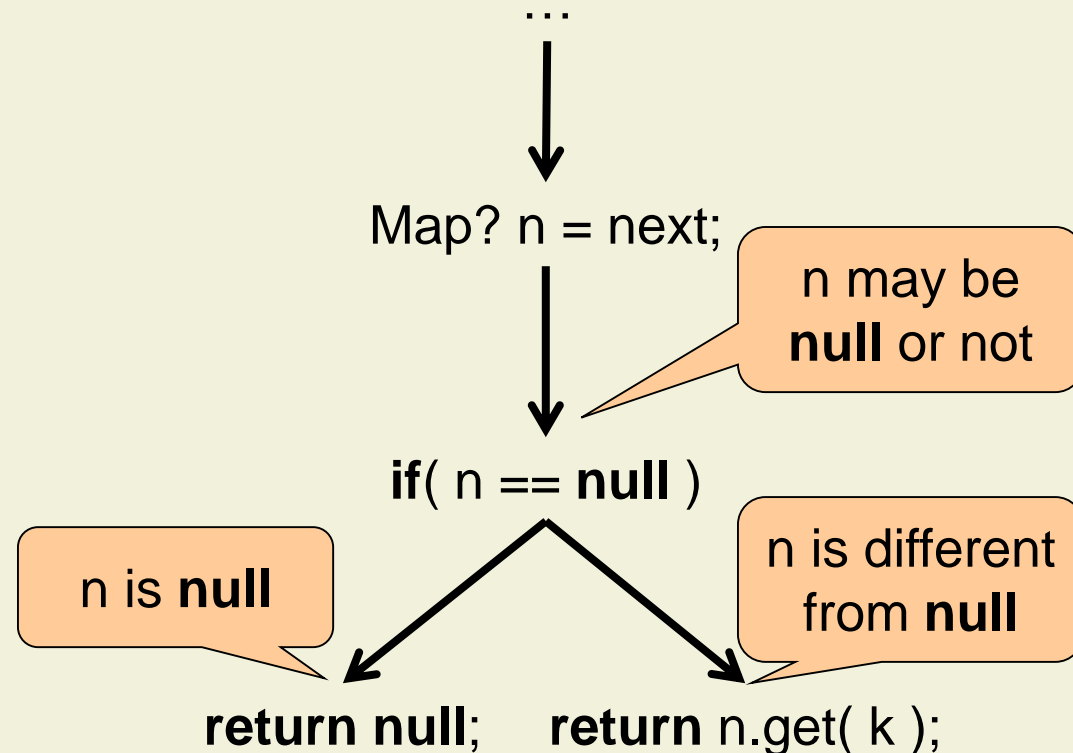
- *Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's control flow graph is used to determine those parts of a program to which a particular value assigned to a variable might propagate.* [Wikipedia]

Comparing against null (cont'd)

```
class Map {  
  Map? next;  
  ...  
  Object? get( Object! k ) {  
    ...  
    Map? n = next;  
    if( n == null ) return null;  
    return n.get( k );  
  }  
}
```

Dataflow analysis
guarantees that
this call is safe

Control Flow Graph



Limitations of Data Flow Analysis

```
class Map {  
    Map? next;  
    ...  
    Object? get( Object! k ) {  
        ...  
        Map? n = next;  
        if( n == null ) return null;  
        return n.get( k );  
    }  
}
```

```
class Map {  
    Map? next;  
    ...  
    Object? get( Object! k ) {  
        ...  
        if( next == null ) return null;  
        return next.get( k );  
    }  
}
```


Limitations of Data Flow Analysis (cont'd)

```
class Map {  
  Map? next;  
  ...  
  Object? get( Object! k ) {  
    ...  
    if( next == null ) return null;  
    someObject.foo( this );  
    return next.get( k );  
  }  
}
```

```
void foo( Map! m ) {  
  m.next = null;  
}
```

- Data flow analysis tracks values of local variables, but not heap locations
 - Tracking heap locations is in general non-modular
- In concurrent programs, **other threads** could modify heap locations

7. Initialization

7.1 Simple Non-Null Types

7.2 Object Initialization

7.3 Initialization of Global Data

Constructing New Objects

```
class Map {  
    Map? next;  
    Object! key;  
    Object! value;  
  
    Map( Object! k, Object! v ) {  
        key = k;  
        value = v;  
    }  
}
```

All fields are
initialized to **null**

Type invariant is
violated here!

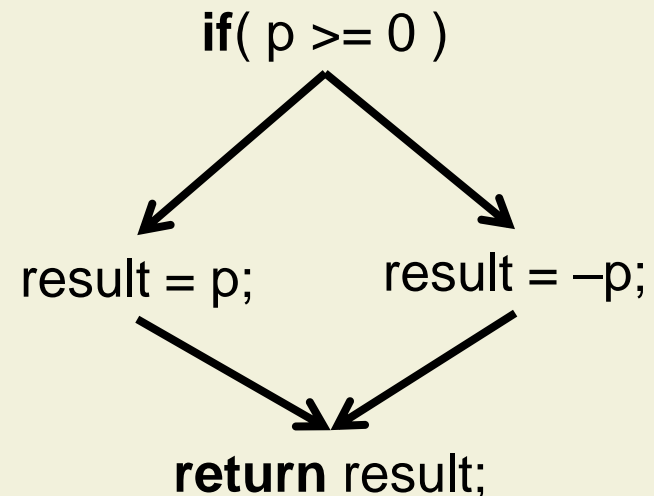
■ Idea:

- Make sure all non-null fields are initialized when the constructor terminates
- Do not rely on non-nullness of fields of objects under construction

Definite Assignment of Local Variables

- Java and C# do not initialize local variables
- **Definite assignment rule**: every local variable must be assigned to before it is first used
 - Checked by compiler using a data flow analysis
 - Also checked during bytecode verification

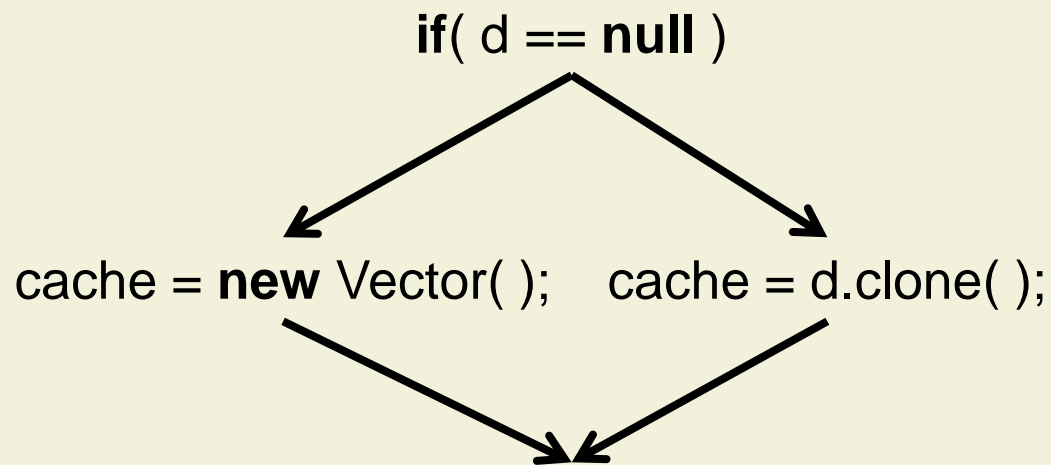
```
int abs( int p ) {  
    int result;  
    if( p >= 0 ) result = p;  
    else result = -p;  
    return result;  
}
```



Definite Assignment of Fields

- Idea: apply definite assignment rule for fields in constructor

```
class Demo {  
  Vector! cache;  
  Demo( Vector? d ) {  
    if( d == null )  
      cache = new Vector( );  
    else  
      cache = d.clone( );  
  }  
}
```



Problem 1: Method Calls

```
class Demo {  
    Vector! cache;  
  
    Demo( ) {  
        int size = optimalSize( );  
        cache = new Vector( size );  
    }  
  
    int optimalSize( ) {  
        return 16;  
    }  
}
```

Dynamically
bound

```
class Sub extends Demo {  
    Vector! data;  
  
    Sub( Vector! d ) {  
        data = d.clone( );  
    }  
  
    int optimalSize( ) {  
        return data.size( ) * 2;  
    }  
}
```

Implicit
super-call

NullPointerException

```
Vector! v = new Vector( );  
Sub! s = new Sub( v );
```


Problem 2: Call-backs

```
class Demo implements Observer {  
    static Subject! subject;  
  
    Demo( ) {  
        subject.register( this );  
    }  
  
    void update( ... ) { }  
}
```

```
class Subject {  
    void register( Observer! o ) {  
        ...  
        o.update( ... );  
    }  
}
```

Dynamically bound

```
class Sub extends Demo  
    Vector! data;  
  
    Sub( Vector! d ) { data = d.clone( ); }  
    void update( ... ) { ... data.size( ) ... }  
}
```

Implicit super-call

```
Vector! v = new Vector( );  
Sub! s = new Sub( v );
```

NullPointerException

Problem 3: Escaping via Method Calls

```
class Demo implements Observer {  
    static Subject! subject;  
  
    Demo( ) {  
        subject.register( this );  
    }  
  
    void update( ... ) { }  
}
```

```
class Sub extends Demo {  
    Vector! data;  
  
    Sub( Vector! d ) { data = d.clone( ); }  
    void update( ... ) { ... data.size( ) ... }  
}
```

NullPointerException

```
class Subject extends Thread {  
    List<Observer!>! list;  
  
    void register( Observer! o )  
    { list.add( o ); }  
  
    void run( ) {  
        while( true ) {  
            if( sensorValueChanged( ) )  
                for( Observer! o: list )  
                    o.update( ... );  
        }  
    }  
    ...  
}
```

No call-back

Call may occur at any time

Problem 4: Escaping via Field Updates

```
class Node {  
    Node! next; // a cyclic list  
    Process! proc;  
  
    Node( Node! after, Process! p ) {  
        this.next = after.next;  
        after.next = this;  
        proc = p;  
    }  
}
```

Assume scheduler
runs now, with
current == after

```
class Scheduler extends Thread {  
    Node! current;  
  
    void run( ) {  
        while( true ) {  
            current.proc.preempt( );  
            current = current.next;  
            current.proc.resume( );  
            Thread.sleep( 1000 );  
        }  
    }  
    ...  
}
```

NullPointerException

Definite Assignment of Fields: Summary

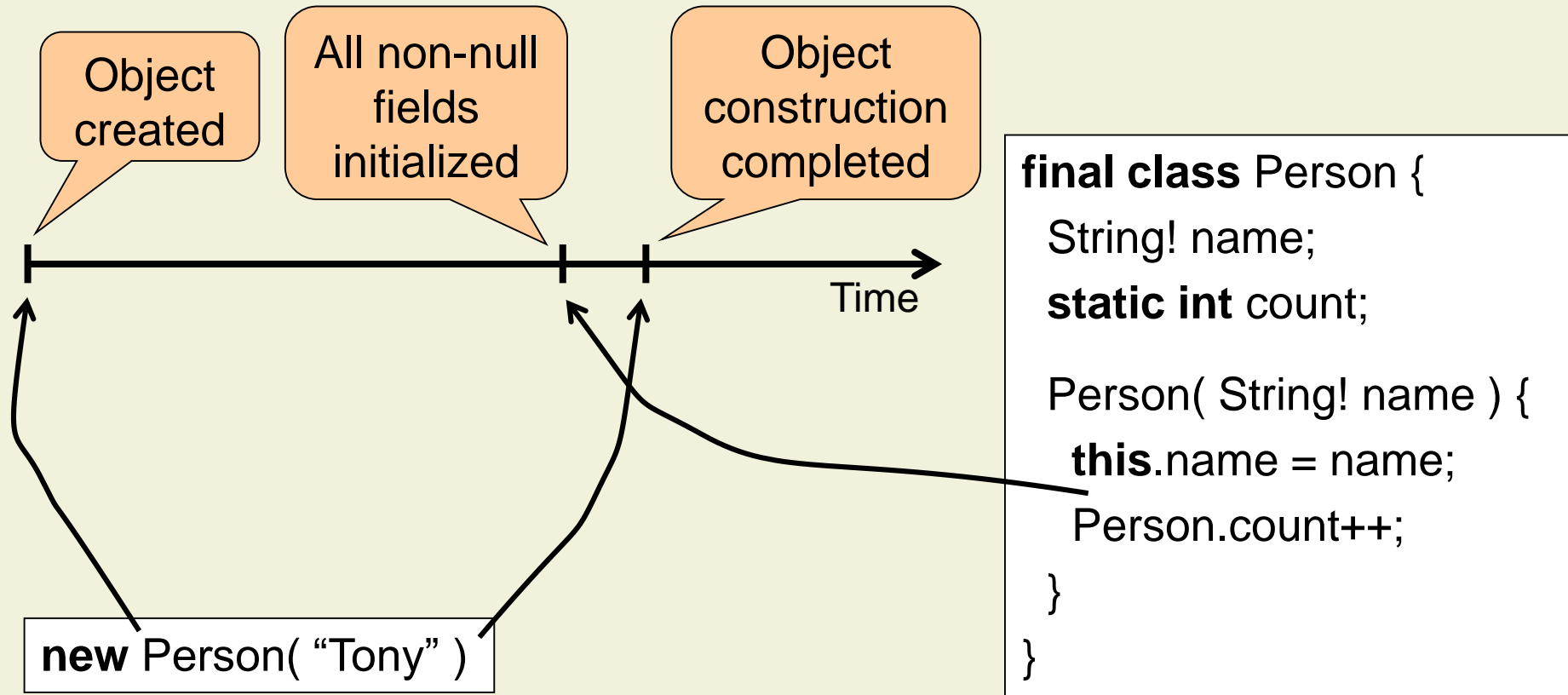
- The simple definite assignment checks for fields are sound only if **partly-initialized objects do not escape** from their constructor
 - Not passed as receiver or argument to a method call
 - Not stored in a field or an array

```
class Node {  
    Node! next; // a  
    String! label;  
  
    Node( String! l ) {  
        this.next = this;  
        this.setLabel( l );  
    }  
  
    void setLabel( String! l ) {  
        this.label = l;  
    }  
}
```

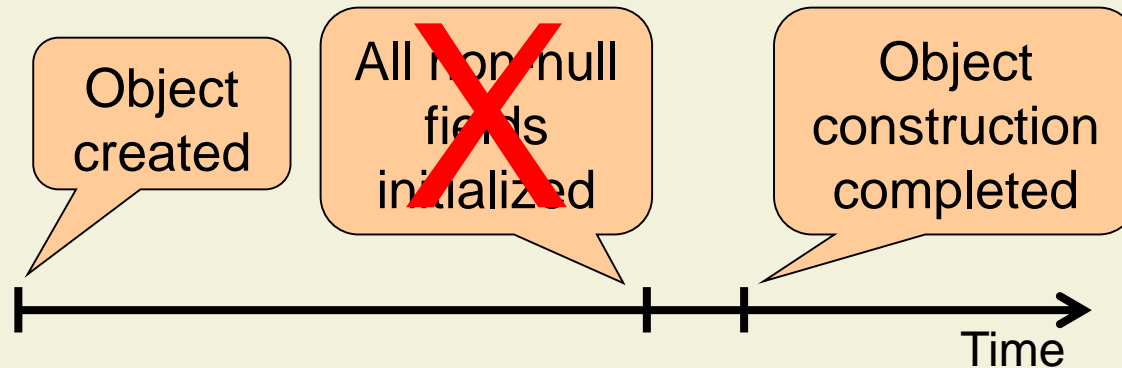
Field update is safe:
object does not
escape

Method call is safe:
no reading of fields
of new object

Initialization Phases



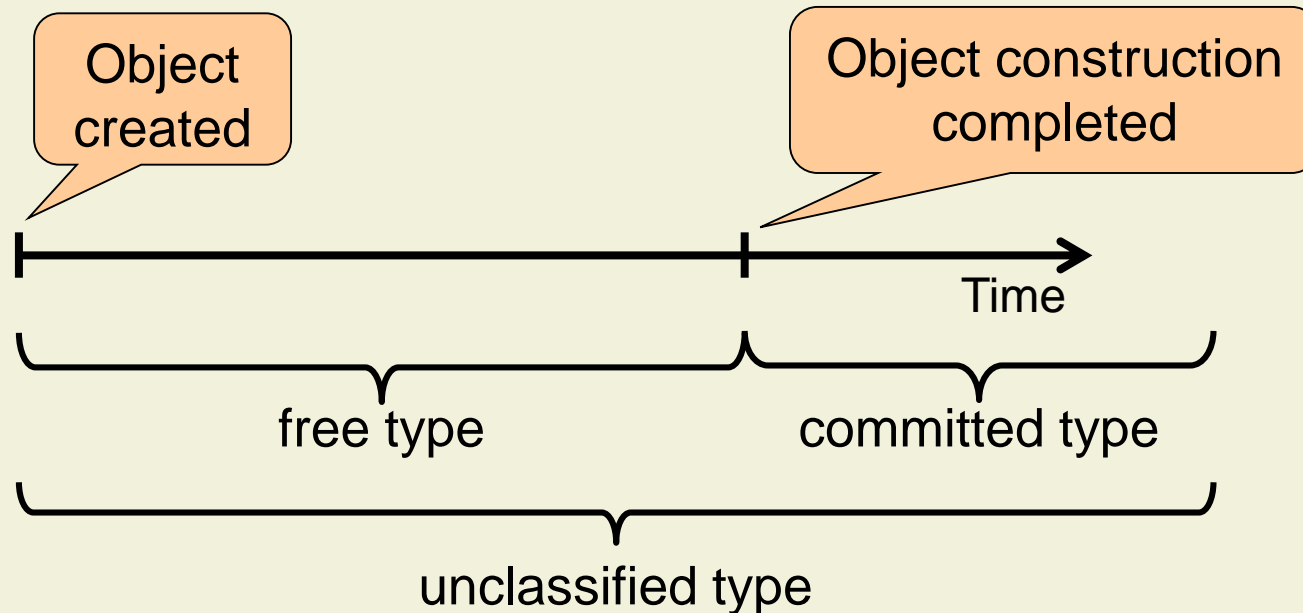
Tracking Object Construction



- Idea: design a type system that tracks **which objects are under construction**
 - For simplicity, we track whether the construction has completed (non-null fields may be initialized earlier)
- (Simplified) type invariant:
If the static type of an expression e is a non-null type then e 's value at run time is different from **null**

Construction Types

- For every class or interface T , we introduce **different types** for references:
 - to objects under construction
 - to objects whose construction is completed



Construction Types: Details

- For a class or interface T , we introduce six types
 - $T!$ and $T?$ (committed types)
 - **free** $T!$ and **free** $T?$ (free types)
 - **unc** $T!$ and **unc** $T?$ (unclassified types)
- Subtyping
 - $T!$ and **free** $T!$ are subtypes of **unc** $T!$
 - $T?$ and **free** $T?$ are subtypes of **unc** $T?$
 - No casts from unclassified to free or committed types

$T! \text{ cT} = \dots$

free $T! \text{ fT} = \dots$

unc $T? \text{ uT} = \dots$

$\text{uT} = \text{cT};$

$\text{uT} = \text{fT};$

unc $T! \text{ nT} = (\text{unc } T!) \text{ uT};$

$\text{fT} = \text{cT};$

$\text{cT} = (T!) \text{ uT};$

$\text{fT} = (\text{free } T!) \text{ uT};$


Requirement 1: Local Initialization

- An object is **locally initialized** if its **non-null fields have non-null values**
- If the static type of an expression e is a **committed** type then e 's value at run time is **locally initialized**
- Non-null type of a field read $e.f$

		non-null type of f	
		!	?
construction type of e	committed	!	?
	free	?	?
	unc	?	?

Heap Traversal

```
class Node {  
    Node! next; // a cyclic list  
    Object! elem;  
  
    boolean contains( Object! e ) {  
        committed Node! ptr = this.next;  
        while( ptr != this ) {  
            if( ptr.elem.equals( e ) )  
                return true;  
            ptr = ptr.next;  
        }  
        return false;  
    }  
}
```

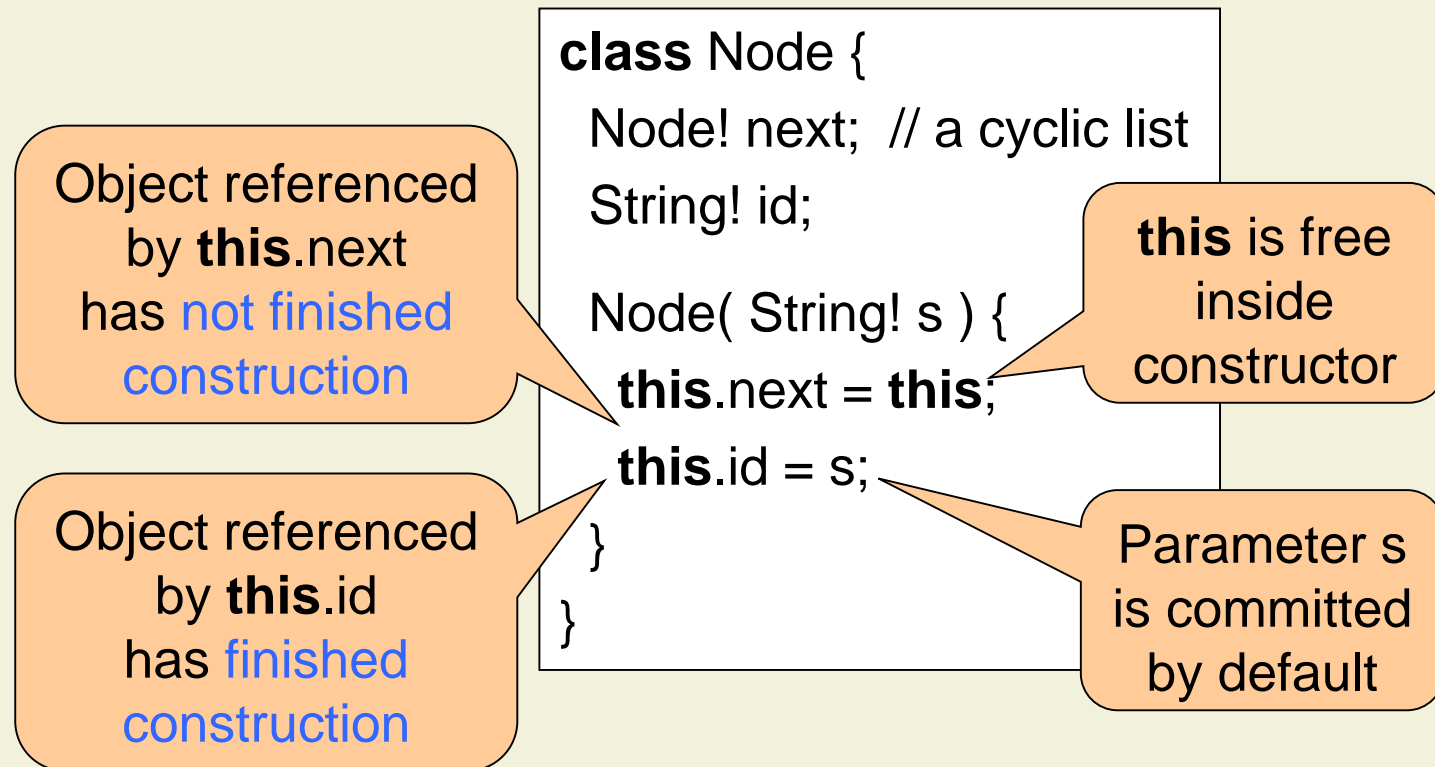


ptr has to be committed

Requirement 2: Transitive Initialization

- An object is **transitively initialized** if **all reachable objects are locally initialized**
- If the static type of an expression e is a **committed** type then e 's value at run time is **transitively initialized**

Requirement 3: Cyclic Structures



Type Rules: Field Write

- A field write $e_1.f = e_2$ is well-typed if
 - e_1 and e_2 are well-typed
 - e_1 's type is a non-null type
 - e_2 's class and non-null type conform to the type of $e_1.f$
 - e_1 's type is free or e_2 's type is committed

		Type of e_2		
		committed	free	unc
Type of e_1	committed	✓	✗	✗
	free	✓	✓	✓
	unc	✓	✗	✗

Type Rules: Field Read

- A field read expression $e.f$ is well-typed if
 - e is well-typed
 - e 's type is a non-null type
- The type of $e.f$ is

		Declared type of f	
		$T!$	$T?$
Type of e	$S!$	$T!$	$T?$
	free $S!$	unc $T?$	unc $T?$
	unc $S!$	unc $T?$	unc $T?$

Type Rules: Constructors

- Constructor signatures include construction types for all parameters
 - Receiver has free, non-null type
- Constructor bodies must assign non-null values to all non-null fields of the receiver

```
class Node
```

```
Node! next
```

```
String! id;
```

```
Node( String! name ) {
```

```
    next = this;
```

```
    id = this.get
```

```
}
```

```
}
```

name is of a committed type

this is of a free type

Non-nullness invariant is

Definite assignment check succeeds

Type Rules: Methods and Calls

- Method signatures include construction types for all parameters
- Calls are type-checked as usual
- Overriding requires the usual co- and contravariance

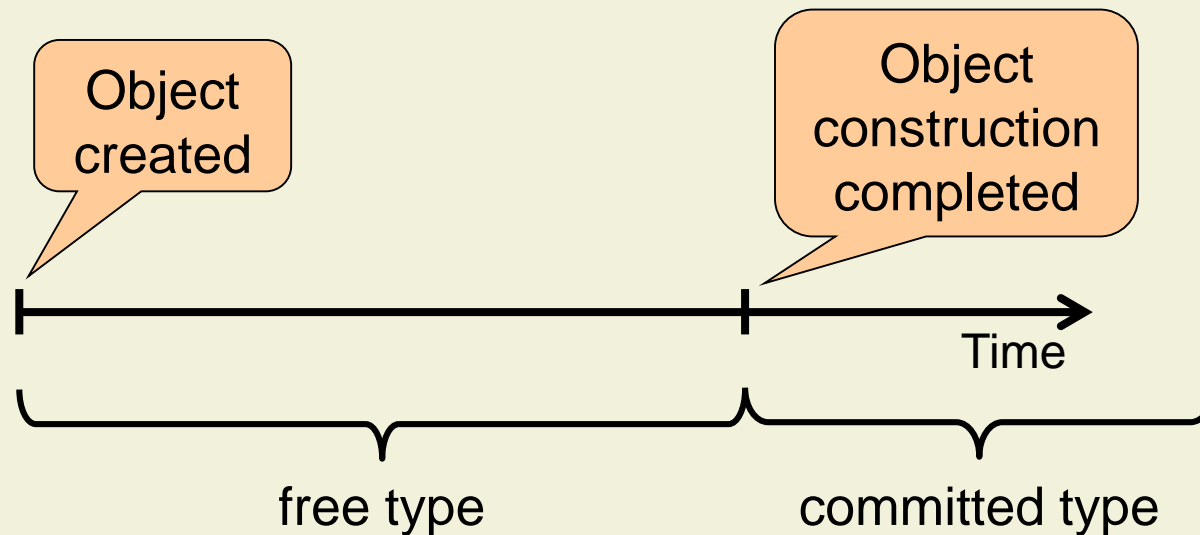
```
class Node {  
  Node! next; // cyclic list  
  String! id;  
  Node( String!  
    next = this;  
    id = this.getId( name );  
}  
  
String! free getId( String! n )  
{ return ...; }
```

Call is
permitted

this is of a
free type

Object Construction

- We have not yet defined **when** the construction of a new object completes



Object Construction (cont'd)

- Is construction finished when the constructor terminates?
- Not if there are subclass constructors, which have not yet executed
 - In general not known modularly

```
class Demo {  
    String! name;  
    Demo( ) {  
        name = "Tony";  
    }  
}
```

this is **not**
completely
constructed

```
class Sub extends Demo {  
    Vector! data;  
  
    Sub( ) {  
        data = new Vector( );  
    }  
}
```


Object Construction (cont'd)

- Is construction finished when the **new**-expression terminates?
- Not if constructor initializes fields with free references

```
class Demo {  
    C! myC;  
    Demo() {  
        C! c = new C( this );  
        c.foo();  
        myC = c;  
    }  
}
```

c is locally, but
not transitively
initialized

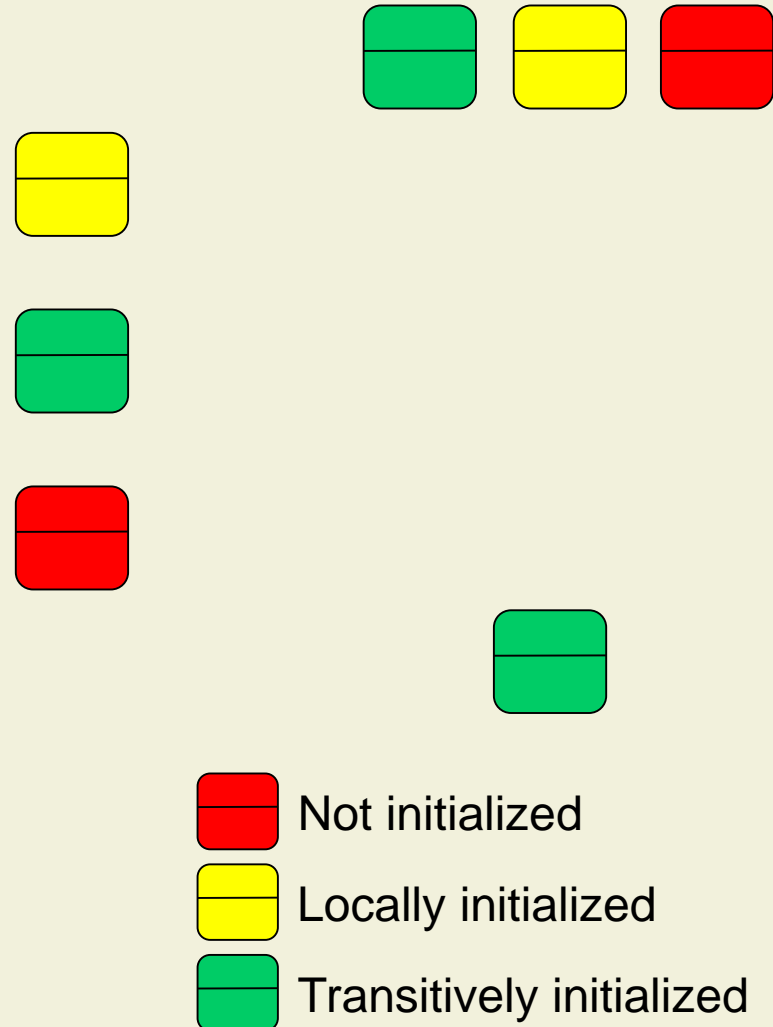
```
class C {  
    Demo! demo;  
    C( free Demo! d ) { demo = d; }  
    String! foo( ) { return demo.myC.toString( ); }  
}
```

NullPointerException

Object Construction

■ Assumptions

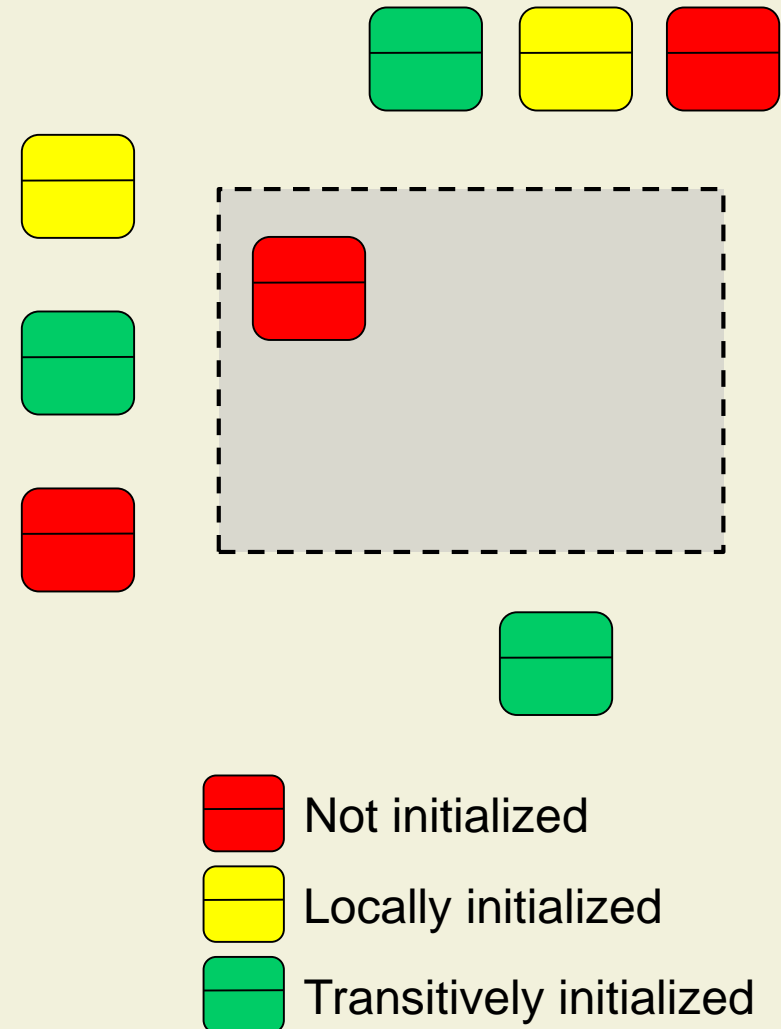
- new-expression takes **only committed arguments**
- Nested new-expressions take **arbitrary arguments**



Object Construction

Assumptions

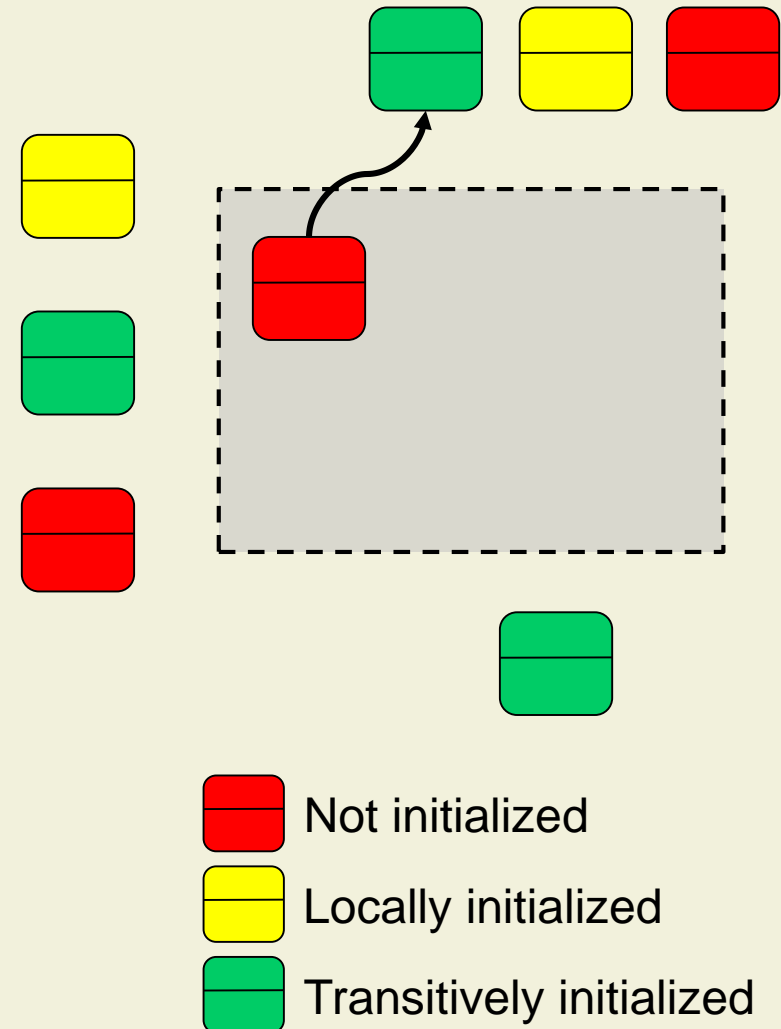
- new-expression takes **only committed arguments**
- Nested new-expressions take **arbitrary arguments**



Object Construction

■ Assumptions

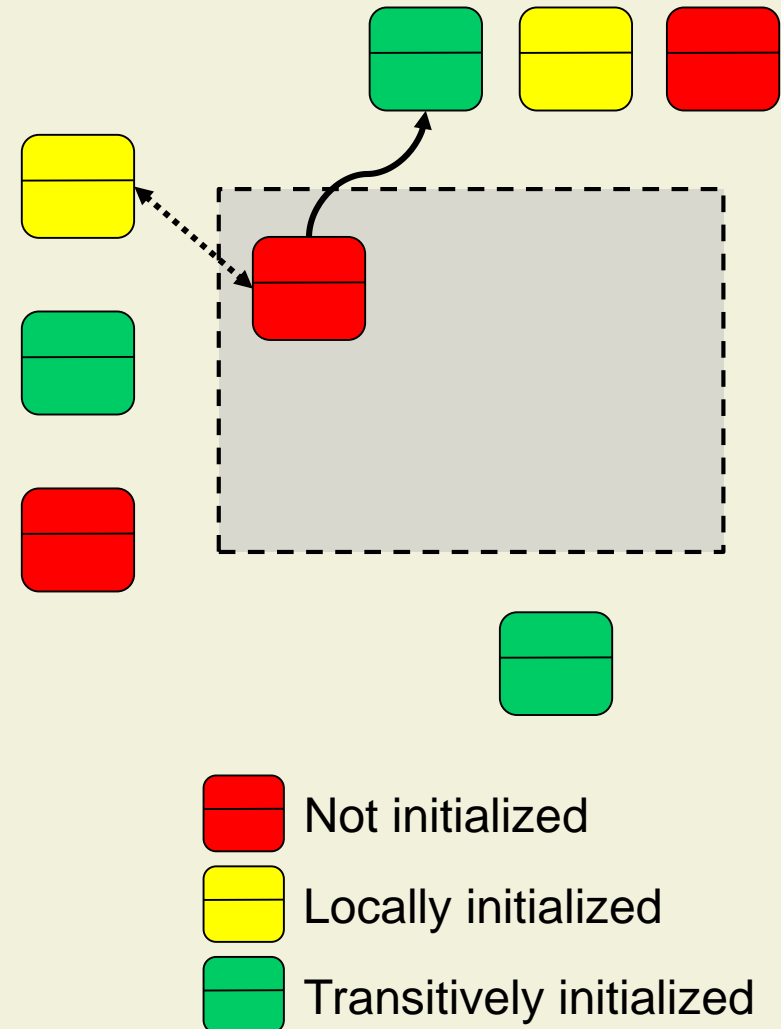
- new-expression takes **only committed arguments**
- Nested new-expressions take **arbitrary arguments**



Object Construction

■ Assumptions

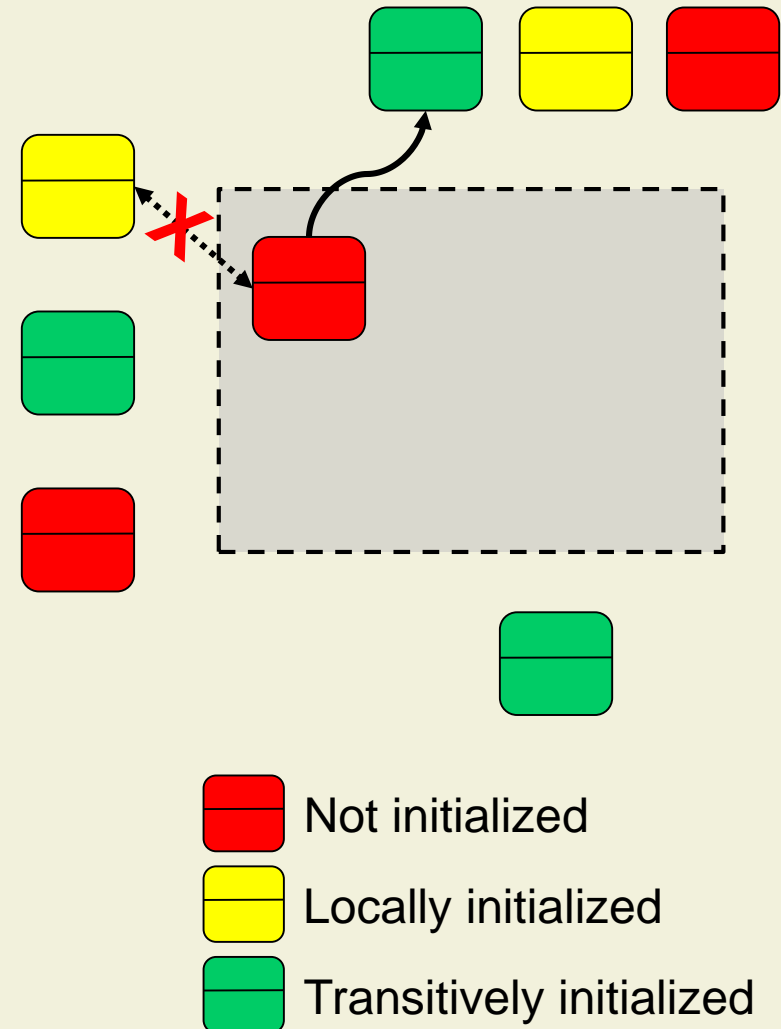
- new-expression takes **only committed arguments**
- Nested new-expressions take **arbitrary arguments**



Object Construction

■ Assumptions

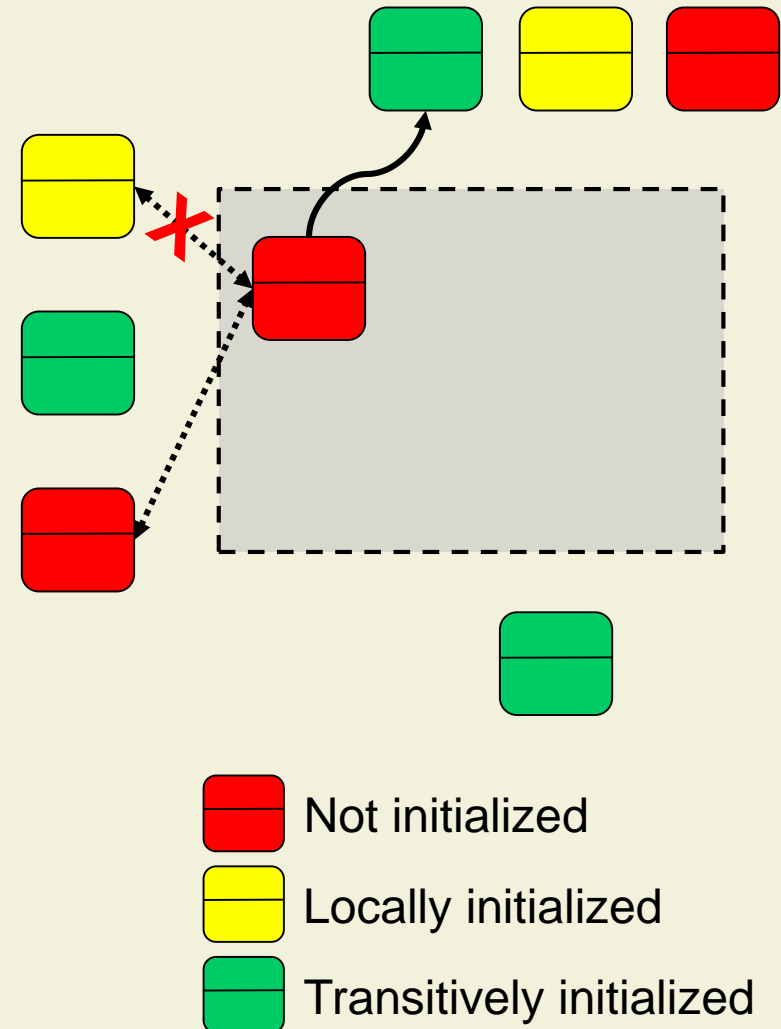
- new-expression takes **only committed arguments**
- Nested new-expressions take **arbitrary arguments**



Object Construction

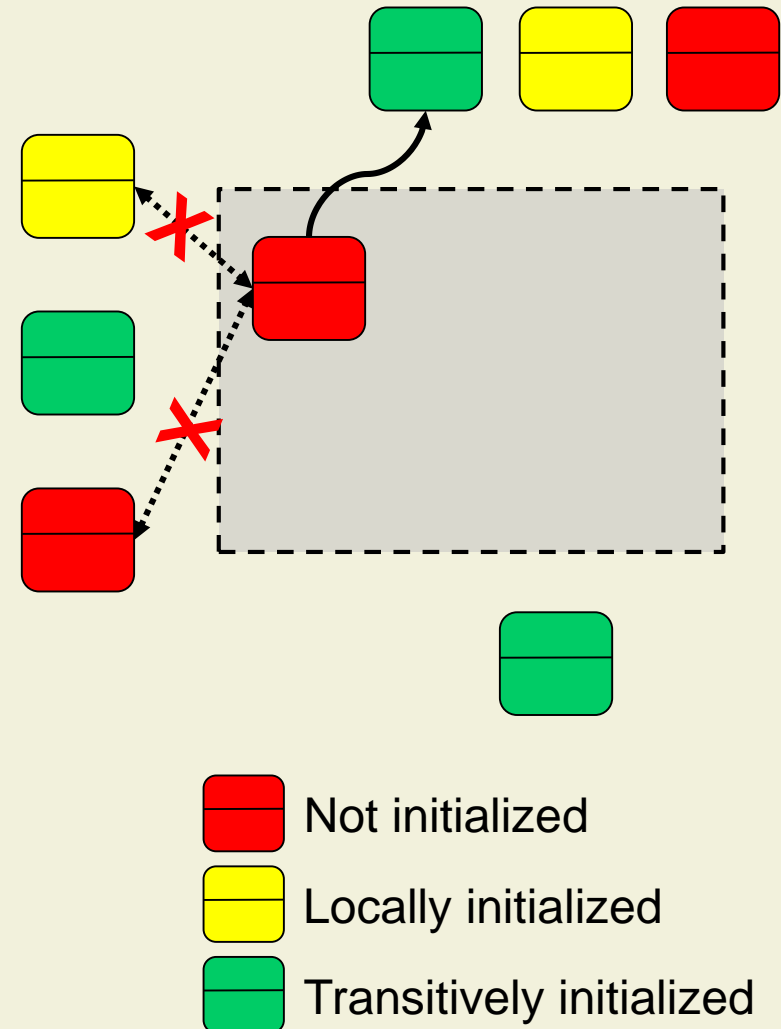
Assumptions

- new-expression takes **only committed arguments**
- Nested new-expressions take **arbitrary arguments**



Object Construction

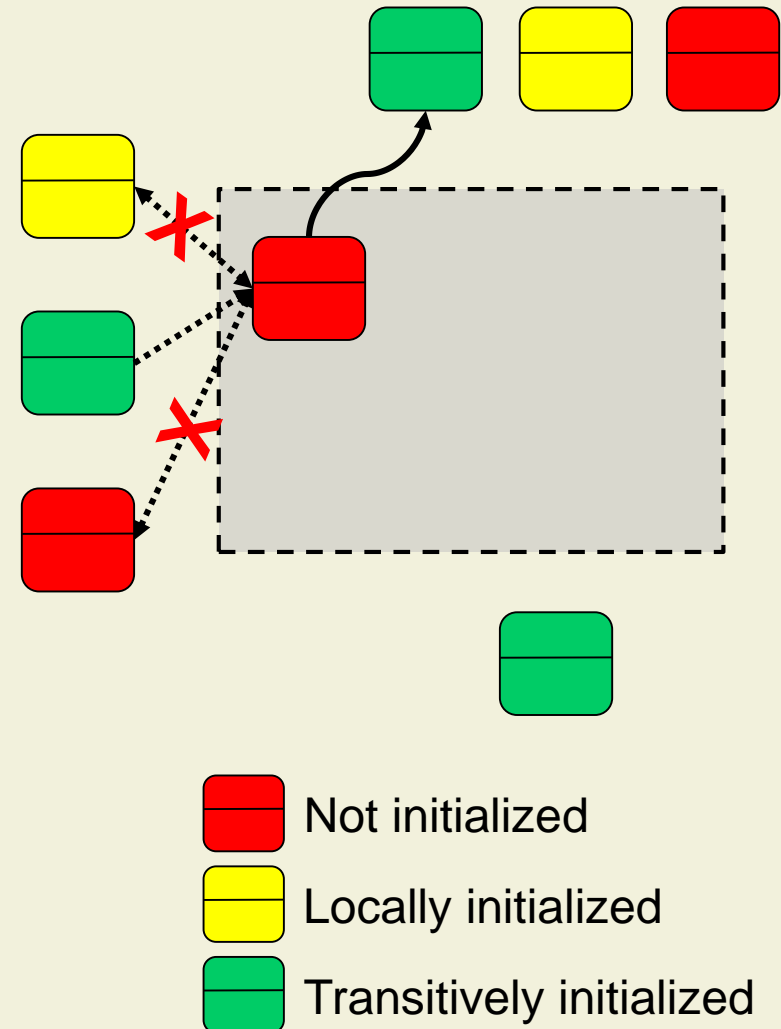
- Assumptions
 - new-expression takes **only committed arguments**
 - Nested new-expressions take **arbitrary arguments**



Object Construction

■ Assumptions

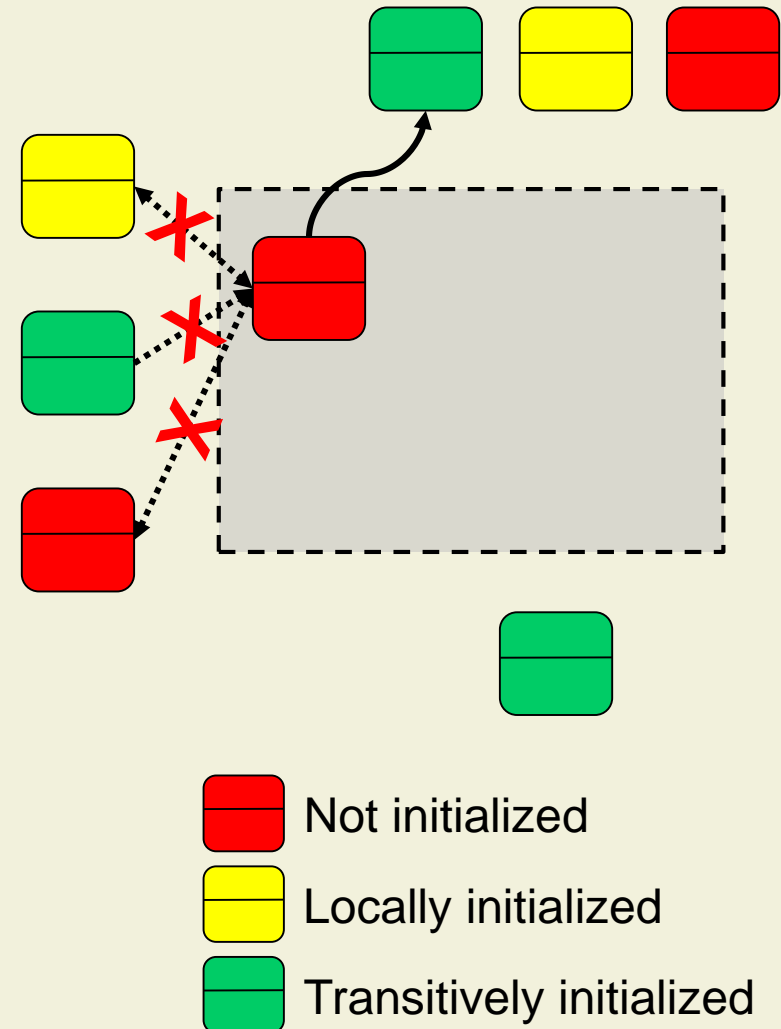
- new-expression takes **only committed arguments**
- Nested new-expressions take **arbitrary arguments**



Object Construction

Assumptions

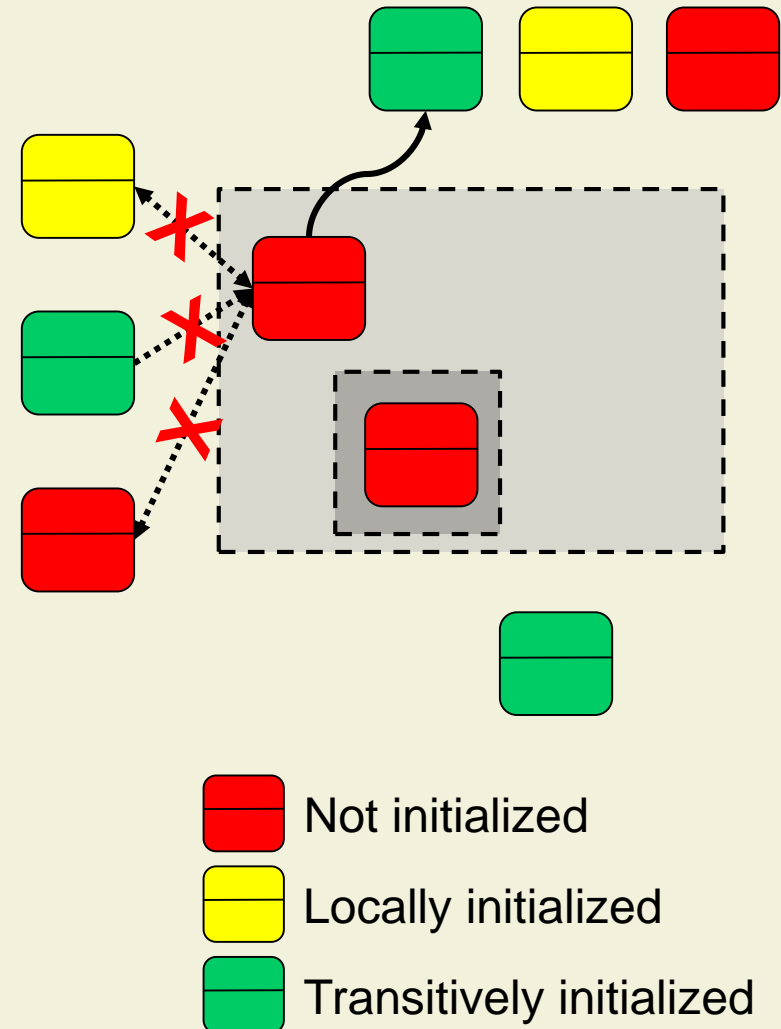
- new-expression takes **only committed arguments**
- Nested new-expressions take **arbitrary arguments**



Object Construction

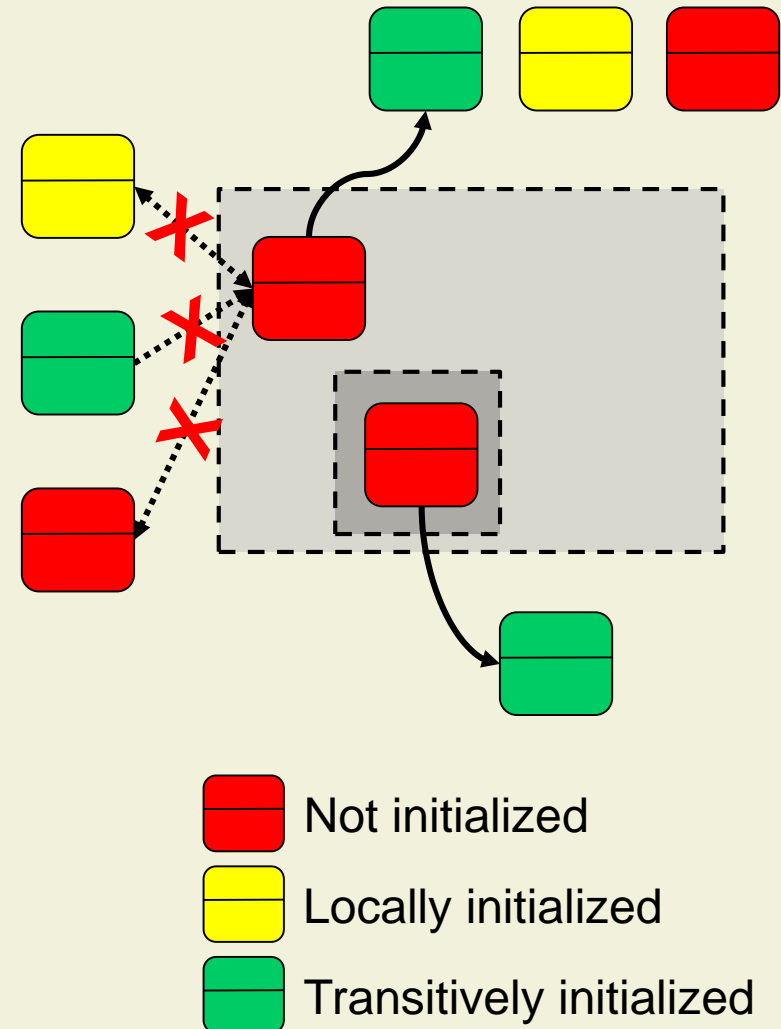
■ Assumptions

- new-expression takes **only committed arguments**
- Nested new-expressions take **arbitrary arguments**



Object Construction

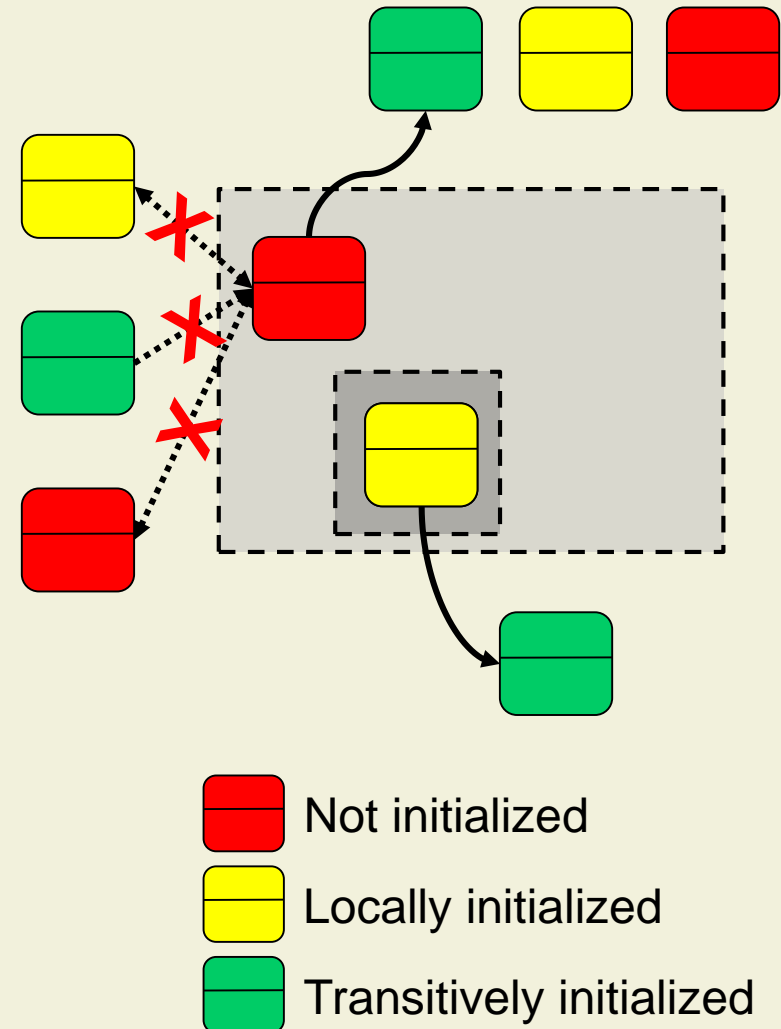
- Assumptions
 - new-expression takes **only committed arguments**
 - Nested new-expressions take **arbitrary arguments**



Object Construction

■ Assumptions

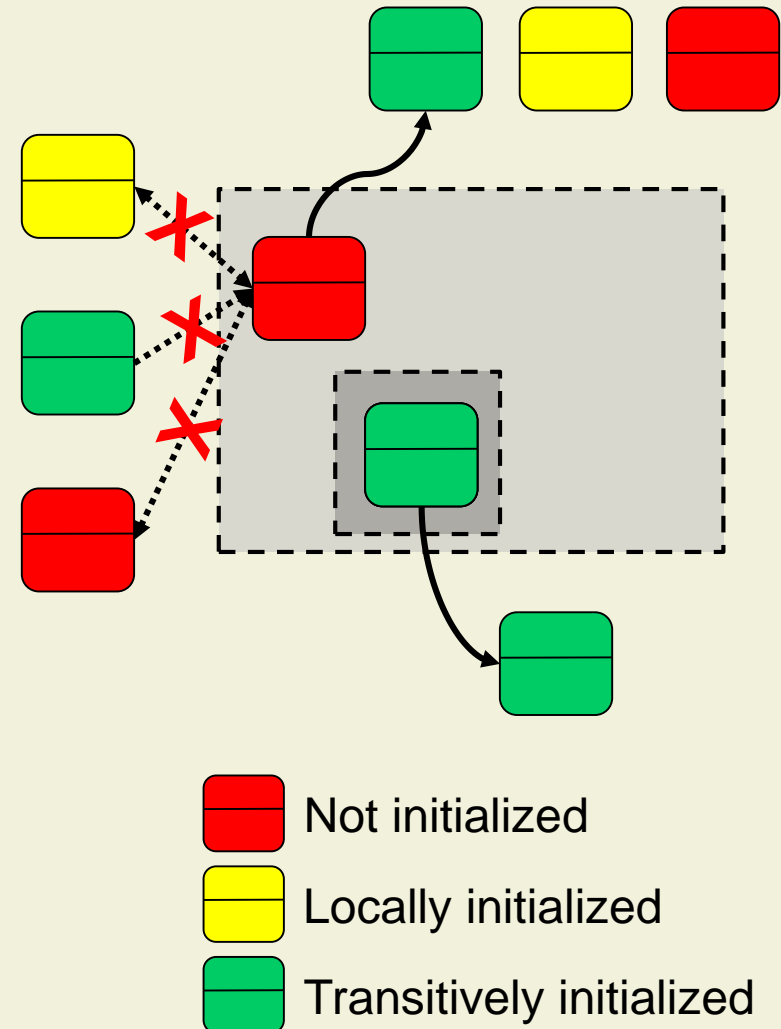
- new-expression takes **only committed arguments**
- Nested new-expressions take **arbitrary arguments**



Object Construction

■ Assumptions

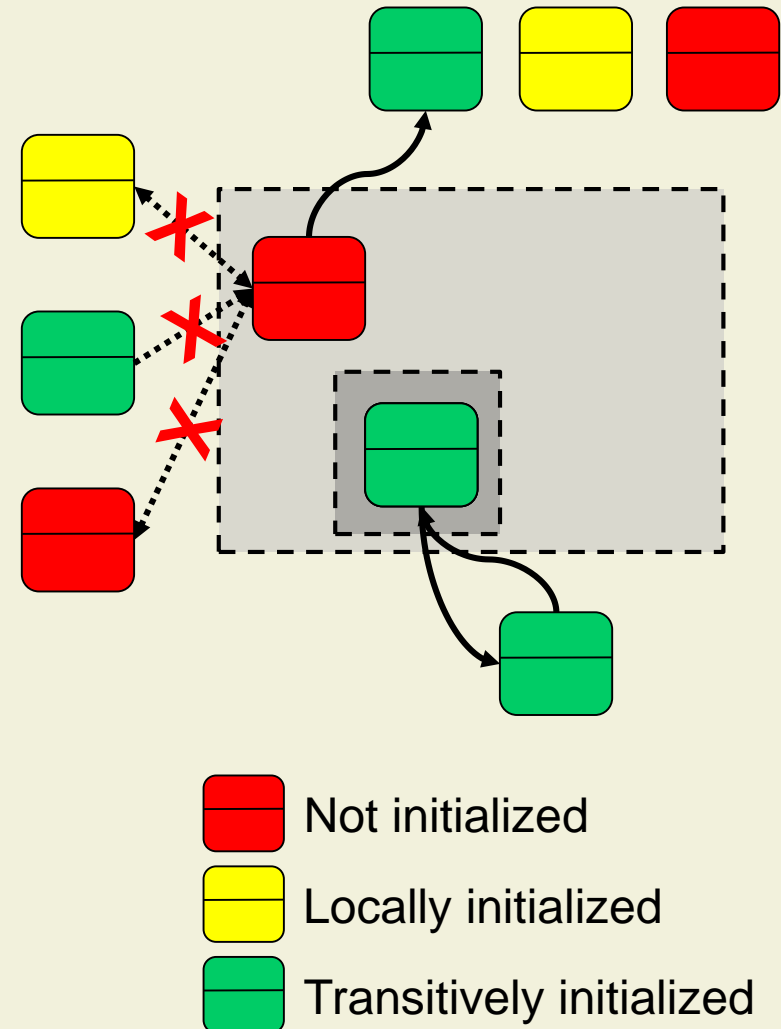
- new-expression takes **only committed arguments**
- Nested new-expressions take **arbitrary arguments**



Object Construction

■ Assumptions

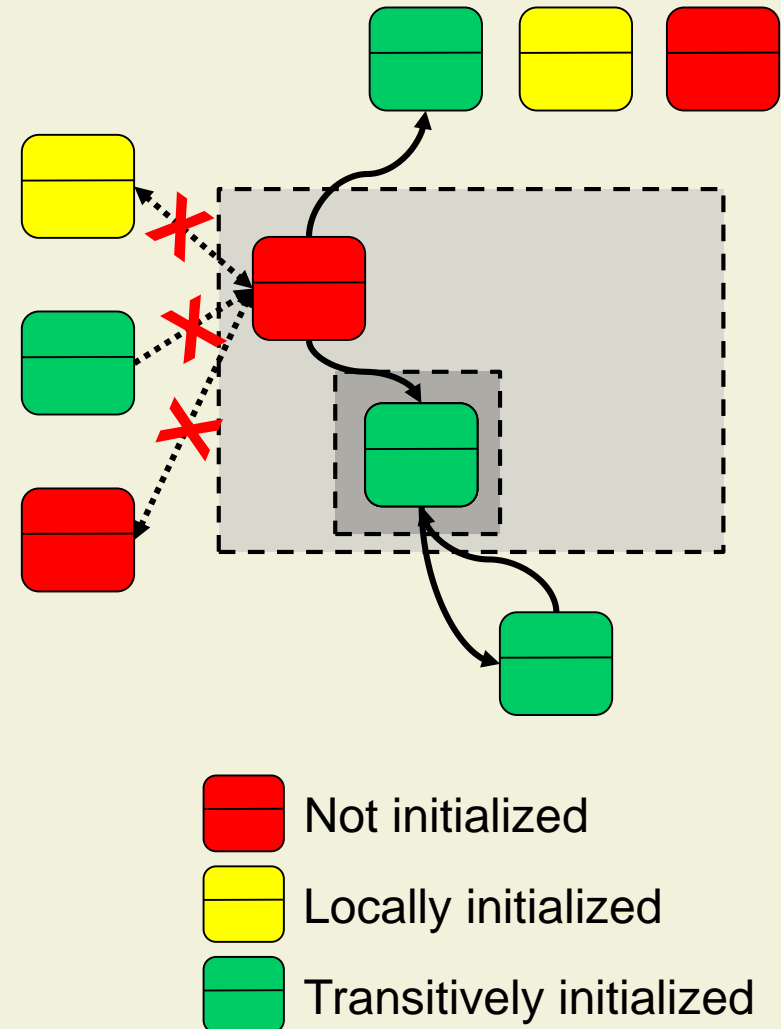
- new-expression takes **only committed arguments**
- Nested new-expressions take **arbitrary arguments**



Object Construction

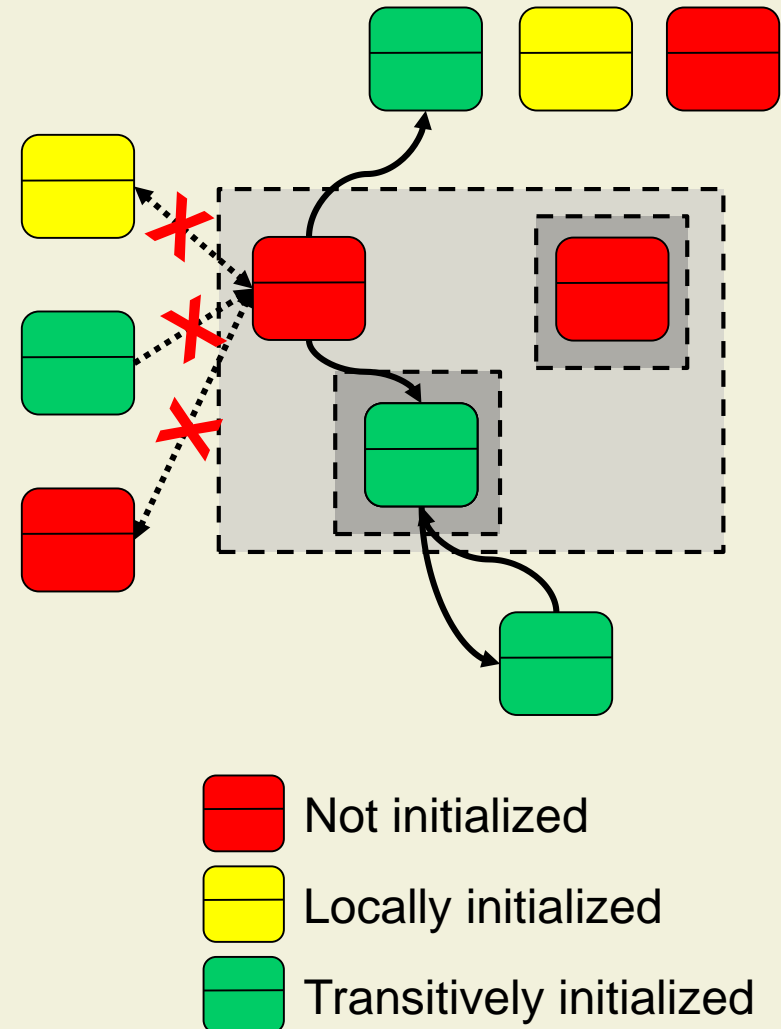
■ Assumptions

- new-expression takes **only committed arguments**
- Nested new-expressions take **arbitrary arguments**



Object Construction

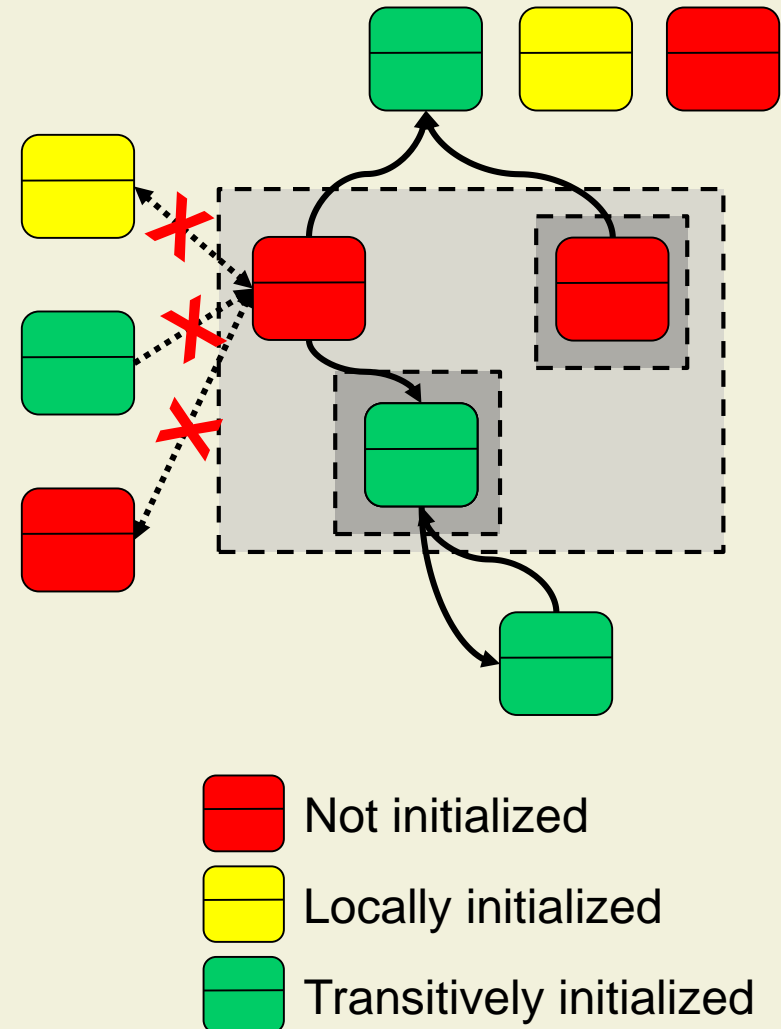
- Assumptions
 - new-expression takes **only committed arguments**
 - Nested new-expressions take **arbitrary arguments**



Object Construction

■ Assumptions

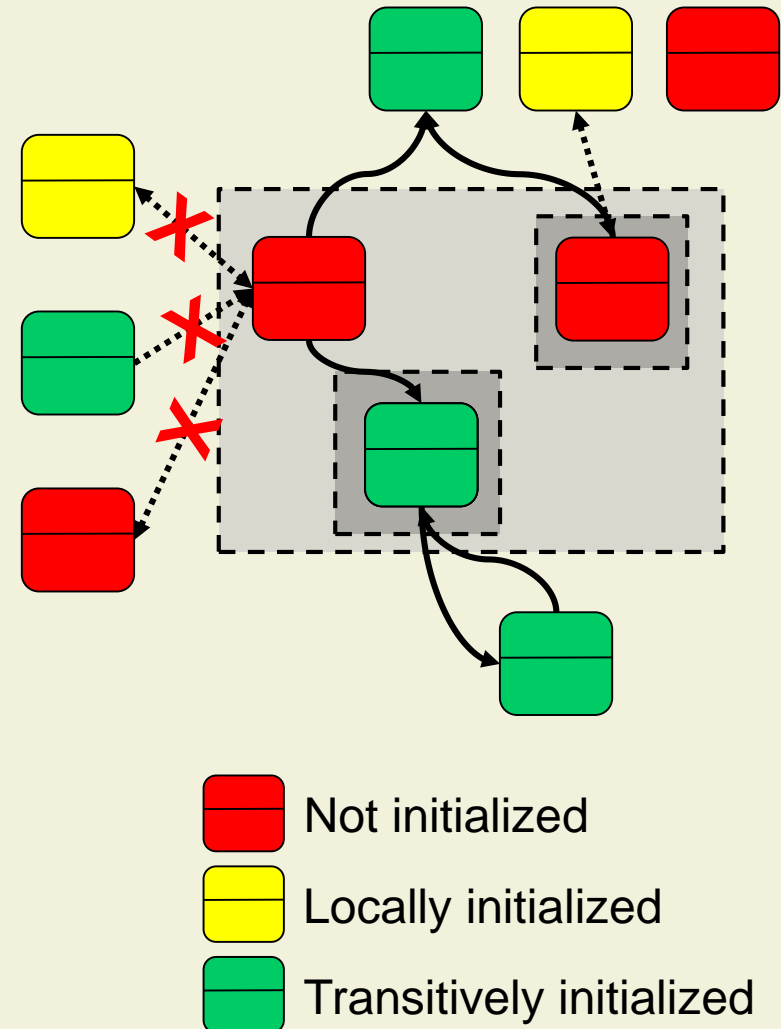
- new-expression takes **only committed arguments**
- Nested new-expressions take **arbitrary arguments**



Object Construction

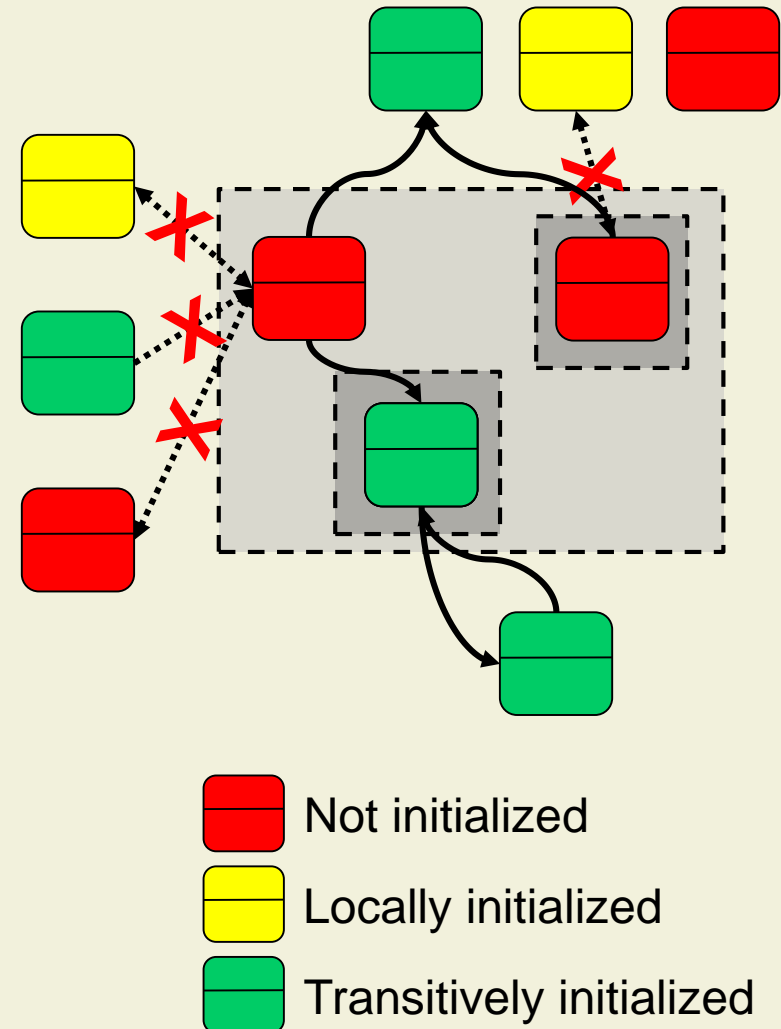
■ Assumptions

- new-expression takes **only committed arguments**
- Nested new-expressions take **arbitrary arguments**



Object Construction

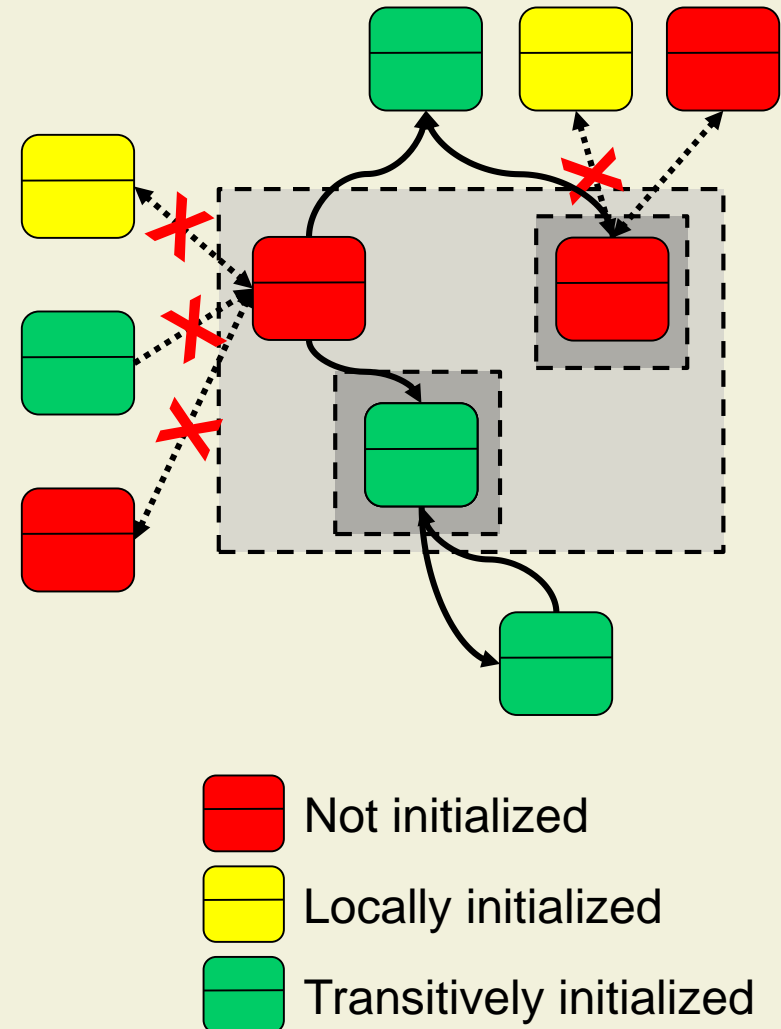
- Assumptions
 - new-expression takes **only committed arguments**
 - Nested new-expressions take **arbitrary arguments**



Object Construction

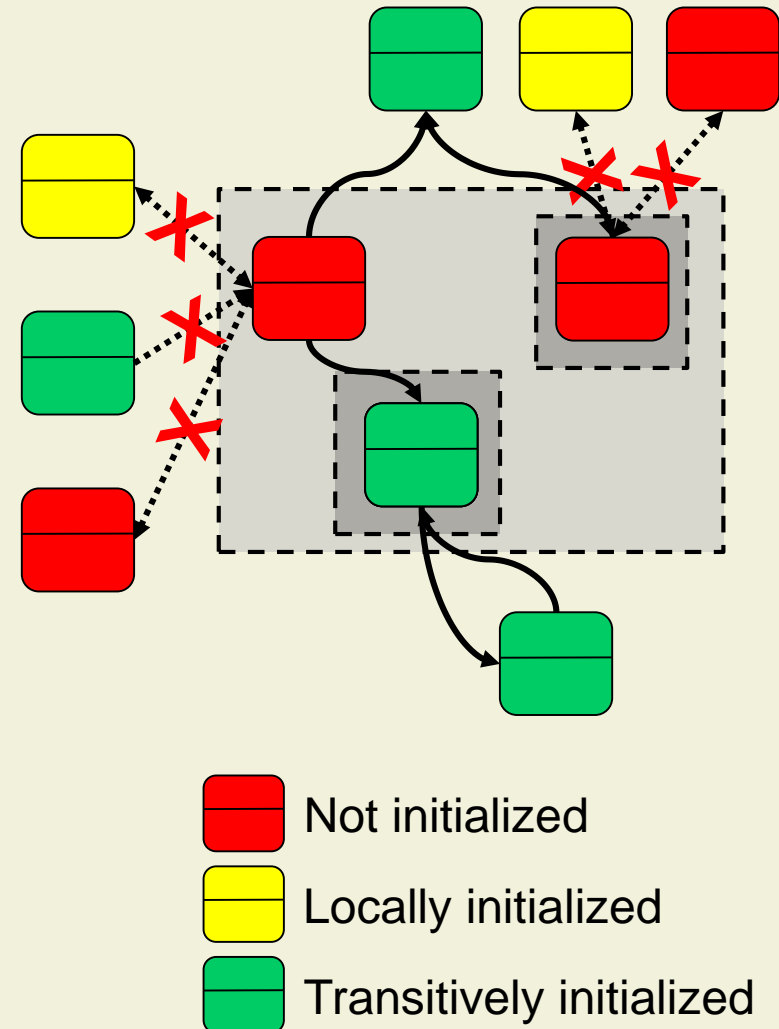
■ Assumptions

- new-expression takes **only committed arguments**
- Nested new-expressions take **arbitrary arguments**



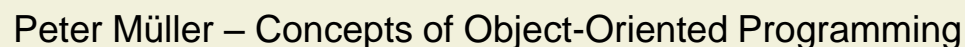
Object Construction

- Assumptions
 - new-expression takes **only committed arguments**
 - Nested new-expressions take **arbitrary arguments**



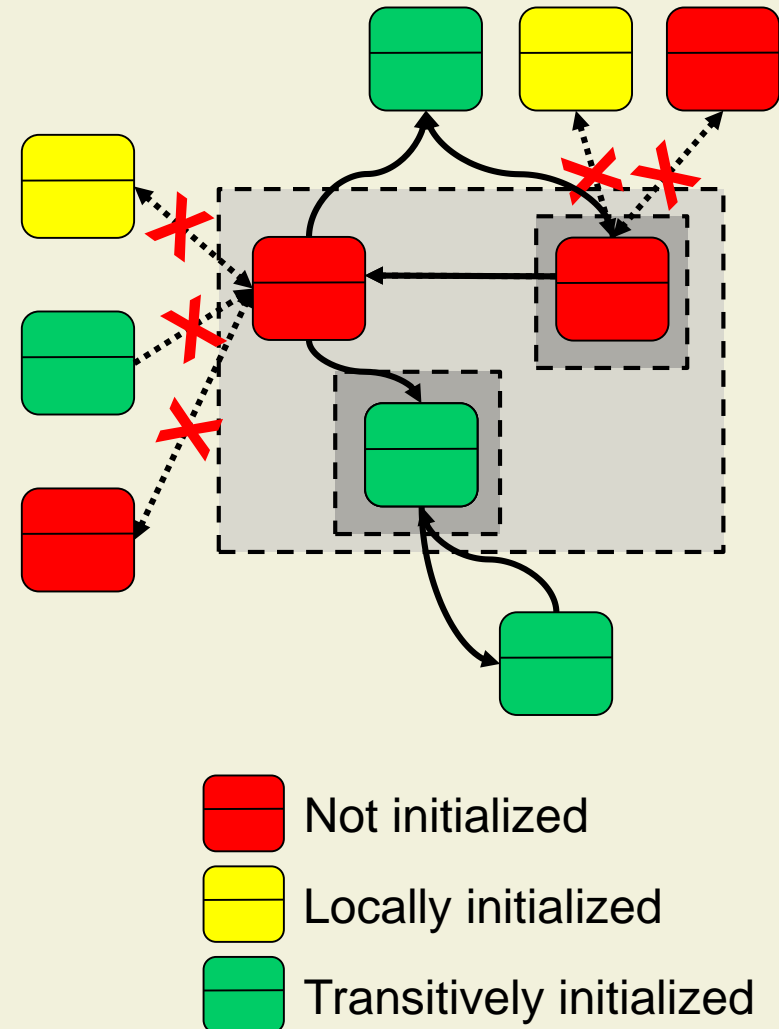
- Assumptions

-
- The diagram shows a directed graph with nodes represented as colored squares (yellow, green, red) divided horizontally. A large gray-shaded area represents a strongly connected component. Within this area, a dashed box highlights a smaller strongly connected component. Red 'X' marks indicate connections that are not part of the strongly connected component.



Object Construction

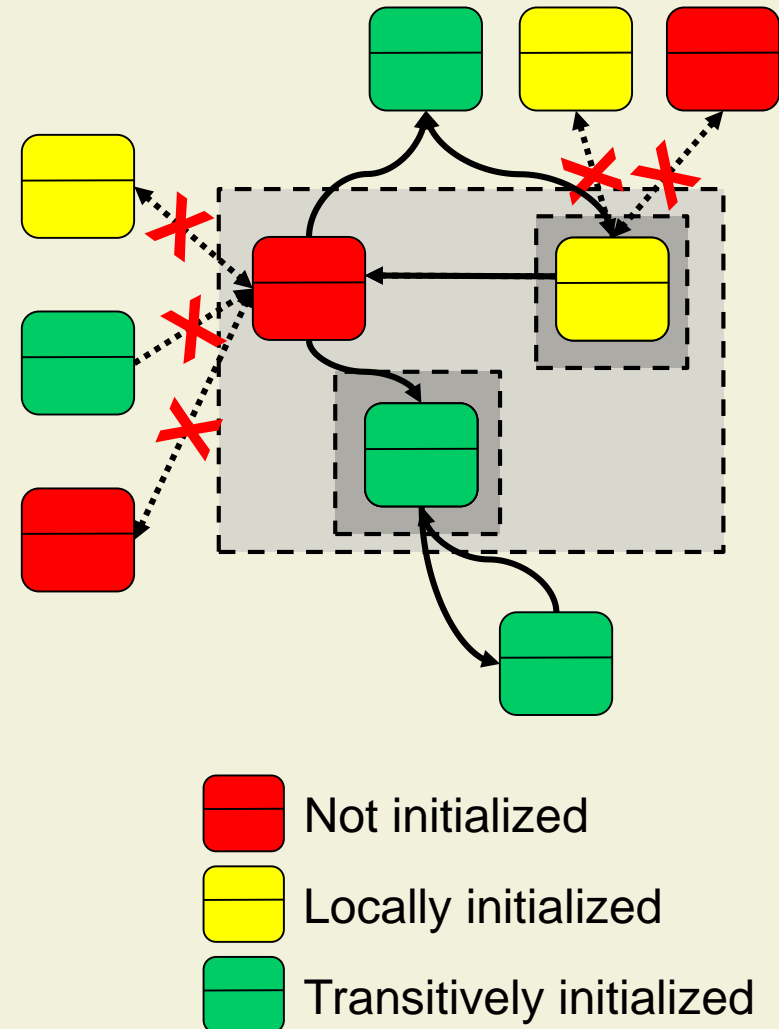
- Assumptions
 - new-expression takes **only committed arguments**
 - Nested new-expressions take **arbitrary arguments**



Object Construction

■ Assumptions

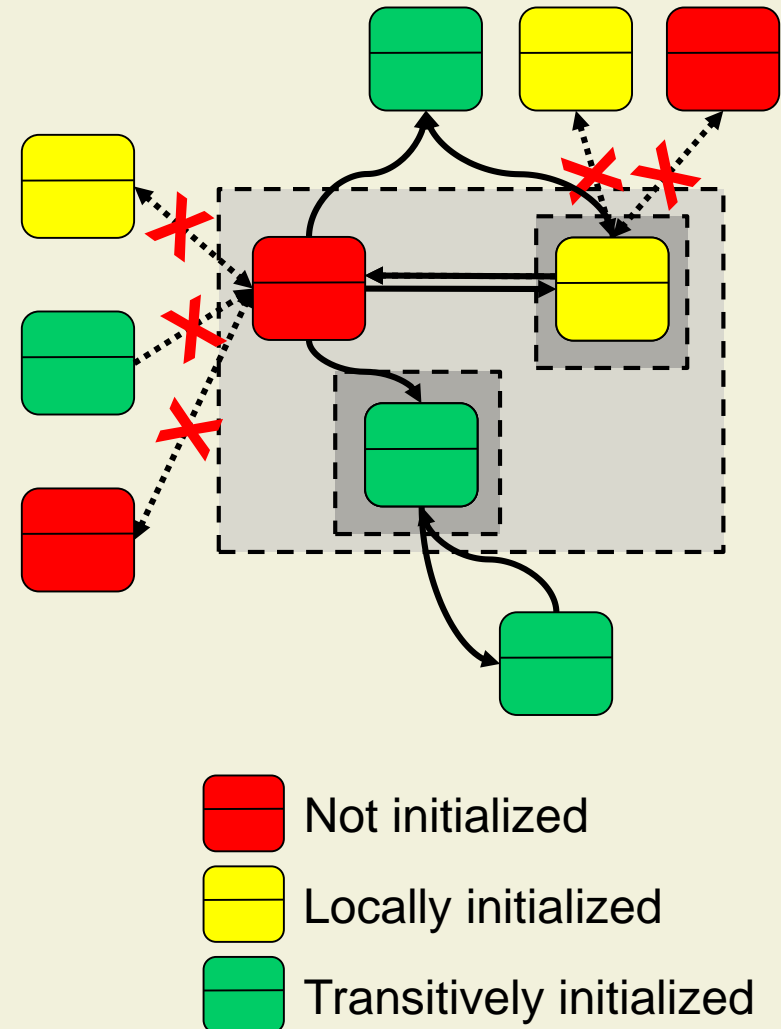
- new-expression takes **only committed arguments**
- Nested new-expressions take **arbitrary arguments**



Object Construction

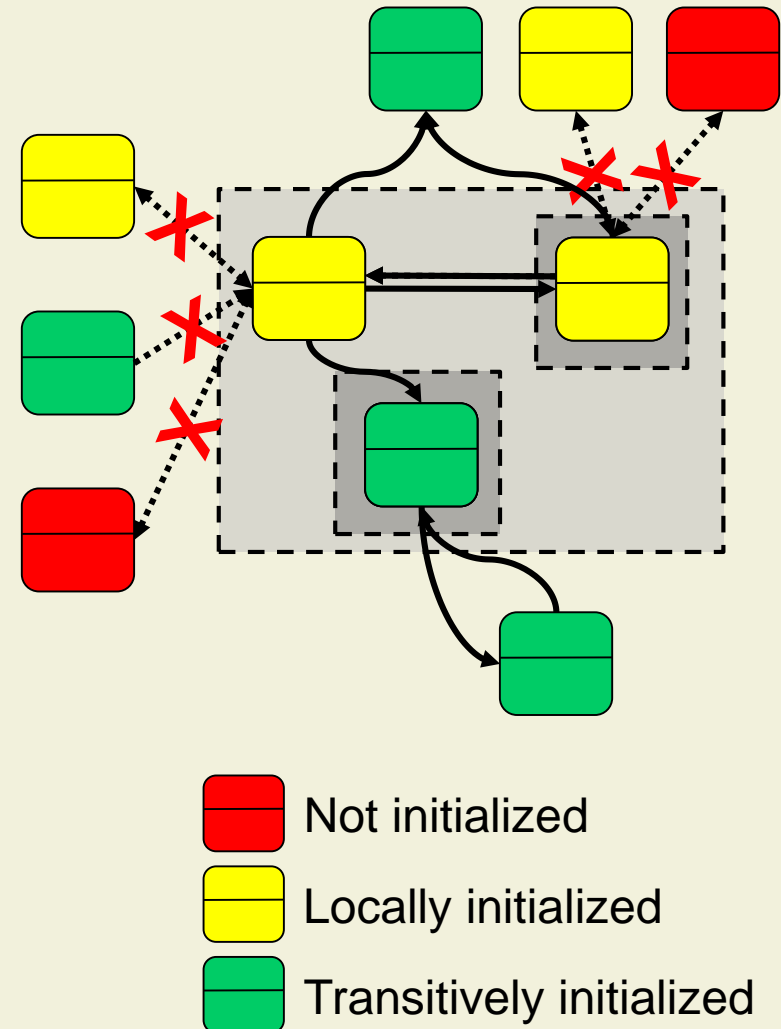
■ Assumptions

- new-expression takes **only committed arguments**
- Nested new-expressions take **arbitrary arguments**



Object Construction

- Assumptions
 - new-expression takes **only committed arguments**
 - Nested new-expressions take **arbitrary arguments**
- After new-expression
 - All new objects are **locally initialized**



- Assumptions

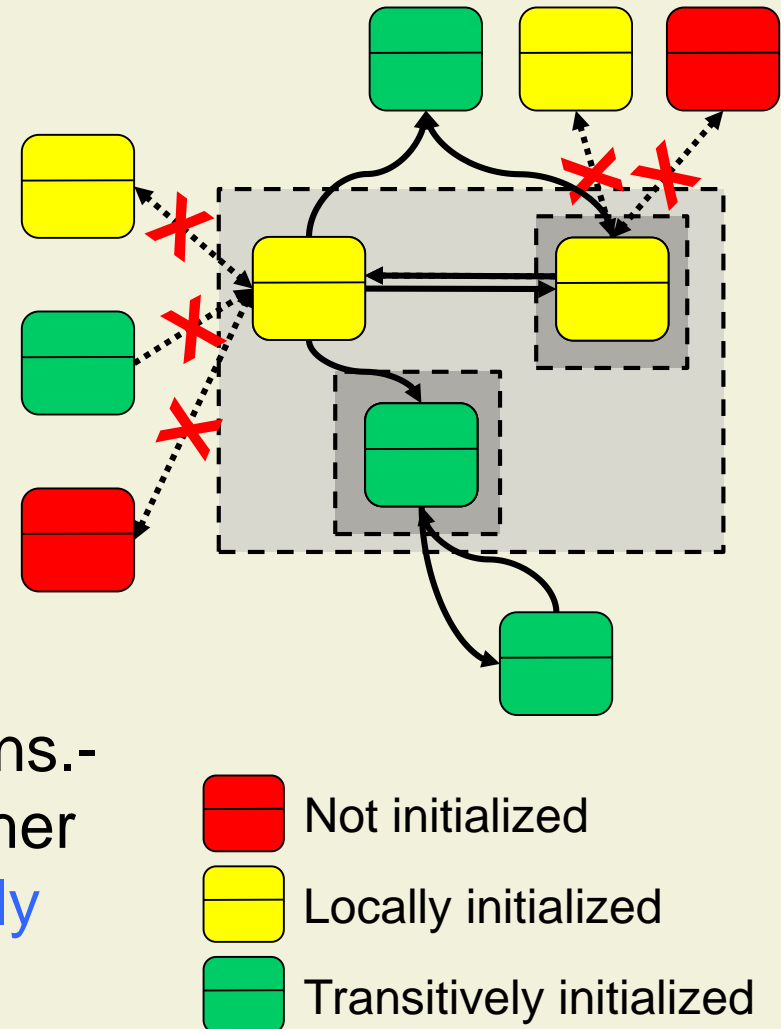
- After new-expression

-



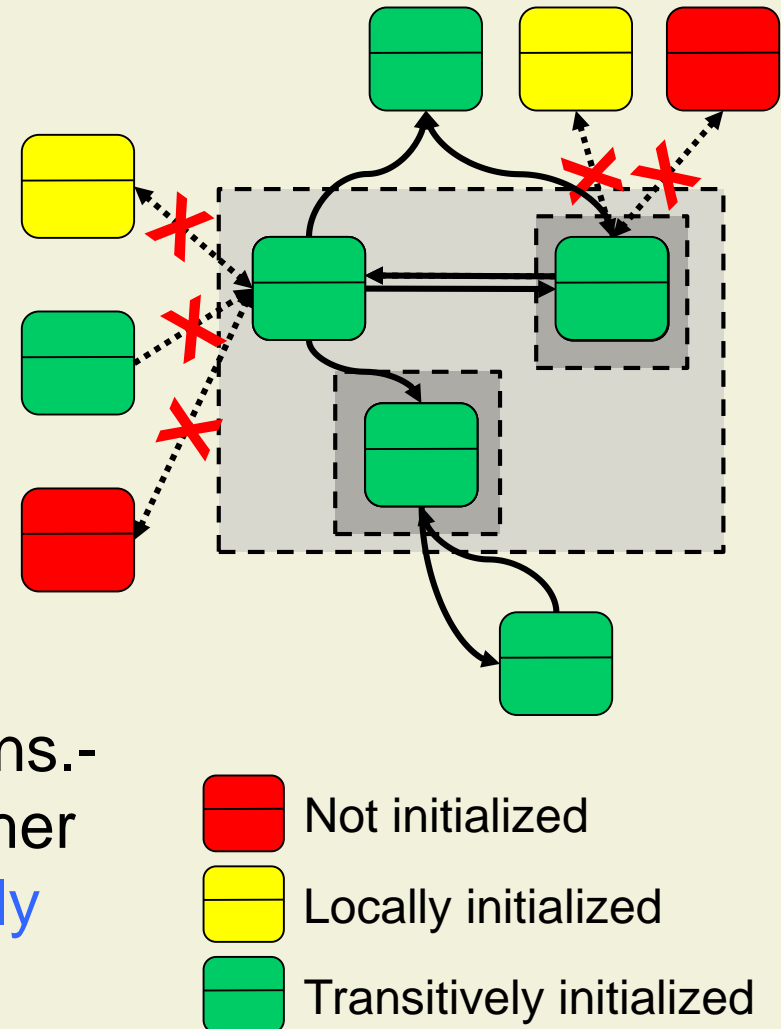
Object Construction

- Assumptions
 - new-expression takes **only committed arguments**
 - Nested new-expressions take **arbitrary arguments**
- After new-expression
 - All new objects are **locally initialized**
 - New objects reference only trans.-initialized objects and each other
 - **All** new objects are **transitively initialized**

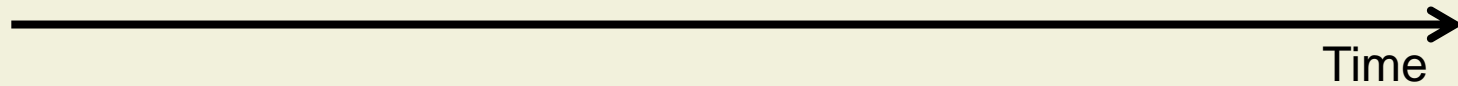


Object Construction

- Assumptions
 - new-expression takes **only committed arguments**
 - Nested new-expressions take **arbitrary arguments**
- After new-expression
 - All new objects are **locally initialized**
 - New objects reference only trans.-initialized objects and each other
 - **All** new objects are **transitively initialized**



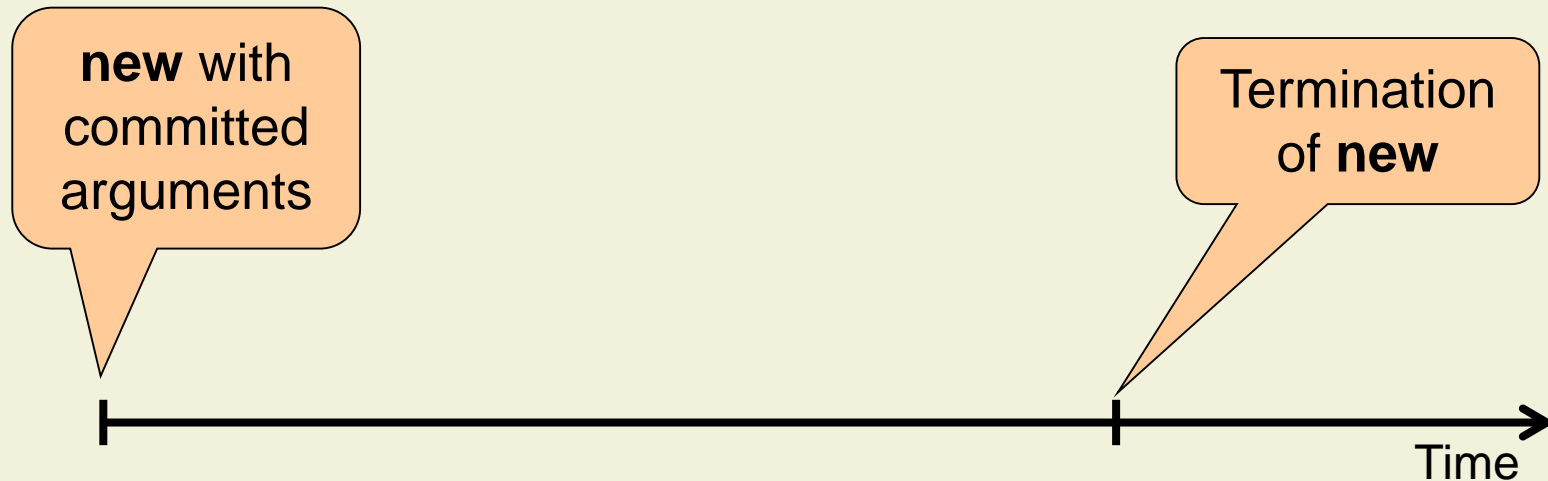
Completing Object Construction



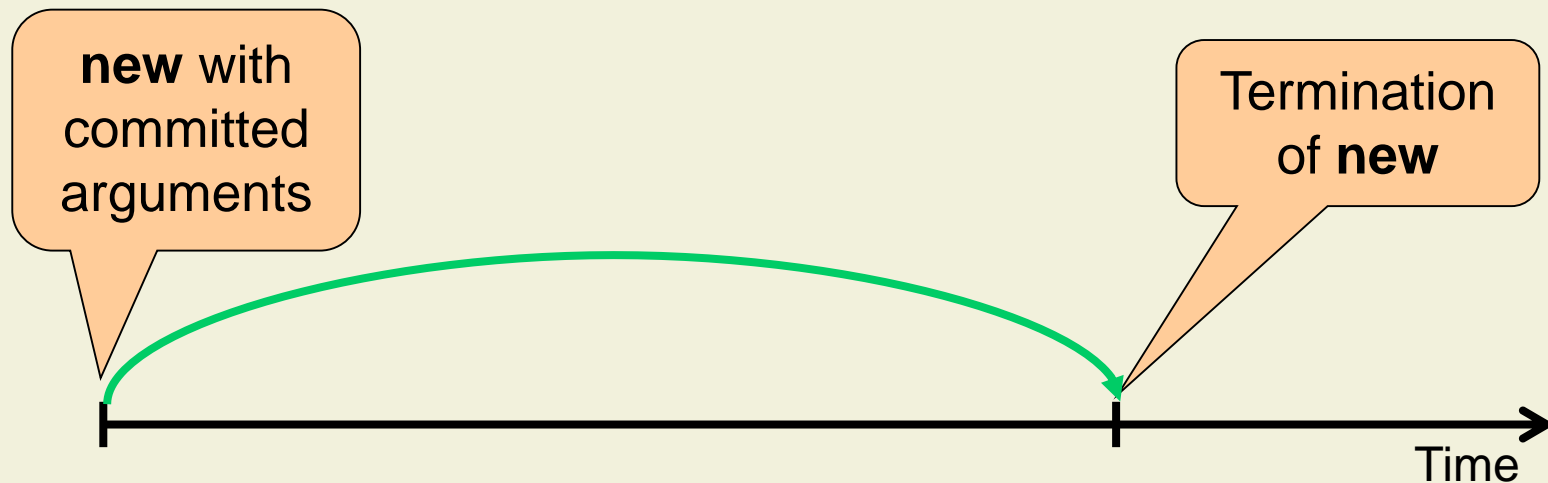
Completing Object Construction



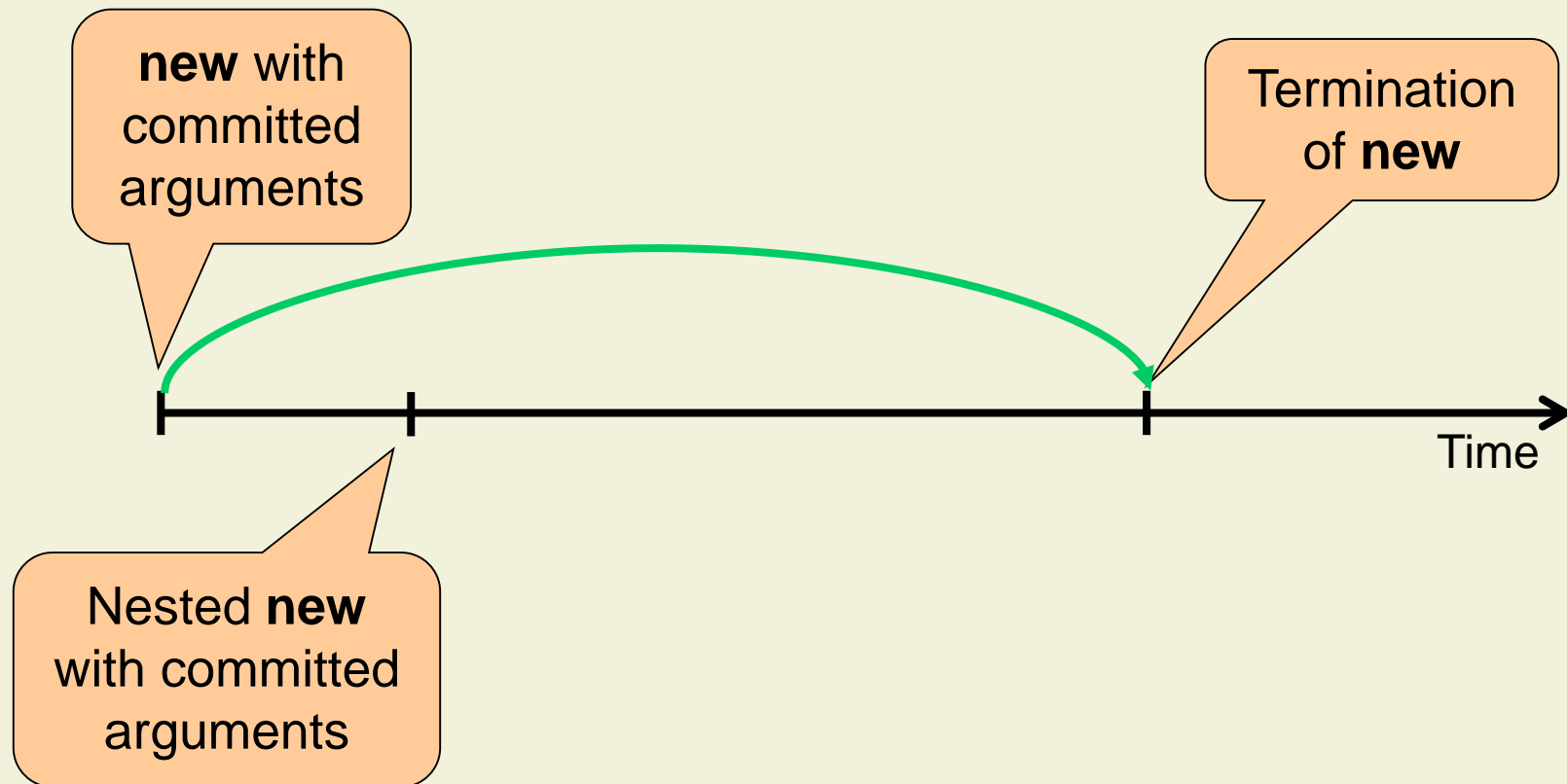
Completing Object Construction



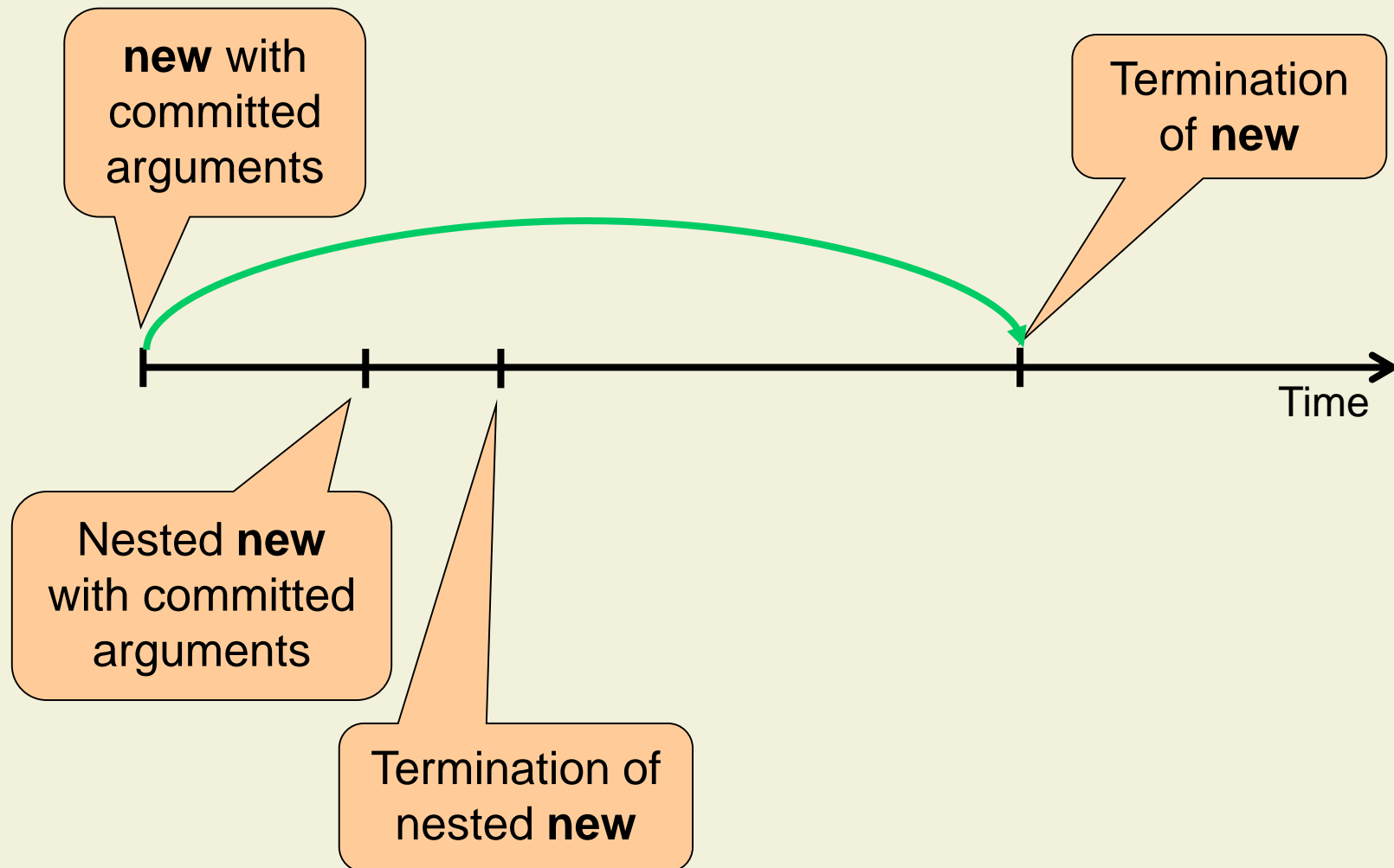
Completing Object Construction



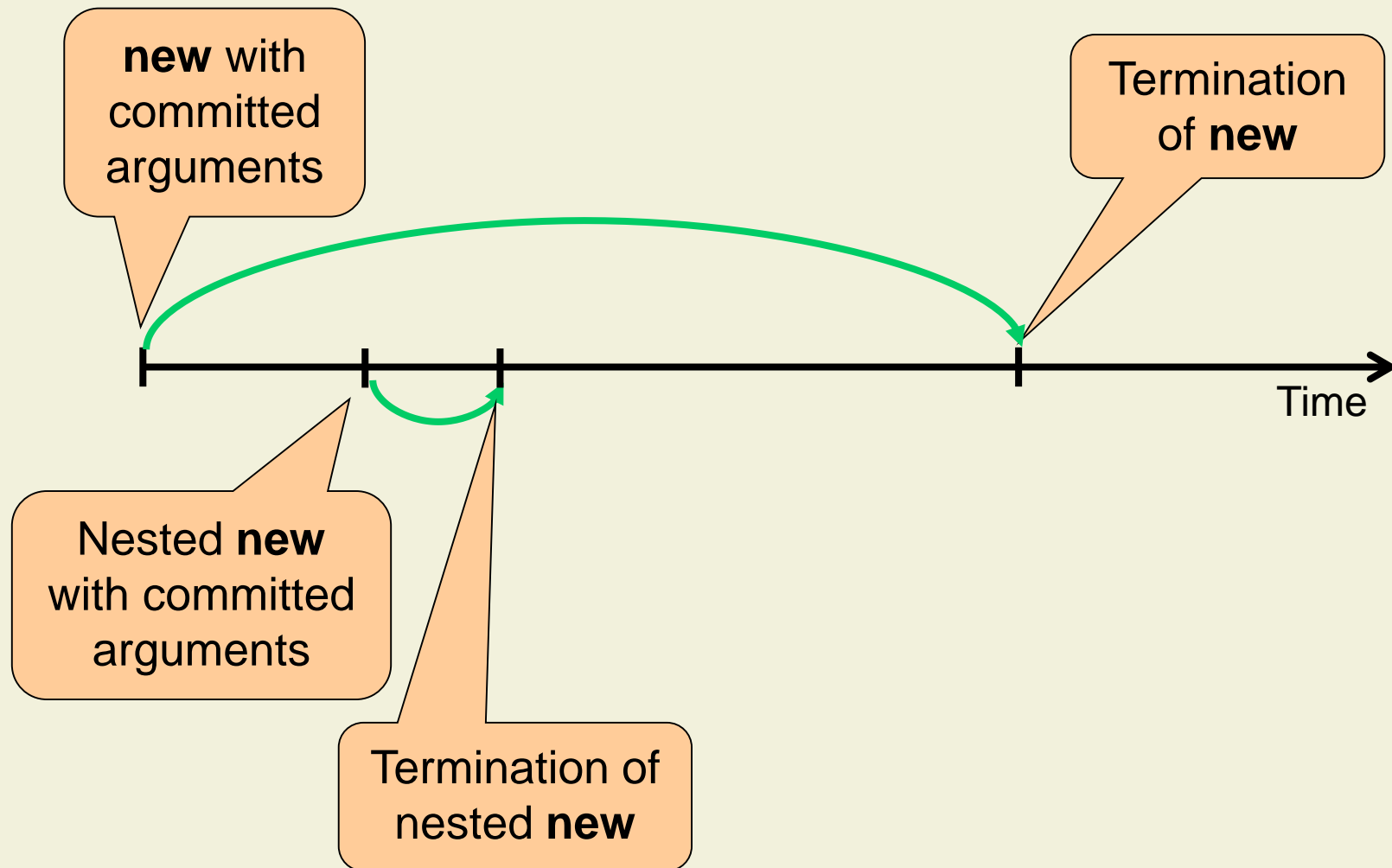
Completing Object Construction



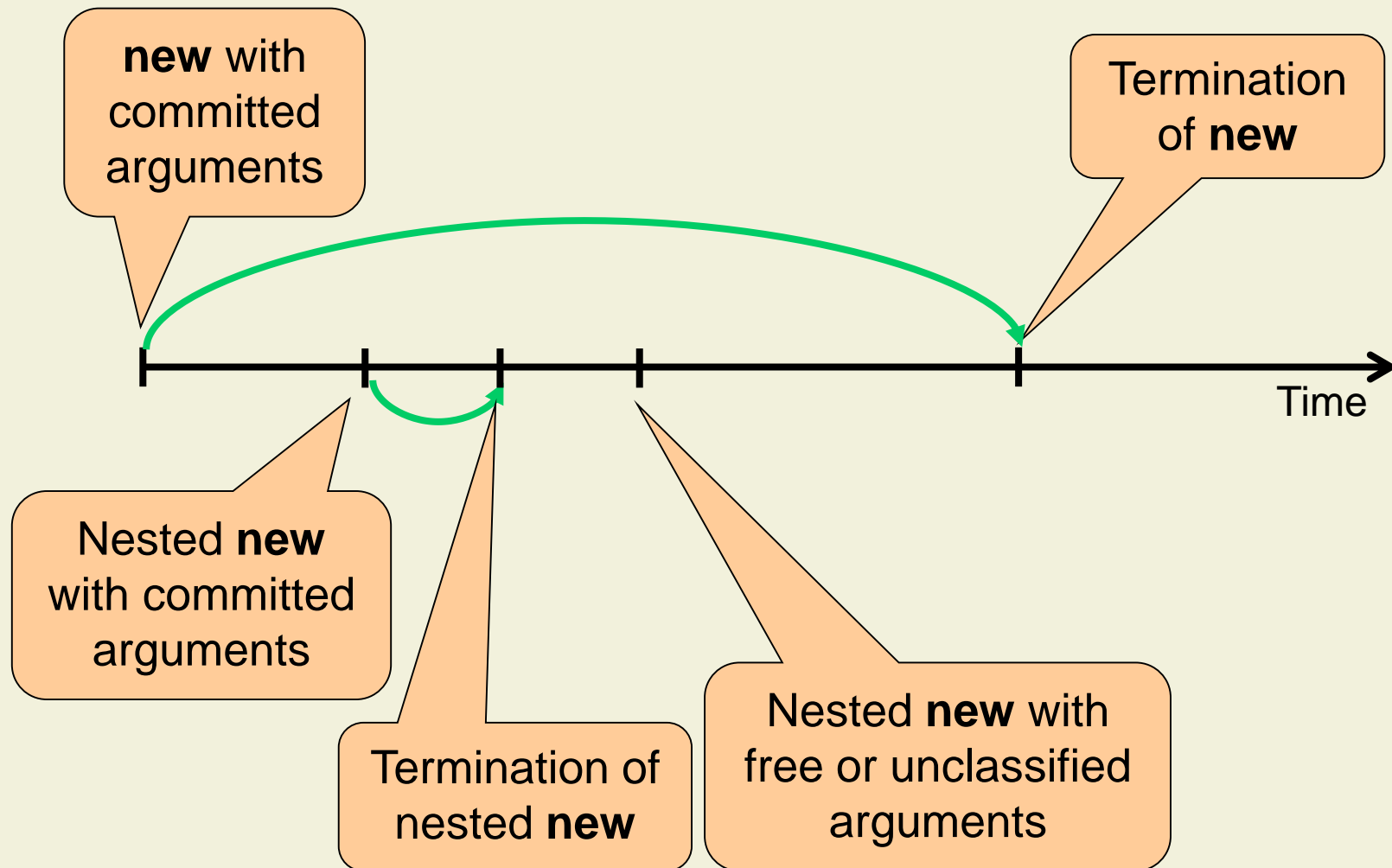
Completing Object Construction



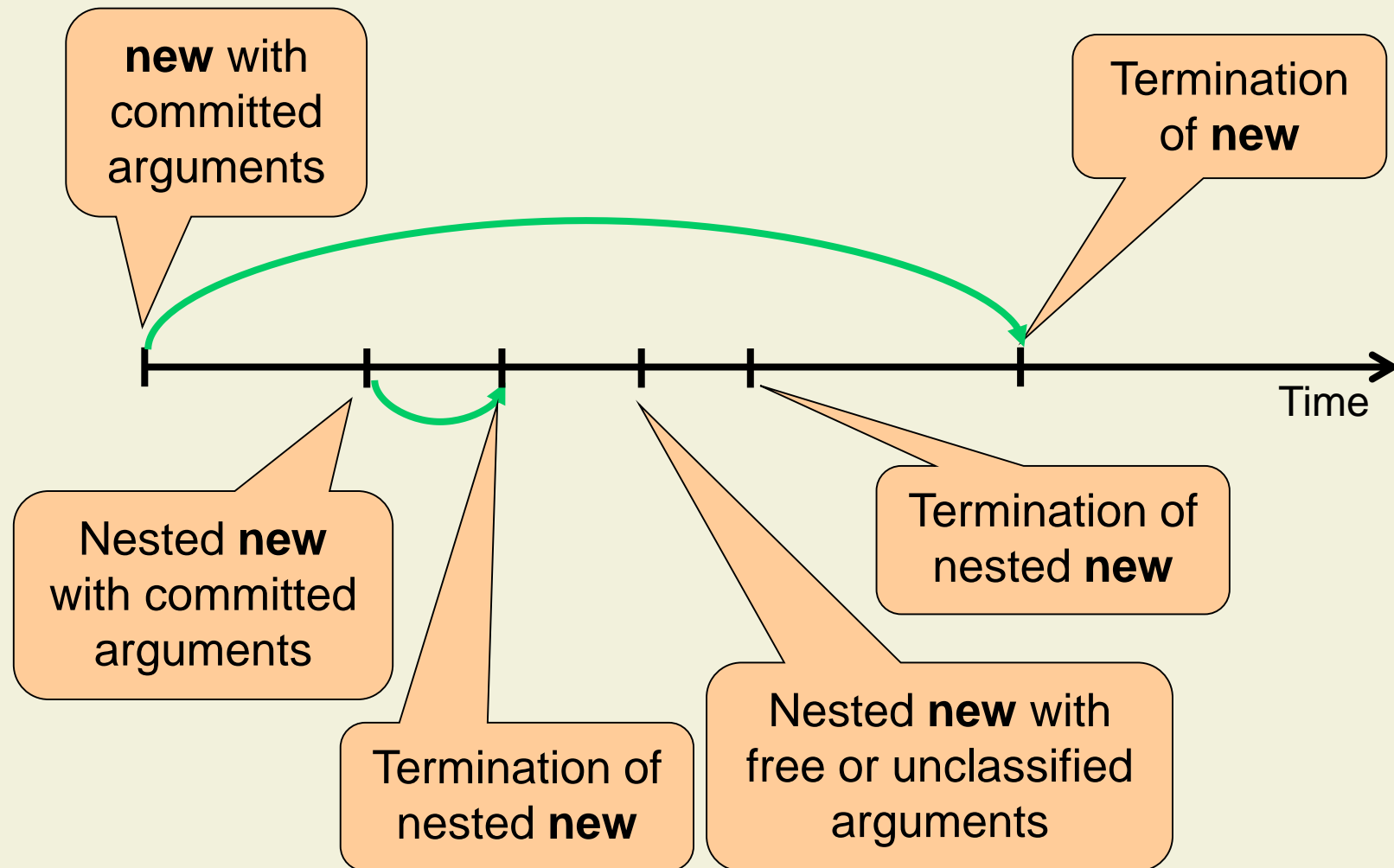
Completing Object Construction



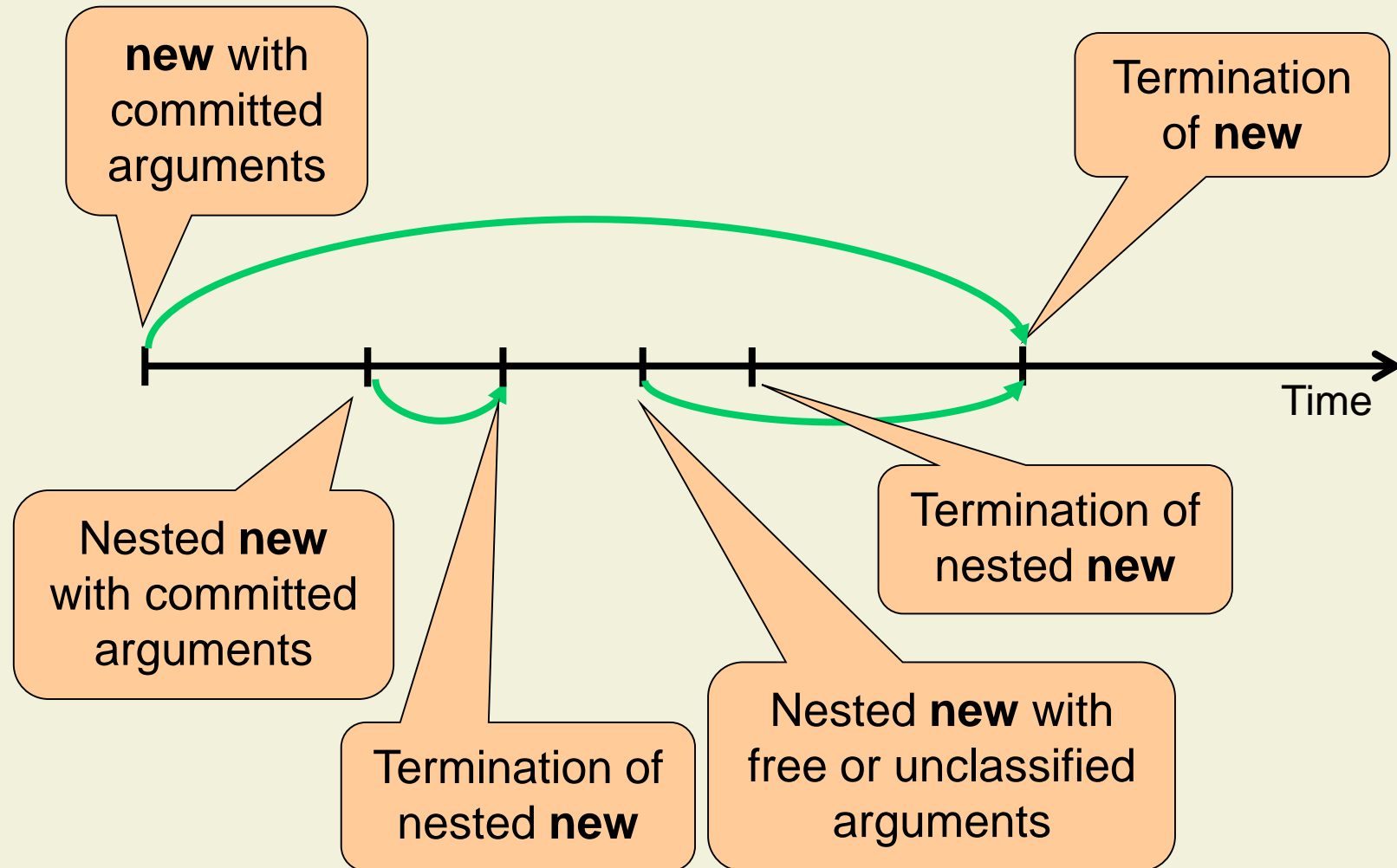
Completing Object Construction



Completing Object Construction



Completing Object Construction



Type Rules: new-Expressions

- An expression **new** C(e_i) is well-typed if
 - All e_i are well-typed
 - Class C contains a constructor with suitable parameter types
- The type of **new** C(e_i) is
 - committed C! if the static types of all e_i are committed
 - free C! otherwise

Type Rules: new-Expressions

- An expression **new** C(e_i) is well-typed if
 - All e_i are well-typed
 - Class C contains a constructor with suitable parameter types
- The type of **new** C(e_i) is
 - committed C! if the **static types of all e_i are committed**
 - free C! otherwise

```
class Demo {  
  C! myC;  
  Demo( ) {  
    free C! c = new C( this );  
    c.foo( );  
    myC = c;  
  }  
}
```


Cyclic Structures: Example

```
class List {  
    List! next; // cyclic  
  
    List( int n ) {  
        if( n == 1 )  
            next = this;  
        else  
            next = new List( this, n );  
    }  
  
    List( free List! last, int n ) {  
        if( n == 2 )  
            next = last;  
        else  
            next = new List( last, n-1 );  
    }  
}
```


Cyclic Structures: Example

```
l = new List( 3 );
```

```
class List {  
    List! next; // cyclic  
  
    List( int n ) {  
        if( n == 1 )  
            next = this;  
        else  
            next = new List( this, n );  
    }  
  
    List( free List! last, int n ) {  
        if( n == 2 )  
            next = last;  
        else  
            next = new List( last, n-1 );  
    }  
}
```


Cyclic Structures: Example

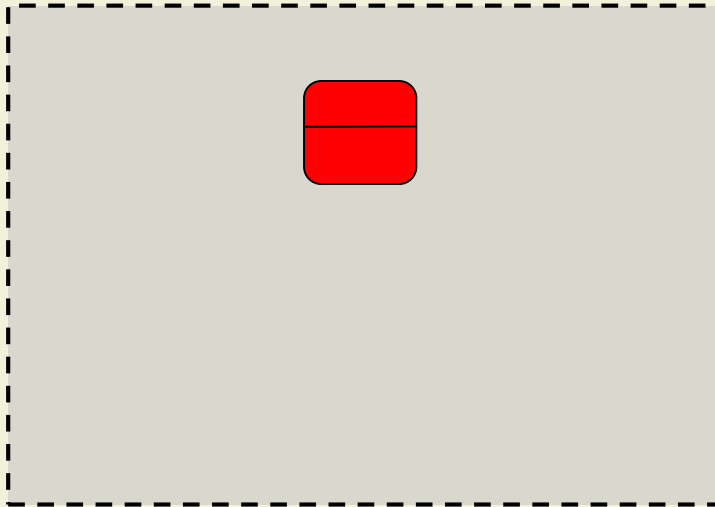
`l = new List(3);`



```
class List {  
    List! next; // cyclic  
  
    List( int n ) {  
        if( n == 1 )  
            next = this;  
        else  
            next = new List( this, n );  
    }  
  
    List( free List! last, int n ) {  
        if( n == 2 )  
            next = last;  
        else  
            next = new List( last, n-1 );  
    }  
}
```


Cyclic Structures: Example

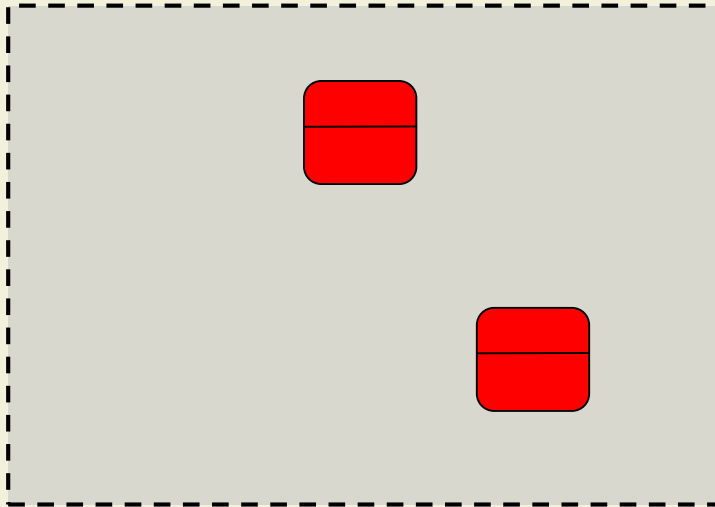
`l = new List(3);`



```
class List {  
    List! next; // cyclic  
  
    List( int n ) {  
        if( n == 1 )  
            next = this;  
        else  
            next = new List( this, n );  
    }  
  
    List( free List! last, int n ) {  
        if( n == 2 )  
            next = last;  
        else  
            next = new List( last, n-1 );  
    }  
}
```


Cyclic Structures: Example

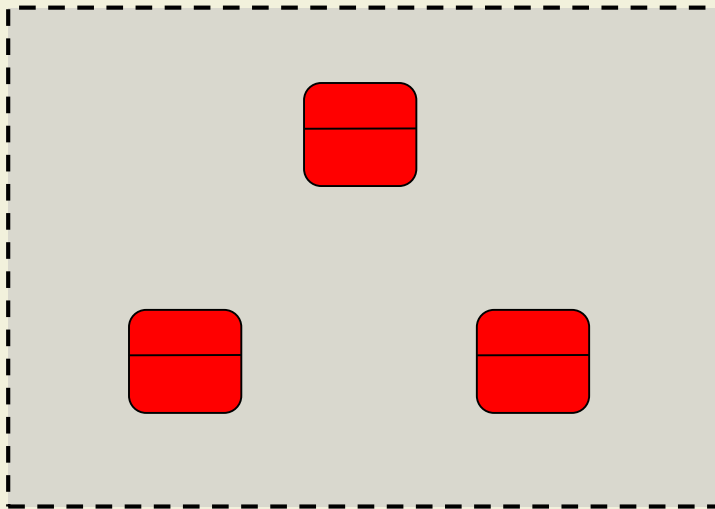
`l = new List(3);`



```
class List {  
    List! next; // cyclic  
  
    List( int n ) {  
        if( n == 1 )  
            next = this;  
        else  
            next = new List( this, n );  
    }  
  
    List( free List! last, int n ) {  
        if( n == 2 )  
            next = last;  
        else  
            next = new List( last, n-1 );  
    }  
}
```


Cyclic Structures: Example

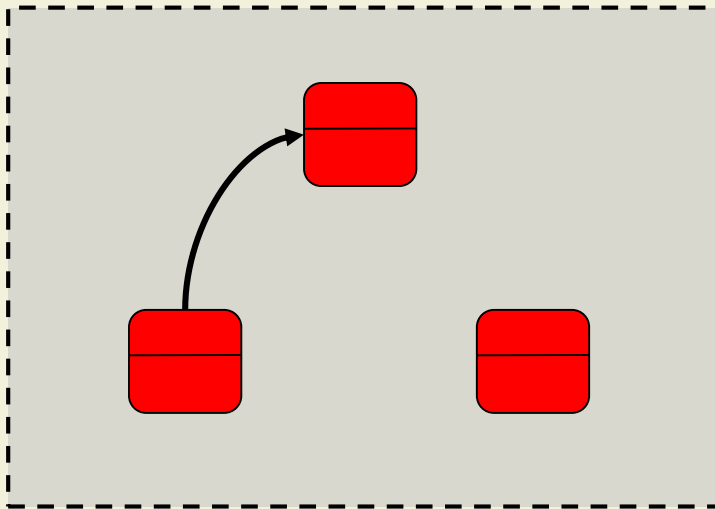
`l = new List(3);`



```
class List {  
    List! next; // cyclic  
  
    List( int n ) {  
        if( n == 1 )  
            next = this;  
        else  
            next = new List( this, n );  
    }  
  
    List( free List! last, int n ) {  
        if( n == 2 )  
            next = last;  
        else  
            next = new List( last, n-1 );  
    }  
}
```


Cyclic Structures: Example

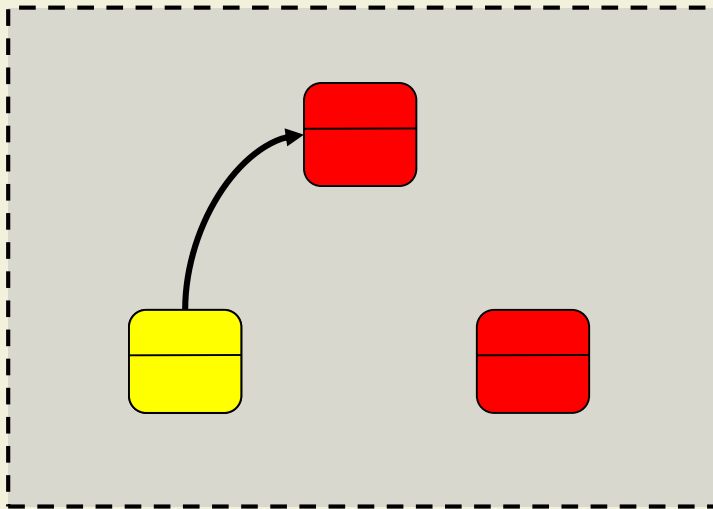
```
l = new List( 3 );
```



```
class List {  
    List! next; // cyclic  
  
    List( int n ) {  
        if( n == 1 )  
            next = this;  
        else  
            next = new List( this, n );  
    }  
  
    List( free List! last, int n ) {  
        if( n == 2 )  
            next = last;  
        else  
            next = new List( last, n-1 );  
    }  
}
```


Cyclic Structures: Example

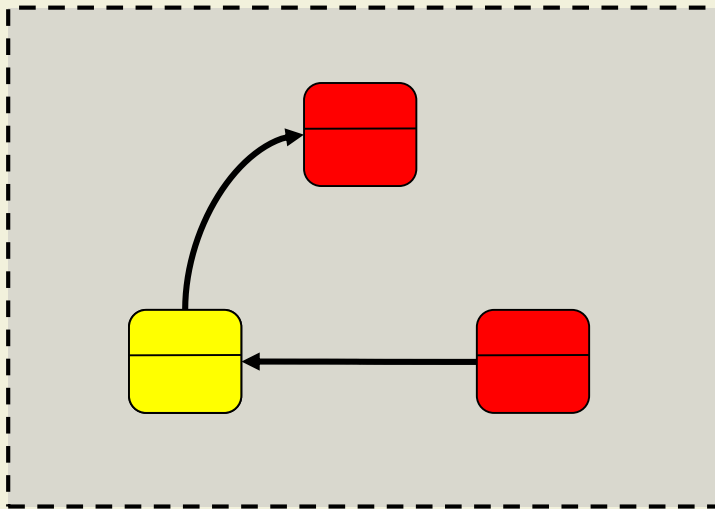
```
l = new List( 3 );
```



```
class List {  
    List! next; // cyclic  
  
    List( int n ) {  
        if( n == 1 )  
            next = this;  
        else  
            next = new List( this, n );  
    }  
  
    List( free List! last, int n ) {  
        if( n == 2 )  
            next = last;  
        else  
            next = new List( last, n-1 );  
    }  
}
```


Cyclic Structures: Example

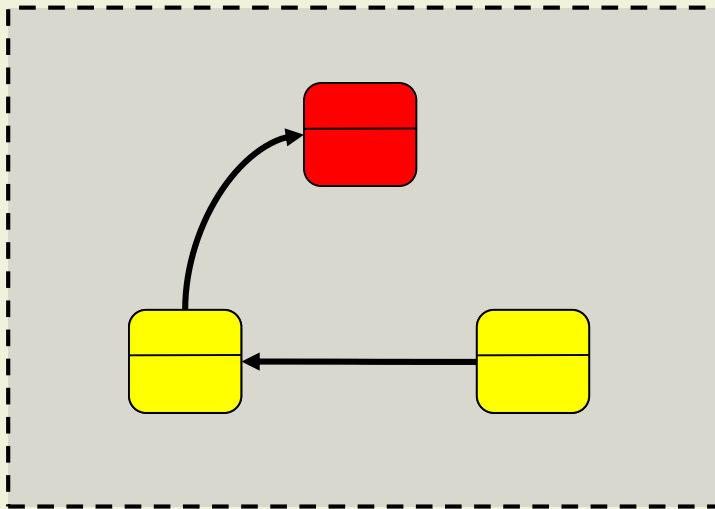
`l = new List(3);`



```
class List {  
    List! next; // cyclic  
  
    List( int n ) {  
        if( n == 1 )  
            next = this;  
        else  
            next = new List( this, n );  
    }  
  
    List( free List! last, int n ) {  
        if( n == 2 )  
            next = last;  
        else  
            next = new List( last, n-1 );  
    }  
}
```


Cyclic Structures: Example

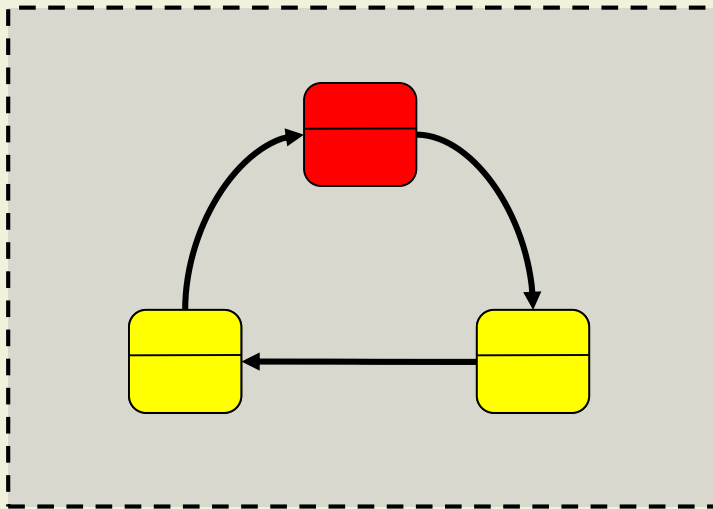
```
l = new List( 3 );
```



```
class List {  
    List! next; // cyclic  
  
    List( int n ) {  
        if( n == 1 )  
            next = this;  
        else  
            next = new List( this, n );  
    }  
  
    List( free List! last, int n ) {  
        if( n == 2 )  
            next = last;  
        else  
            next = new List( last, n-1 );  
    }  
}
```


Cyclic Structures: Example

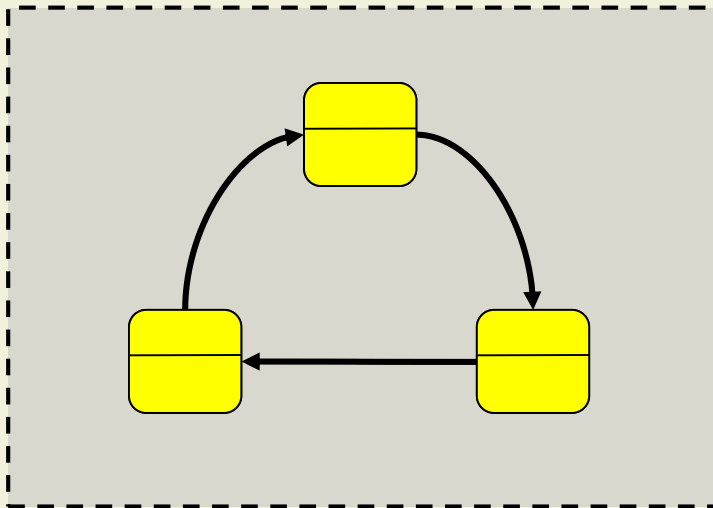
`l = new List(3);`



```
class List {  
  List! next; // cyclic  
  
  List( int n ) {  
    if( n == 1 )  
      next = this;  
    else  
      next = new List( this, n );  
  }  
  
  List( free List! last, int n ) {  
    if( n == 2 )  
      next = last;  
    else  
      next = new List( last, n-1 );  
  }  
}
```


Cyclic Structures: Example

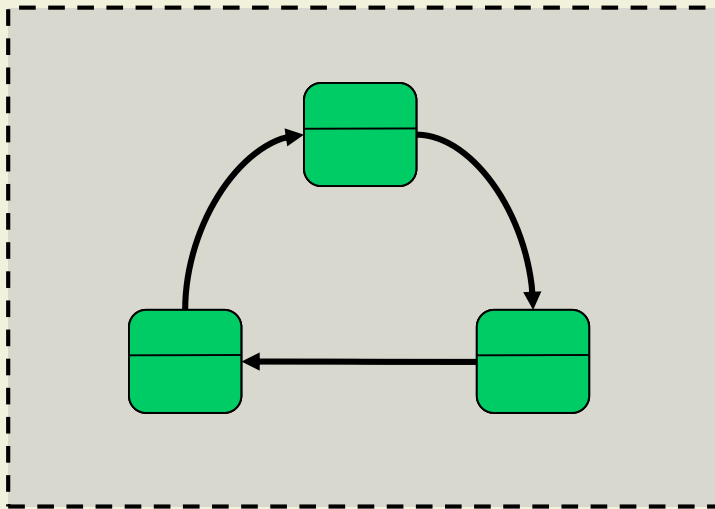
```
l = new List( 3 );
```



```
class List {  
  List! next; // cyclic  
  
  List( int n ) {  
    if( n == 1 )  
      next = this;  
    else  
      next = new List( this, n );  
  }  
  
  List( free List! last, int n ) {  
    if( n == 2 )  
      next = last;  
    else  
      next = new List( last, n-1 );  
  }  
}
```


Cyclic Structures: Example

`l = new List(3);`



```
class List {  
    List! next; // cyclic  
  
    List( int n ) {  
        if( n == 1 )  
            next = this;  
        else  
            next = new List( this, n );  
    }  
  
    List( free List! last, int n ) {  
        if( n == 2 )  
            next = last;  
        else  
            next = new List( last, n-1 );  
    }  
}
```


Problem 1: Method Calls Revisited

```
class Demo {  
    Vector! cache;  
  
    Demo( ) {  
        int size = optimalSize( );  
        cache = new Vector( size );  
    }  
  
    int free optimalSize( ) {  
        return 16;  
    }  
}
```

Called from
constructor

```
class Sub extends Demo {  
    Vector! data;  
  
    Sub( Vector! d ) {  
        data = d.clone( );  
    }  
  
    int free optimalSize( ) {  
        return this.data.size( ) * 2;  
    }  
}
```

Compile-
time error

```
Vector! v = new Vector( );  
Sub! s = new Sub( v );
```


Problem 2: Call-backs Revisited

```
class Demo implements Observer {
    static Subject! subject;

    Demo( ) {
        subject.register( this );
    }

    void free update( ... ) { }
}
```

Called on free
reference

```
class Subject {
    void register( free Observer! o ) {
        ...
        o.update( ... );
    }
}
```

```
class Sub extends Demo {
    Vector! data;

    Sub( Vector! d ) { data = d.clone( ); }

    void free update(
        { ... this.data.size( )
    }
}
```

Compile-
time error

```
Vector! v = new Vector( );
Sub! s = new Sub( v );
```


Problem 3: Escaping via Calls Revisited

```
class Demo implements Observer {
    static Subject! subject;

    Demo( ) {
        subject.register( this );
    }

    void update( ... ) { }
}
```

```
class Sub extends Demo {
    Vector! data;

    Sub( Vector! d ) { data = d.clone( ); }
    void update( ... ) { ... data.size( ) ... }
}
```

```
class Subject extends Thread {
    List<Observer!>! list;

    void register( free Observer! o )
    { list.add( o ); }

    void run( ) {
        while( true
            if( sensor( ... ) )
                for( Observer! o: list )
                    o.update( ... );
        }
    }
    ...
}
```

add requires committed argument

Problem 4: Escaping via Fields Revisited

```
class Node {  
    Node! next; // a cyclic list  
    Process! proc;  
  
    Node( Node! after, Process! p ) {  
        this.next = after.next;  
        after.next = this;  
        proc = p;  
    }  
}
```

New node cannot
be inserted before
construction is
complete

```
class Scheduler extends Thread {  
    Node! current;  
  
    void run( ) {  
        while( true ) {  
            current.proc.preempt( );  
            current = current.next;  
            current.proc.resume( );  
            Thread.sleep( 1000 );  
        }  
    }  
    ...  
}
```


Lazy Initialization

- Creating objects and initializing their fields is time consuming
 - Long application start-up time
- Lazy initialization: initialize fields when they are first used
 - Spreads initialization effort over longer time period

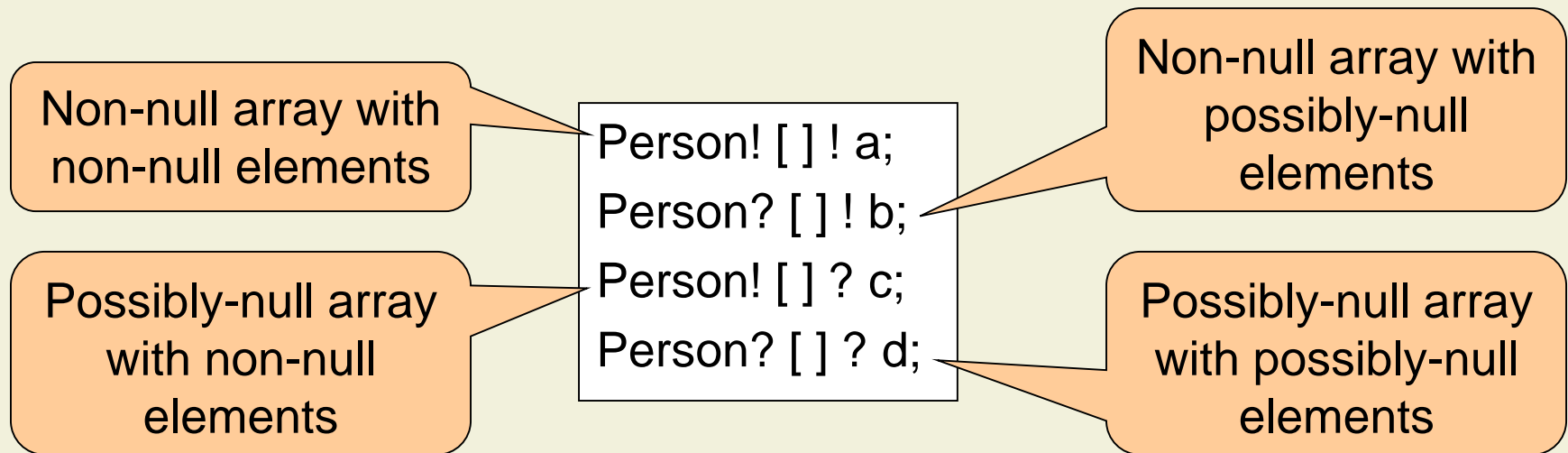
```
class Demo {  
    private Vector? data;  
  
    Demo() {  
        // do not init  
    }  
  
    public Vector! getData() {  
        Vector? d = data;  
        if( d == null ) {  
            d = new Vector(); data = d;  
        }  
        return d;  
    }  
}
```

Not initialized
by constructor

Clients get non-
null guarantee

Non-Null Arrays

- Arrays are objects whose fields are numbered
- An array type describes two kinds of references
 - The reference to the array object
 - The references to the array elements
 - Both can be non-null or possibly-null



Problems of Array Initialization

- Our solution for non-null fields does not work for non-null array elements
 - No constructor for arrays
 - Arrays are typically initialized using loops
 - Static analyses ignore loop conditions
- In general, definite assignment cannot be checked by compiler

```
class Demo {  
    String! [ ] s;  
  
    Demo( int l ) {  
        if( l % 2 == 1 )  
            l = l + 1;  
        s = new String! [ l ];  
  
        for( int i = 0; i < l / 2; i++ ) {  
            s[ i*2 ] = "Even";  
            s[ i*2 + 1 ] = "Odd";  
        }  
    }  
}
```

When do the elements have to contain non-null references?

Are all elements of s initialized?

Array Initialization: (Partial) Solutions

- Array initializers

```
String! [ ] ! s = { "array", "of", "non-null", "String" };
```

- Pre-filling the array

```
my_array: attached ARRAY [ attached STRING ]  
create my_array.make_filled ( " ", 1, l )
```

Eiffel

- Not clear why a default object is any better than **null**

- Run-time checks

```
free String! [ ] ! s = new String! [ l ];  
for( int i = 0; i < l / 2; i++ ) { /* as before */  
  NonNullType.AssertInitialized( s );
```

Changes type from
free to committed

Spec#

Summary

- Object initialization has to establish invariants
 - Non-nullness of fields is just an example
- General guidelines for writing constructors
 - Avoid calling dynamically-bound methods on **this**
 - Be careful when new object escapes from constructor
 - Be aware of subclass constructors that have not run yet
- Non-null types are available in Spec#
 - specsharp.codeplex.com

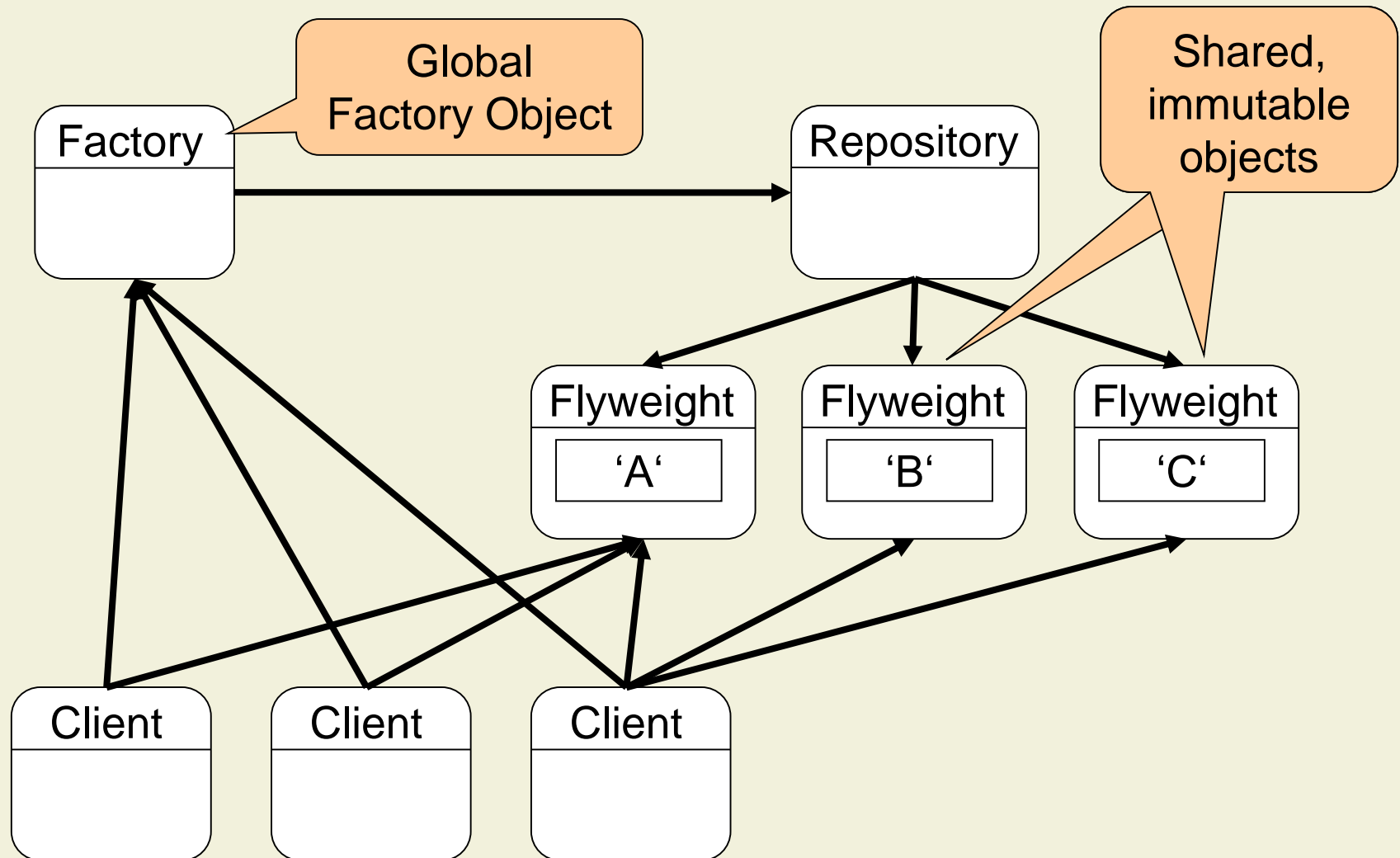
7. Initialization

7.1 Simple Non-Null Types

7.2 Object Initialization

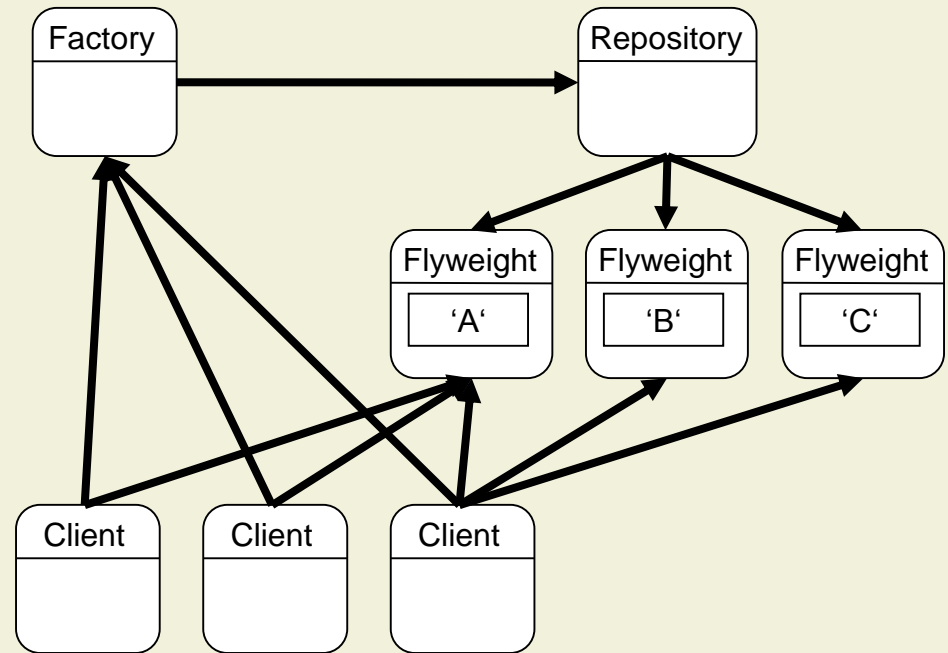
7.3 Initialization of Global Data

The Flyweight Pattern



Global Data

- Most software systems maintain global data
 - Factories
 - Caches
 - Flyweights
 - Singletons
- Main issues
 - How do clients access the global data?
 - How is the global data initialized?



Initialization of Globals: Design Goals

- Effectiveness
 - Ensure that global data is **initialized before first access**
 - Example: non-nullness
- Clarity
 - Initialization has a **clean semantics** and facilitates reasoning
- Laziness
 - Global data is **initialized lazily** to reduce start-up time

Solution 1: Global Vars and Init-Methods

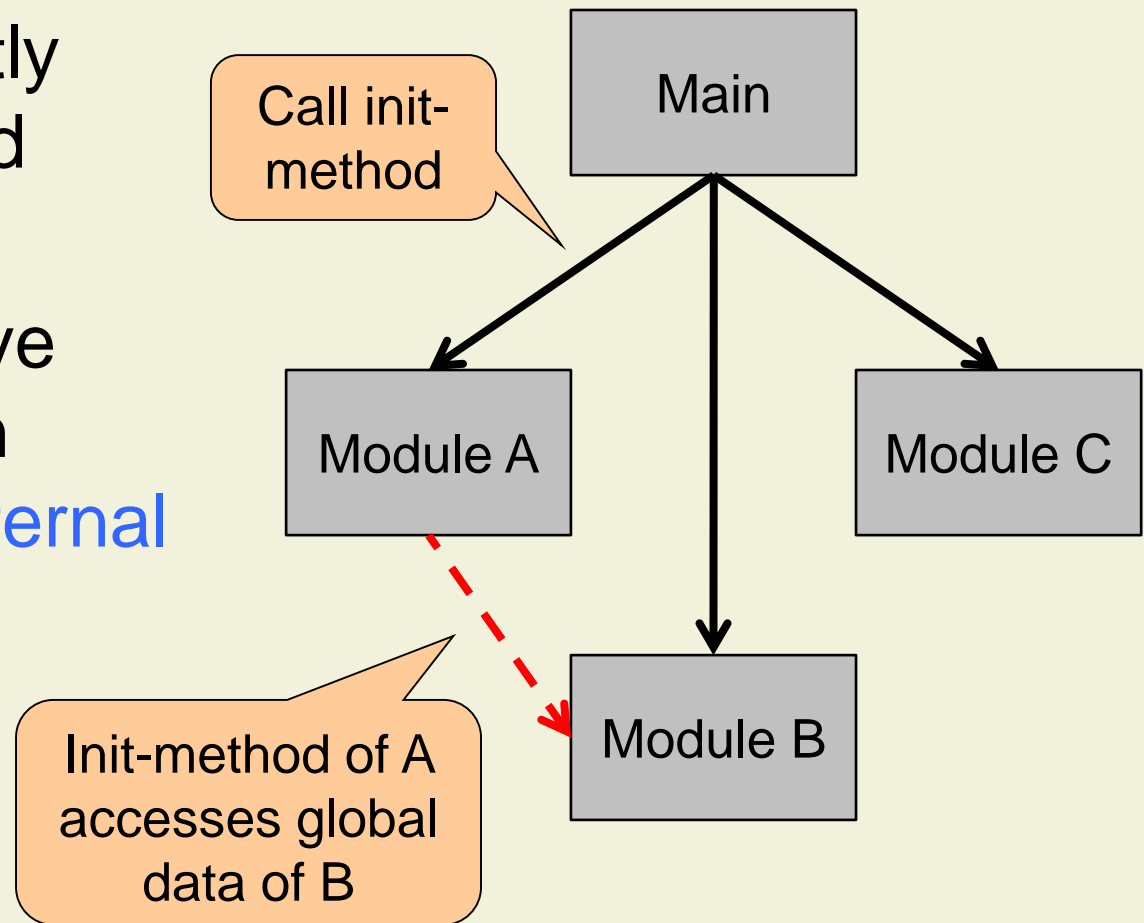
- **Global variables**
store references to
global data
- Initialization is done
by **explicit calls** to
init-methods

```
global Factory theFactory;  
  
void init( ) {  
    theFactory = new Factory( );  
}  
  
class Factory {  
    HashMap flyweights;  
  
    Flyweight create( Data d ) { ... }  
    ...  
}
```

```
Flyweight f = theFactory.create( ... );
```


Globals and Init-Methods: Dependencies

- Init-methods are called directly or indirectly from main-method
- To ensure effective initialization, main needs to know internal dependencies of modules



Globals and Init-Methods: Summary

- Effectiveness
 - Initialization order needs to be **coded manually**
 - Error-prone
- Clarity
 - Dependency information **compromises information hiding**
- Laziness
 - Needs to be **coded manually**

Variation: C++ Initializers

- Global variables can have initializers
- Initializers are executed before execution of main-method
 - No explicit calls needed
 - No support for lazy initialization
- Order of execution determined by order of appearance in the source code
 - Programmer has to manage dependencies

```
class Factory {  
    HashMap* flyweights;  
  
    Flyweight* create( Data* d ) { ... }  
  
    ...  
};  
  
Factory* theFactory = new Factory( );
```

C++

Solution 2: Static Fields and Initializers

- **Static fields** store references to global data
- Static initializers are executed by the system **immediately before a class is used**

```
class Factory {  
    static Factory theFactory;  
    HashMap flyweights;  
  
    static {  
        theFactory = new Factory( );  
    }  
  
    Flyweight create( Data d ) { ... }  
    ...  
}
```

Java

```
Factory o = Factory.theFactory;  
Flyweight f = o.create( ... );
```


Execution of Static Initializers

- A class C's static initializer runs **immediately before first**
 - Creation of a C-instance
 - Call to a static method of C
 - Access to a static field of Cand before static initializers of C's subclasses
- Initialization is done **lazily**
- System manages dependencies

```
class Factory {  
    static Factory theFactory;  
    HashMap flyweights;  
  
    static {  
        theFactory = new Factory( );  
    }  
  
    Flyweight create( Data d ) { ... }  
  
    ...  
}
```

Initialization triggered here

Java

```
Factory o = Factory.theFactory;  
Flyweight f = o.create( ... );
```


Static Initializers: Mutual Dependencies

```
class Debug {  
    static int session;  
    static Vector logfile;  
  
    static {  
        session = UniqueID.getID( );  
        logfile = new Vector( );  
    }  
  
    static void log( String msg ) {  
        logfile.add( msg );  
    }  
}
```

Initialize
UniqueID

Initialize
Debug

NullPointerException

```
class UniqueID {  
    static int next;  
  
    static {  
        next = 1;  
        Debug.log( "..." );  
    }  
  
    static int getID( ) {  
        return next++;  
    }  
}
```

Initialization
already in progress

```
Debug.log( "Start of program execution" );
```

Java

Static Initializers: Side Effects

- Static initializers may have **arbitrary side effects**
- Reasoning about programs with static initializers is **non-modular**
 - Need to know when initializers run

```
class C {  
    static int x;  
  
    ...  
}
```

```
class D {  
    static char y;  
  
    static { C.x = C.x + 1; }  
}
```

```
C.x = 0;  
D.y = '?';  
assert C.x == 0;
```


Static Initializers: Summary

- Effectiveness
 - Static initializers may be **interrupted**
 - **Reading un-initialized fields** is possible
- Clarity
 - Reasoning requires to **keep track** of which initializers have run already
 - **Side effects through implicit executions** of static initializers can be surprising
- Laziness
 - Static initializers are not called upfront (but also not as late as possible)

Scala's Singleton Objects

- Scala provides language support for **singletons**
 - Singleton objects may extend classes or traits
 - But they **cannot be specialized**
- Not every global object is a singleton
- Initialization is **defined by translation to Java**
 - Inherits all pros and cons of static initializers

```
object Factory {  
  val flyweights: HashMap[ ... ]  
  
  def  
  create( d: Data ): Flyweight =  
    ...  
  ...  
}
```

Scala

Solution 3: Eiffel's Once Methods

- Once methods are **executed only once**
- **Result** of first execution **is cached** and returned for subsequent calls

```
class FlyweightMgr
feature
  theFactory: Factory
  once
    create Result
  end
  ...
end
```

Eiffel

```
o := manager.theFactory
f := o.createFlyweight( ... )
```


Once Methods: Mutual Dependencies

- Mutual dependencies lead to recursive calls
- Recursive calls return the **current value of Result**
 - Typically not a meaningful value

```
factorial ( i: INTEGER ): INTEGER
  require 0 <= i
  once
    if i <= 1 then Result := 1
    else
      Result := i * factorial ( i - 1 )
    end
  end
```

Eiffel

```
check factorial( 3 ) = 0 end
check factorial( 30 ) = 0 end
```


Once Methods: Parameters

- Arguments to once methods are used for the first execution
- Arguments to subsequent calls are ignored

```
factorial ( i: INTEGER ): INTEGER
  require 0 <= i
  once
    if i <= 1 then Result := 1
    else
      Result := i * factorial ( i - 1 )
    end
  end
```

Eiffel

```
check factorial( 3 ) = 0 end
check factorial( 30 ) = 0 end
check factorial( 1 ) = 0 end
```

```
check factorial( 1 ) = 1 end
check factorial( 3 ) = 1 end
check factorial( 30 ) = 1 end
```


Once Methods: Summary

- Effectiveness
 - Mutual dependencies lead to recursive calls
 - Reading un-initialized fields is possible
- Clarity
 - Reasoning requires to keep track of which once methods have run already (use of arguments, side effects)
- Laziness
 - Once methods are executed only when result is needed (as late as possible)

Initialization of Global Data: Summary

- No solution ensures that global data is initialized before it is accessed
 - How to establish invariants over global data?
 - For instance, solutions would not be suitable to ensure that global non-null variables have non-null values

- No solution handles mutual dependencies

References

- Manuel Fähndrich and K. Rustan M. Leino: *Declaring and Checking Non-Null Types in an Object-Oriented Language*. OOPSLA 2003
- Alexander J. Summers and Peter Müller: *Freedom Before Commitment – A Lightweight Type System for Object Initialisation*. OOPSLA 2011