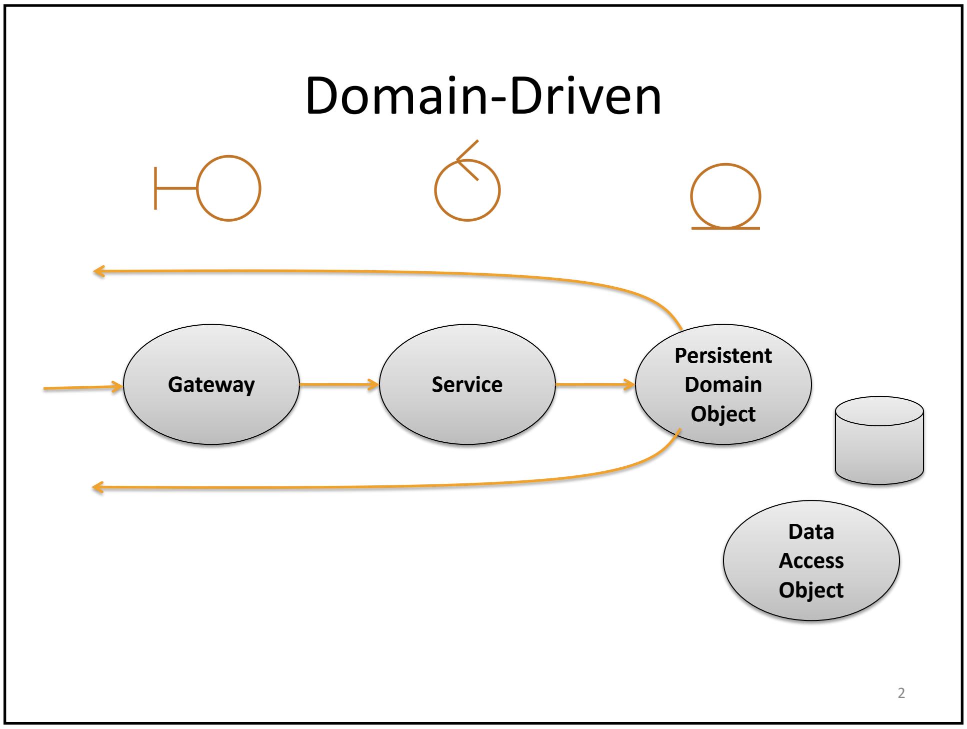
Service Oriented Architecture

Dominic Duggan

Stevens Institute of Technology

1



Domain-Driven

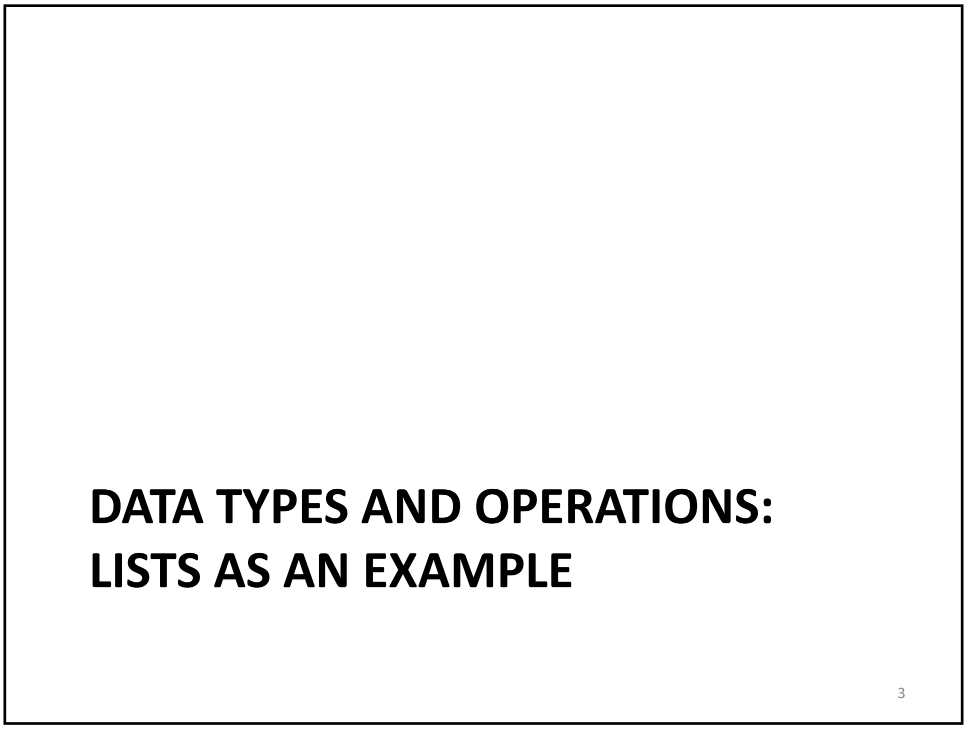
Gateway

Service

Persistent Domain Object

Data Access Object

2



DATA TYPES AND OPERATIONS: LISTS AS AN EXAMPLE

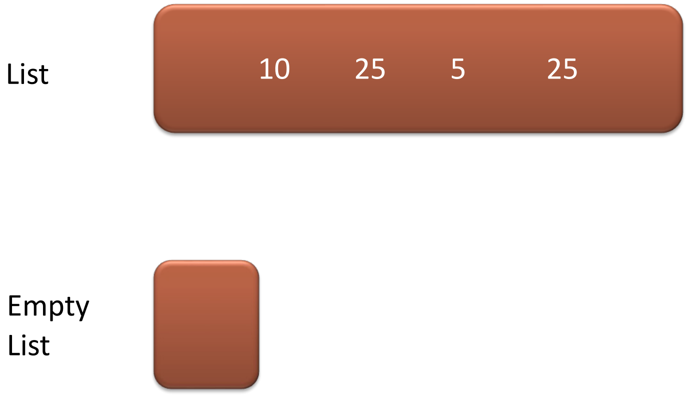
3

List == “Grocery list”

Apples
Oranges
Lettuce
Carrots
Bread
Wine
Eggs
Sausages
…

4

Lists



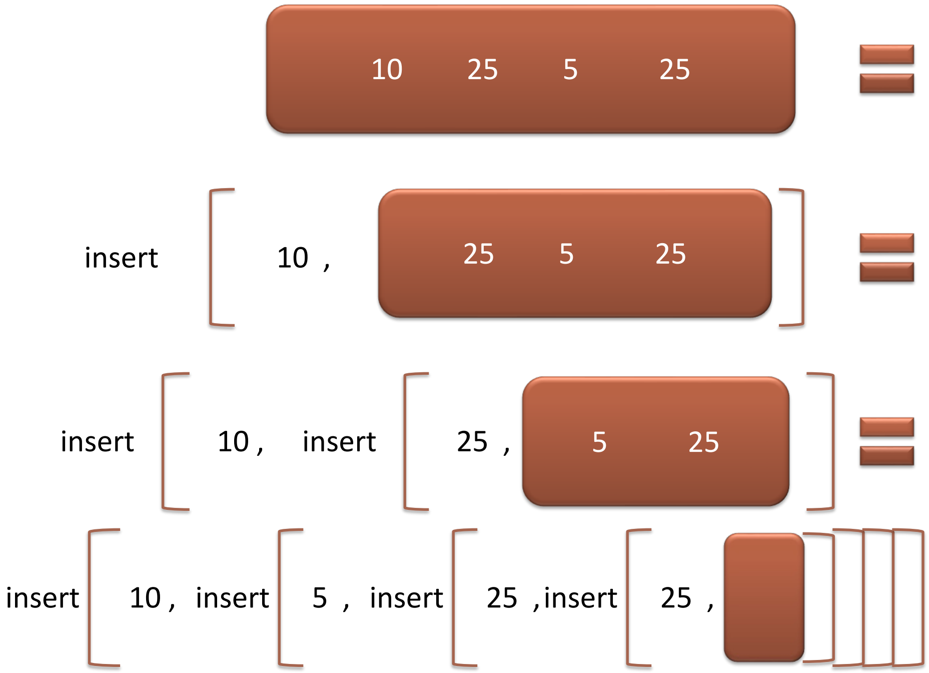
List

10 25 5 25

Empty List

5

5



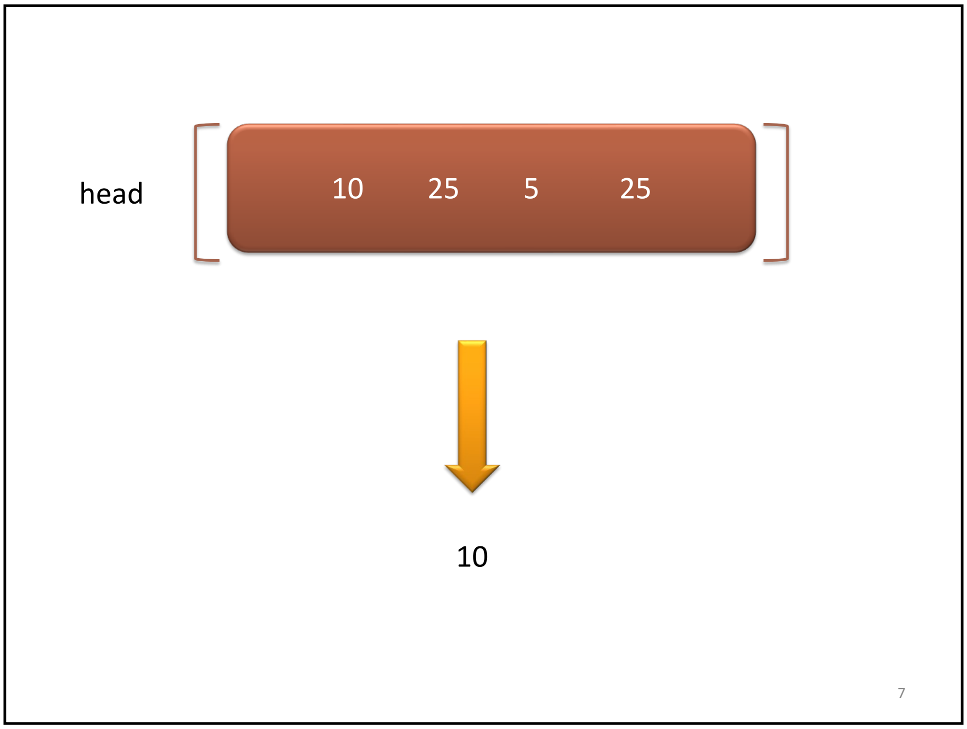
10 25 5 25

insert 10 , 25 5 25

insert 10 , insert 25 , 5 25

insert 10 , insert 5 , insert 25 ,insert 25 ,

6



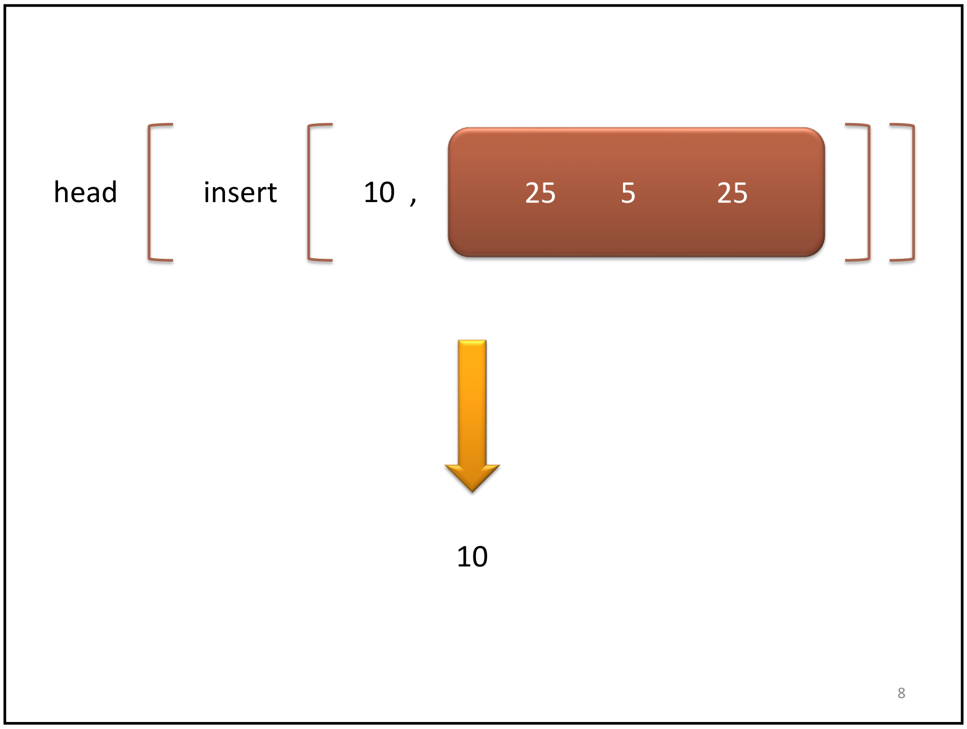
head

10 25 5 25

10

7

7



head

insert

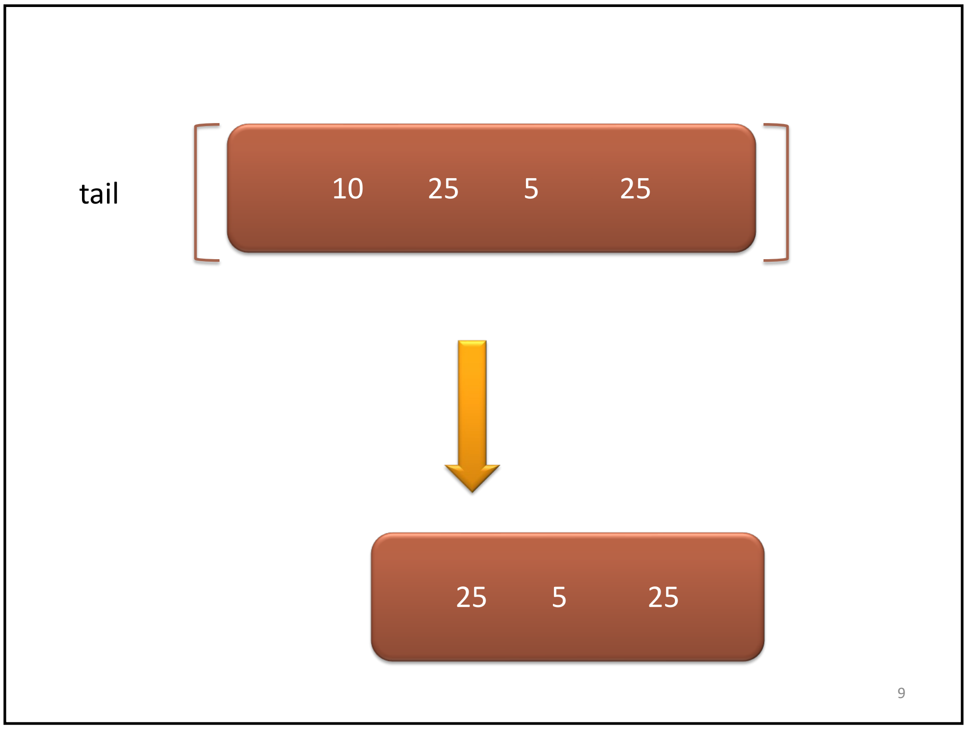
10 ,

25 5 25

10

8

8

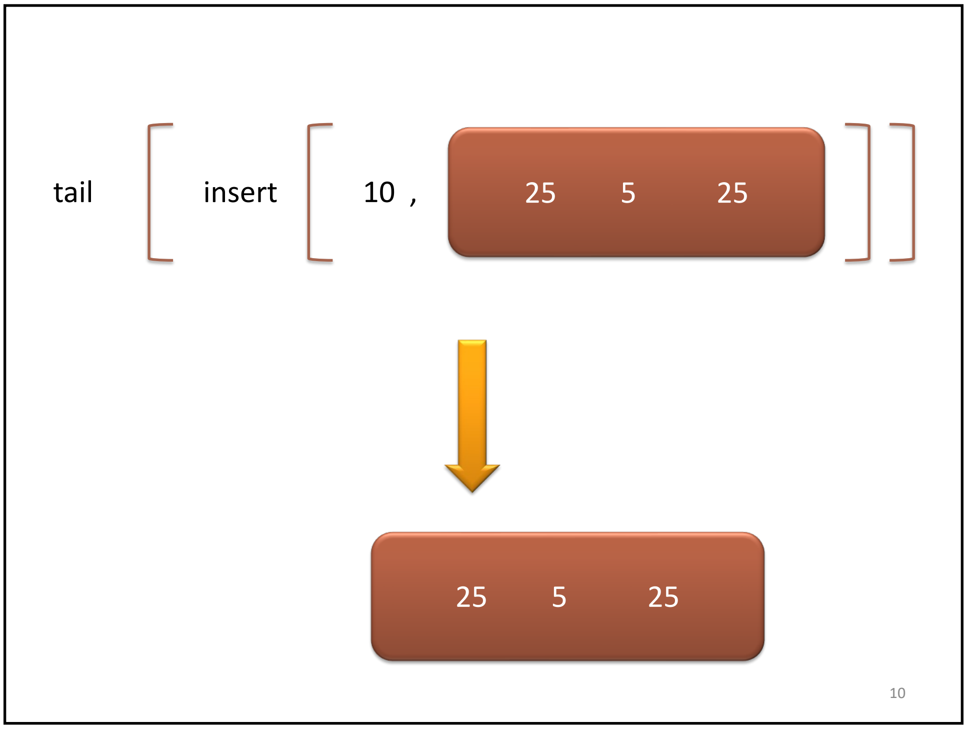


tail [10 25 5 25]

25 5 25

9

9

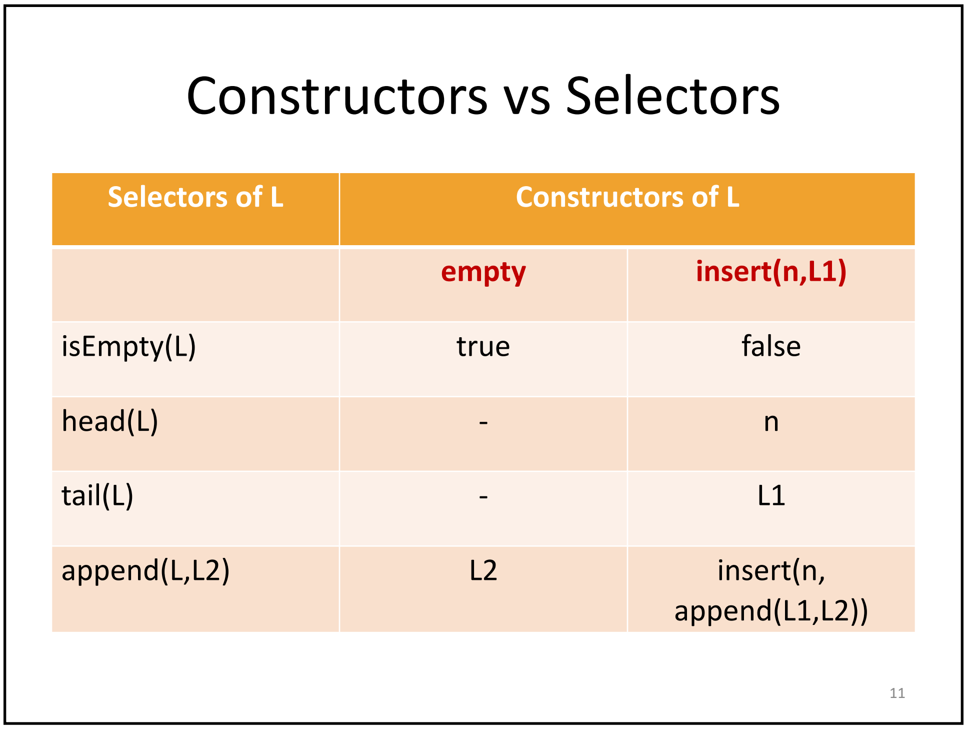


tail [insert [10 , 25 5 25]]

25 5 25

10

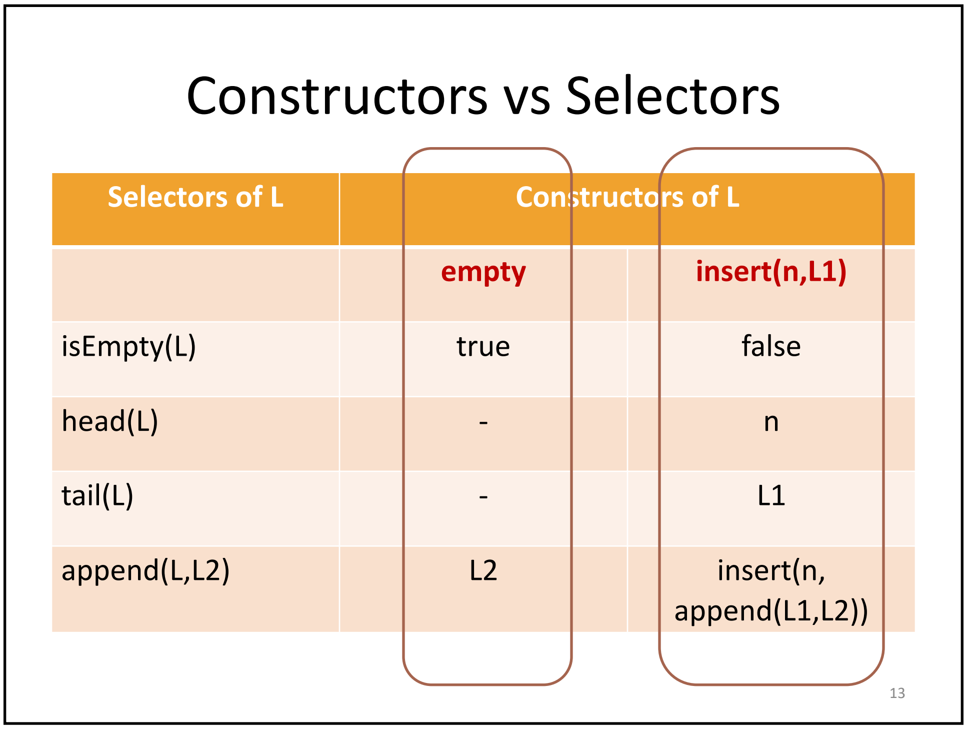
10



Constructors vs Selectors

Selectors of L	Constructors of L	
	empty	**insert(n,L1)**
isEmpty(L)	true	false
head(L)	-	n
tail(L)	-	L1
append(L,L2)	L2	insert(n, append(L1,L2))

CONSTRUCTOR-ORIENTED REPRESENTATION

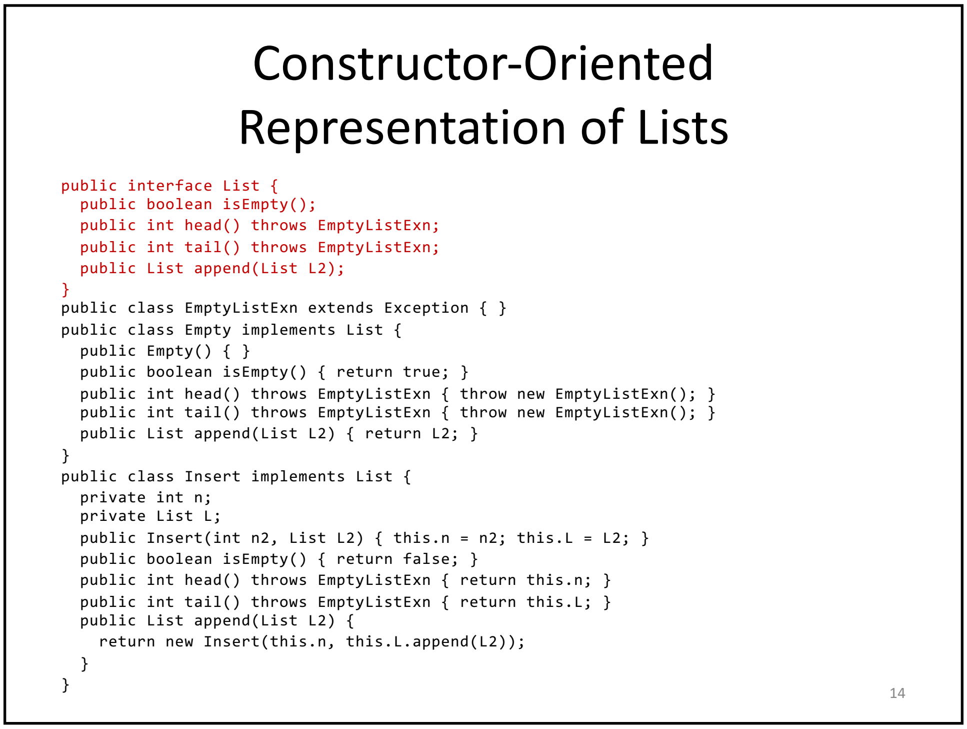


Constructors vs Selectors

Selectors of L	Constructors of L	
	empty	**insert(n,L1)**
isEmpty(L)	true	false
head(L)	-	n
tail(L)	-	L1
append(L,L2)	L2	insert(n, append(L1,L2))

13

13

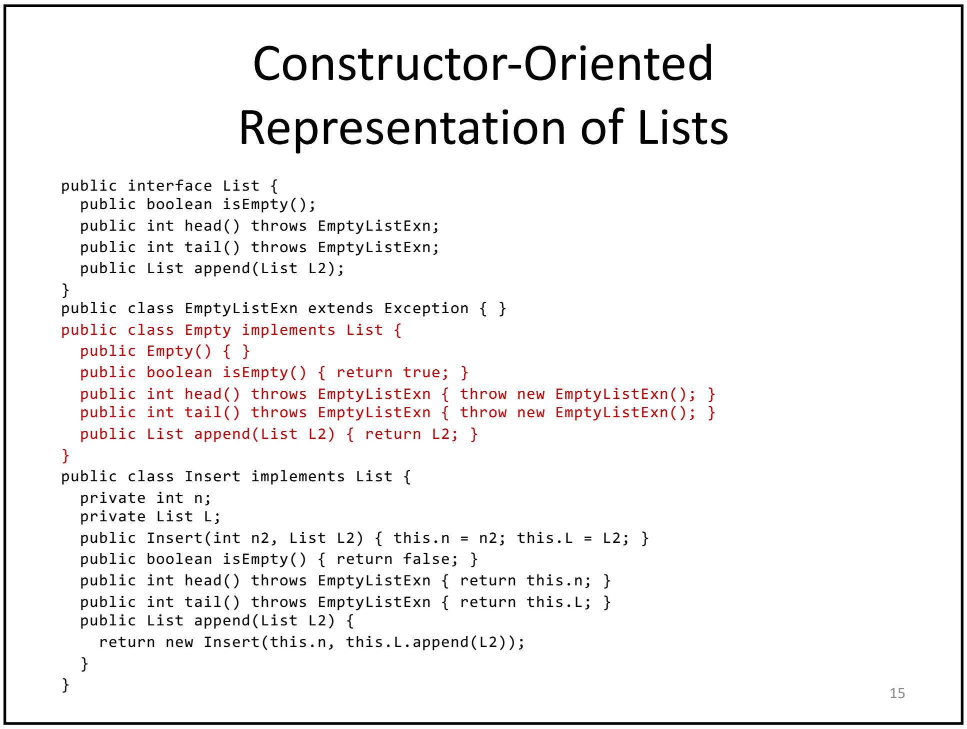


Constructor-Oriented Representation of Lists

```
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 { }
public class Empty implements List {
  public Empty() { }
  public boolean isEmpty() { return true; }
  public int head() throws EmptyListExn { throw new EmptyListExn(); }
  public int tail() throws EmptyListExn { 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));
  }
}
```

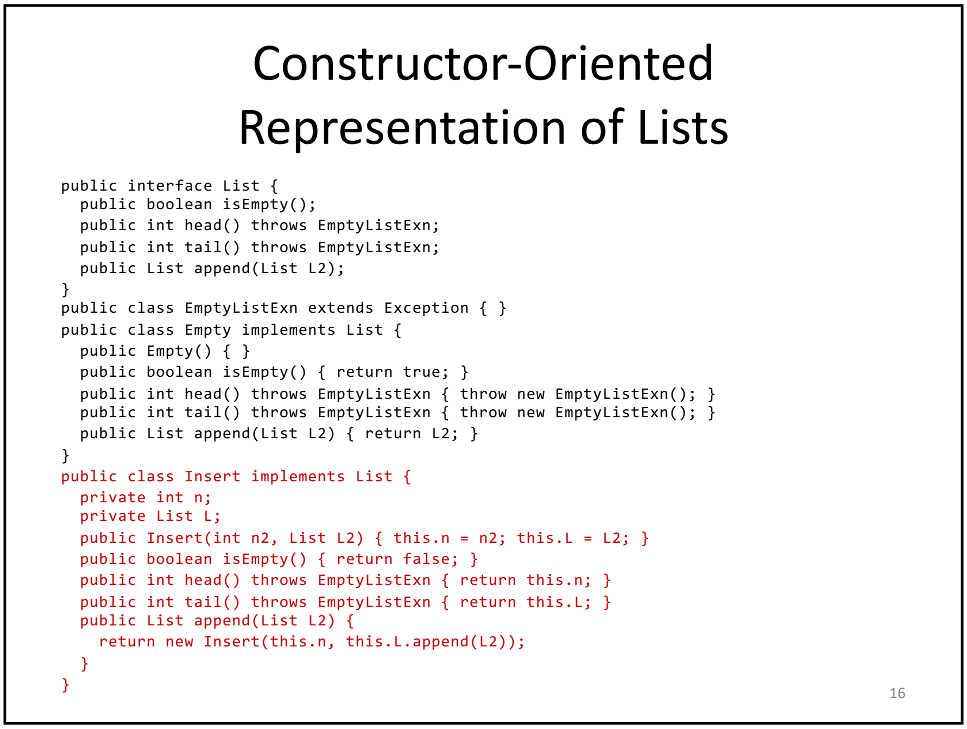
14

14



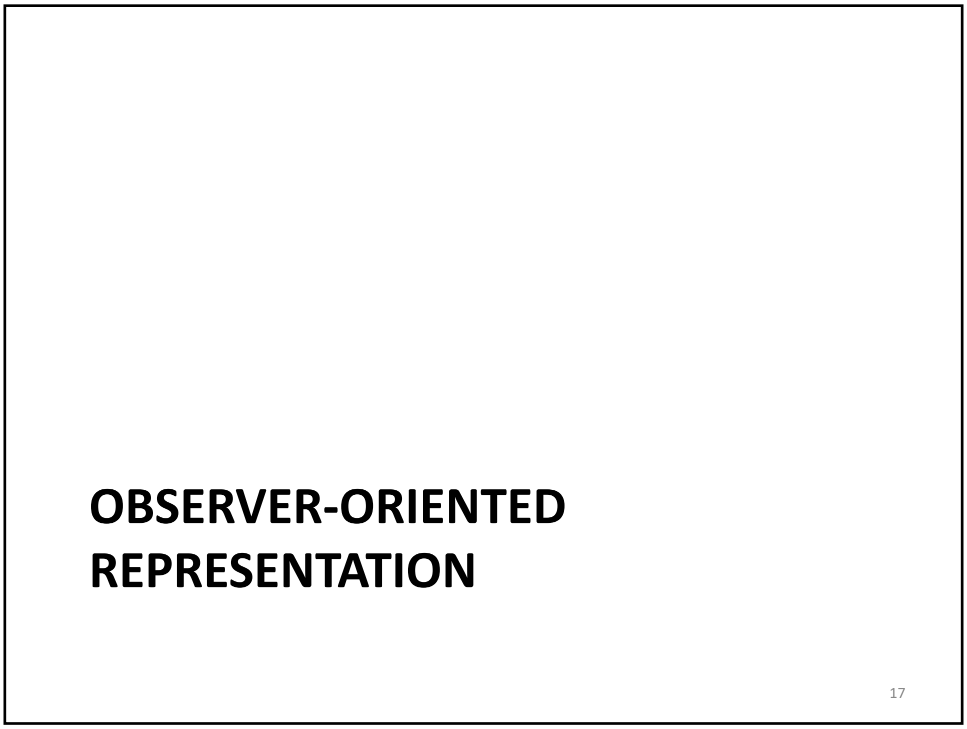
Constructor-Oriented Representation of Lists

```
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 { }
public class Empty implements List {
  public Empty() { }
  public boolean isEmpty() { return true; }
  public int head() throws EmptyListExn { throw new EmptyListExn(); }
  public int tail() throws EmptyListExn { 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));
  }
}
```

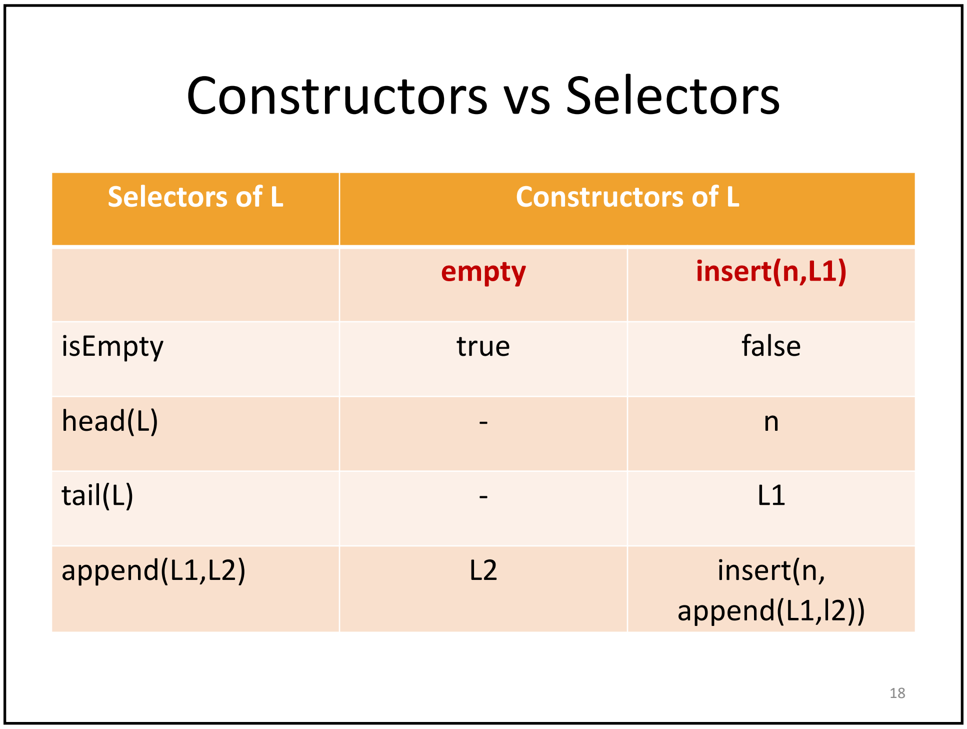


Constructor-Oriented Representation of Lists

```
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 { }
public class Empty implements List {
  public Empty() { }
  public boolean isEmpty() { return true; }
  public int head() throws EmptyListExn { throw new EmptyListExn(); }
  public int tail() throws EmptyListExn { 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));
  }
}
```

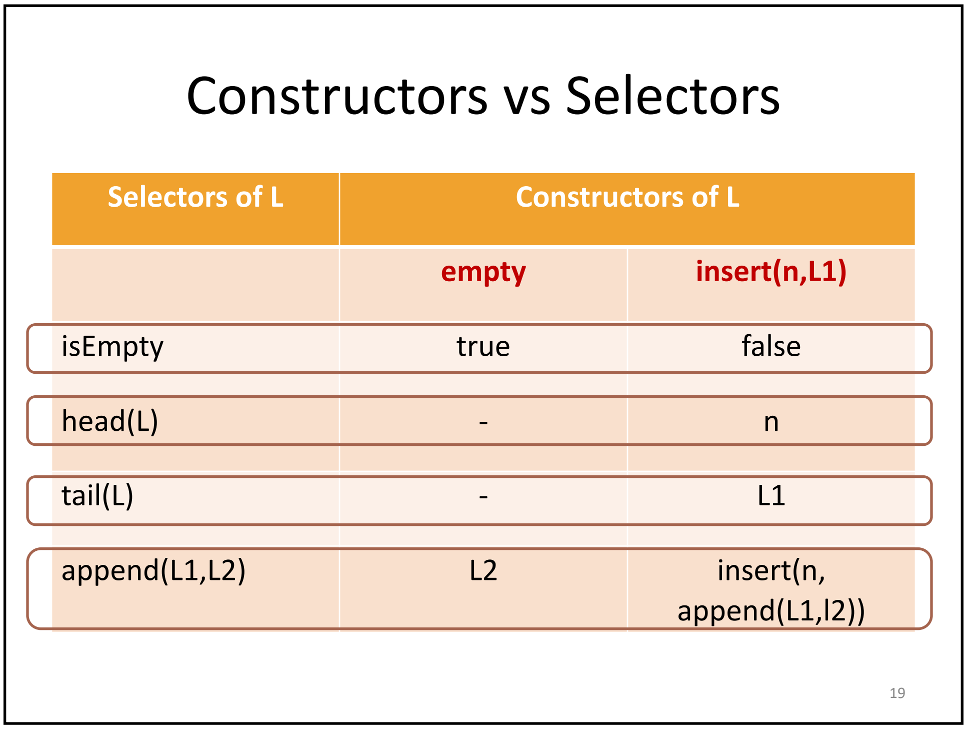


OBSERVER-ORIENTED REPRESENTATION



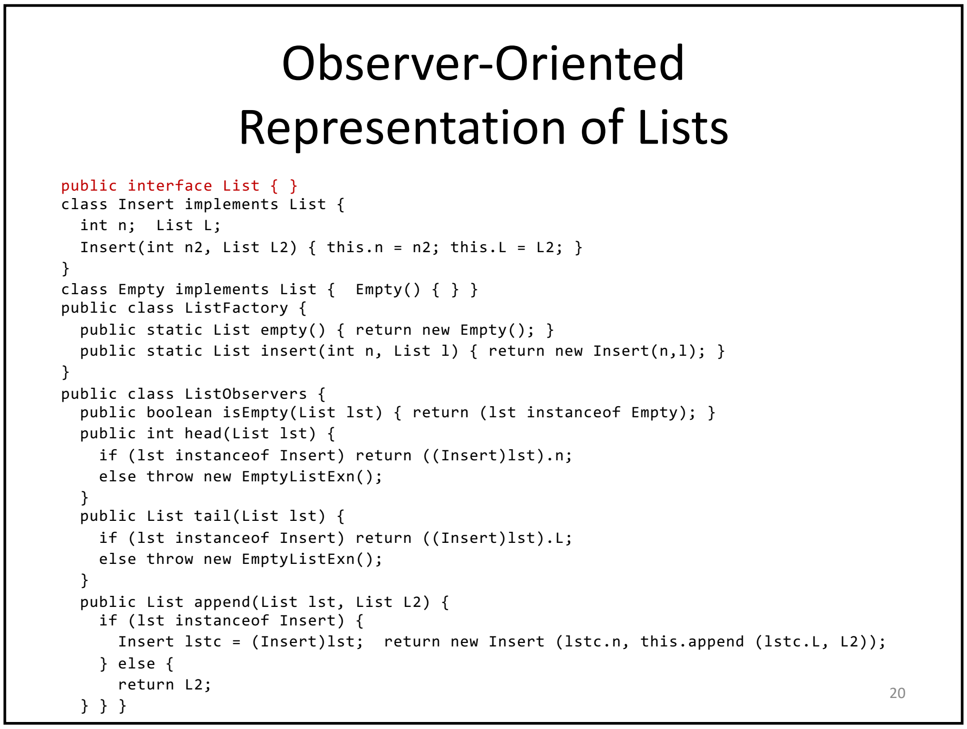
Constructors vs Selectors

Selectors of L	Constructors of L	
	empty	**insert(n,L1)**
isEmpty	true	false
head(L)	-	n
tail(L)	-	L1
append(L1,L2)	L2	insert(n, append(L1,l2))



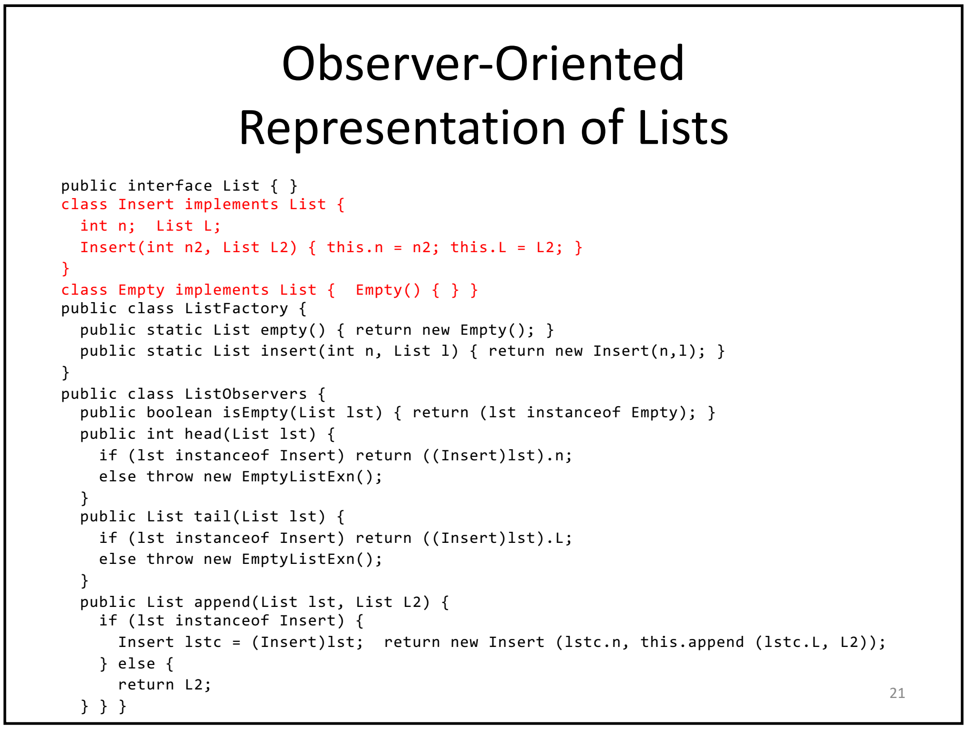
Constructors vs Selectors

Selectors of L	Constructors of L	
	empty	**insert(n,L1)**
isEmpty	true	false
head(L)	-	n
tail(L)	-	L1
append(L1,L2)	L2	insert(n, append(L1,l2))



Observer-Oriented Representation of Lists

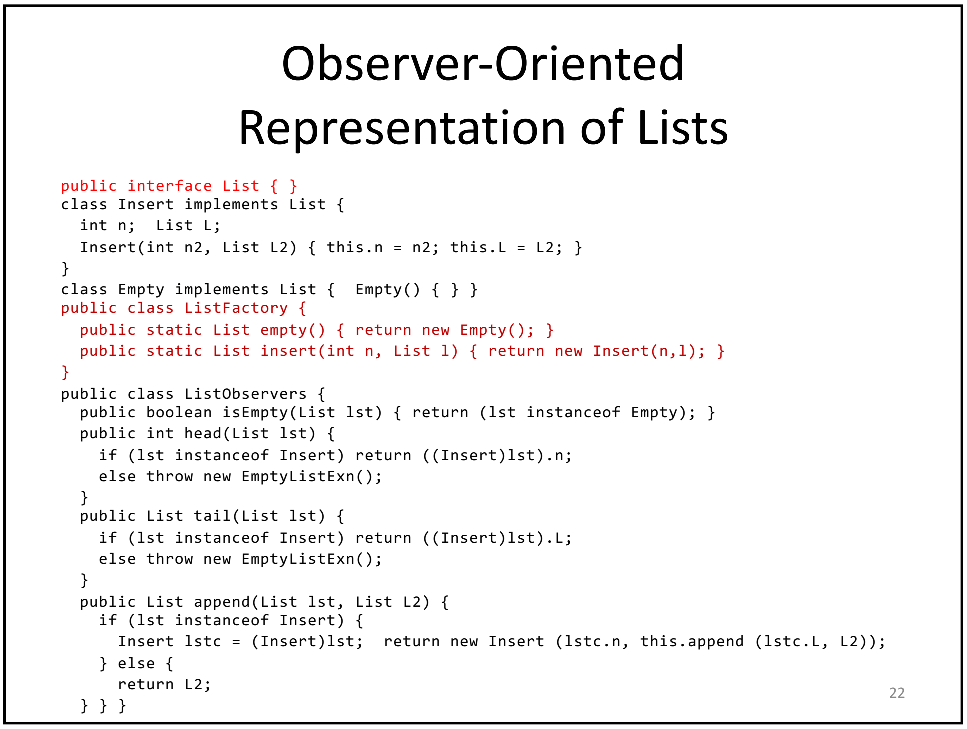
```
public interface List { }
class Insert implements List {
  int n;  List L;
  Insert(int n2, List L2) { this.n = n2; this.L = L2; }
}
class Empty implements List {  Empty() { } }
public class ListFactory {
  public static List empty() { return new Empty(); }
  public static List insert(int n, List l) { return new Insert(n,l); }
}
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, this.append (lstc.L, L2));
    } else {
      return L2;
  } } }
```



Observer-Oriented Representation of Lists

```
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); }
}
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, this.append (lstc.L, L2));
    } else {
      return L2;
  } } }
```

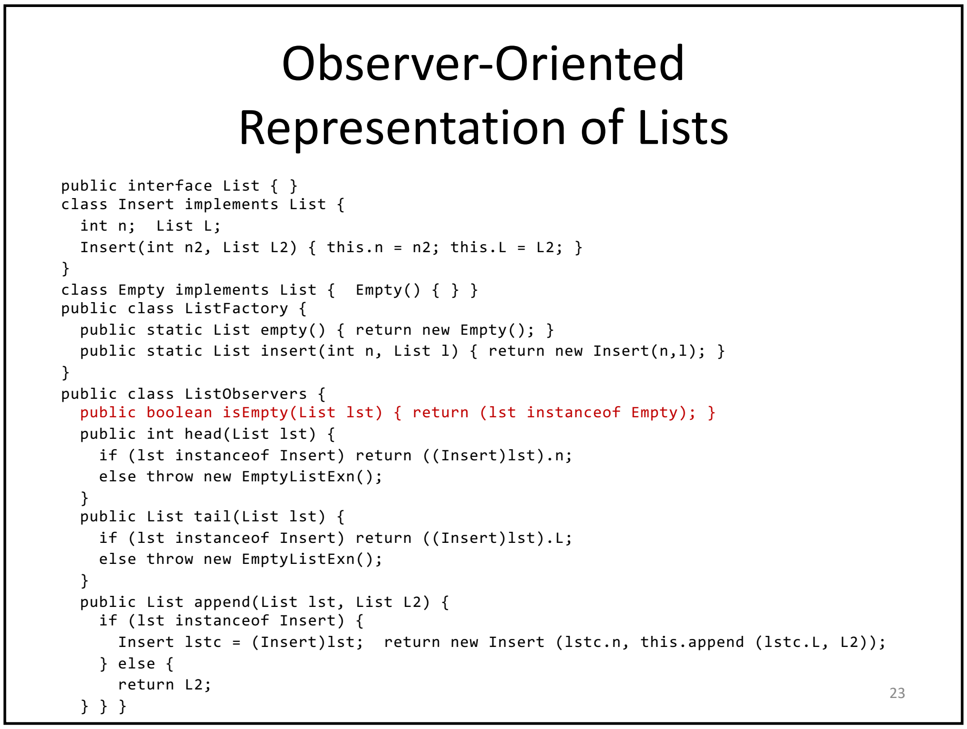
21



Observer-Oriented Representation of Lists

```
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); }
}
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, this.append (lstc.L, L2));
    } else {
      return L2;
  } } }
```

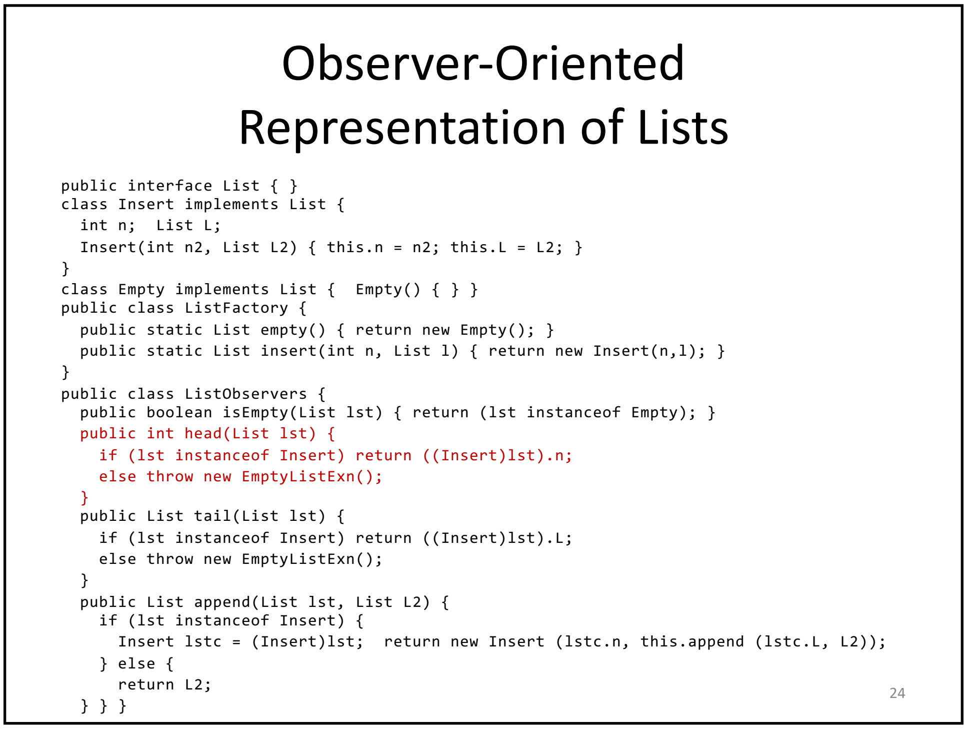
22



Observer-Oriented Representation of Lists

```
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); }
}
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, this.append (lstc.L, L2));
    } else {
      return L2;
  } } }
```

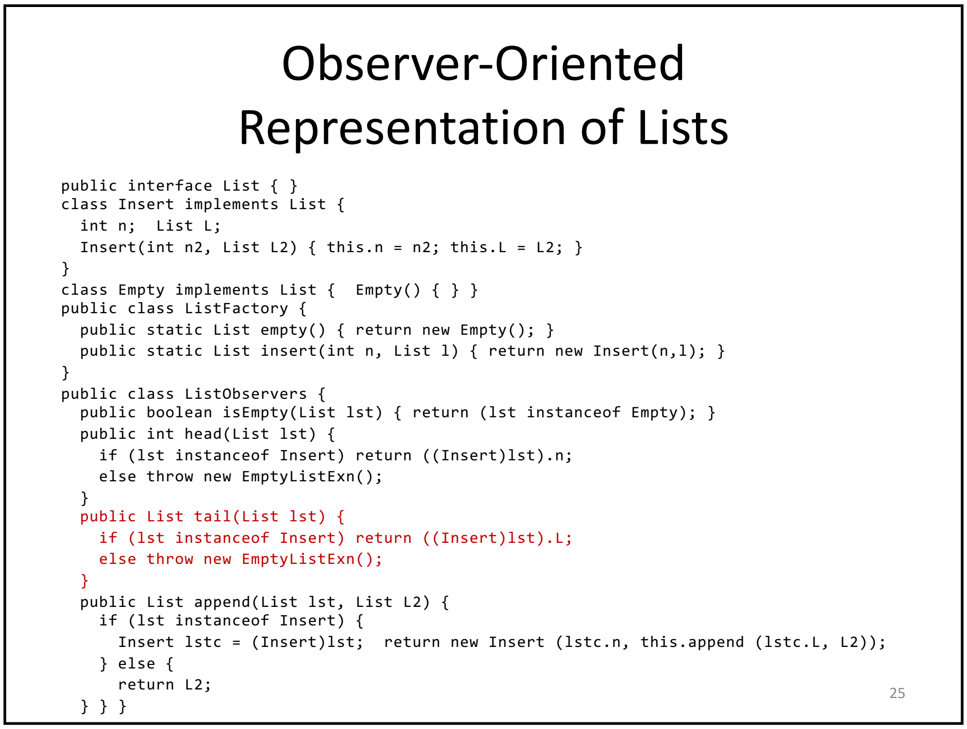
23



Observer-Oriented Representation of Lists

```
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); }
}
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, this.append (lstc.L, L2));
    } else {
      return L2;
  } } }
```

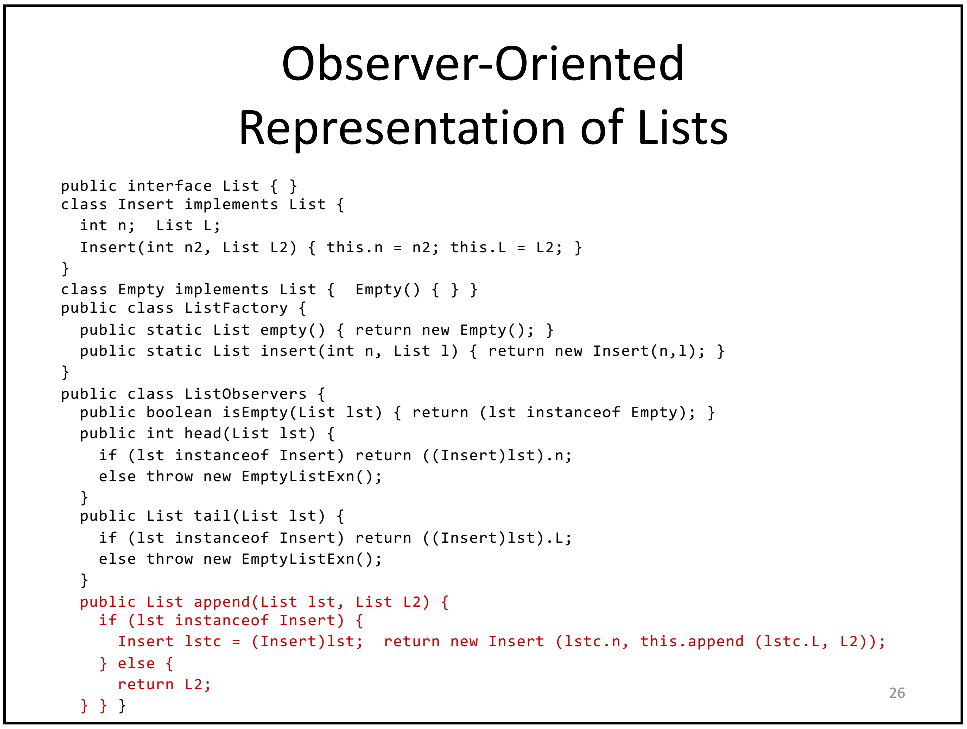
24



Observer-Oriented Representation of Lists

```
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); }
}
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, this.append (lstc.L, L2));
    } else {
      return L2;
  } } }
```

25



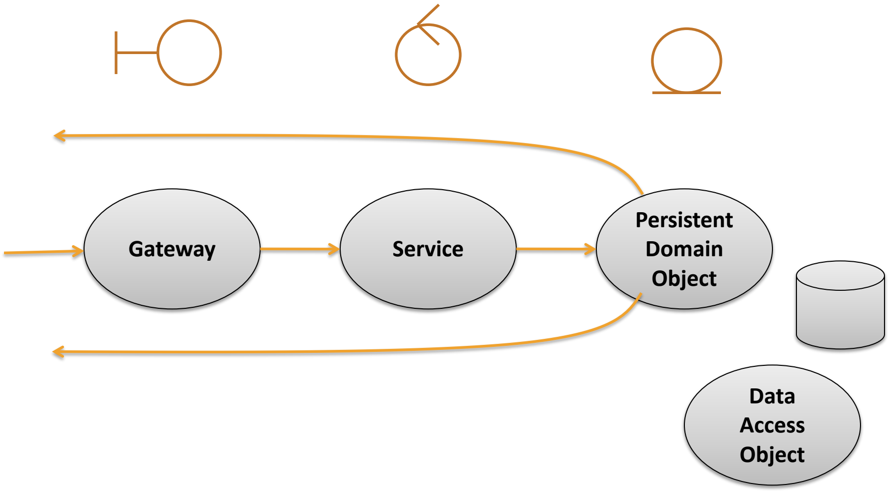
Observer-Oriented Representation of Lists

```
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); }
}
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, this.append (lstc.L, L2));
    } else {
      return L2;
  } } }
```

26

SERVICE-ORIENTED ARCHITECTURE

Domain-Driven



Gateway

Service

Persistent Domain Object

Data Access Object

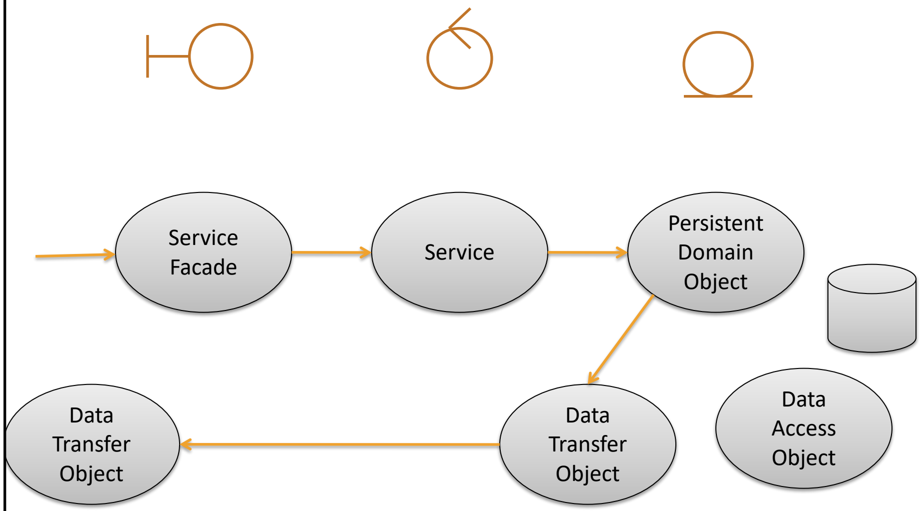
Patterns

- Data Access Object (DAO)
  - Encapsulates and abstracts logic for data access and storage
- Persistence Domain Object (PDO)
  - Used to persist a domain entity object in the database
- **Data Transfer Object (DTO)**
  - Container for entity state to be transferred
  - Not the same as a value object

Service Oriented Architecture

- Move domain logic out of the entity objects
  - Out of the PDOs
- Entity object becomes data transfer object (DTO)
- Logic is enshrined (as use case logic) in **services**
- **Service Façade Pattern:** Collection of procedures encapsulates resources and domain logic, abstracts domain details

Service-Oriented



DDA vs SOA

- DDA
  - Key pattern: Gateway
  - Expose domain objects
  - Data in PDOs
  - Domain logic in PDOs
- SOA
  - Key pattern: Service Façade
  - Encapsulate & abstract domain objects
  - Data in DTOs
  - Domain logic in services