Secure the Clones:

Static Enforcement of Policies for Secure Object Copying

Thomas Jensen and David Pichardie
INRIA Rennes - Bretagne Atlantique
FRANCE
thomas.jensen@inria.fr

OWASP

The OWASP Foundation
http://www.owasp.org

Accepting objects from strangers

Accepting objects from strangers

```
package TheGood;

class StoreMyHoneyPots {

  void main(String[] args) {
```

Accepting objects from strangers

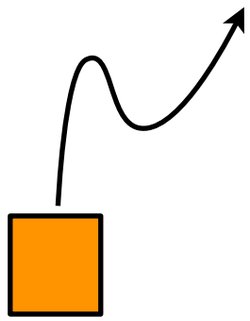
```
package TheGood;

class StoreMyHoneyPots {

  void main(String[] args) {

    BookShelf bs = new BookShelf();
```

Accepting objects from strangers

```
package TheGood;

class StoreMyHoneyPots {

  void main(String[] args) {

    BookShelf bs = new BookShelf();
```

```
package TheBad;

class BookShelf {

  private Object[] content;

  public BookShelf()
    { ...;

  public add(Object o) { ... }

  public BookShelf clone() { ... }


}
```

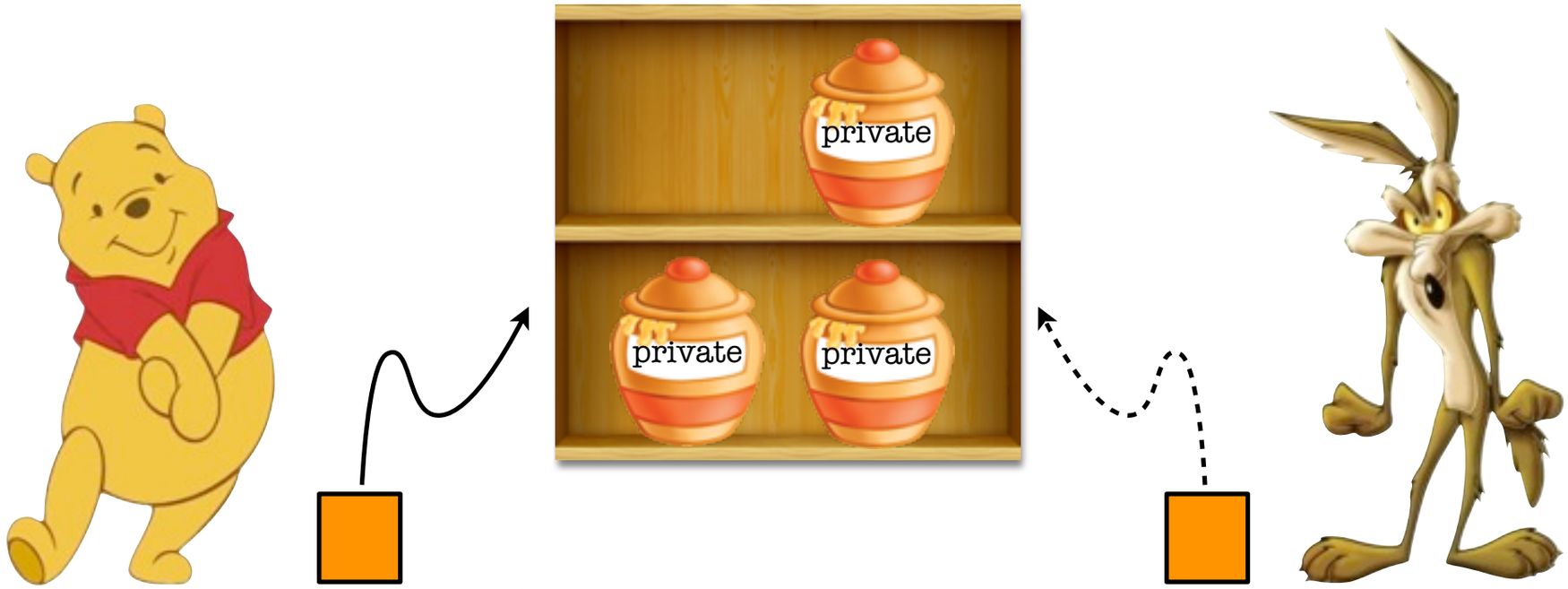
Accepting objects from strangers



```
package TheGood;

class StoreMyHoneyPots {

  void main(String[] args) {

    BookShelf bs = new BookShelf();



    bs.add(pot1);
    bs.add(pot2);
    bs.add(pot3);

  }
}
```

```
package TheBad;

class BookShelf {

  private Object[] content;

  public BookShelf()
    { ...;

  public add(Object o) { ... }

  public BookShelf clone() { ... }



}
```

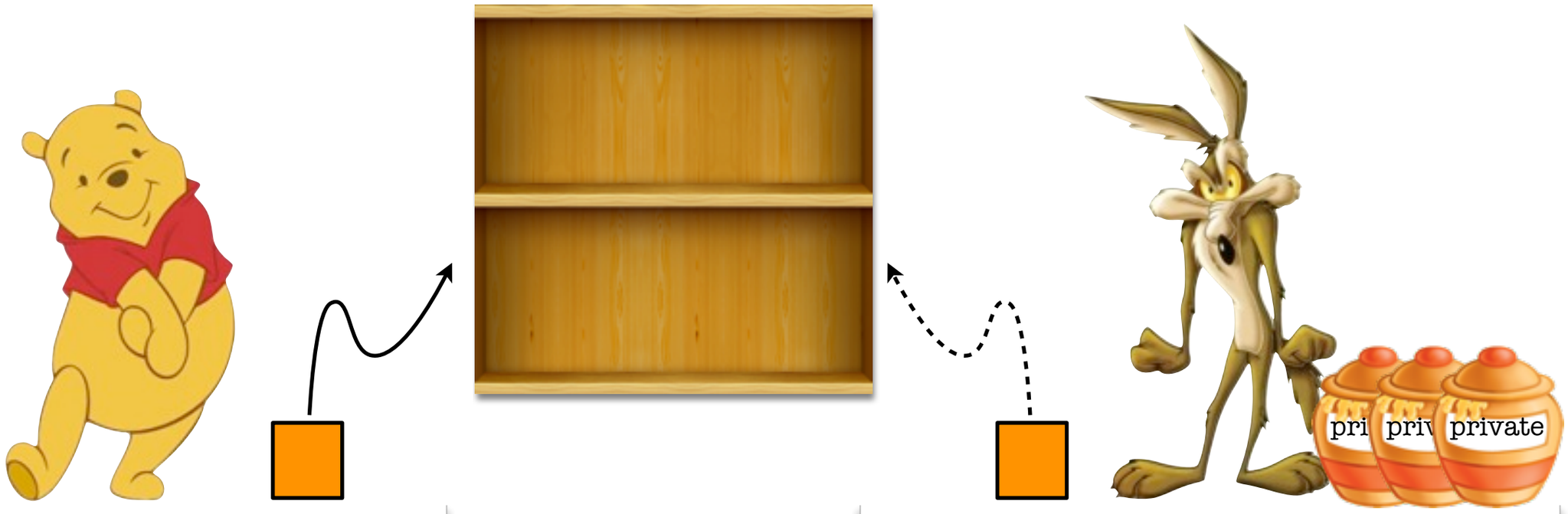
Accepting objects from strangers




```
package TheGood;

class StoreMyHoneyPots {

  void main(String[] args) {

    BookShelf bs = new BookShelf();



    bs.add(pot1);
    bs.add(pot2);
    bs.add(pot3);

  }
}
```

```
package TheBad;

class BookShelf {

  private Object[] content;

  public BookShelf()
    { ...; backdoor = this; }

  public add(Object o) { ... }

  public BookShelf clone() { ... }



}
```

Accepting objects from strangers




```
package TheGood;

class StoreMyHoneyPots {

  void main(String[] args) {

    BookShelf bs = new BookShelf();



    bs.add(pot1);
    bs.add(pot2);
    bs.add(pot3);

  }
}
```

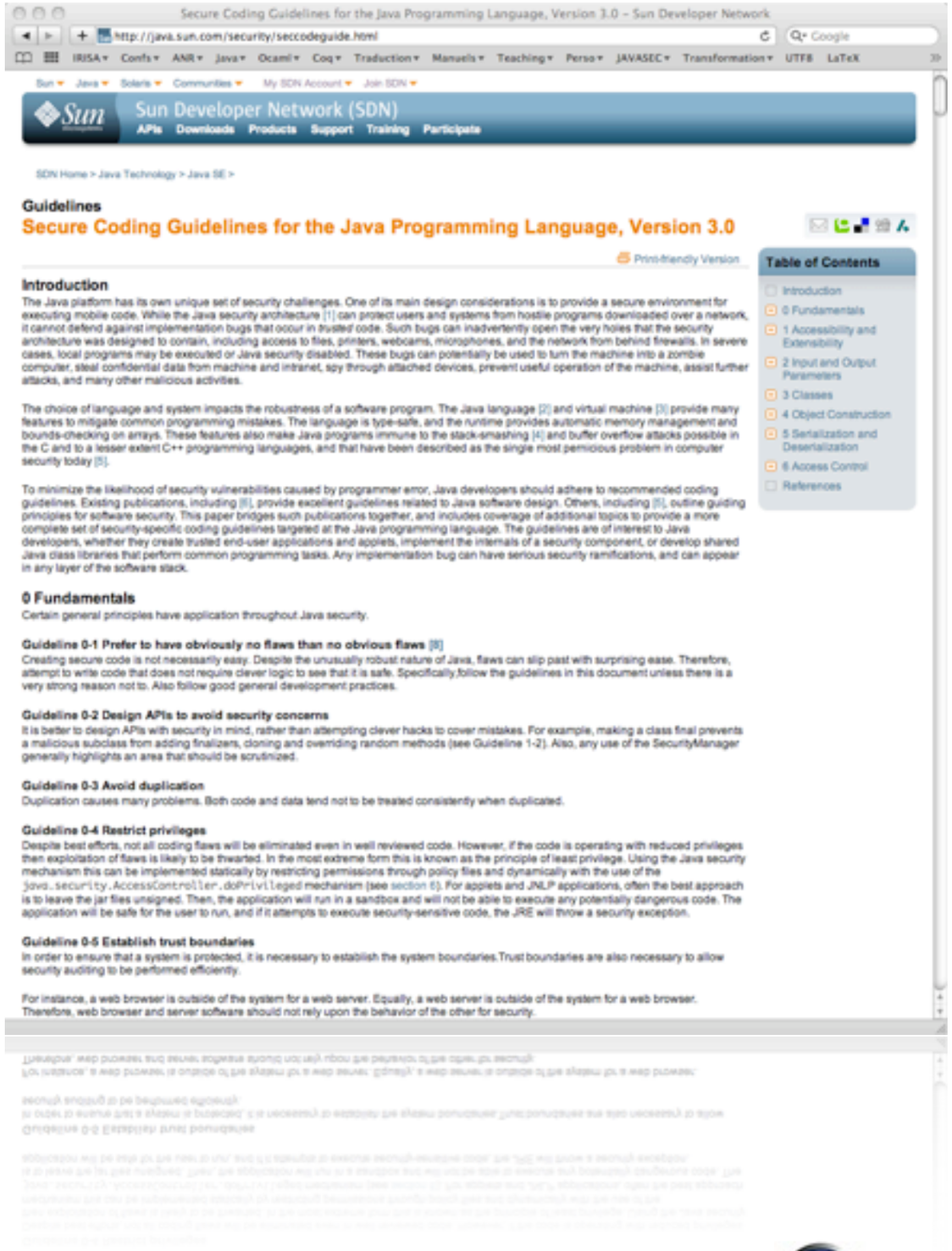
```
package TheBad;

class BookShelf {

  private Object[] content;

  public BookShelf()
    { ...; backdoor = this; }

  public add(Object o) { ... }

  public BookShelf clone() { ... }

  private badThings() {
    robAll(backdoor.content);
}
```

Secure Coding in Java

- Programs are complex collections of trusted and untrusted classes.
- Sensitive objects may migrate
  - from trusted parts to untrusted parts
  - from untrusted parts to trusted parts
- Enhance security by copying of objects.

http://java.sun.com/security/seccodeguide.html

- **Guideline 2-2 Create copies of mutable outputs**
  - If a method returns a reference to an internal mutable object, then client code may modify internal state. Therefore, copy mutable objects before returning, unless the intention is to share state.
  - To create a copy of a trusted mutable object, call a copy constructor or clone method.



Secure Coding Guidelines for the Java Programming Language, Version 3.0 - Sun Developer Network

http://java.sun.com/security/seccodeguide.html

Sun Developer Network (SDN)

APIs Downloads Products Support Training Participate

SDN Home > Java Technology > Java SE >

Guidelines

Secure Coding Guidelines for the Java Programming Language, Version 3.0

Print-friendly Version

Table of Contents

Introduction

The Java platform has its own unique set of security challenges. One of its main design considerations is to provide a secure environment for executing mobile code. While the Java security architecture [1] can protect users and systems from hostile programs downloaded over a network, it cannot defend against implementation bugs that occur in trusted code. Such bugs can inadvertently open the very holes that the security architecture was designed to contain, including access to files, printers, webcams, microphones, and the network from behind firewalls. In severe cases, local programs may be executed or Java security disabled. These bugs can potentially be used to turn the machine into a zombie computer, steal confidential data from machine and intranet, spy through attached devices, prevent useful operation of the machine, assist further attacks, and many other malicious activities.

The choice of language and system impacts the robustness of a software program. The Java language [2] and virtual machine [3] provide many features to mitigate common programming mistakes. The language is type-safe, and the runtime provides automatic memory management and bounds-checking on arrays. These features also make Java programs immune to the stack-smashing [4] and buffer overflow attacks possible in the C and to a lesser extent C++ programming languages, and that have been described as the single most pernicious problem in computer security today [5].

To minimize the likelihood of security vulnerabilities caused by programmer error, Java developers should adhere to recommended coding guidelines. Existing publications, including [6], provide excellent guidelines related to Java software design. Others, including [5], outline guiding principles for software security. This paper bridges such publications together, and includes coverage of additional topics to provide a more complete set of security-specific coding guidelines targeted at the Java programming language. The guidelines are of interest to Java developers, whether they create trusted end-user applications and applets, implement the internals of a security component, or develop shared Java class libraries that perform common programming tasks. Any implementation bug can have serious security ramifications, and can appear in any layer of the software stack.

0 Fundamentals

Certain general principles have application throughout Java security.

Guideline 0-1 Prefer to have obviously no flaws than no obvious flaws [8]

Creating secure code is not necessarily easy. Despite the unusually robust nature of Java, flaws can slip past with surprising ease. Therefore, attempt to write code that does not require clever logic to see that it is safe. Specifically,follow the guidelines in this document unless there is a very strong reason not to. Also follow good general development practices.

Guideline 0-2 Design APIs to avoid security concerns

It is better to design APIs with security in mind, rather than attempting clever hacks to cover mistakes. For example, making a class final prevents a malicious subclass from adding finalizers, cloning and overriding random methods (see Guideline 1-2). Also, any use of the SecurityManager generally highlights an area that should be scrutinized.

Guideline 0-3 Avoid duplication

Duplication causes many problems. Both code and data tend not to be treated consistently when duplicated.

Guideline 0-4 Restrict privileges

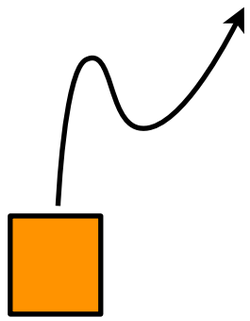
Despite best efforts, not all coding flaws will be eliminated even in well reviewed code. However, if the code is operating with reduced privileges then exploitation of flaws is likely to be thwarted. In the most extreme form this is known as the principle of least privilege. Using the Java security mechanism this can be implemented statically by restricting permissions through policy files and dynamically with the use of the `java.security.AccessController.doPrivileged` mechanism (see section 6). For applets and JNLP applications, often the best approach is to leave the jar files unsigned. Then, the application will run in a sandbox and will not be able to execute any potentially dangerous code. The application will be safe for the user to run, and if it attempts to execute security-sensitive code, the JRE will throw a security exception.

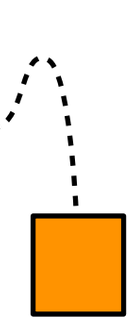
Guideline 0-5 Establish trust boundaries

In order to ensure that a system is protected, it is necessary to establish the system boundaries.Trust boundaries are also necessary to allow security auditing to be performed efficiently.

For instance, a web browser is outside of the system for a web server. Equally, a web server is outside of the system for a web browser. Therefore, web browser and server software should not rely upon the behavior of the other for security.

Clone Defense

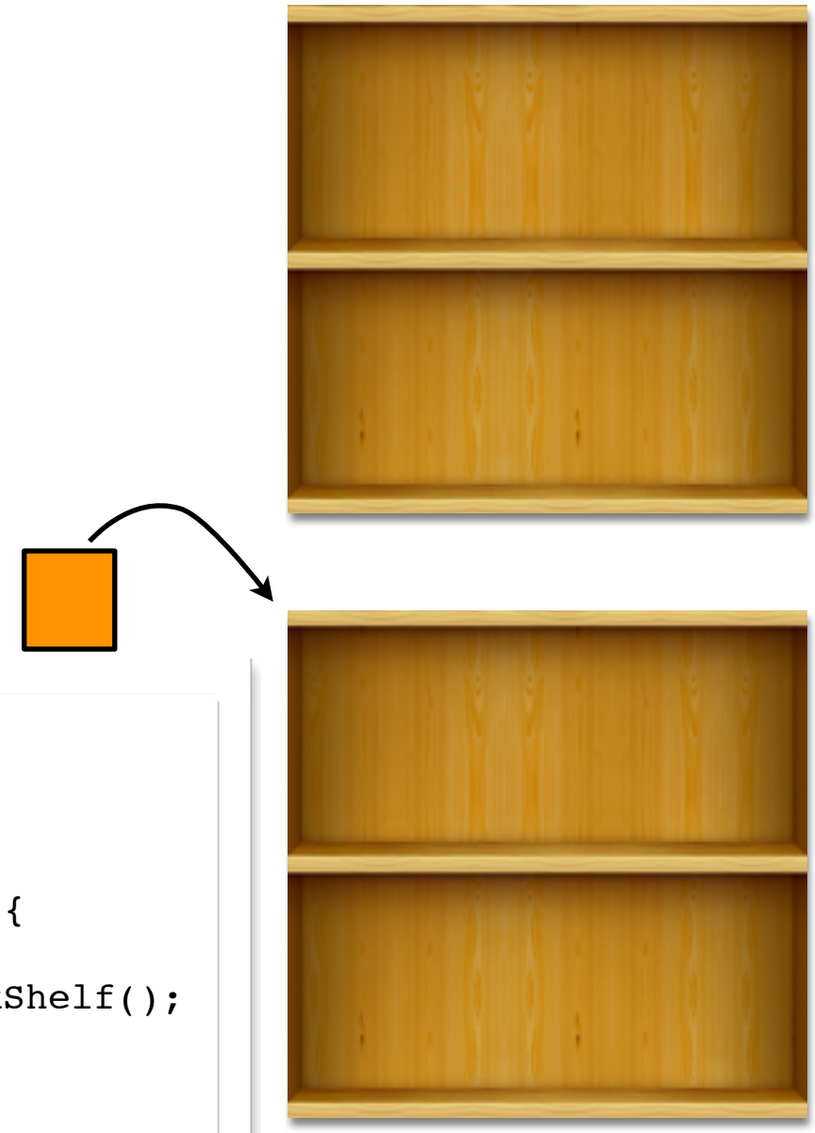




```
package TheGood;

class StoreMyHoneyPots {

  void main(String[] args) {

    BookShelf bs = new BookShelf();





  }
}
```

```
package TheBad;

class BookShelf {

  private Object[] content;

  public BookShelf()
    { ...; backdoor = this; }

  public add(Object o) { ... }

  public BookShelf clone() { ... }

  private badThings() {
    robAll(backdoor.content);
}
```

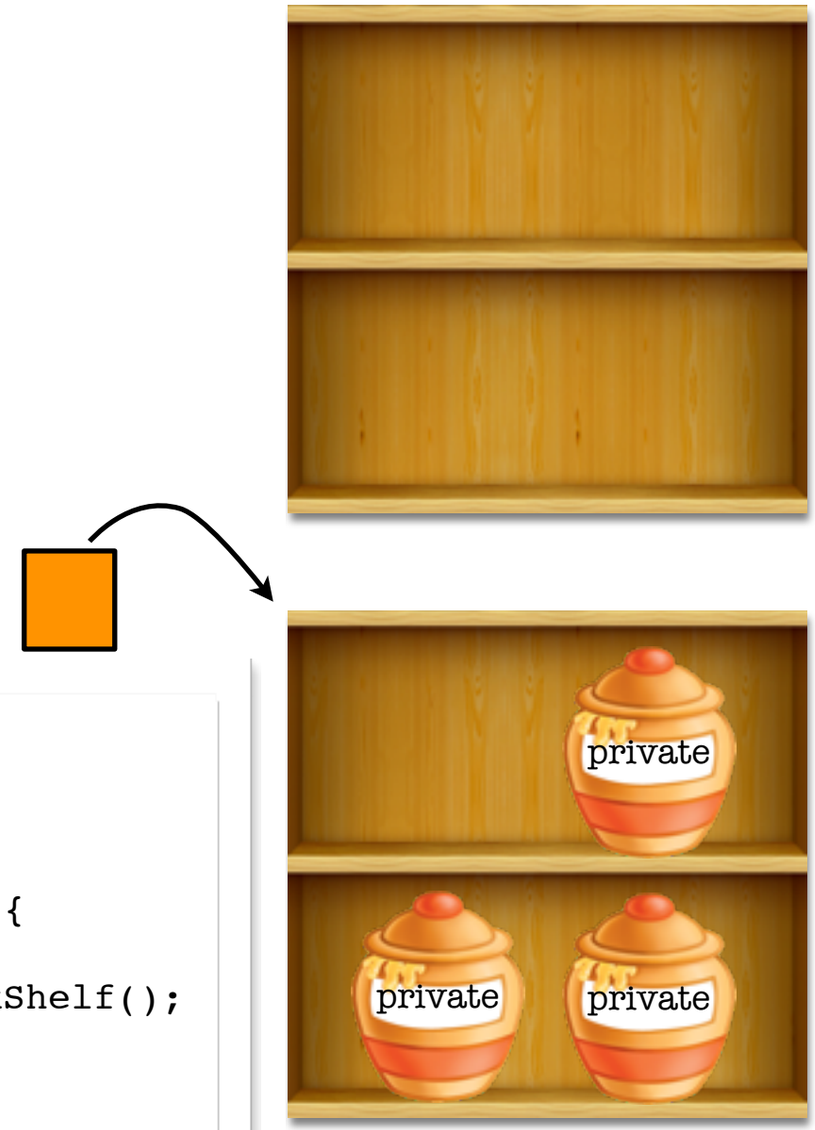
Clone Defense



```
package TheGood;

class StoreMyHoneyPots {

  void main(String[] args) {

    BookShelf bs = new BookShelf();

    bs = bs.clone();




  }
}
```

```
package TheBad;

class BookShelf {

  private Object[] content;

  public BookShelf()
    { ...; backdoor = this; }

  public add(Object o) { ... }

  public BookShelf clone() { ... }

  private badThings() {
    robAll(backdoor.content);
}
```

Clone Defense



```
package TheGood;

class StoreMyHoneyPots {

  void main(String[] args) {

    BookShelf bs = new BookShelf();

    bs = bs.clone();

    bs.add(pot1);
    bs.add(pot2);
    bs.add(pot3);

  }
}
```

```
package TheBad;

class BookShelf {

  private Object[] content;

  public BookShelf()
    { ...; backdoor = this; }

  public add(Object o) { ... }

  public BookShelf clone() { ... }

  private badThings() {
    robAll(backdoor.content);
}
```

Copy In Java

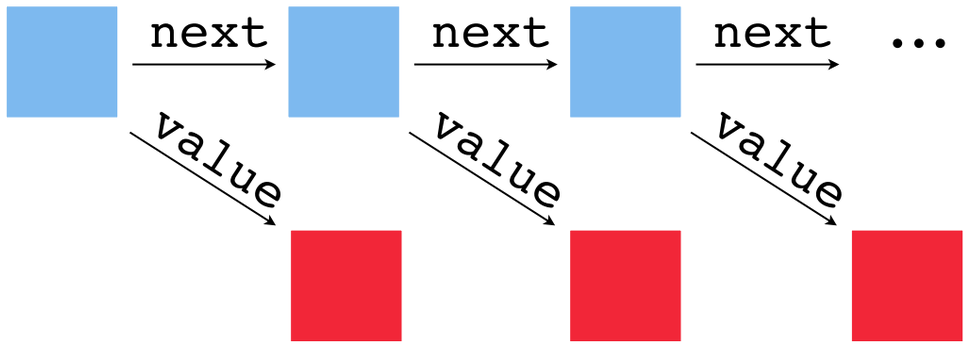
- No default copy
- Copy methods must be provided by programmers
  - by implementing Cloneable interface

```
class A implements Cloneable {
   Object clone() {...}
}
```

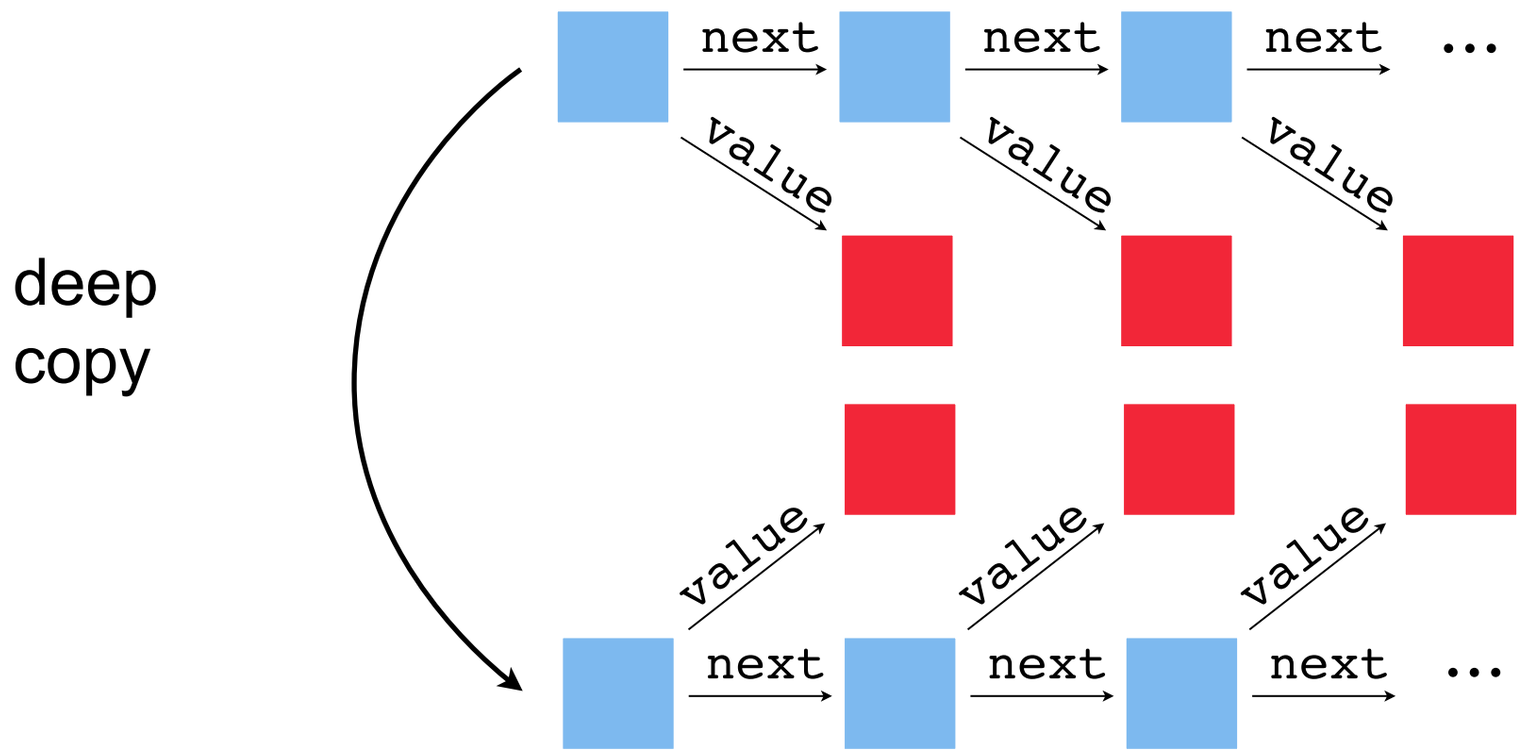
  - or a copy constructor

```
class A {
   A (A source) {...}
}
```

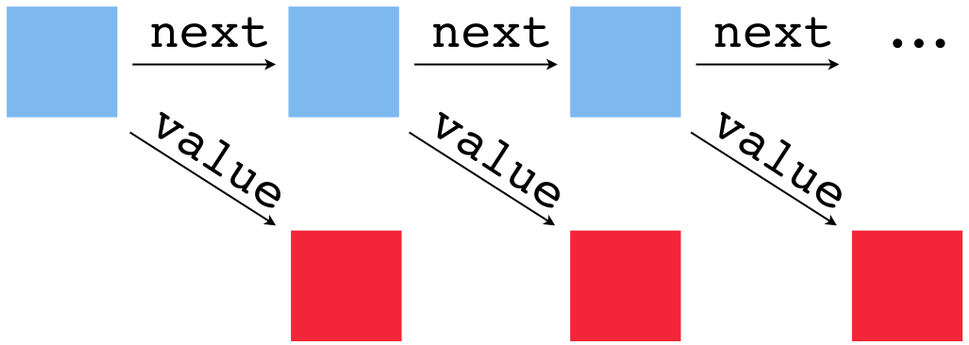
Shallow/Deep Copy



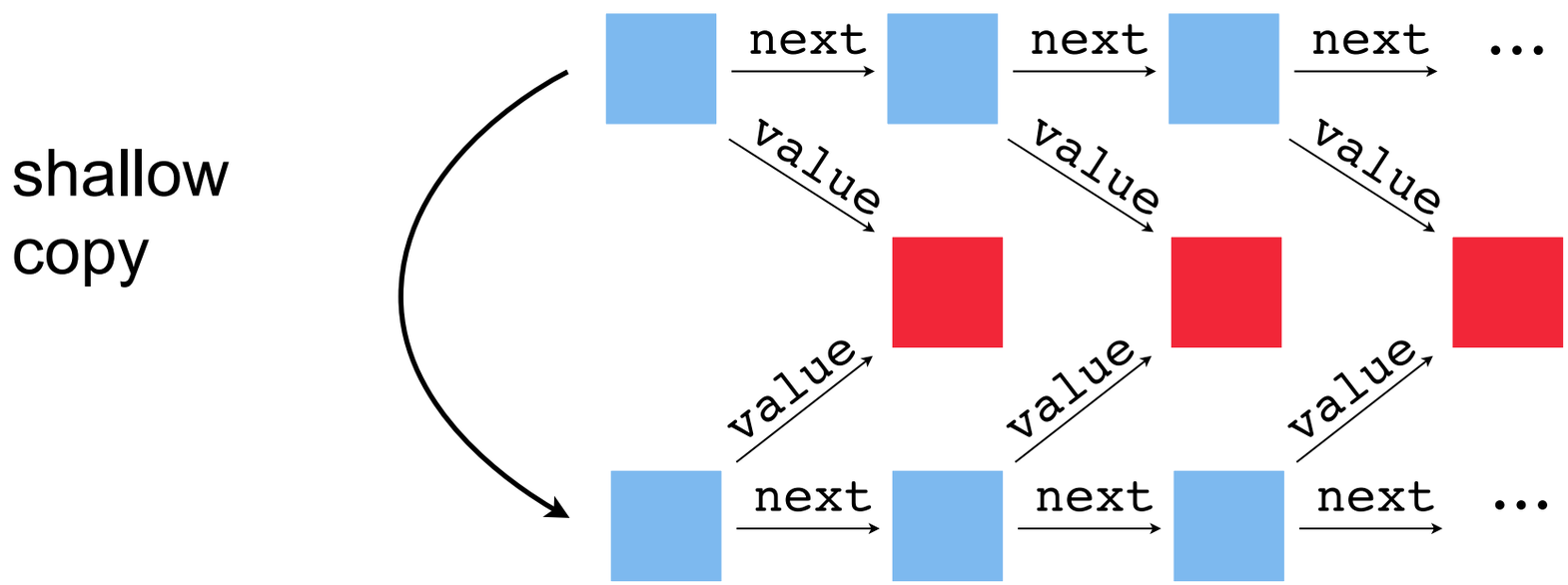
Shallow/Deep Copy



Shallow/Deep Copy



Shallow/Deep Copy



Shallow/Deep Copy

```
class List {
  V value;
  List next;

  List(V value, List next) {
    this.value = value;
    this.next = next;
  }


  List clone() {
    return new List(value,
                    (next==null ? null : next.clone()));
  }


  List deepClone() {
    return new List((V) value.clone(),
                    (next==null ? null : next.deepClone()));
  }
}
```

Copy in Java

Sub-classing and overriding complicate matters

This field is final but content is mutable

```
public class CopyOutput {
    private final java.util.Date date;
    ...
    public java.util.Date getDate() {
        return (java.util.Date)date.clone();
    }
}
```

Copy in Java

Sub-classing and overriding complicate matters

```
public class CopyOutput {
    private final java.util.Date date;
    ...
    public java.util.Date getDate() {
        return (java.util.Date)date.clone();
    }
}
```

This field is final but content is mutable

Does it really perform a copy ?

It may call an overridden version of Date.clone() !

What we want:

- Copy policy language
  - expressive
  - simple and modular
  - compatible with subclassing and method overriding
  - with semantic foundations
- Enforcement mechanism
  - sound
  - precise enough for «reasonable» clone methods
  - compatible with class-based bytecode verification
    - fast enough to be used at class loading time
    - can handle legacy code

Secure Cloning

- A new annotation system for expressing copy policies
- Enforcement: a type system that
  - enforces non-sharing
  - but does not guarantee exact copy
- Copy policies can be checked in the presence of overriding.
  - accumulate constraints on "deep"-ness of the copy

Shallow/Deep Copy

```
class List {
  V value;
  List next;

  List(V value, List next) {
    this.value = value;
    this.next = next;
  }

  @Copy(default){ @Shallow value; @Deep(default) next;}
  List clone() {
    return new List(value,
                    (next==null ? null : next.clone()));
  }

}
```

Copy signature

Shallow/Deep Copy

```
class List {
  @Shallow V value;
  @Deep List next;

  List(V value, List next) {
    this.value = value;
    this.next = next;
  }

  @Copy
  List clone() {
    return new List(value,
                    (next==null ? null : next.clone()));
  }

}
```

Default copy signature

Shallow/Deep Copy

```
class List {
  @Shallow V value;
  @Deep List next;

  List(V value, List next) {
    this.value = value;
    this.next = next;
  }

  @Copy
  List clone() {
    return new List(value,
                    (next==null ? null : next.clone()));
  }

  @Copy(List.Deep){ @Deep value; @Deep(List.Deep) next;}
  List deepClone() {
    return new List((V) value.clone(),
                    (next==null ? null : next.deepClone()));
  }
}
```

Default copy signature

Explicit copy signature

Copy Policy Grammar

$$
\begin{array}{rcrcl}
\tau & \in & Policy & ::= & \mathsf{Copy}(X)\{af; \ldots; af\} \\
af & \in & AnnotField & ::= & a\ f \\
a & \in & Annot & ::= & \mathsf{Shallow} \mid \mathsf{Deep}(X)
\end{array}
$$

A heap cell reachable from the result of a call

`m(...)`

by following only fields marked

`Deep`

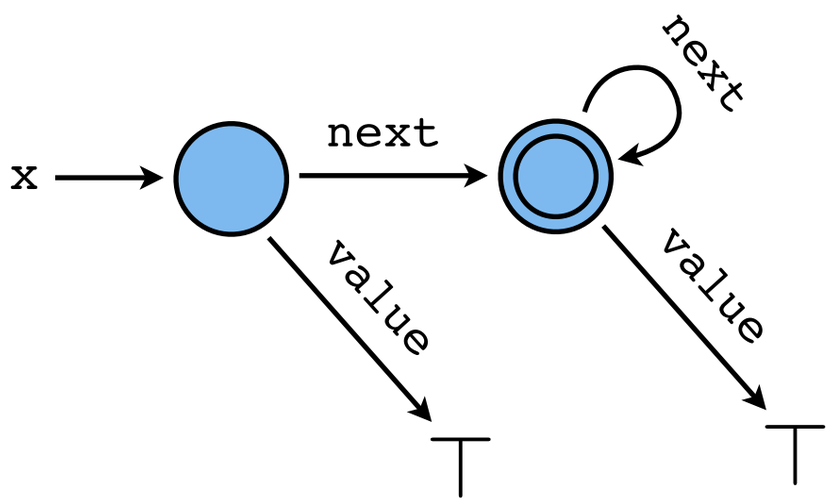
is not reachable from any local variable.

Observations

- Enforcing non-sharing of dynamically allocated structure is complex
- Copy methods are generally simple (and sometimes only straight-line code)
- We adopt a local shape analysis technique

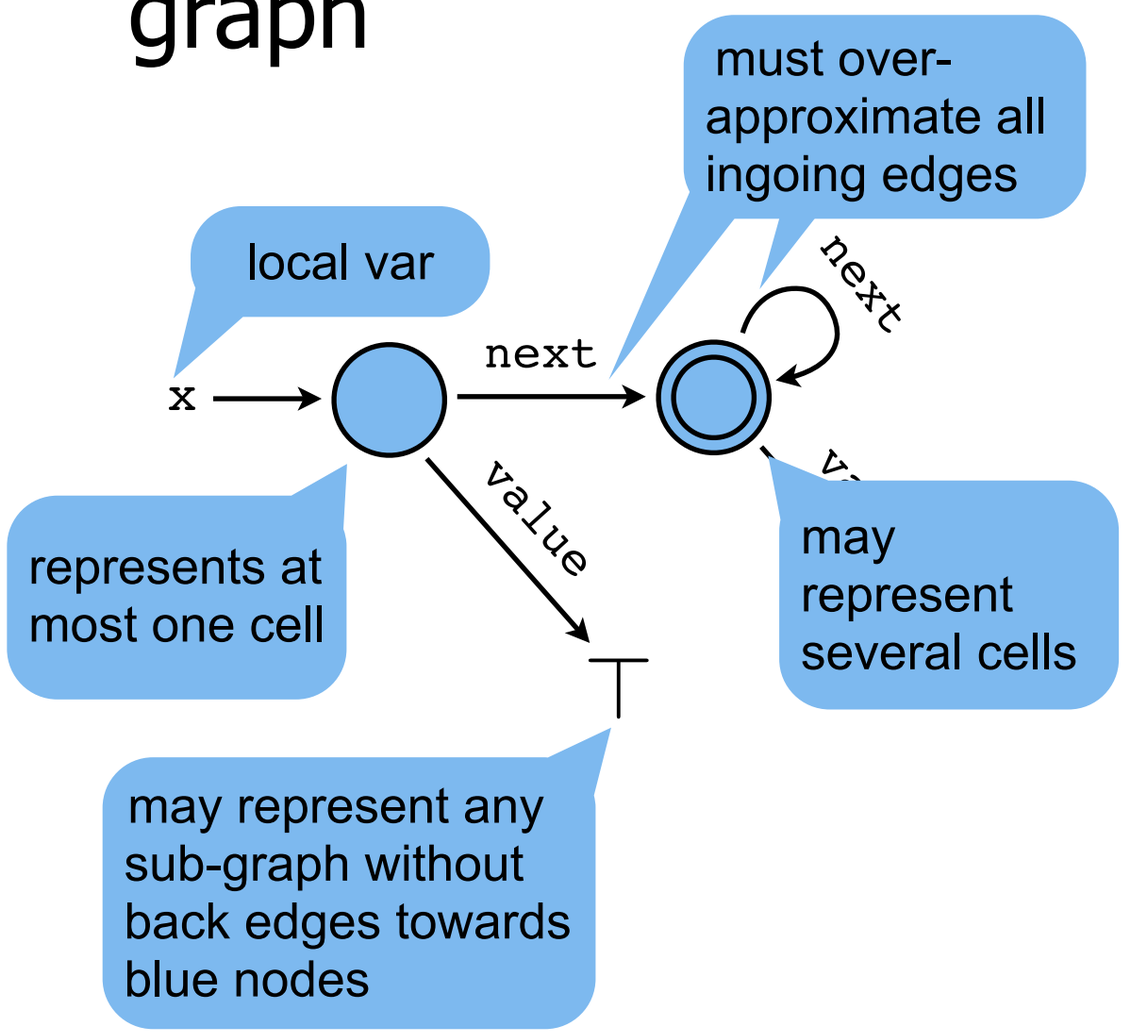
(Our) Shape Analysis

- We abstract the memory by a shape graph



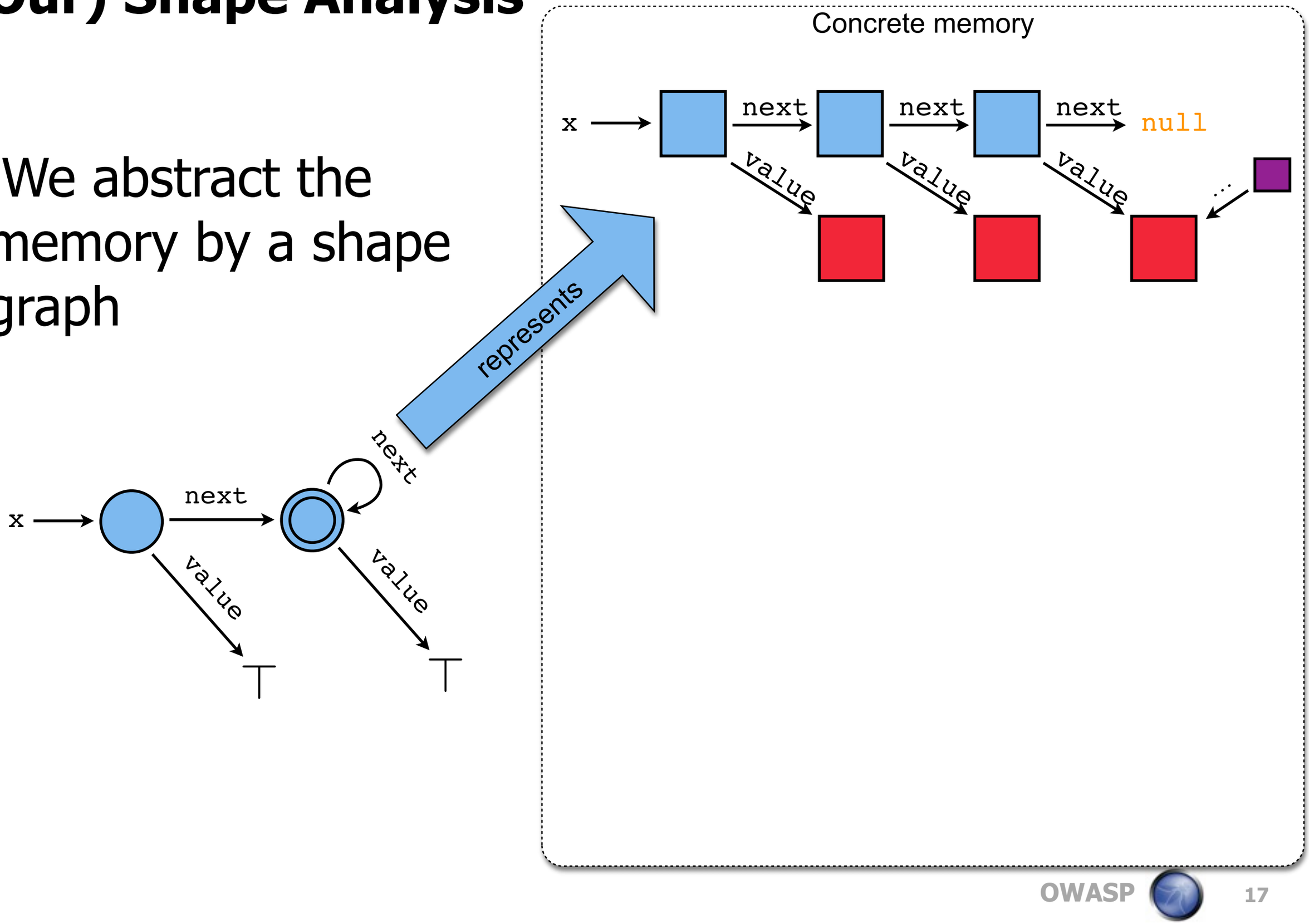
(Our) Shape Analysis

- We abstract the memory by a shape graph



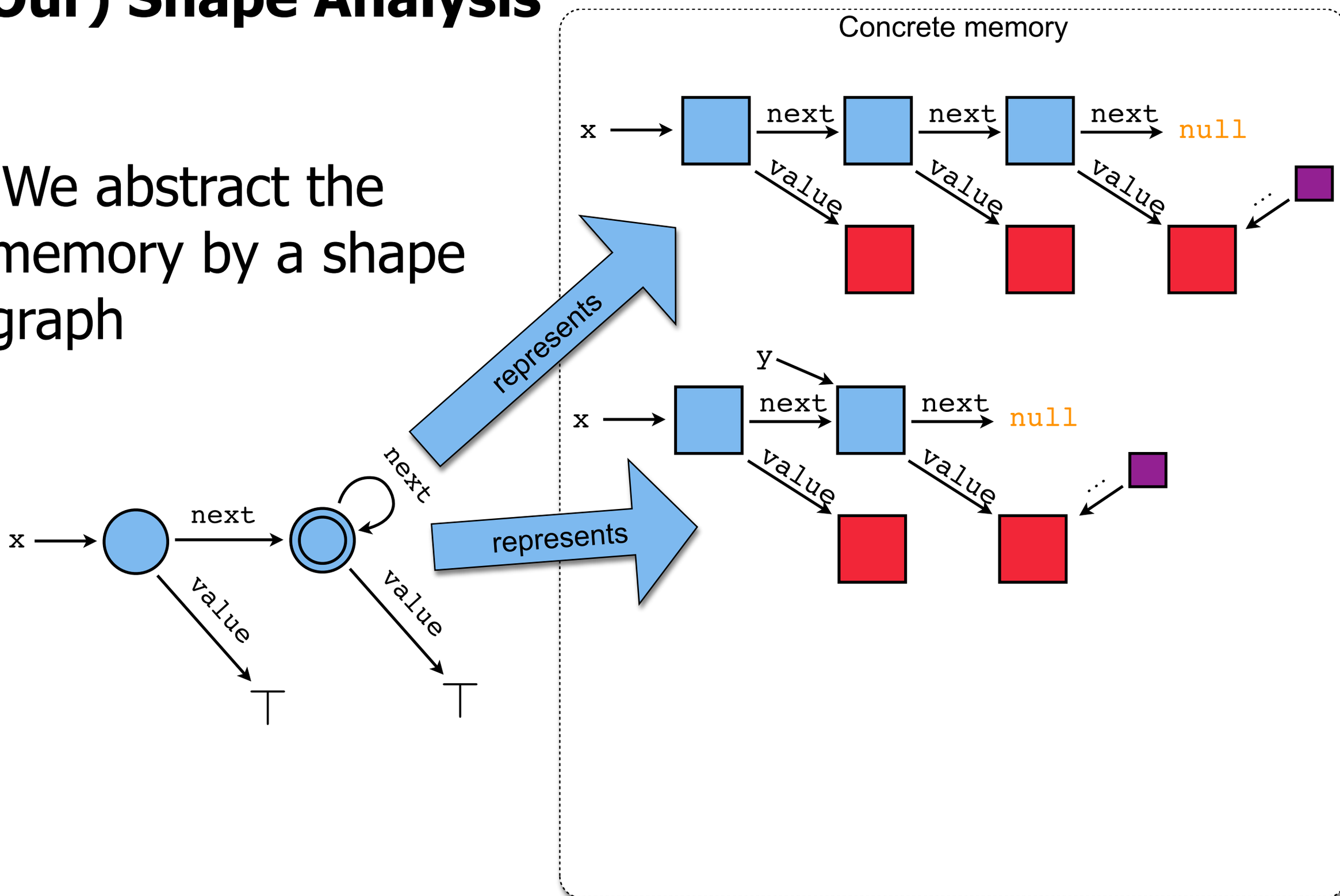
(Our) Shape Analysis

- We abstract the memory by a shape graph



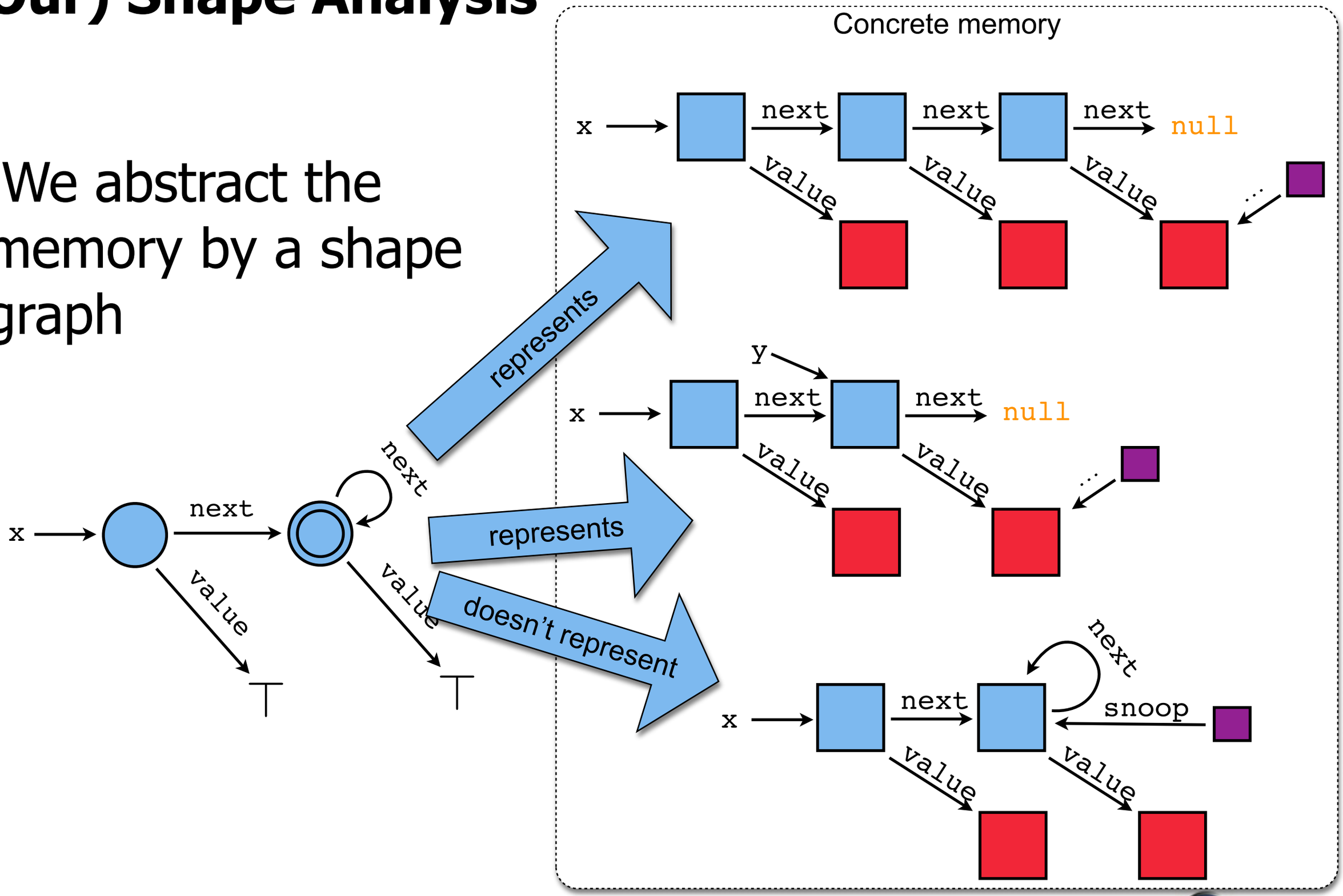
(Our) Shape Analysis

- We abstract the memory by a shape graph



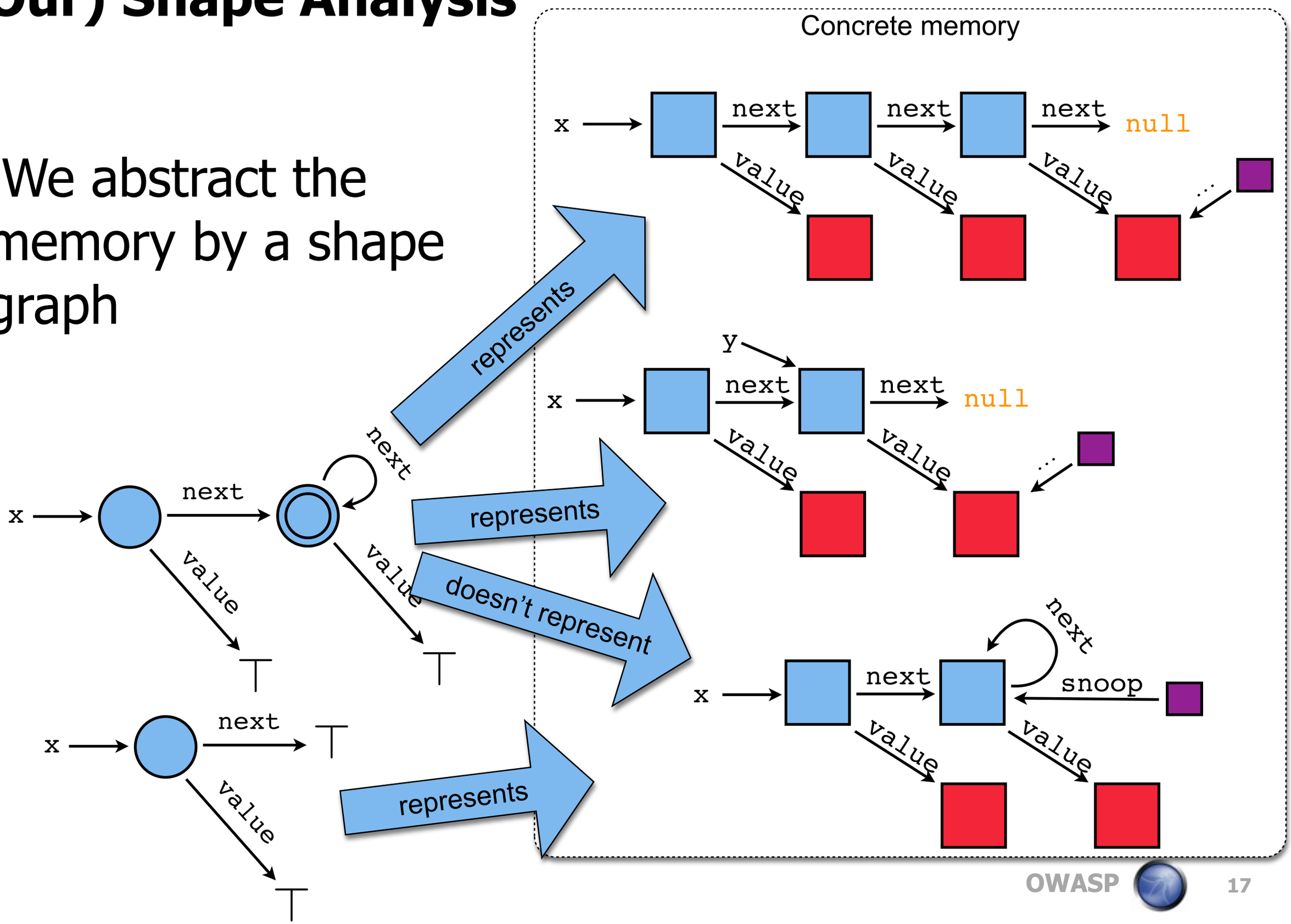
(Our) Shape Analysis

- We abstract the memory by a shape graph

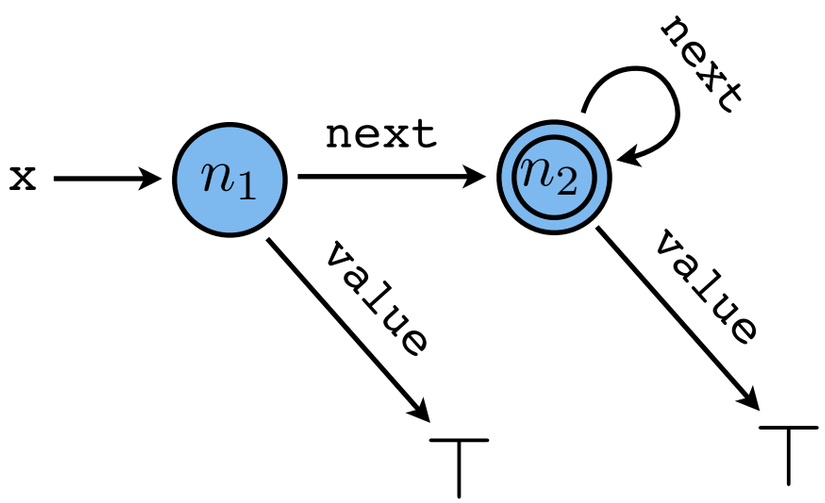


(Our) Shape Analysis

- We abstract the memory by a shape graph



Shape Graphs: formal notation



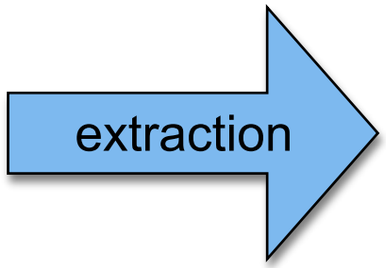
$$
\begin{aligned}
\Gamma &= [\ \mathtt{x} \mapsto n_1, \mathtt{y} \mapsto \top] \\
\Delta &= [(n_1, \mathtt{next}) \mapsto n_2, \\
&\quad (n_2, \mathtt{next}) \mapsto n_2, \\
&\quad (n_1, \mathtt{value}) \mapsto \top, \\
&\quad (n_2, \mathtt{value}) \mapsto \top\ ] \\
\Theta &= \{n_1\}
\end{aligned}
$$

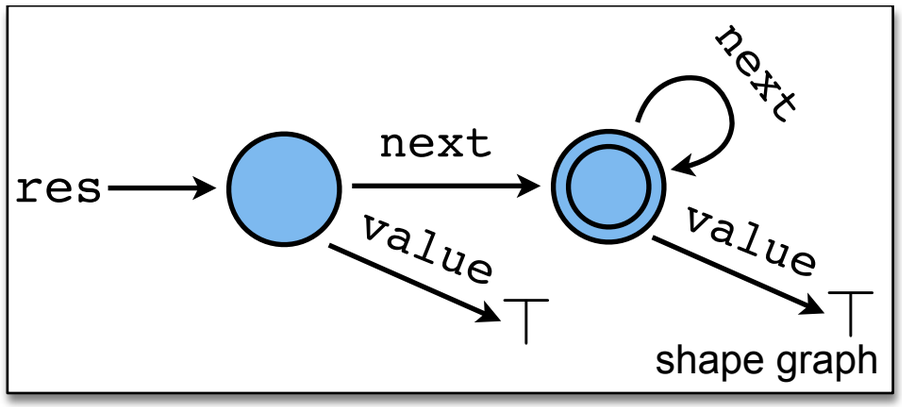
How to check a copying method m ?

1. Extract a graph type from a copy policy

```
class List {
  @Shallow V value;
  @Deep List next;

  @Copy Object clone() {...}
}
```

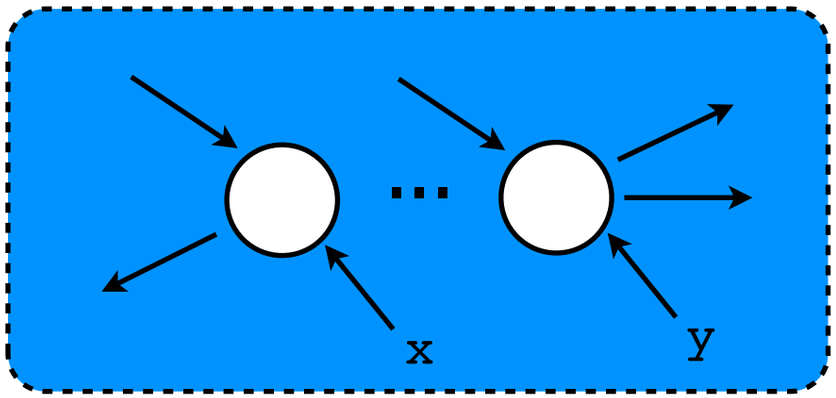
copy policy

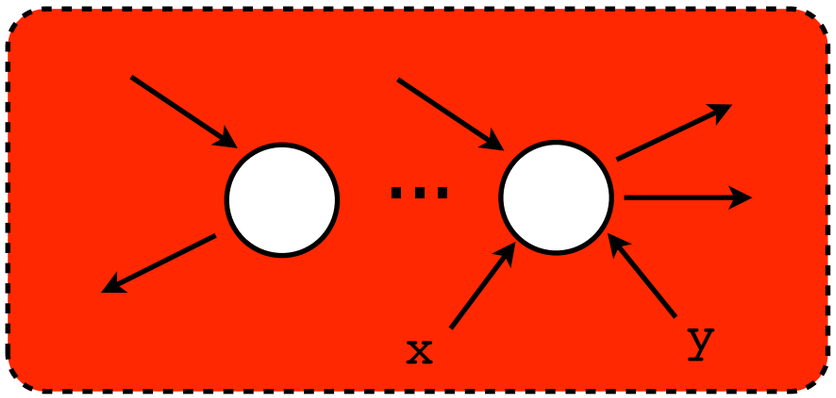




2. A type system checks if m satisfies the graph type
3. Type inference is required at join points
4. Needs a subtyping relation $\sqsubseteq$ on graph types

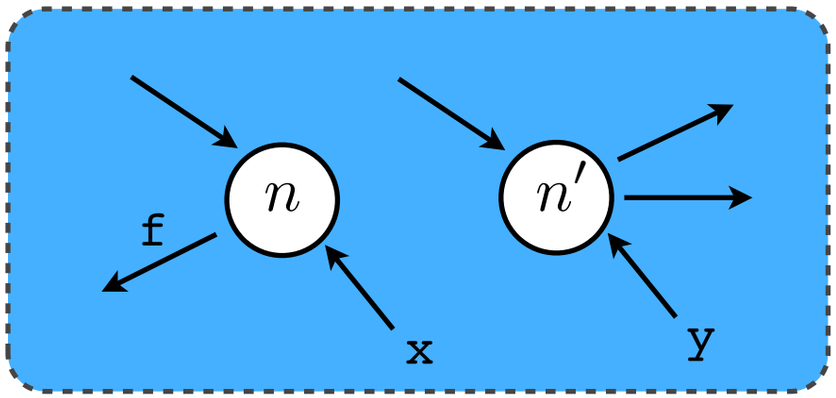
Typing Rules

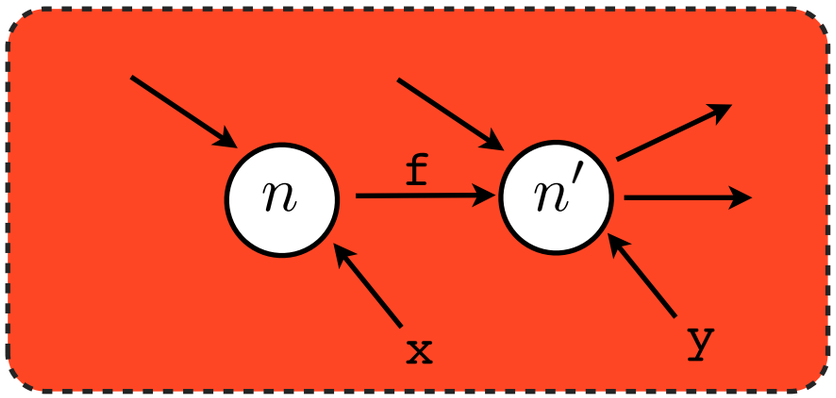




$$\frac{}{\Gamma, \Delta, \Theta \vdash x{:=}y : \Gamma[x \mapsto \Gamma(y)], \Delta, \Theta}$$

Typing Rules



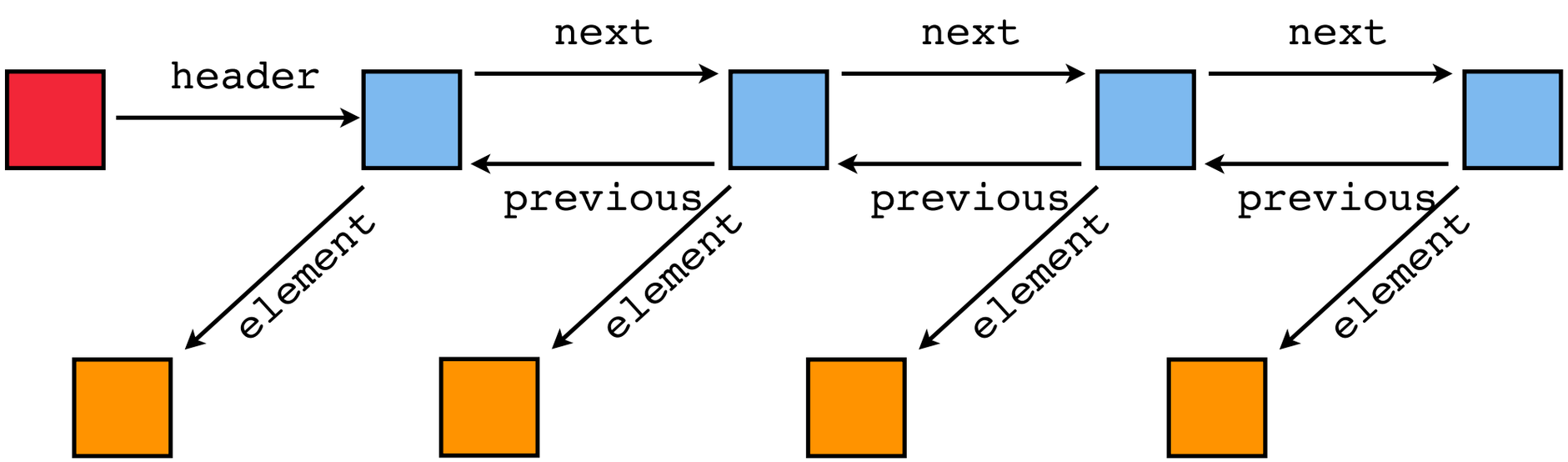


$$\frac{\Gamma(x) = n \quad n \in \Theta \quad \Gamma(y) = n'}{\Gamma, \Delta, \Theta \vdash x.f := y : \Gamma, \Delta[(n, f) \mapsto n'], \Theta}$$

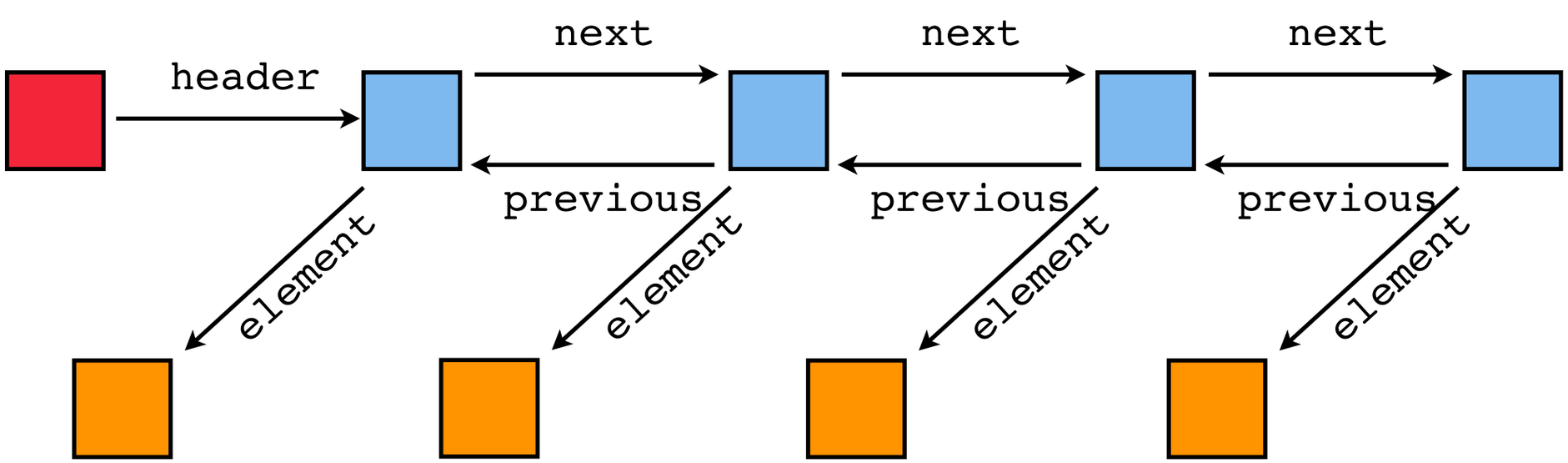
Case Study

`java.util.LinkedList`

```
public class LinkedList<E> implements Cloneable {
   private Entry<E> header;

   private static class Entry<E> {
         E element;
         Entry<E> next;
         Entry<E> previous;
      }
}
```



```
public class LinkedList<E> implements Cloneable {
   private @Deep Entry<E> header;

   private static class Entry<E> {
         @Shallow E element;
         @Deep Entry<E> next;
         @Deep Entry<E> previous;
      }
}
```



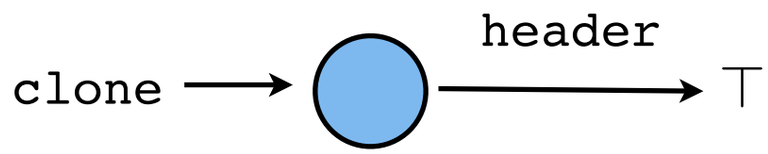
```java
public Object clone() {
  LinkedList<E> clone = null;
  clone = (LinkedList<E>) super.clone();
  clone.header = new Entry<E>;
  clone.header.next = clone.header.previous = clone.header;
  for (Entry<E> e = this.header.next; e != this.header; e = e.next) {
      Entry<E> newEntry = new Entry<E>;
      newEntry.element = e.element;
      newEntry.next = clone.header;
      newEntry.previous = clone.header.previous;
      newEntry.previous.next = newEntry;
      newEntry.next.previous = newEntry;
  }
  return clone;
}
```

```
public Object clone() {
  LinkedList<E> clone = null;
```

clone ⟶ ⊥

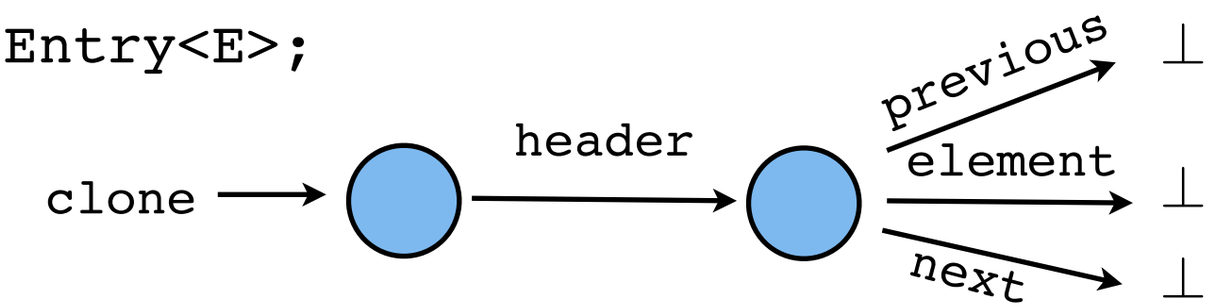
```
  clone = (LinkedList<E>) super.clone();
  clone.header = new Entry<E>;
  clone.header.next = clone.header.previous = clone.header;
  for (Entry<E> e = this.header.next; e != this.header; e = e.next) {
      Entry<E> newEntry = new Entry<E>;
      newEntry.element = e.element;
      newEntry.next = clone.header;
      newEntry.previous = clone.header.previous;
      newEntry.previous.next = newEntry;
      newEntry.next.previous = newEntry;
  }
  return clone;
}
```

```
public Object clone() {
  LinkedList<E> clone = null;
  clone = (LinkedList<E>) super.clone();
```



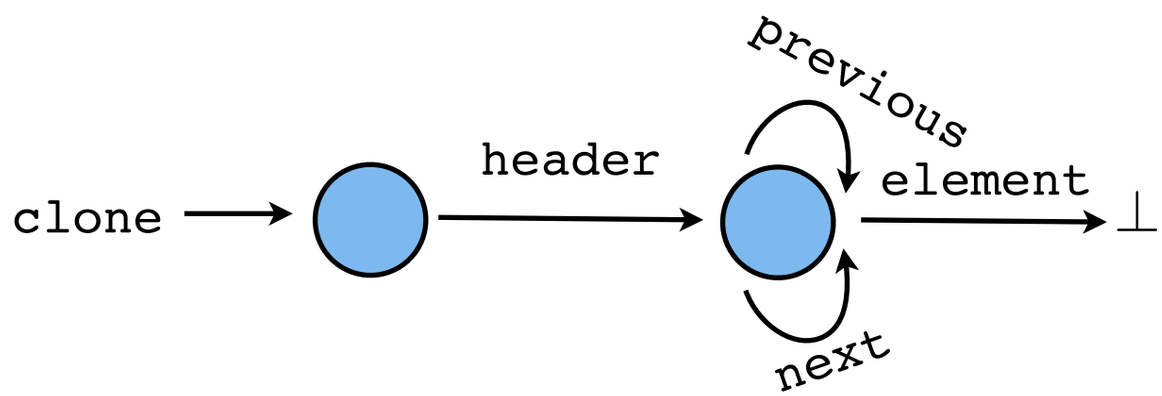

```
  clone.header = new Entry<E>;
  clone.header.next = clone.header.previous = clone.header;
  for (Entry<E> e = this.header.next; e != this.header; e = e.next) {
      Entry<E> newEntry = new Entry<E>;
      newEntry.element = e.element;
      newEntry.next = clone.header;
      newEntry.previous = clone.header.previous;
      newEntry.previous.next = newEntry;
      newEntry.next.previous = newEntry;
  }
  return clone;
}
```

```
public Object clone() {
  LinkedList<E> clone = null;
  clone = (LinkedList<E>) super.clone();
  clone.header = new Entry<E>;
```



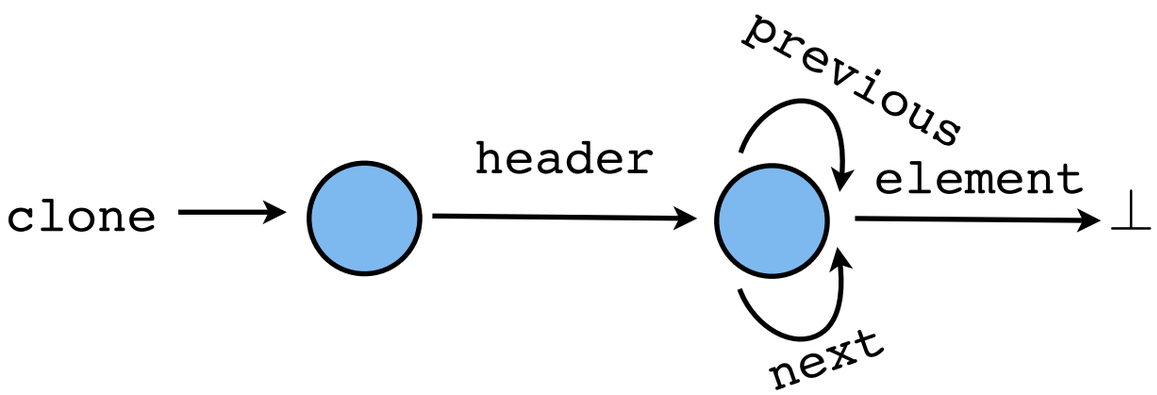

```
  clone.header.next = clone.header.previous = clone.header;
  for (Entry<E> e = this.header.next; e != this.header; e = e.next) {
      Entry<E> newEntry = new Entry<E>;
      newEntry.element = e.element;
      newEntry.next = clone.header;
      newEntry.previous = clone.header.previous;
      newEntry.previous.next = newEntry;
      newEntry.next.previous = newEntry;
  }
  return clone;
}
```

```
public Object clone() {
  LinkedList<E> clone = null;
  clone = (LinkedList<E>) super.clone();
  clone.header = new Entry<E>;
  clone.header.next = clone.header.previous = clone.header;
```



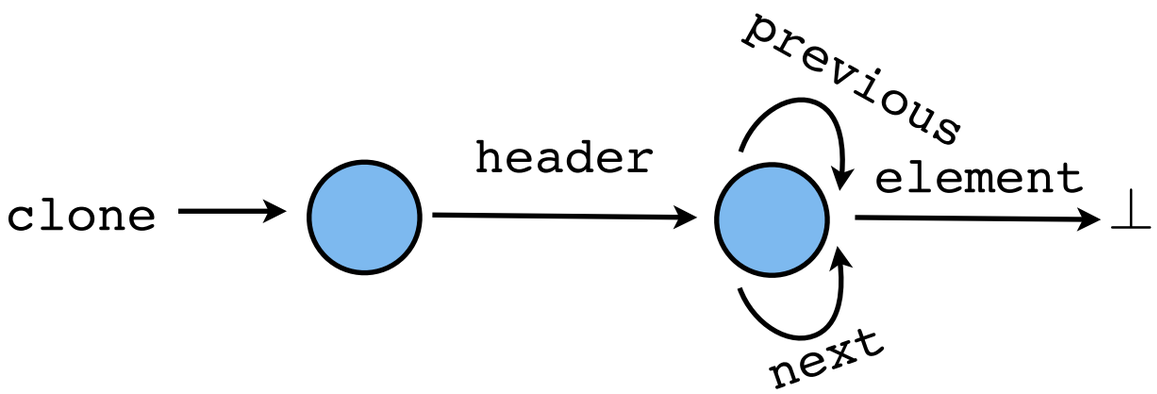

```
  for (Entry<E> e = this.header.next; e != this.header; e = e.next) {
      Entry<E> newEntry = new Entry<E>;
      newEntry.element = e.element;
      newEntry.next = clone.header;
      newEntry.previous = clone.header.previous;
      newEntry.previous.next = newEntry;
      newEntry.next.previous = newEntry;
  }
  return clone;
}
```

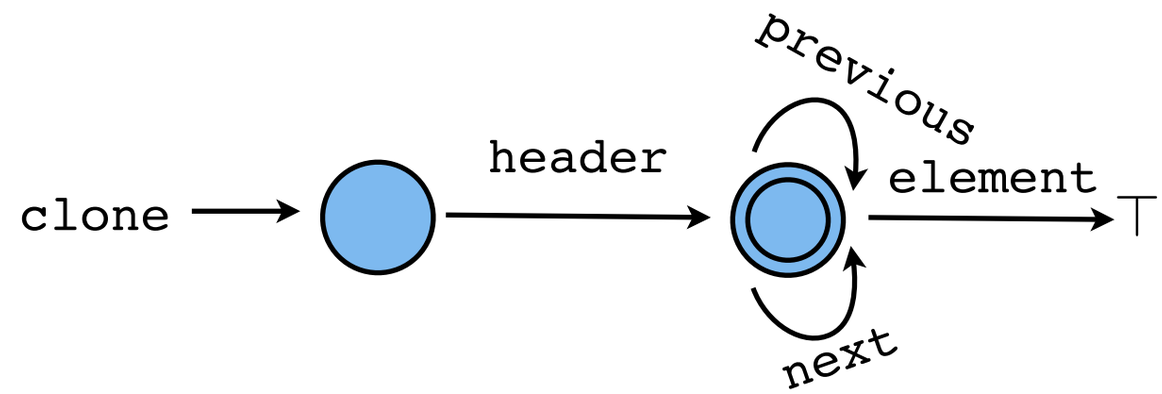
```
public Object clone() {
   LinkedList<E> clone = null;
   clone = (LinkedList<E>) super.clone();
   clone.header = new Entry<E>;
   clone.header.next = clone.header.previous = clone.header;
```




```
   for (Entry<E> e = this.header.next; e != this.header; e = e.next) {
       Entry<E> newEntry = new Entry<E>;
       newEntry.element = e.element;
       newEntry.next = clone.header;
       newEntry.previous = clone.header.previous;
       newEntry.previous.next = newEntry;
       newEntry.next.previous = newEntry;
   }
   return clone;
}
```

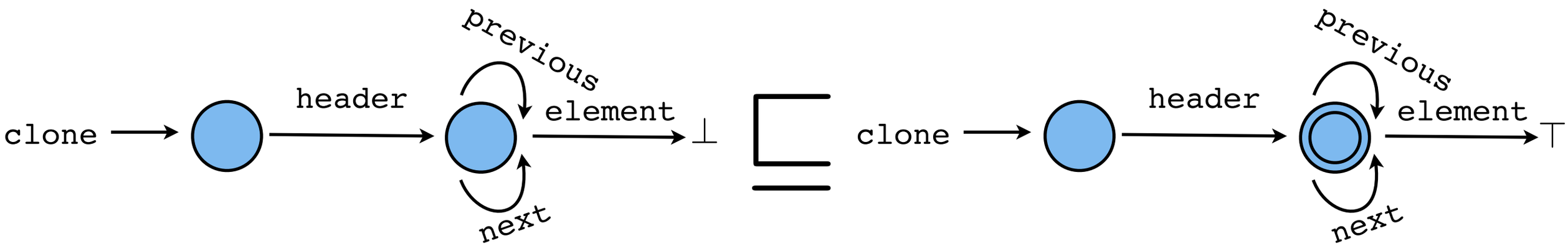
```
public Object clone() {
   LinkedList<E> clone = null;
   clone = (LinkedList<E>) super.clone();
   clone.header = new Entry<E>;
   clone.header.next = clone.header.previous = clone.header;
```





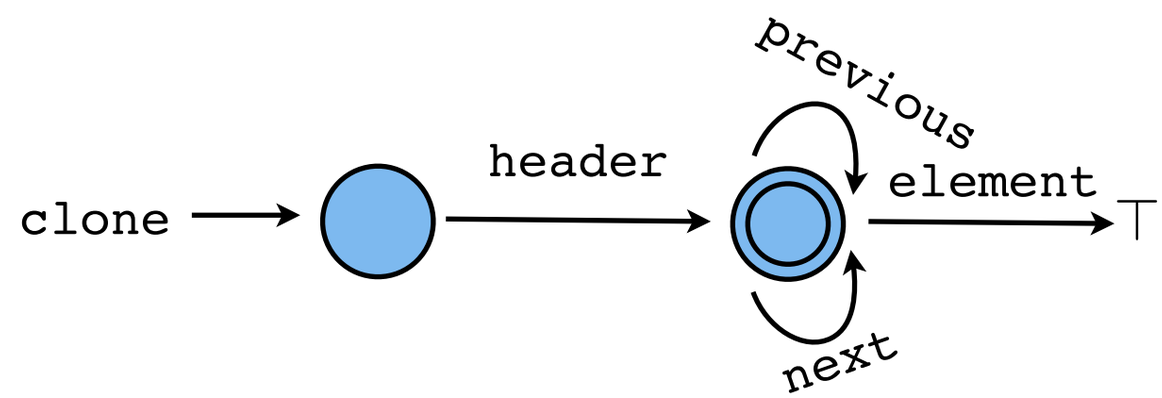

```
   for (Entry<E> e = this.header.next; e != this.header; e = e.next) {
      Entry<E> newEntry = new Entry<E>;
      newEntry.element = e.element;
      newEntry.next = clone.header;
      newEntry.previous = clone.header.previous;
      newEntry.previous.next = newEntry;
      newEntry.next.previous = newEntry;
   }
   return clone;
}
```

```
public Object clone() {
   LinkedList<E> clone = null;
   clone = (LinkedList<E>) super.clone();
   clone.header = new Entry<E>;
   clone.header.next = clone.header.previous = clone.header;
```



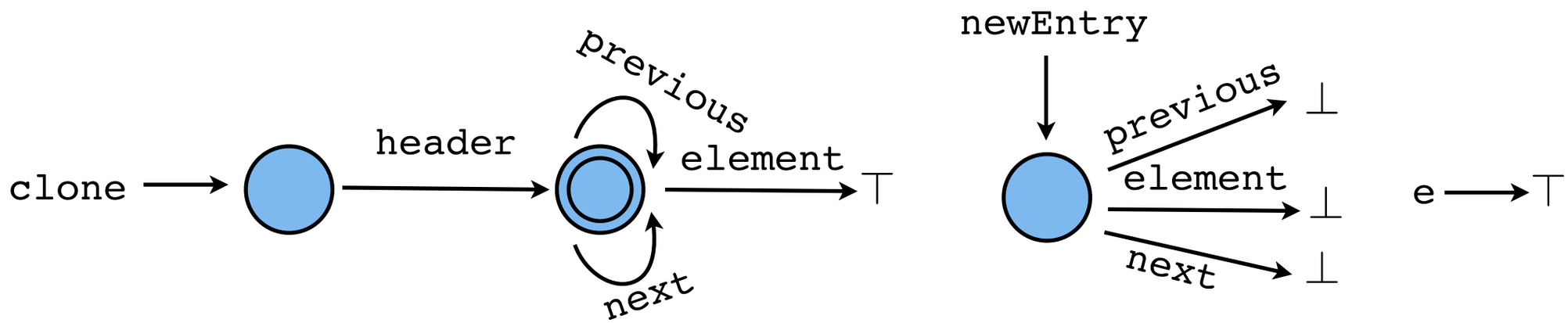

```
   for (Entry<E> e = this.header.next; e != this.header; e = e.next) {
       Entry<E> newEntry = new Entry<E>;
       newEntry.element = e.element;
       newEntry.next = clone.header;
       newEntry.previous = clone.header.previous;
       newEntry.previous.next = newEntry;
       newEntry.next.previous = newEntry;
   }
   return clone;
}
```

```
public Object clone() {
  LinkedList<E> clone = null;
  clone = (LinkedList<E>) super.clone();
  clone.header = new Entry<E>;
  clone.header.next = clone.header.previous = clone.header;
```



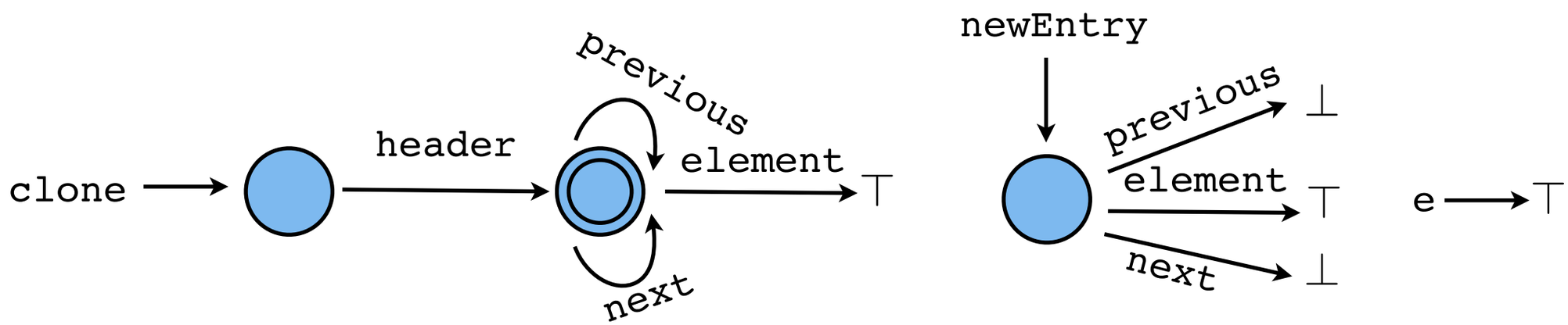

```
  for (Entry<E> e = this.header.next; e != this.header; e = e.next) {
      Entry<E> newEntry = new Entry<E>;
      newEntry.element = e.element;
      newEntry.next = clone.header;
      newEntry.previous = clone.header.previous;
      newEntry.previous.next = newEntry;
      newEntry.next.previous = newEntry;
  }
  return clone;
}
```

```
public Object clone() {
  LinkedList<E> clone = null;
  clone = (LinkedList<E>) super.clone();
  clone.header = new Entry<E>;
  clone.header.next = clone.header.previous = clone.header;
  for (Entry<E> e = this.header.next; e != this.header; e = e.next) {
      Entry<E> newEntry = new Entry<E>;
```



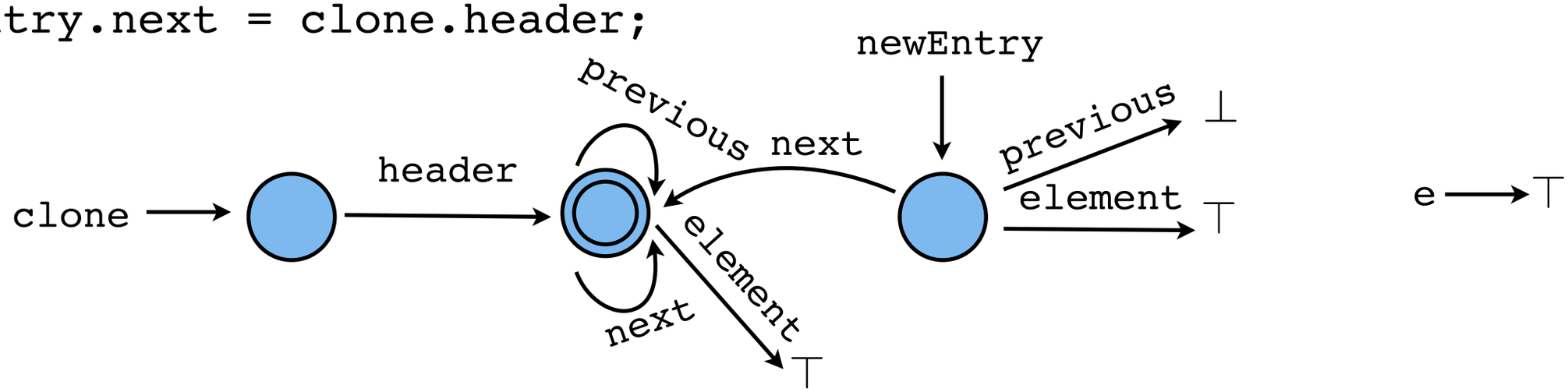

```
      newEntry.element = e.element;
      newEntry.next = clone.header;
      newEntry.previous = clone.header.previous;
      newEntry.previous.next = newEntry;
      newEntry.next.previous = newEntry;
  }
  return clone;
}
```

```
public Object clone() {
  LinkedList<E> clone = null;
  clone = (LinkedList<E>) super.clone();
  clone.header = new Entry<E>;
  clone.header.next = clone.header.previous = clone.header;
  for (Entry<E> e = this.header.next; e != this.header; e = e.next) {
      Entry<E> newEntry = new Entry<E>;
      newEntry.element = e.element;
```



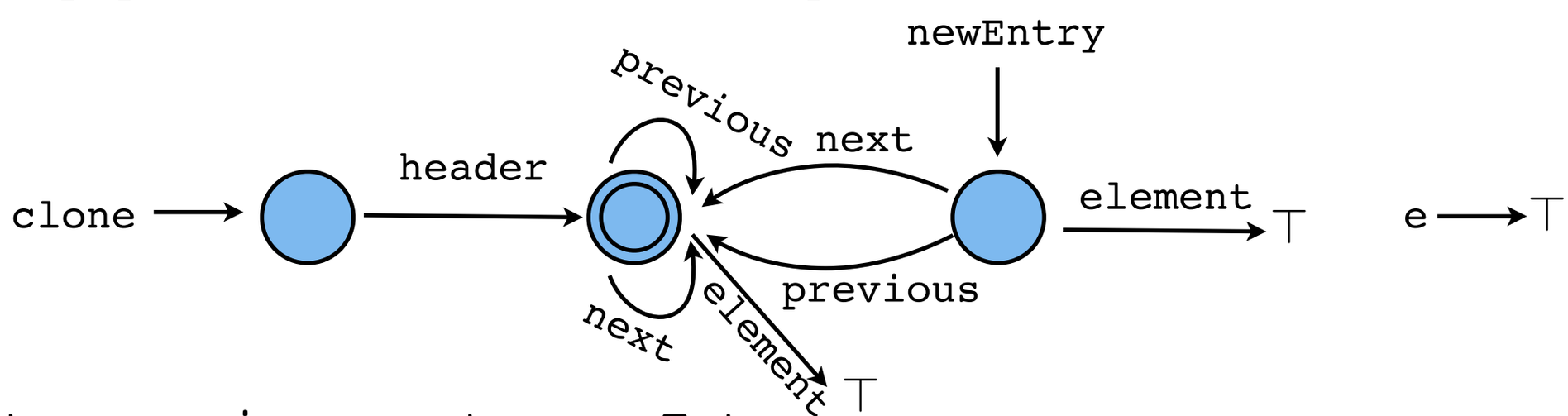

```
      newEntry.next = clone.header;
      newEntry.previous = clone.header.previous;
      newEntry.previous.next = newEntry;
      newEntry.next.previous = newEntry;
  }
  return clone;
}
```

```
public Object clone() {
  LinkedList<E> clone = null;
  clone = (LinkedList<E>) super.clone();
  clone.header = new Entry<E>;
  clone.header.next = clone.header.previous = clone.header;
  for (Entry<E> e = this.header.next; e != this.header; e = e.next) {
      Entry<E> newEntry = new Entry<E>;
      newEntry.element = e.element;
      newEntry.next = clone.header;
```



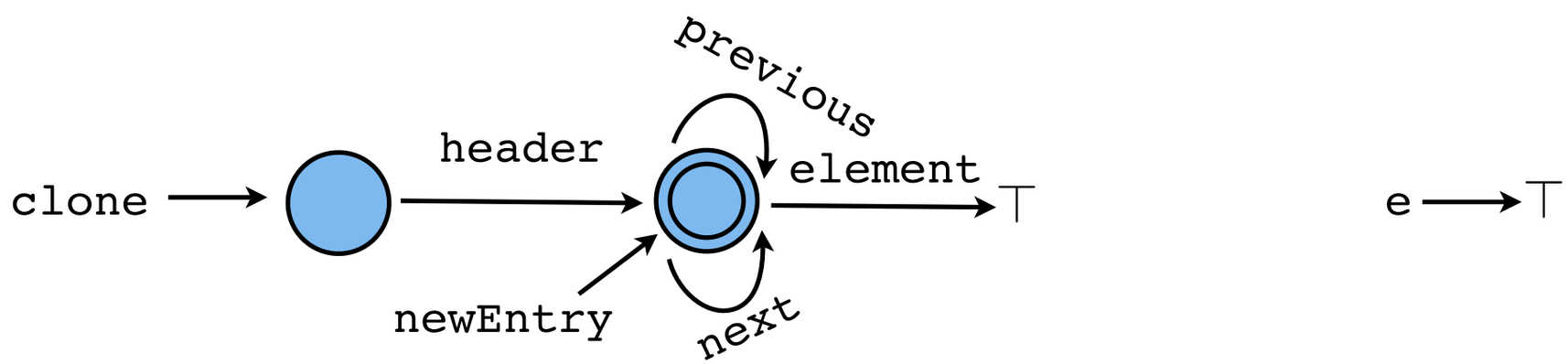

```
      newEntry.previous = clone.header.previous;
      newEntry.previous.next = newEntry;
      newEntry.next.previous = newEntry;
  }
  return clone;
}
```

```
public Object clone() {
  LinkedList<E> clone = null;
  clone = (LinkedList<E>) super.clone();
  clone.header = new Entry<E>;
  clone.header.next = clone.header.previous = clone.header;
  for (Entry<E> e = this.header.next; e != this.header; e = e.next) {
      Entry<E> newEntry = new Entry<E>;
      newEntry.element = e.element;
      newEntry.next = clone.header;
      newEntry.previous = clone.header.previous;
```



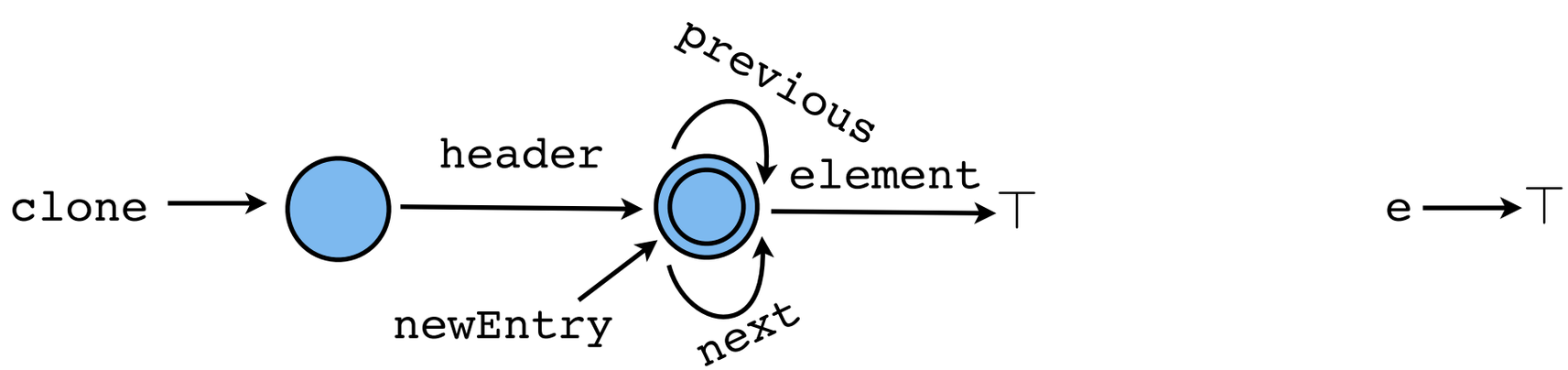

```
      newEntry.previous.next = newEntry;
      newEntry.next.previous = newEntry;
  }
  return clone;
}
```

```
public Object clone() {
  LinkedList<E> clone = null;
  clone = (LinkedList<E>) super.clone();
  clone.header = new Entry<E>;
  clone.header.next = clone.header.previous = clone.header;
  for (Entry<E> e = this.header.next; e != this.header; e = e.next) {
      Entry<E> newEntry = new Entry<E>;
      newEntry.element = e.element;
      newEntry.next = clone.header;
      newEntry.previous = clone.header.previous;
      newEntry.previous.next = newEntry;
```



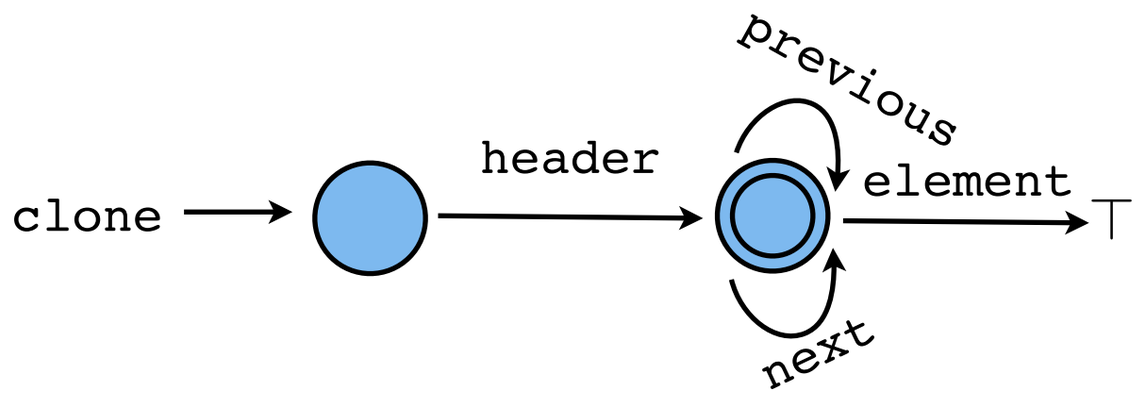

```
      newEntry.next.previous = newEntry;
  }
  return clone;
}
```

```
public Object clone() {
  LinkedList<E> clone = null;
  clone = (LinkedList<E>) super.clone();
  clone.header = new Entry<E>;
  clone.header.next = clone.header.previous = clone.header;
  for (Entry<E> e = this.header.next; e != this.header; e = e.next) {
      Entry<E> newEntry = new Entry<E>;
      newEntry.element = e.element;
      newEntry.next = clone.header;
      newEntry.previous = clone.header.previous;
      newEntry.previous.next = newEntry;
      newEntry.next.previous = newEntry;
```

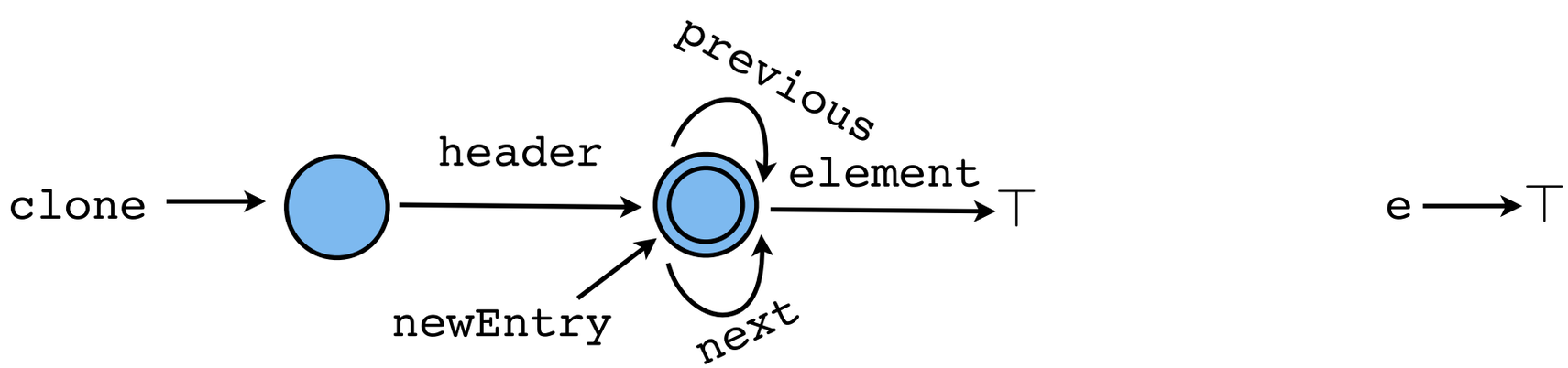



```
  }
  return clone;
}
```

```
public Object clone() {
  LinkedList<E> clone = null;
  clone = (LinkedList<E>) super.clone();
  clone.header = new Entry<E>;
  clone.header.next = clone.header.previous = clone.header;
```

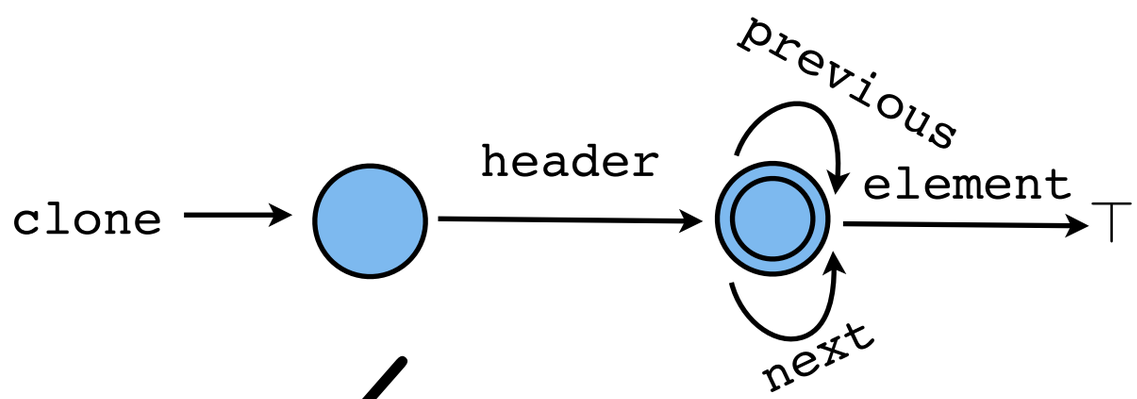



```
  for (Entry<E> e = this.header.next; e != this.header; e = e.next) {
      ...
```

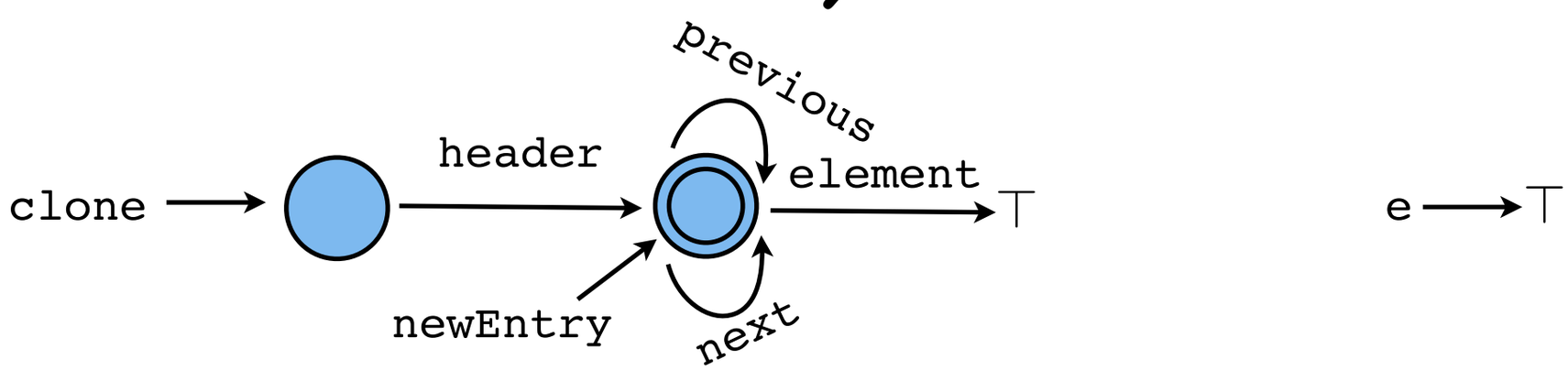



```
  }
  return clone;
}
```

```
public Object clone() {
  LinkedList<E> clone = null;
  clone = (LinkedList<E>) super.clone();
  clone.header = new Entry<E>;
  clone.header.next = clone.header.previous = clone.header;
```

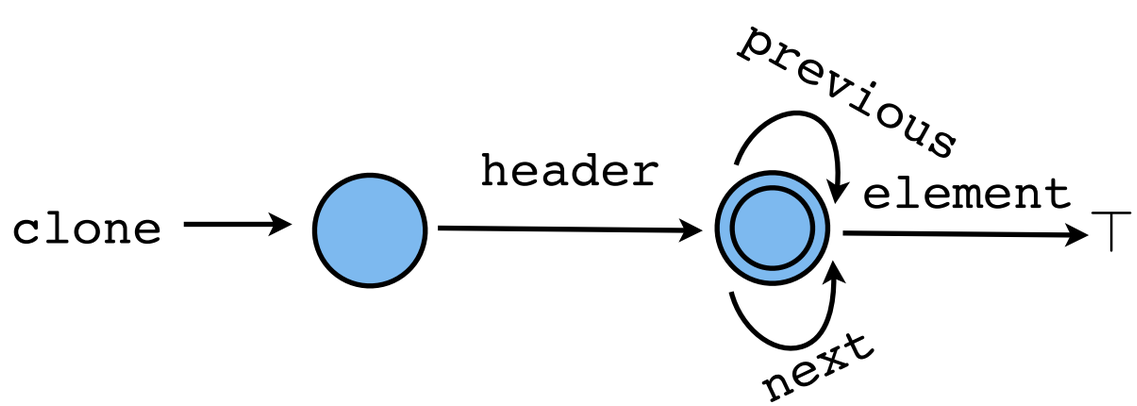



```
  for (Entry<E> e = this.header.next; e != this.header; e = e.next) {
      ...
```

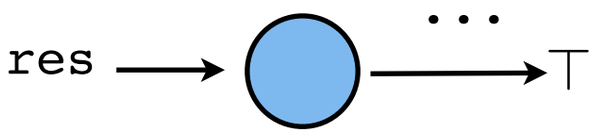



```
  }
  return clone;
}
```

```java
public Object clone() {
  LinkedList<E> clone = null;
  clone = (LinkedList<E>) super.clone();
  clone.header = new Entry<E>;
  clone.header.next = clone.header.previous = clone.header;
  for (Entry<E> e = this.header.next; e != this.header; e = e.next) {
      Entry<E> newEntry = new Entry<E>;
      newEntry.element = e.element;
      newEntry.next = clone.header;
      newEntry.previous = clone.header.previous;
      newEntry.previous.next = newEntry;
      newEntry.next.previous = newEntry;
  }
  return clone;
}
```



Experiments

- Tested on GNU ClassPath
- 285 clone methods
  - 253 return at least 

  - 78 methods produced more complex graphs
- Some case studies on more complex copy policies
  - java.lang.LinkedList
  - verification of «deep clone» in gnu.xml.Stylesheet

The JAVASEC project

- Commissioned by the French National Agency for Security of Information systems
- Analysis of security of Java and JVMs
  - Language features
  - Secure programming guidelines
  - Strengthening of a JVM
    - extended byte code verification
    - secure memory
  - Evaluation/certification of secure JVM

- http://www.ssi.gouv.fr/site_article226.html

Conclusions

- Cloning was left to the programmer
  - a source of mis-understanding
  - no semantics
- Declare copy policies for copy methods
- Check copy policies with a type system
  - Implemented
  - Formalized