The benefits of indirection

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 6

Abstract data types

- Separate public interface from private implementation

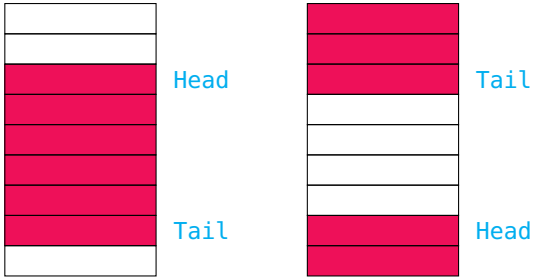
Abstract data types

- Separate public interface from private implementation
- For instance, a (generic) queue

```
public class Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

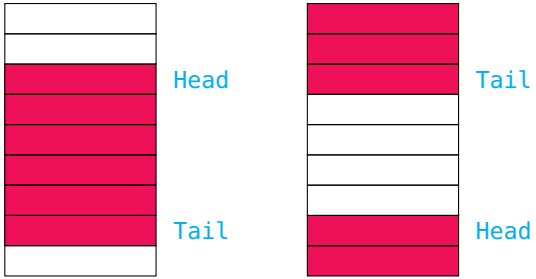
Abstract data types

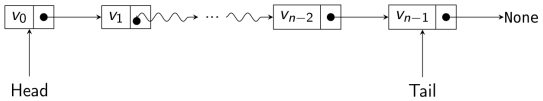
- Separate public interface from private implementation
- For instance, a (generic) queue
- Concrete implementation could be a circular array



Abstract data types

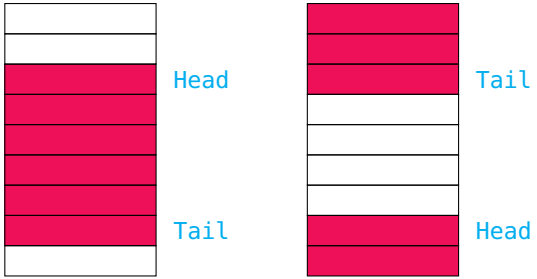
- Separate public interface from private implementation
- For instance, a (generic) queue
- Concrete implementation could be a circular array
- Or a linked list

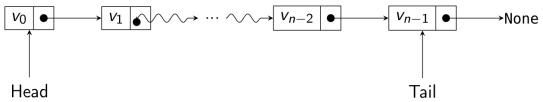




Abstract data types

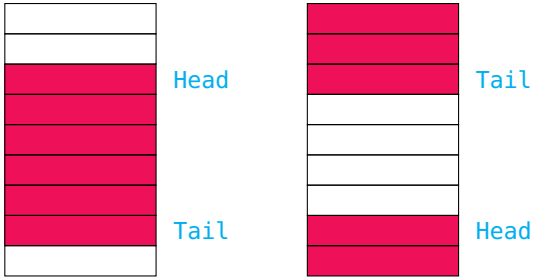
- Separate public interface from private implementation
- For instance, a (generic) queue
- Concrete implementation could be a circular array
- Or a linked list
- Implementer of class `Queue` can choose either one





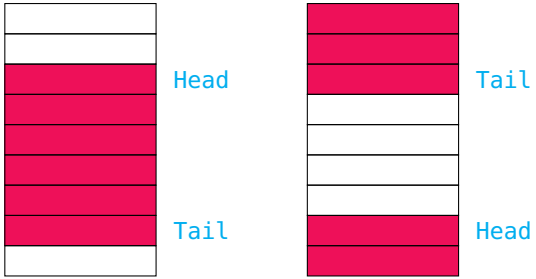
Abstract data types

- Separate public interface from private implementation
- For instance, a (generic) queue
- Concrete implementation could be a circular array
- Or a linked list
- Implementer of class `Queue` can choose either one
- Public interface is unchanged



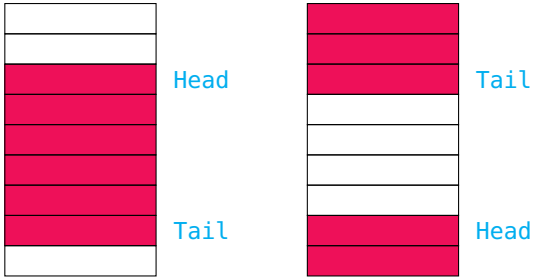
Abstract data types . . .

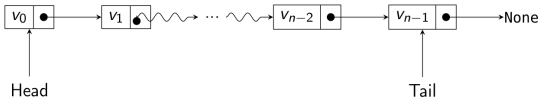
- Is the user indifferent to choice of implementation?



Abstract data types . . .

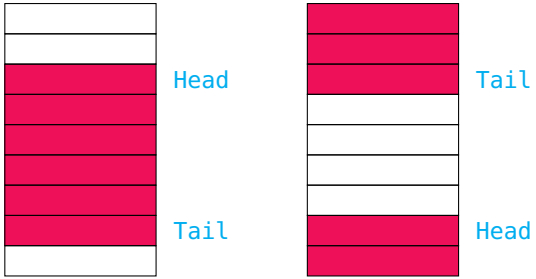
- Is the user indifferent to choice of implementation?
- Interface does not capture other aspects

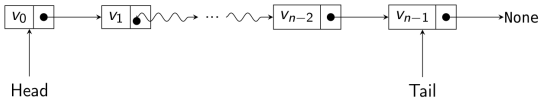




Abstract data types . . .

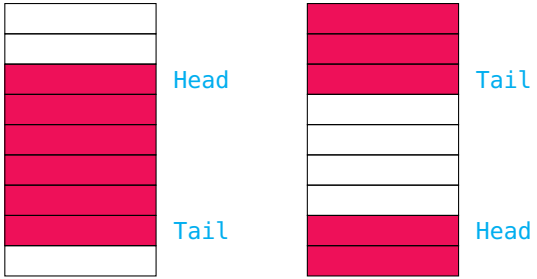
- Is the user indifferent to choice of implementation?
- Interface does not capture other aspects
- Efficiency
  - Circular array is better — one time storage allocation

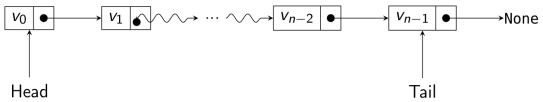




Abstract data types . . .

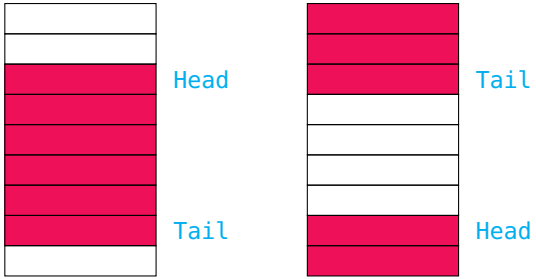
- Is the user indifferent to choice of implementation?
- Interface does not capture other aspects
- Efficiency
  - Circular array is better — one time storage allocation
- Flexibility
  - Linked list is better — circular array has bounded size





Abstract data types . . .

- Is the user indifferent to choice of implementation?
- Interface does not capture other aspects
- Efficiency
  - Circular array is better — one time storage allocation
- Flexibility
  - Linked list is better — circular array has bounded size
- Offer user a choice of implementation?



Multiple impementations

- Create two separate implementations

```
public class CircularArrayQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

```
public class LinkedListQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Multiple impementations

- Create two separate implementations
- User chooses

```
CircularArrayQueue<Date> dateq;
LinkedListQueue<String> stringq;

dateq =
   new CircularArrayQueue<Date>();
stringq =
   new LinkedListQueue<String>();
}
```

```
public class CircularArrayQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

```
public class LinkedListQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Multiple impementations

- Create two separate implementations
- User chooses

```
CircularArrayQueue<Date> dateq;
LinkedListQueue<String> stringq;

dateq =
   new CircularArrayQueue<Date>();
stringq =
   new LinkedListQueue<String>();
}
```

- What if we later realize we need a flexible size `dateq`?

```
public class CircularArrayQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

```
public class LinkedListQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Multiple impementations

- Create two separate implementations
- User chooses

```
CircularArrayQueue<Date> dateq;
LinkedListQueue<String> stringq;

dateq =
   new CircularArrayQueue<Date>();
stringq =
   new LinkedListQueue<String>();
}
```

- What if we later realize we need a flexible size `dateq`?
- Change declaration for `dateq`

```
public class CircularArrayQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

```
public class LinkedListQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Multiple impementations

- Create two separate implementations
- User chooses

```
CircularArrayQueue<Date> dateq;
LinkedListQueue<String> stringq;

dateq =
   new CircularArrayQueue<Date>();
stringq =
   new LinkedListQueue<String>();
}
```

- What if we later realize we need a flexible size `dateq`?
- Change declaration for `dateq`
- And also every function header, auxiliary variable, . . . associated with it

```
public class CircularArrayQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

```
public class LinkedListQueue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Adding indirection

- Instead, create a `Queue` interface

```
public interface Queue<E> {
  abstract void add (E element);
  abstract E remove();
  abstract int size();
}
```

Adding indirection

- Instead, create a `Queue` interface
- Concrete implementations implement the interface

```
public interface Queue<E> {
  abstract void add (E element);
  abstract E remove();
  abstract int size();
}
public class CircularArrayQueue<E>
    implements Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
public class LinkedListQueue<E>
    implements Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Adding indirection

- Instead, create a Queue interface
- Concrete implementations implement the interface
- Use the interface to declare variables

```
Queue<Date> dateq;
Queue<String> stringq;

dateq =
   new CircularArrayQueue<Date>();
stringq =
   new LinkedListQueue<String>();
}
```

```
public interface Queue<E> {
  abstract void add (E element);
  abstract E remove();
  abstract int size();
}
public class CircularArrayQueue<E>
    implements Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
public class LinkedListQueue<E>
    implements Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Adding indirection

- Instead, create a `Queue` interface
- Concrete implementations implement the interface
- Use the interface to declare variables

```
Queue<Date> dateq;
Queue<String> stringq;

dateq =
   new CircularArrayQueue<Date>();
stringq =
   new LinkedListQueue<String>();
}
```

- Benefit of indirection — to use a different implementation for `dateq`, only need to update the instantiation

```
public interface Queue<E> {
  abstract void add (E element);
  abstract E remove();
  abstract int size();
}
public class CircularArrayQueue<E>
    implements Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
public class LinkedListQueue<E>
    implements Queue<E> {
  public void add (E element){...};
  public E remove(){...};
  public int size(){...};
  ...
}
```

Summary

- Use interfaces to flexibly choose between multiple concrete implementations

Summary

- Use interfaces to flexibly choose between multiple concrete implementations
- Interfaces add a level of indirection

Summary

- Use interfaces to flexibly choose between multiple concrete implementations
- Interfaces add a level of indirection
- Indirection in real life
  - Organization provides senior staff with an office car

Summary

- Use interfaces to flexibly choose between multiple concrete implementations
- Interfaces add a level of indirection
- Indirection in real life
  - Organization provides senior staff with an office car
  - Concrete: each official has an assigned car — what if it breaks down?

Summary

- Use interfaces to flexibly choose between multiple concrete implementations
- Interfaces add a level of indirection
- Indirection in real life
  - Organization provides senior staff with an office car
  - Concrete: each official has an assigned car — what if it breaks down?
  - Indirection: a pool of office cars, use any that is available

Summary

- Use interfaces to flexibly choose between multiple concrete implementations
- Interfaces add a level of indirection
- Indirection in real life
  - Organization provides senior staff with an office car
  - Concrete: each official has an assigned car — what if it breaks down?
  - Indirection: a pool of office cars, use any that is available
  - Don't want to maintain a pool of cars? Contract with a taxi service

Summary

- Use interfaces to flexibly choose between multiple concrete implementations
- Interfaces add a level of indirection
- Indirection in real life
  - Organization provides senior staff with an office car
  - Concrete: each official has an assigned car — what if it breaks down?
  - Indirection: a pool of office cars, use any that is available
  - Don't want to maintain a pool of cars? Contract with a taxi service
  - Don't want to negotiate tenders? Reimburse taxi bills

Summary

- Use interfaces to flexibly choose between multiple concrete implementations
- Interfaces add a level of indirection
- Indirection in real life
  - Organization provides senior staff with an office car
  - Concrete: each official has an assigned car — what if it breaks down?
  - Indirection: a pool of office cars, use any that is available
  - Don't want to maintain a pool of cars? Contract with a taxi service
  - Don't want to negotiate tenders? Reimburse taxi bills

"Fundamental theorem of software engineering"

All problems in computer science can be solved by another level of indirection.

Butler Lampson, Turing Award 1992