Programming Language Concepts: Lecture 6

Madhavan Mukund

Chennai Mathematical Institute

madhavan@cmi.ac.in

http://www.cmi.ac.in/~madhavan/courses/pl2009

PLC 2009, Lecture 6, 04 February 2009

Private classes

Lists in Java

```
public class Node {
  public Object data;
  public Node next;
  ...
}
```

Private classes

Lists in Java

```
public class Node {
  public Object data;
  public Node next;
  ...
}
```

```
public class LinkedList{
  private int size;
  private Node first;

  public Object head(){
    Object returnval = null;
    if (first != null){
      returnval = first.data;
      first = first.next;
    }
    return returnval;
  }
}

 public void insert(Object newdata){
   ...
 }
```

Private classes

Lists in Java

```
public class Node {
  public Object data;
  public Node next;
  ...
}
```

Why should `Node` be exposed as a public class?

```
public class LinkedList{
  private int size;
  private Node first;

  public Object head(){
    Object returnval = null;
    if (first != null){
      returnval = first.data;
      first = first.next;
    }
    return returnval;
  }
}

 public void insert(Object newdata){
   ...
 }
```

Private classes . . .

Instead, make `Node` a `private` class defined inside `LinkedList`

```
public class LinkedList{
  private int size;
  private Node first;

  public Object head(){ ... }

  public void insert(Object newdata){
    ...
  }

  private class Node {
    public Object data;
    public Node next;
    ...
  }
}
```

Interfaces and capabilities

Implementing a call-back facility

Interfaces and capabilities

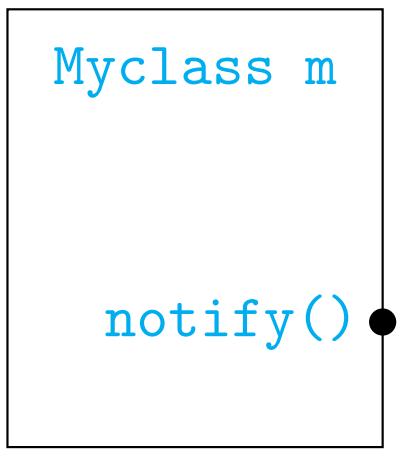
Implementing a call-back facility

- `Myclass m` creates a `Timer t` and starts it in parallel
  - `Myclass m` continues to run
  - Will see later how to invoke parallel execution in Java!

Interfaces and capabilities

Implementing a call-back facility

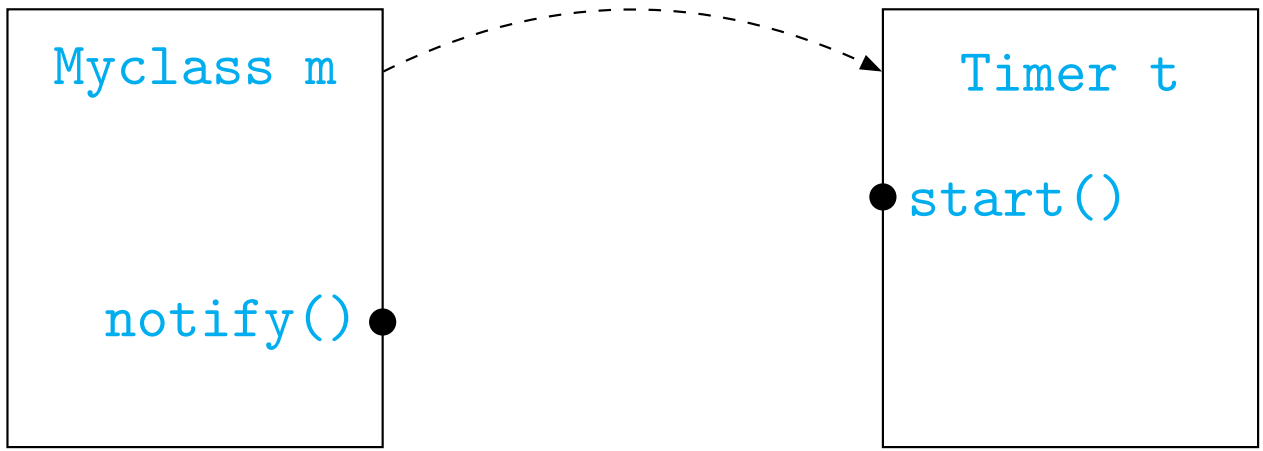
- `Myclass m` creates a `Timer t` and starts it in parallel
  - `Myclass m` continues to run
  - Will see later how to invoke parallel execution in Java!



Interfaces and capabilities

Implementing a call-back facility

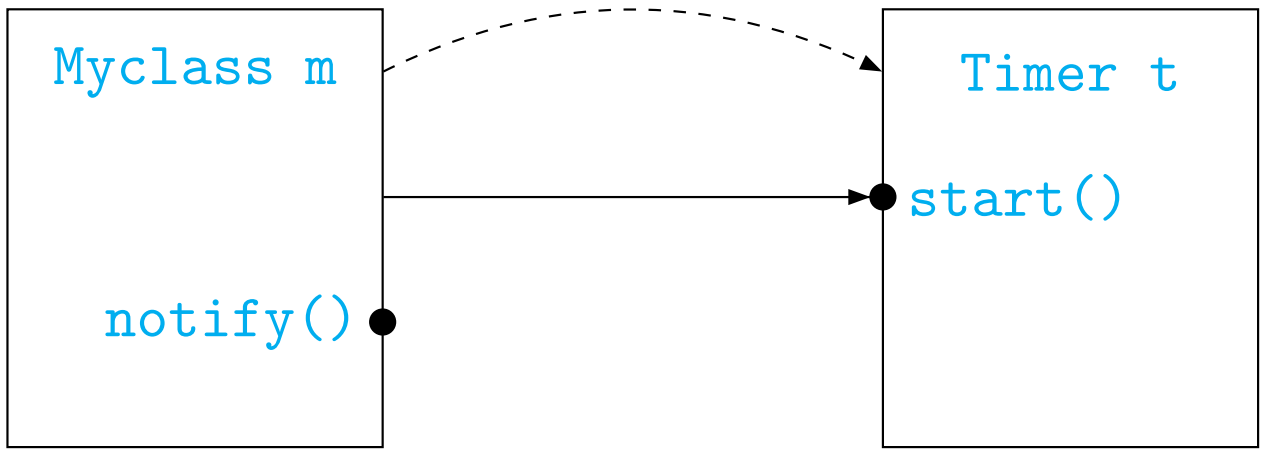
- `Myclass m` creates a `Timer t` and starts it in parallel
  - `Myclass m` continues to run
  - Will see later how to invoke parallel execution in Java!



Interfaces and capabilities

Implementing a call-back facility

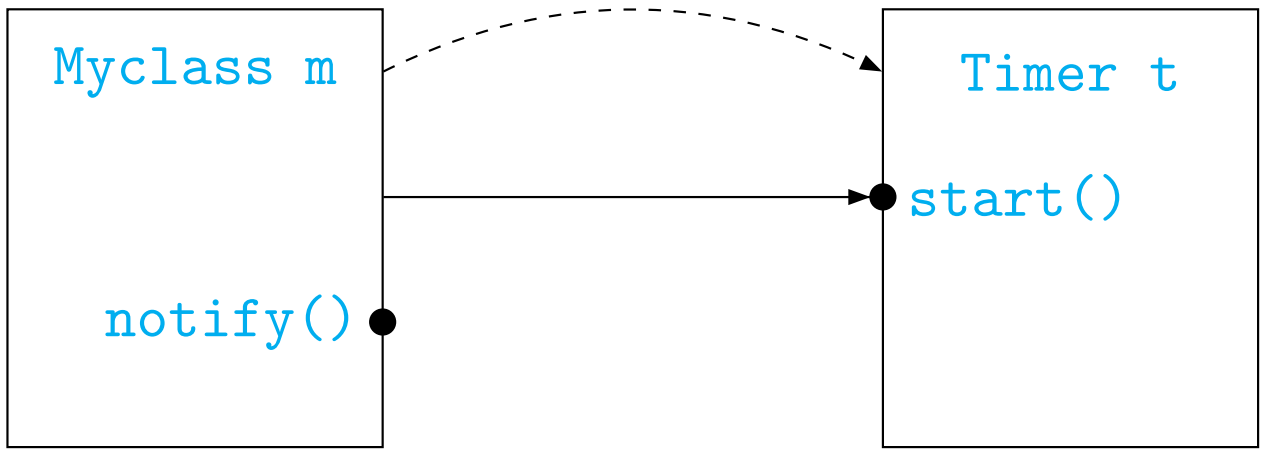
- `Myclass m` creates a `Timer t` and starts it in parallel
  - `Myclass m` continues to run
  - Will see later how to invoke parallel execution in Java!



Interfaces and capabilities

Implementing a call-back facility

- `Myclass m` creates a `Timer t` and starts it in parallel
  - `Myclass m` continues to run
  - Will see later how to invoke parallel execution in Java!

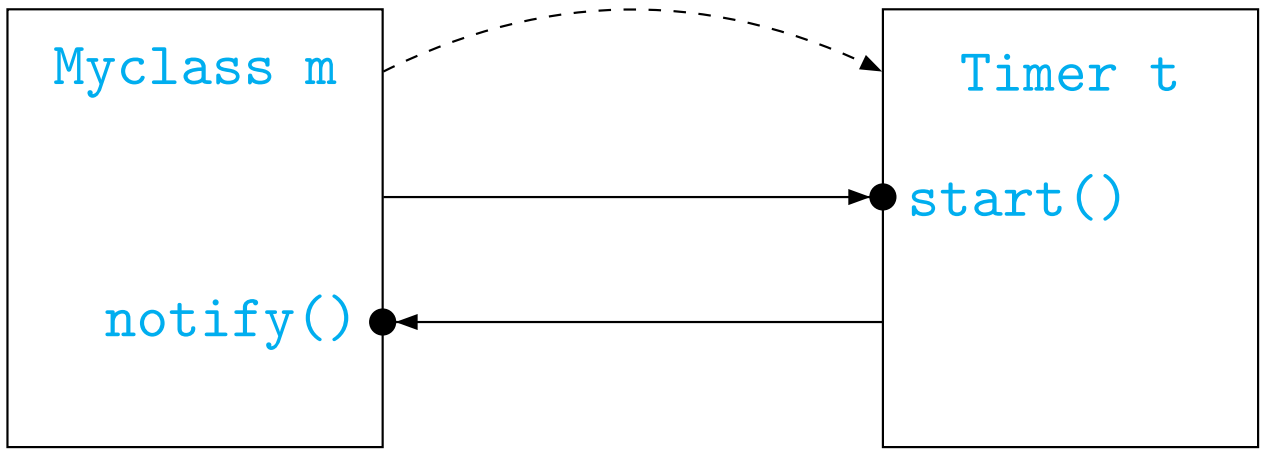


- `Timer t` notifies `Myclass m` when the time limit expires
  - Assume `Myclass m` has a function `notify()`

Interfaces and capabilities

Implementing a call-back facility

- `Myclass m` creates a `Timer t` and starts it in parallel
  - `Myclass m` continues to run
  - Will see later how to invoke parallel execution in Java!



- `Timer t` notifies `Myclass m` when the time limit expires
  - Assume `Myclass m` has a function `notify()`
- `Timer t` should know whom to notify
  - `Myclass m` passes its identity when it creates `Timer t`

Callback . . .

```
class Myclass{
  ...
  f(){
    ..
    Timer t = new Timer(this);
     // this object created t
    ...
    t.start(); // Start t
    ...
  }
  ..
  public void notify(){...}
}
```

Callback . . .

```
class Myclass{
  ...
  f(){
    ..
    Timer t = new Timer(this);
     // this object created t
    ...
    t.start(); // Start t
    ...
  }
  ..
  public void notify(){...}
}
```

```
class Timer implements Runnable{
  // Timer can be invoked in parallel
  private Myclass owner;

  public Timer(Myclass o){
    owner = o;  // My creator
  }
  ...
  public void start(){
    ...
    o.notify(); //  I'm done
  }
  ...
}
```

Callback . . .

```
class Myclass{
  ...
  f(){
    ..
    Timer t = new Timer(this);
     // this object created t
    ...
    t.start(); // Start t
    ...
  }
  ..
  public void notify(){...}
}
```

```
class Timer implements Runnable{
  // Timer can be invoked in parallel
  private Myclass owner;

  public Timer(Myclass o){
    owner = o;  // My creator
  }
  ...
  public void start(){
    ...
    o.notify(); //  I'm done
  }
  ...
}
```

- `Timer` is specific to `Myclass`

Callback . . .

```
class Myclass{
  ...
  f(){
    ..
    Timer t = new Timer(this);
     // this object created t
    ...
    t.start(); // Start t
    ...
  }
  ..
  public void notify(){...}
}
```

```
class Timer implements Runnable{
  // Timer can be invoked in parallel
  private Myclass owner;

  public Timer(Myclass o){
    owner = o;  // My creator
  }
  ...
  public void start(){
    ...
    o.notify(); //  I'm done
  }
  ...
}
```

- `Timer` is specific to `Myclass`
- Can we create a generic `Timer`?

Callback . . .

```
class Myclass{
  ...
  f(){
    ..
    Timer t = new Timer(this);
     // this object created t
    ...
    t.start(); // Start t
    ...
  }
  ..
  public void notify(){...}
}
```

```
class Timer implements Runnable{
  // Timer can be invoked in parallel
  private Object owner;

  public Timer(Object o){
    owner = o;  // My creator
  }
  ...
  public void start(){
    ...
    o.notify(); //  I'm done
  }
  ...
}
```

Callback . . .

```
class Myclass{
  ...
  f(){
    ..
    Timer t = new Timer(this);
     // this object created t
    ...
    t.start(); // Start t
    ...
  }
  ..
  public void notify(){...}
}
```

```
class Timer implements Runnable{
  // Timer can be invoked in parallel
  private Object owner;

  public Timer(Object o){
    owner = o;  // My creator
  }
  ...
  public void start(){
    ...
    o.notify(); //  I'm done
  }
  ...
}
```

- Must cast `owner` from `Object` to `Myclass`!

Callback . . .

- Define an interface for callback

```
interface Timerowner{
  public abstract void notify();
}
```

Callback . . .

- Define an interface for callback

```
interface Timerowner{
  public abstract void notify();
}
```

- Modify `Myclass` to implement `Timerowner`

Callback . . .

- Define an interface for callback

```
interface Timerowner{
  public abstract void notify();
}
```

- Modify `Myclass` to implement `Timerowner`
- Modify `Timer` so that `owner` is compatible with `Timerowner`

Callback . . .

```
class Myclass implements
  Timerowner{
  ...
  f(){
    ..
    Timer t = new Timer(this);
     // this object created t
    ...
    t.start(); // Start t
    ...
  }
  ..
  public void notify(){...}
}
```

```
class Timer implements Runnable{
  // Timer can be invoked in parallel
  private Timerowner owner;

  public Timer(Timerowner o){
    owner = o;  // My creator
  }
  ...
  public void start(){
    ...
    o.notify(); //  I'm done
  }
  ...
}
```

Callback . . .

```
class Myclass implements
  Timerowner{
  ...
  f(){
    ..
    Timer t = new Timer(this);
     // this object created t
    ...
    t.start(); // Start t
    ...
  }
  ..
  public void notify(){...}
}
```

```
class Timer implements Runnable{
  // Timer can be invoked in parallel
  private Timerowner owner;

  public Timer(Timerowner o){
    owner = o;  // My creator
  }
  ...
  public void start(){
    ...
    o.notify(); //  I'm done
  }
  ...
}
```

- Must cast `owner` from `Object` to `Myclass`!

Interactions with state

- Interaction with objects is through methods

Interactions with state

- Interaction with objects is through methods
- Internal variables remember state of object

Interactions with state

- Interaction with objects is through methods
- Internal variables remember state of object
- Sometimes we need to remember state of interaction

Interactions with state

- Interaction with objects is through methods
- Internal variables remember state of object
- Sometimes we need to remember state of interaction
  - Login to a bank account

Interactions with state

- Interaction with objects is through methods
- Internal variables remember state of object
- Sometimes we need to remember state of interaction
  - Login to a bank account
  - Abort after three wrong passwords

Interactions with state

- Interaction with objects is through methods
- Internal variables remember state of object
- Sometimes we need to remember state of interaction
  - Login to a bank account
  - Abort after three wrong passwords
  - This is a property of the interaction, not of the overall bank account!

Interactions with state

Iterators

- Linear list is a generic list of objects

Interactions with state

Iterators

- Linear list is a generic list of objects

```
class Linearlist {
  // Array implementation
  private int limit = 100;
  private Object[] data = new Object[limit];
  private int size;  // Current size of the list

  public Linearlist(){ size = 0; } // Constructor

  public void append(Object o){
    data[size] = o;
    size++;
    ...
  }
  ...
}
```

Iterators . . .

- Linked list implementation

```
class Linearlist {
  private Node head;
  private int size;

  public Linearlist(){ size = 0; }

  public void append(Object o){
    Node m;

    for (m = head; m.next != null; m = m.next){}
    Node n = new Node(o);
    m.next = n;

    size++;
  }
  ...
  private class Node (...}
}
```

Iterators . . .

Want a loop to run through all values in a linear list

Iterators . . .

Want a loop to run through all values in a linear list

- If the list is an array with public access, we write

```
int i;
for (i = 0; i < data.length; i++){
  ... // do something with data[i]
}
```

Iterators . . .

Want a loop to run through all values in a linear list

- If the list is an array with public access, we write

```
int i;
for (i = 0; i < data.length; i++){
  ... // do something with data[i]
}
```

- For a linked list with public access, we could write

```
Node m;
for (m = head; m != null; m = m.next}
  ... // do something with m.data
}
```

Iterators . . .

Want a loop to run through all values in a linear list

- If the list is an array with public access, we write

```
int i;
for (i = 0; i < data.length; i++){
  ... // do something with data[i]
}
```

- For a linked list with public access, we could write

```
Node m;
for (m = head; m != null; m = m.next}
  ... // do something with m.data
}
```

- We don't have public access . . .

Iterators . . .

Want a loop to run through all values in a linear list

- If the list is an array with public access, we write

```
int i;
for (i = 0; i < data.length; i++){
  ... // do something with data[i]
}
```

- For a linked list with public access, we could write

```
Node m;
for (m = head; m != null; m = m.next}
  ... // do something with m.data
}
```

- We don't have public access . . .
- . . . and we don't know which implementation is in use!

Iterators . . .

Need the following abstraction

```
Start at the beginning of the list;
while (there is a next element){
  get the next element;
  do something with it
}
```

Iterators . . .

Need the following abstraction

```
Start at the beginning of the list;
while (there is a next element){
  get the next element;
  do something with it
}
```

Encapsulate this functionality in an interface called `Iterator`

```
public interface Iterator{
  public abstract boolean has_next();
  public abstract Object get_next();
}
```

Iterators . . .

- How do we implement `Iterator` in `Linearlist`?

Iterators . . .

- How do we implement `Iterator` in `Linearlist`?
- Need a "pointer" to remember position of the iterator

Iterators . . .

- How do we implement `Iterator` in `Linearlist`?
- Need a "pointer" to remember position of the iterator
- How do we handle nested loops?

```
for (i = 0; i < data.length; i++){
  for (j = 0; j < data.length; j++){
     ... // do something with data[i] and data[j]
  }
}
```

Iterators . . .

- Solution: Create an `Iterator` object and export it!

```
public class Linearlist{
  ...
  private class Iter implements Iterator{
    private Node position;
    public Iter(){...}   // Constructor
    public boolean has_next(){...}
    public Object get_next(){...}
  }
  ...
 // Export a fresh iterator
  public Iterator get_iterator(){
    Iter it = new Iter();
    return it;
  }
  ...
 }
```

Iterators . . .

- Solution: Create an `Iterator` object and export it!

```
public class Linearlist{
  ...
  private class Iter implements Iterator{
    private Node position;
    public Iter(){...}   // Constructor
    public boolean has_next(){...}
    public Object get_next(){...}
  }
  ...
 // Export a fresh iterator
  public Iterator get_iterator(){
    Iter it = new Iter();
    return it;
  }
  ...
}
```

- Definition of `Iter` depends on linear list implementation

Iterators . . .

Now, we can traverse the list externally as follows:

```
Linearlist l = new Linearlist();
...
Object o;
Iterator i = l.get_iterator()

while (i.has_next()){
  o = i.get_next();
  ...   // do something with o
}
...
```

Iterators . . .

Now, we can traverse the list externally as follows:

```
Linearlist l = new Linearlist();
...
Object o;
Iterator i = l.get_iterator()

while (i.has_next()){
  o = i.get_next();
  ...   // do something with o
}
...
```

For nested loops, acquire multiple iterators!