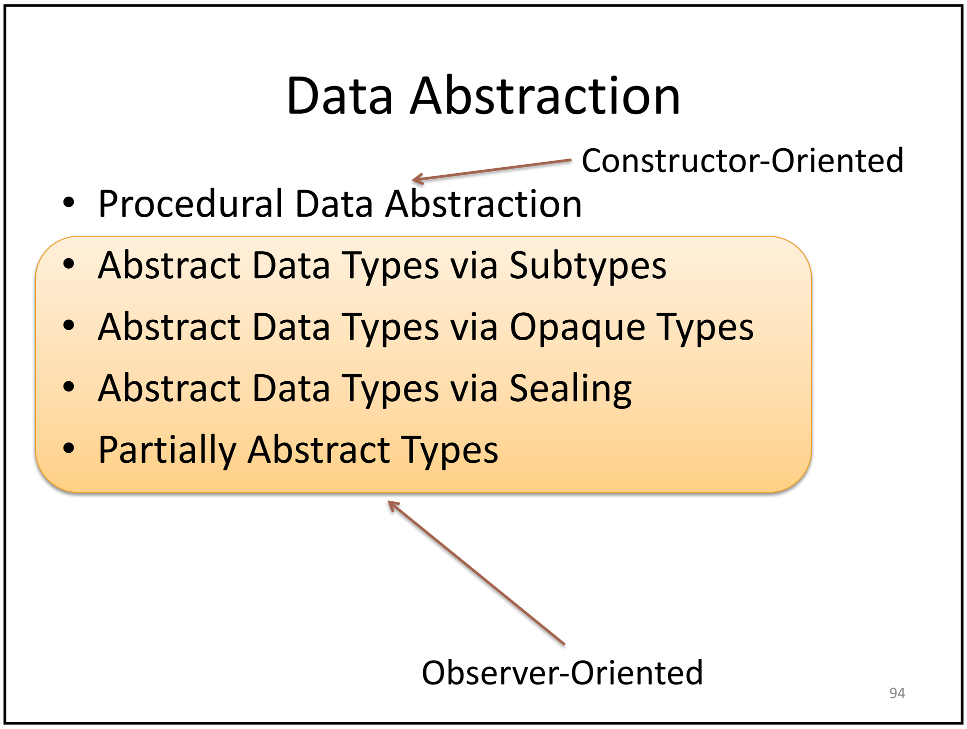
SERVICE ABSTRACTION: DATA ABSTRACTION

93



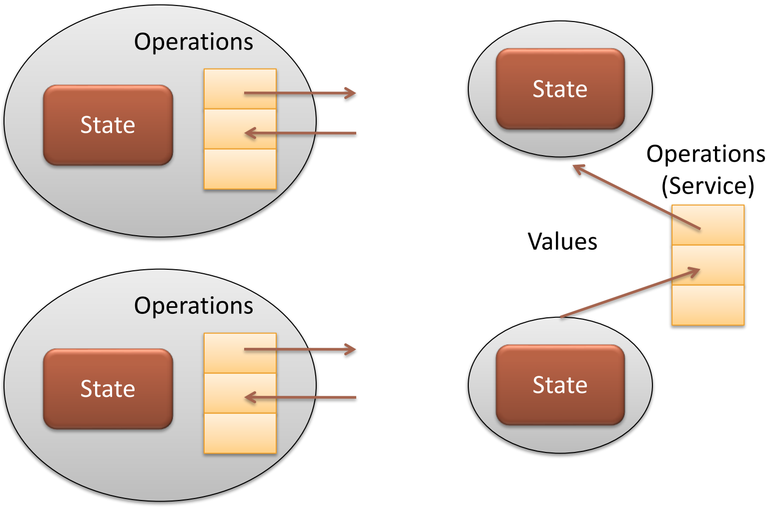
Data Abstraction

Constructor-Oriented

- Procedural Data Abstraction
- Abstract Data Types via Subtypes
- Abstract Data Types via Opaque Types
- Abstract Data Types via Sealing
- Partially Abstract Types

Observer-Oriented

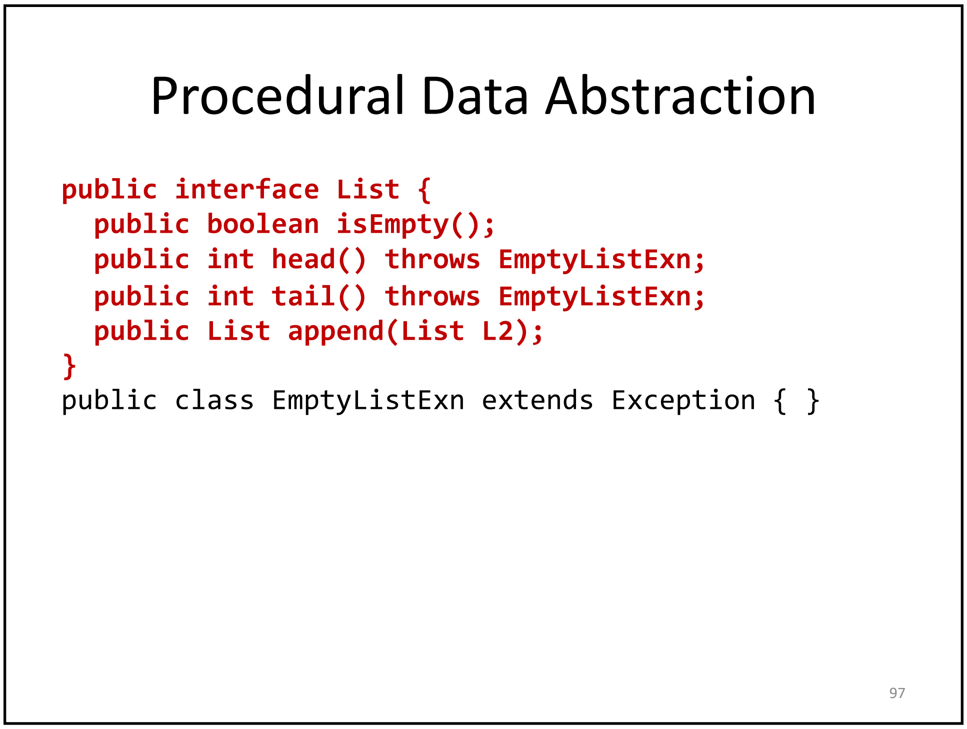
94



Procedural Data Abstraction

Abstract Data Types

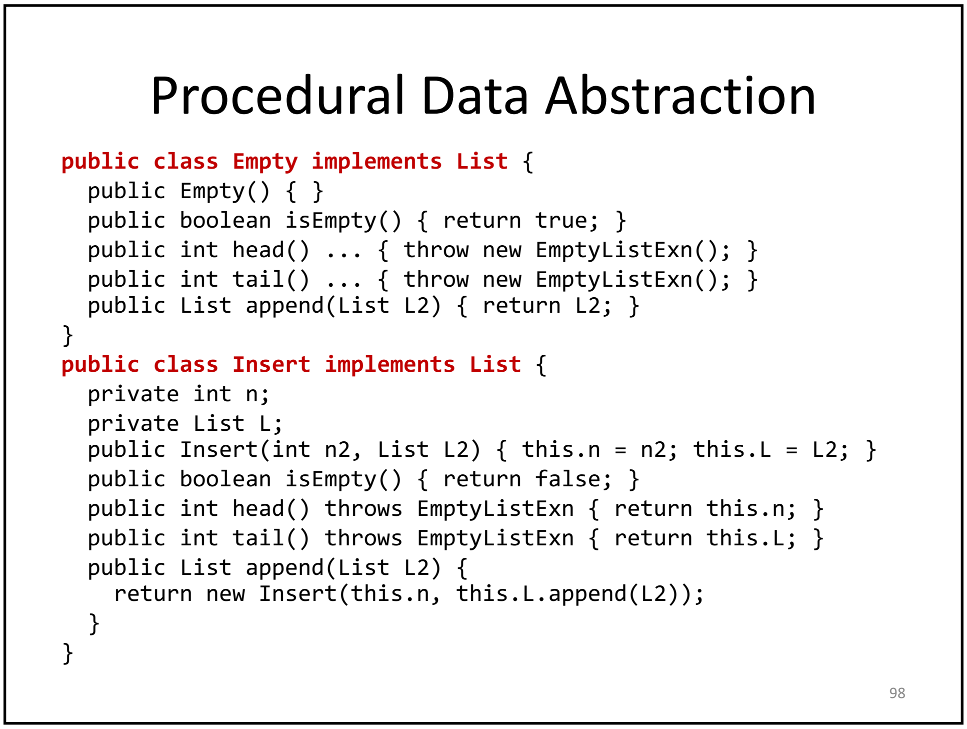
SERVICE ABSTRACTION: PROCEDURAL DATA ABSTRACTION



Procedural Data Abstraction

```
public interface List {
  public boolean isEmpty();
  public int head() throws EmptyListExn;
  public int tail() throws EmptyListExn;
  public List append(List L2);
}
public class EmptyListExn extends Exception { }
```

97



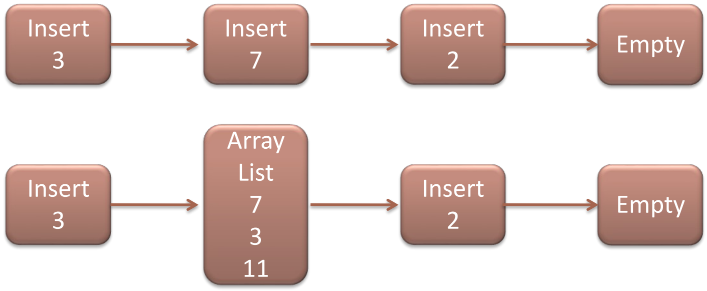
Procedural Data Abstraction

```
public class Empty implements List {
  public Empty() { }
  public boolean isEmpty() { return true; }
  public int head() ... { throw new EmptyListExn(); }
  public int tail() ... { throw new EmptyListExn(); }
  public List append(List L2) { return L2; }
}
public class Insert implements List {
  private int n;
  private List L;
  public Insert(int n2, List L2) { this.n = n2; this.L = L2; }
  public boolean isEmpty() { return false; }
  public int head() throws EmptyListExn { return this.n; }
  public int tail() throws EmptyListExn { return this.L; }
  public List append(List L2) {
    return new Insert(this.n, this.L.append(L2));
  }
}
```

98

Procedural Data Abstraction: Extension

- Suppose we want to add a new case in our list data type
  - Two cases already: Empty and Insert
  - Add a third case: ArrayList



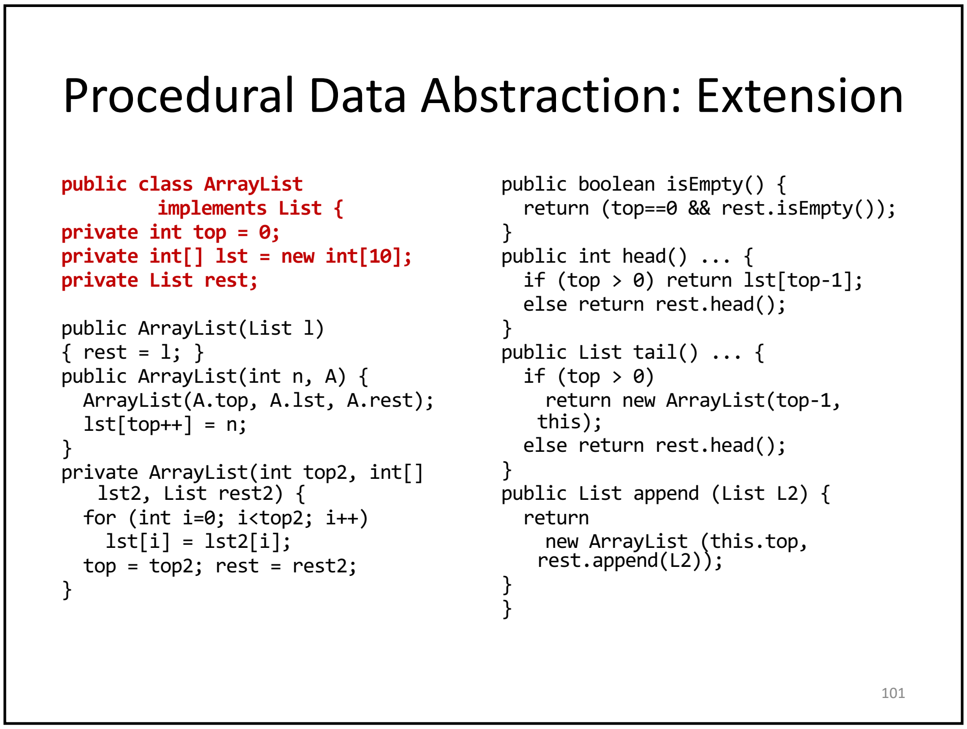
Insert 3 → Insert 7 → Insert 2 → Empty

Insert 3 → Array List 7 3 11 → Insert 2 → Empty

Procedural Data Abstraction: Extension

```
public class ArrayList
      implements List {

private int top = 0;

private int[] lst =
            new int[10];

private List rest;
```

```
public boolean isEmpty()
  { ... }

public int head()
  throws EmptyListExn
  { ... }

public List tail()
  throws EmptyListExn
  { ... }

public List append (List L2)
  { ... }
}
```



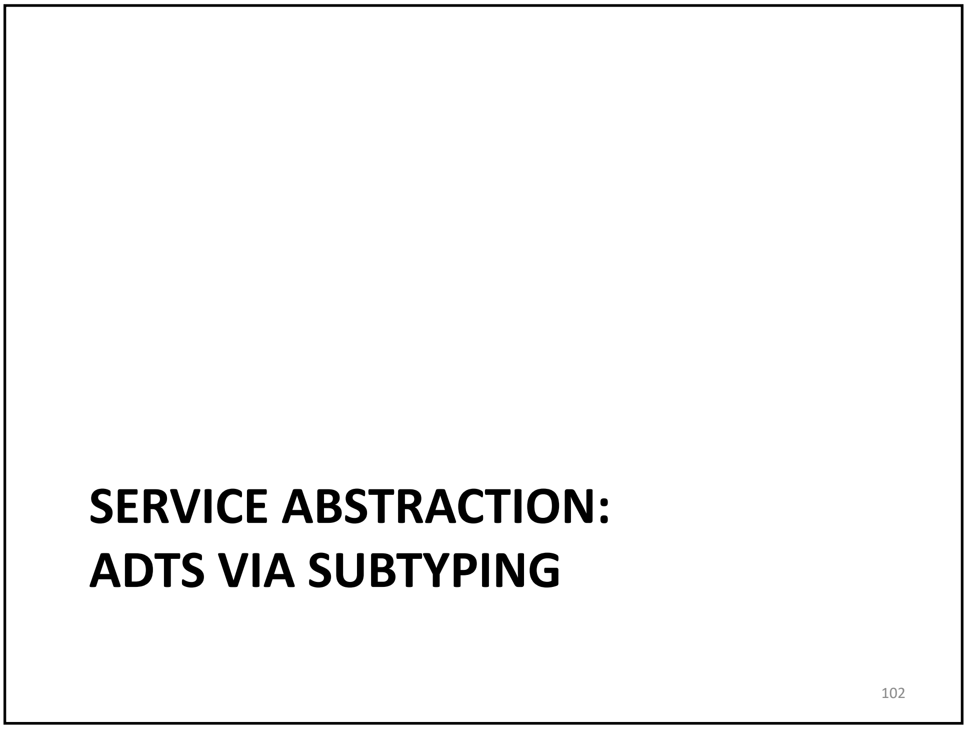
Procedural Data Abstraction: Extension

```
public class ArrayList
        implements List {
private int top = 0;
private int[] lst = new int[10];
private List rest;

public ArrayList(List l)
{ rest = l; }
public ArrayList(int n, A) {
  ArrayList(A.top, A.lst, A.rest);
  lst[top++] = n;
}
private ArrayList(int top2, int[]
   lst2, List rest2) {
  for (int i=0; i<top2; i++)
    lst[i] = lst2[i];
  top = top2; rest = rest2;
}
```

```
public boolean isEmpty() {
  return (top==0 && rest.isEmpty());
}
public int head() ... {
  if (top > 0) return lst[top-1];
  else return rest.head();
}
public List tail() ... {
  if (top > 0)
    return new ArrayList(top-1,
   this);
  else return rest.head();
}
public List append (List L2) {
  return
    new ArrayList (this.top,
   rest.append(L2));
}
}
```

101

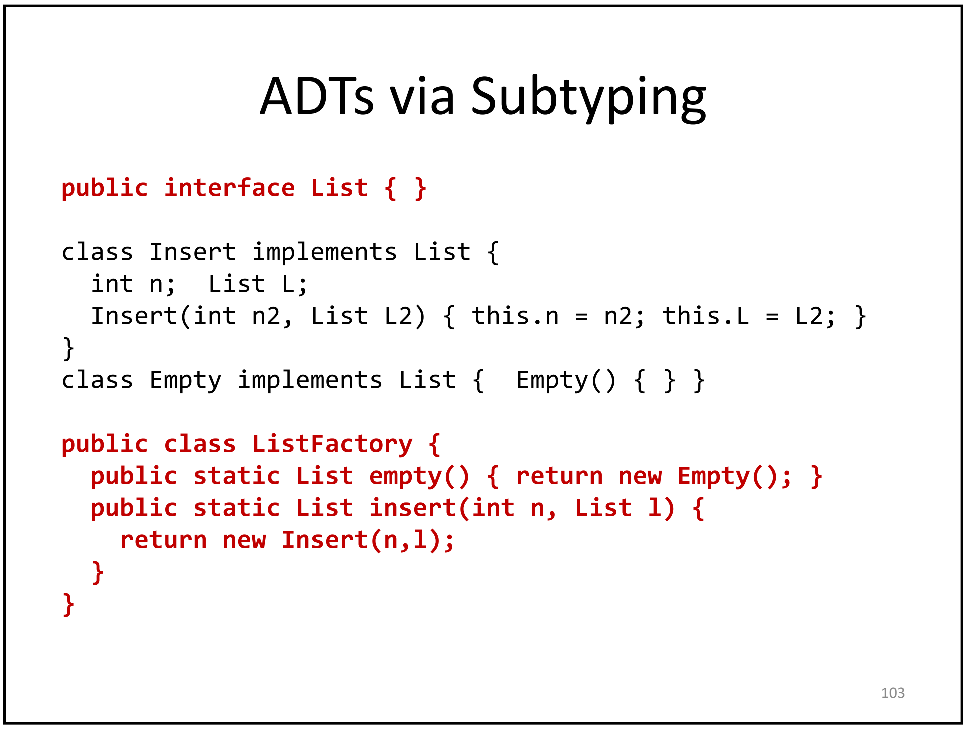
101



SERVICE ABSTRACTION:
ADTS VIA SUBTYPING

102

102

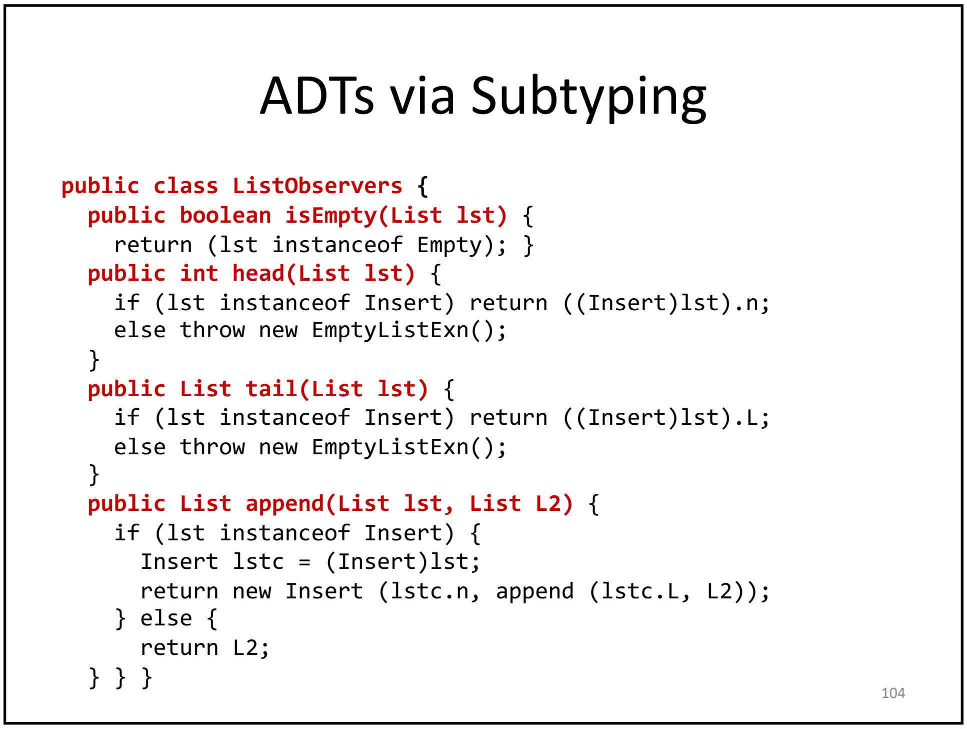


ADTs via Subtyping

```
public interface List { }

class Insert implements List {
  int n;  List L;
  Insert(int n2, List L2) { this.n = n2; this.L = L2; }
}
class Empty implements List {  Empty() { } }

public class ListFactory {
  public static List empty() { return new Empty(); }
  public static List insert(int n, List l) {
    return new Insert(n,l);
  }
}
```

103

103

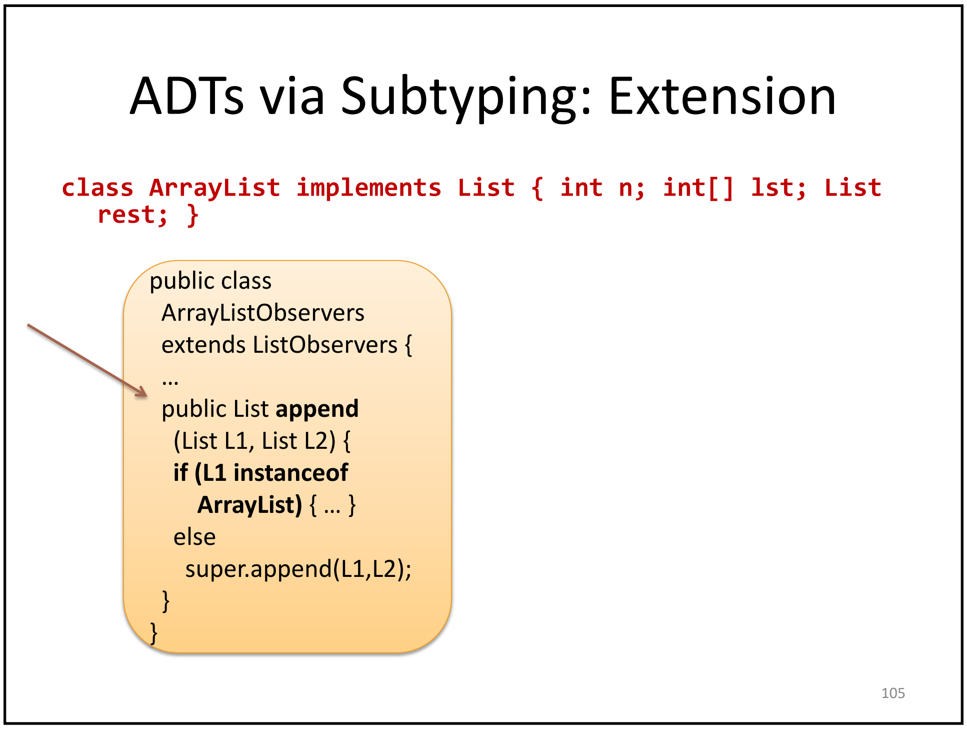


ADTs via Subtyping

```
public class ListObservers {
  public boolean isEmpty(List lst) {
    return (lst instanceof Empty); }
  public int head(List lst) {
    if (lst instanceof Insert) return ((Insert)lst).n;
    else throw new EmptyListExn();
  }
  public List tail(List lst) {
    if (lst instanceof Insert) return ((Insert)lst).L;
    else throw new EmptyListExn();
  }
  public List append(List lst, List L2) {
    if (lst instanceof Insert) {
      Insert lstc = (Insert)lst;
      return new Insert (lstc.n, append (lstc.L, L2));
    } else {
      return L2;
  } } }
```

104

104



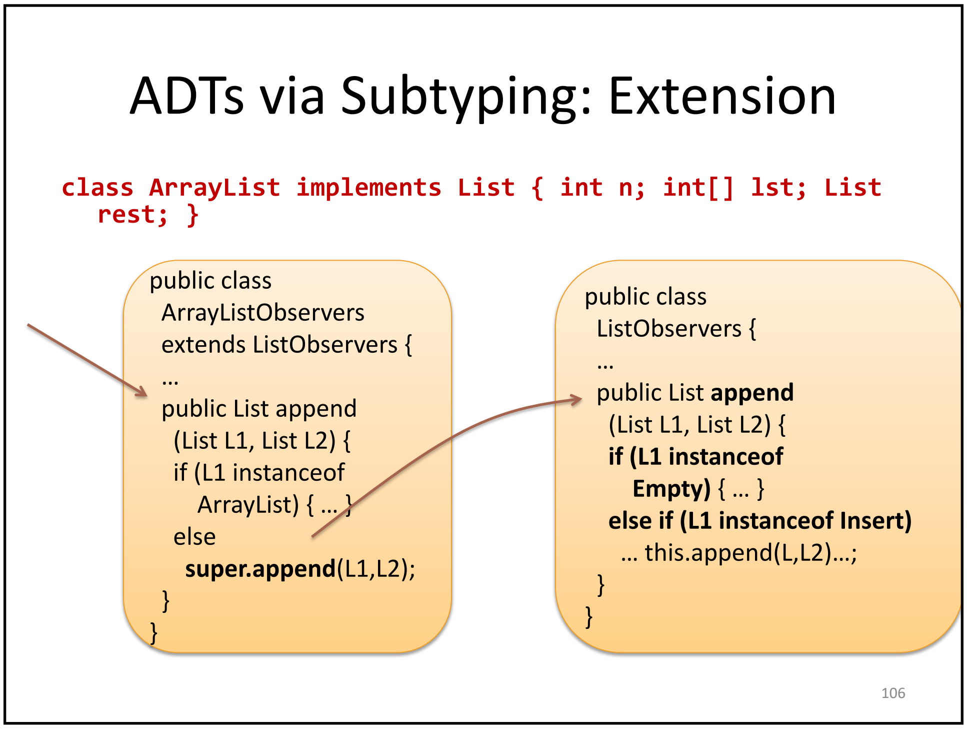
ADTs via Subtyping: Extension

```
class ArrayList implements List { int n; int[] lst; List
   rest; }
```

```
public class
  ArrayListObservers
  extends ListObservers {
  …
  public List append
    (List L1, List L2) {
    if (L1 instanceof
       ArrayList) { … }
    else
      super.append(L1,L2);
  }
}
```

105

105



ADTs via Subtyping: Extension

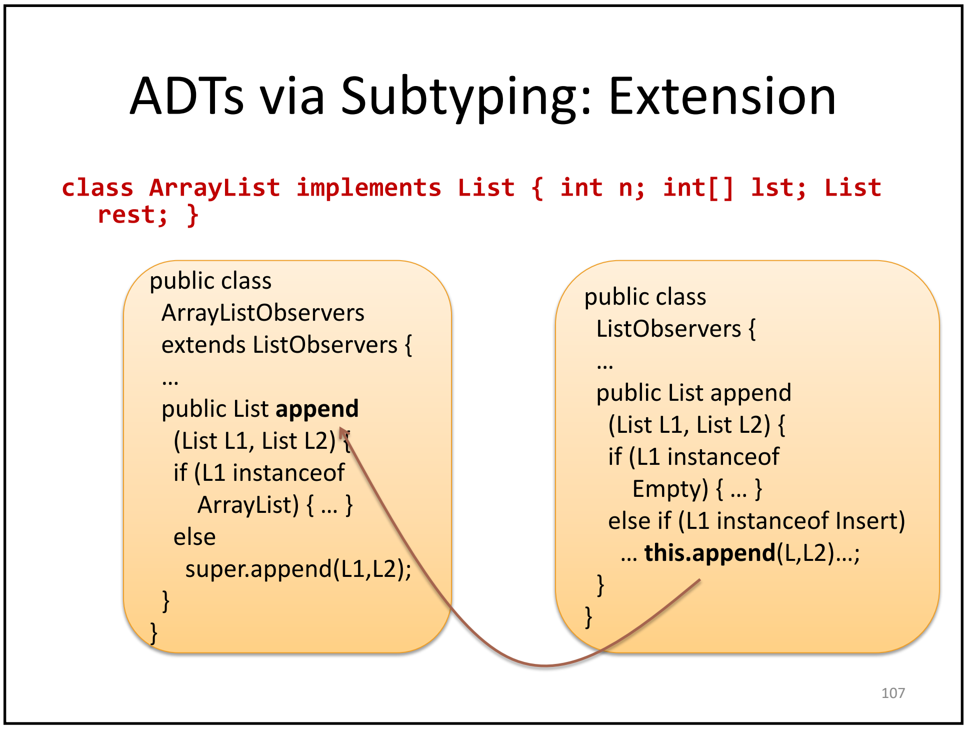
```
class ArrayList implements List { int n; int[] lst; List
   rest; }
```

```
public class
  ArrayListObservers
  extends ListObservers {
  …
  public List append
    (List L1, List L2) {
    if (L1 instanceof
       ArrayList) { … }
    else
      super.append(L1,L2);
  }
}
```

```
public class
  ListObservers {
  …
  public List append
    (List L1, List L2) {
    if (L1 instanceof
       Empty) { … }
    else if (L1 instanceof Insert)
      … this.append(L,L2)…;
  }
}
```

106

106



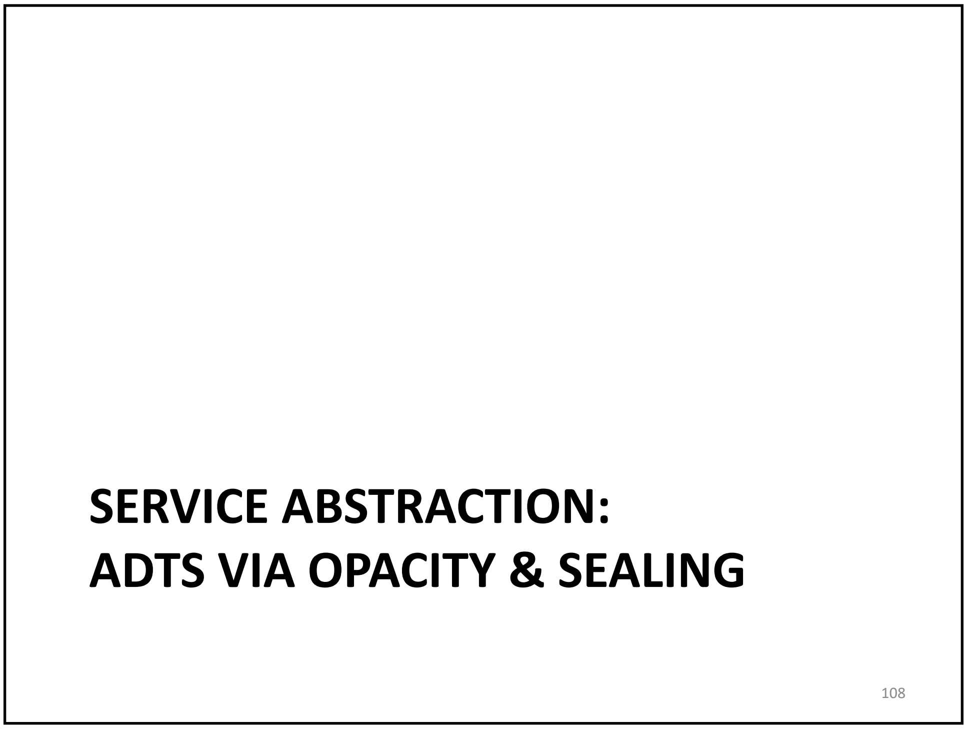
ADTs via Subtyping: Extension

```
class ArrayList implements List { int n; int[] lst; List
   rest; }
```

```
public class
  ArrayListObservers
  extends ListObservers {
  …
  public List append
    (List L1, List L2) {
    if (L1 instanceof
        ArrayList) { … }
    else
      super.append(L1,L2);
  }
}
```

```
public class
  ListObservers {
  …
  public List append
    (List L1, List L2) {
    if (L1 instanceof
        Empty) { … }
    else if (L1 instanceof Insert)
      … this.append(L,L2)…;
  }
}
```

107



SERVICE ABSTRACTION:
ADTS VIA OPACITY & SEALING

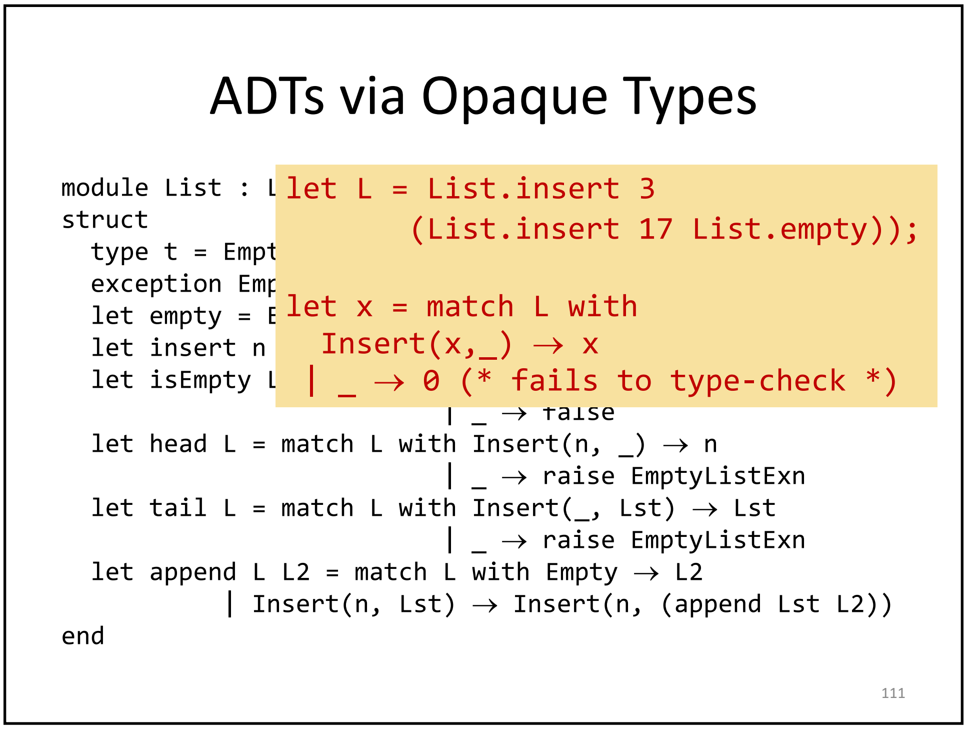
108

ADTs via Opaque Types

```
module type LIST =
sig
  type t
  exception EmptyListExn
  val empty : t
  val insert : e → t → t
  val isEmpty : t → bool
  val head : t → e
  val tail : t → t
  val append : t → t → t
end
```

ADTs via Opaque Types

```
module List : LIST =
struct
  type t = Empty | Insert of int * t
  exception EmptyListExn
  let empty = Empty
  let insert n L = Insert(n, L)
  let isEmpty L = match L with Empty → true
                             | _ → false
  let head L = match L with Insert(n, _) → n
                             | _ → raise EmptyListExn
  let tail L = match L with Insert(_, Lst) → Lst
                             | _ → raise EmptyListExn
  let append L L2 = match L with Empty → L2
                 | Insert(n, Lst) → Insert(n, (append Lst L2))
end
```



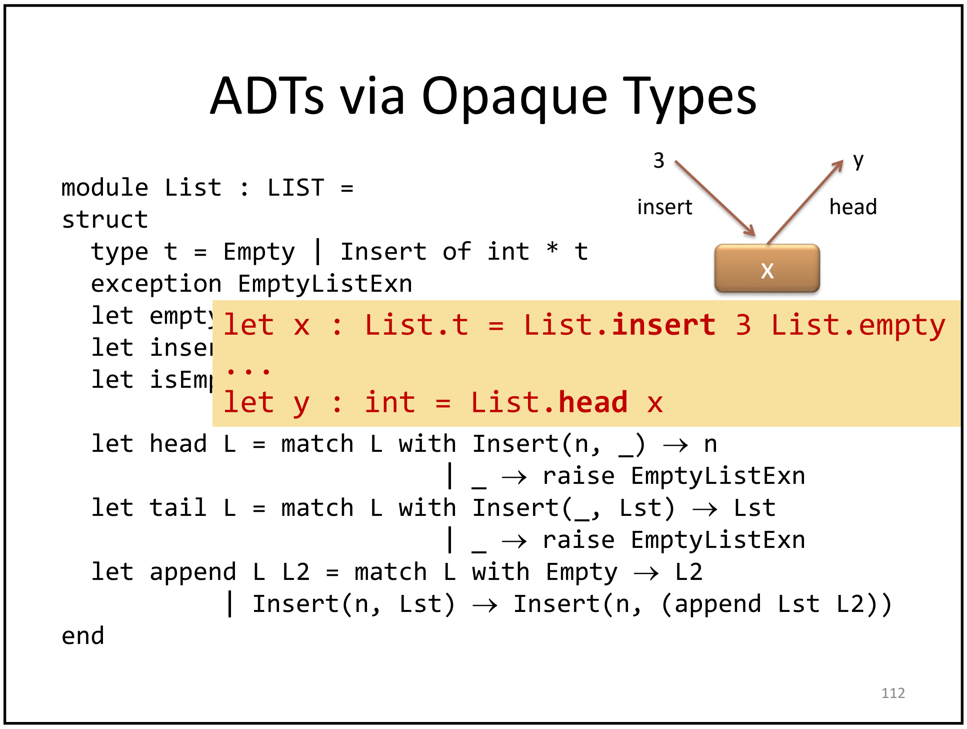
ADTs via Opaque Types

```
module List : L
struct
  type t = Empt
  exception Emp
  let empty = E
  let insert n
  let isEmpty L
                      | _ → false
  let head L = match L with Insert(n, _) → n
                          | _ → raise EmptyListExn
  let tail L = match L with Insert(_, Lst) → Lst
                          | _ → raise EmptyListExn
  let append L L2 = match L with Empty → L2
             | Insert(n, Lst) → Insert(n, (append Lst L2))
end
```

```
let L = List.insert 3
        (List.insert 17 List.empty));

let x = match L with
   Insert(x,_) → x
 | _ → 0 (* fails to type-check *)
```

111

111



ADTs via Opaque Types

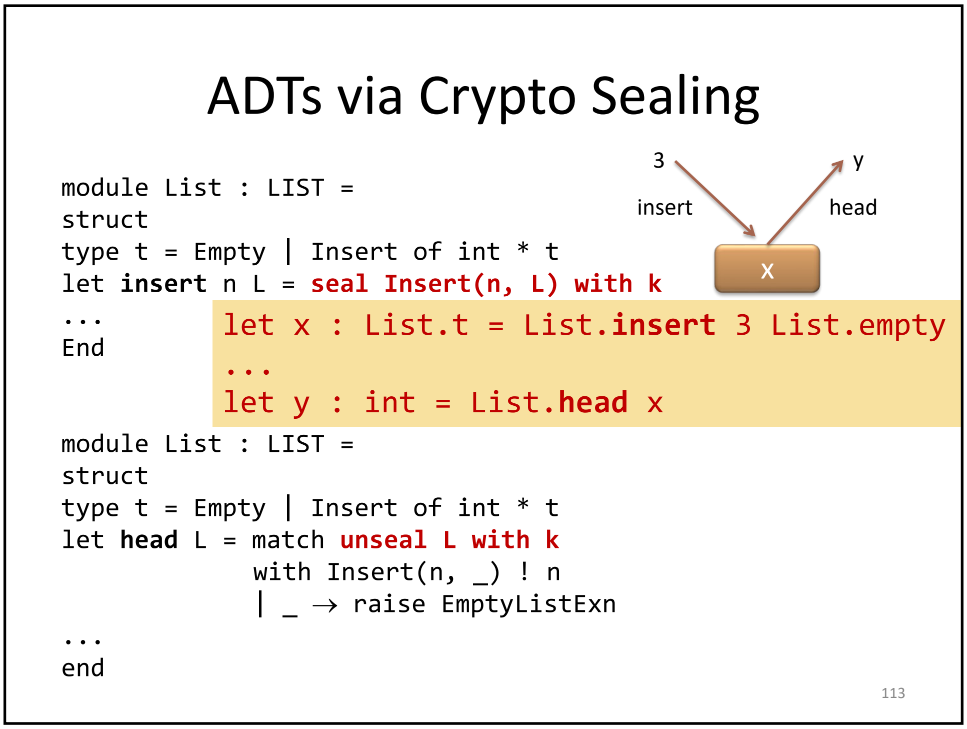
3
insert
y
head
x

```
module List : LIST =
struct
  type t = Empty | Insert of int * t
  exception EmptyListExn
  let empty
  let inse
  let isEm

  let head L = match L with Insert(n, _) → n
                          | _ → raise EmptyListExn
  let tail L = match L with Insert(_, Lst) → Lst
                          | _ → raise EmptyListExn
  let append L L2 = match L with Empty → L2
             | Insert(n, Lst) → Insert(n, (append Lst L2))
end
```

```
let x : List.t = List.insert 3 List.empty
...
let y : int = List.head x
```

112

112



ADTs via Crypto Sealing

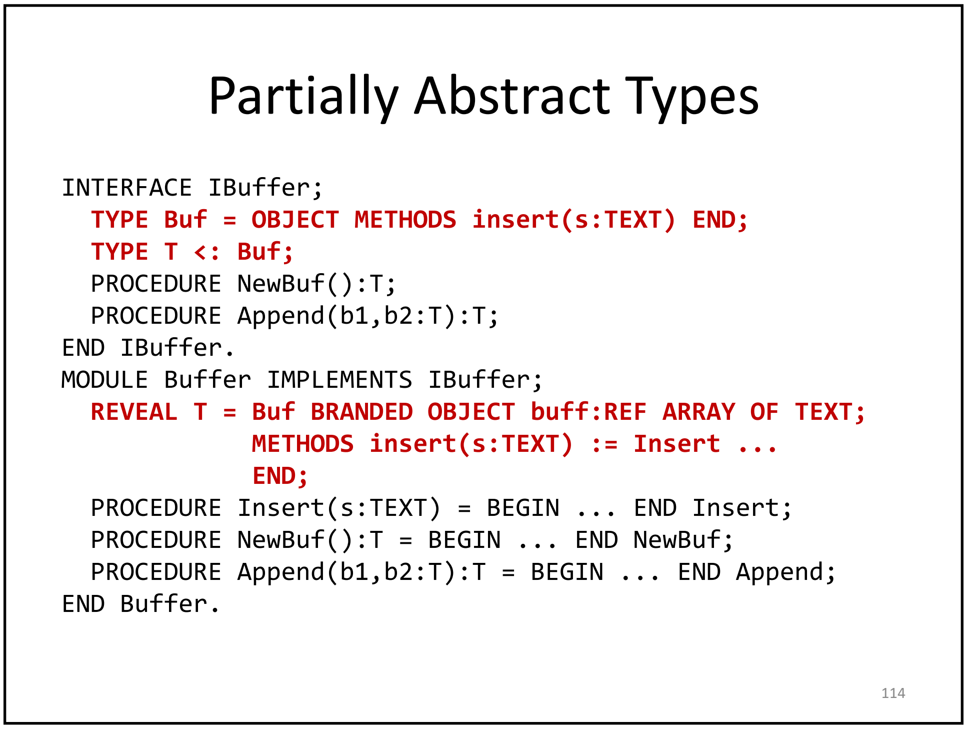
3
insert
y
head
x

```
module List : LIST =
struct
type t = Empty | Insert of int * t
let insert n L = seal Insert(n, L) with k
...
End
```

```
let x : List.t = List.insert 3 List.empty
...
let y : int = List.head x
```

```
module List : LIST =
struct
type t = Empty | Insert of int * t
let head L = match unseal L with k
             with Insert(n, _) ! n
             | _ → raise EmptyListExn
...
end
```

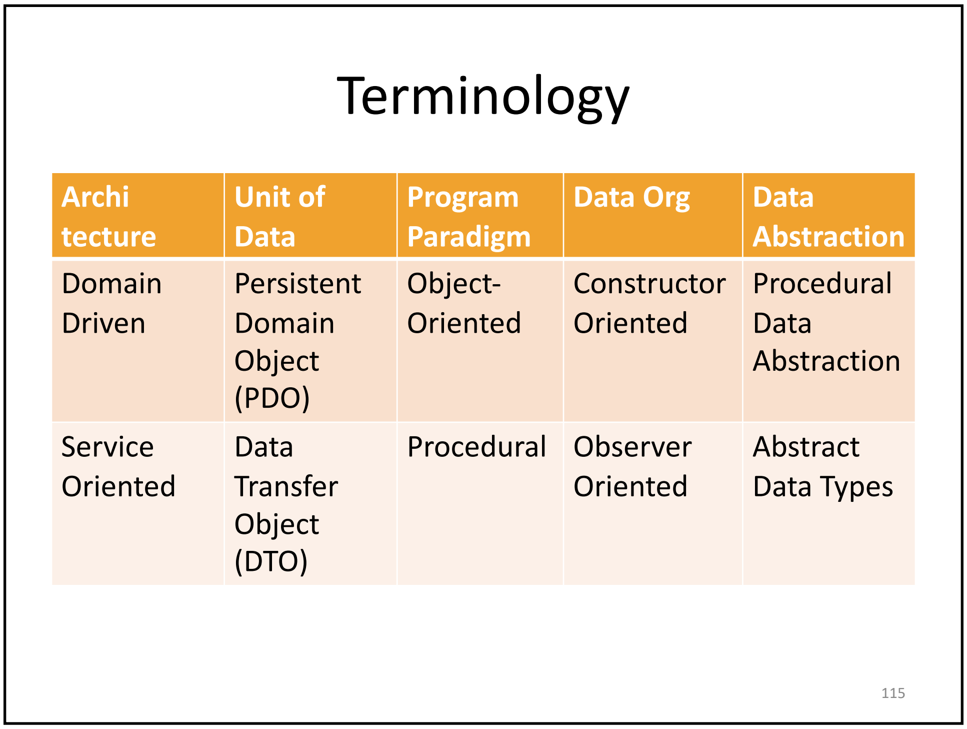
113



Partially Abstract Types

```
INTERFACE IBuffer;
  TYPE Buf = OBJECT METHODS insert(s:TEXT) END;
  TYPE T <: Buf;
  PROCEDURE NewBuf():T;
  PROCEDURE Append(b1,b2:T):T;
END IBuffer.
MODULE Buffer IMPLEMENTS IBuffer;
  REVEAL T = Buf BRANDED OBJECT buff:REF ARRAY OF TEXT;
             METHODS insert(s:TEXT) := Insert ...
             END;
  PROCEDURE Insert(s:TEXT) = BEGIN ... END Insert;
  PROCEDURE NewBuf():T = BEGIN ... END NewBuf;
  PROCEDURE Append(b1,b2:T):T = BEGIN ... END Append;
END Buffer.
```

114



Terminology

Archi tecture	Unit of Data	Program Paradigm	Data Org	Data Abstraction
Domain Driven	Persistent Domain Object (PDO)	Object-Oriented	Constructor Oriented	Procedural Data Abstraction
Service Oriented	Data Transfer Object (DTO)	Procedural	Observer Oriented	Abstract Data Types