Higher order functions

Madhavan Mukund

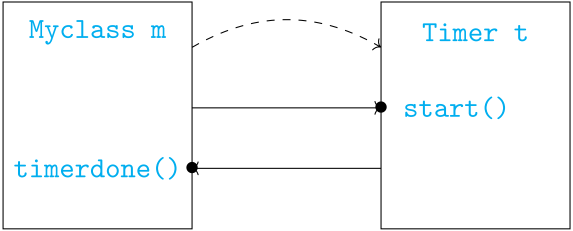
`https://www.cmi.ac.in/~madhavan`

Programming Concepts using Java

Week 8

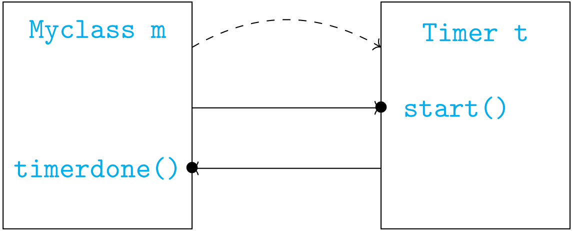
Passing functions

- Recall callbacks
  - Myclass m creates a Timer t
  - t starts running in parallel
  - t notifies m when the time limit expires



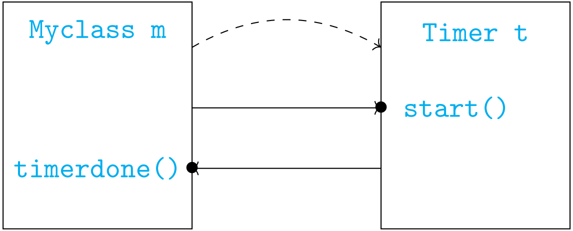
Passing functions

- Recall callbacks
  - `Myclass m` creates a `Timer t`
  - `t` starts running in parallel
  - `t` notifies `m` when the time limit expires
- `m` needs to pass `timerdone()` to `t`



Passing functions

- Recall callbacks
  - Myclass m creates a Timer t
  - t starts running in parallel
  - t notifies m when the time limit expires
- m needs to pass timerdone() to t
- Achieved this through an interface



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

public class Myclass
       extends Timerowner{
  ...
}
```

```
public class Timer implements Runnable{
  private Timerowner owner;
  ...
  public void start(){
    ...
    owner.timerdone();
  }
}
```

Passing functions

- Customize `Arrays.sort`

Passing functions

- Customize `Arrays.sort`
- `Comparator` interface provides signature for comparison function

```
public interface Comparator<T>{
  public abstract int compare(T o1, T o2);
}
```

Passing functions

- Customize `Arrays.sort`
- `Comparator` interface provides signature for comparison function
- Implement `Comparator`

```
public interface Comparator<T>{
  public abstract int compare(T o1, T o2);
}


public class StringCompare
  implements Comparator<String>{

  public int compare(String s1, String s2){
    return s1.length() - s2.length();
  }
}
```

Passing functions

- Customize `Arrays.sort`
- `Comparator` interface provides signature for comparison function
- Implement `Comparator`
- Pass to `Arrays.sort`

```
public interface Comparator<T>{
  public abstract int compare(T o1, T o2);
}


public class StringCompare
  implements Comparator<String>{

  public int compare(String s1, String s2){
    return s1.length() - s2.length();
  }
}


String[] strarr = new ...;
Arrays.sort(strarr,StringCompare);
```

Functional interfaces

- Interfaces that define a single function are called functional interfaces
  - Comparator, Timerowner

```
public interface Comparator<T>{
  public abstract int compare(T o1, T o2);
}


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

Functional interfaces

- Interfaces that define a single function are called functional interfaces
  - `Comparator`, `Timerowner`
- How can we directly pass the required function?

```
public interface Comparator<T>{
  public abstract int compare(T o1, T o2);
}
```

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

Functional interfaces

- Interfaces that define a single function are called functional interfaces
  - `Comparator`, `Timerowner`
- How can we directly pass the required function?
- In Python, function names are similar to variable names
  - Define a function
  - Pass it as an argument to another function
  - `map` is a higher order function

```
public interface Comparator<T>{
  public abstract int compare(T o1, T o2);
}
```

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

```
def square(x):
  return(x*x)

l = list(map(square,range(100)))
```

Lambda expressions

- Lambda expressions denote anonymous functions
  - `(Parameters) -> Body`
  - Return value and type are implicit

```
(String s1, String s2) ->
   s1.length() - s2.length()
```

Lambda expressions

- Lambda expressions denote anonymous functions
  - `(Parameters) -> Body`
  - Return value and type are implicit
- From λ-calculus (Alonzo Church)
  - Foundational model for computing, parallel to Alan Turing's machines
  - Basis for functional programming: Lisp, Scheme, ML, Haskell, . . .

```
(String s1, String s2) ->
   s1.length() - s2.length()
```

Lambda expressions

- Lambda expressions denote anonymous functions
  - `(Parameters) -> Body`
  - Return value and type are implicit
- From λ-calculus (Alonzo Church)
  - Foundational model for computing, parallel to Alan Turing's machines
  - Basis for functional programming: Lisp, Scheme, ML, Haskell, …
- Substitute wherever a functional interface is specified

```
(String s1, String s2) ->
   s1.length() - s2.length()
```

```
String[] strarr = new ...;
Arrays.sort(strarr,
            (String s1, String s2) ->
               s1.length() - s2.length());
```

Lambda expressions

- Lambda expressions denote anonymous functions
  - `(Parameters) -> Body`
  - Return value and type are implicit
- From λ-calculus (Alonzo Church)
  - Foundational model for computing, parallel to Alan Turing's machines
  - Basis for functional programming: Lisp, Scheme, ML, Haskell, ...
- Substitute wherever a functional interface is specified
- Limited type inference is also possible
  - Java infers `s1` and `s2` are `String`

```
(String s1, String s2) ->
   s1.length() - s2.length()
```

```
String[] strarr = new ...;
Arrays.sort(strarr,
            (String s1, String s2) ->
               s1.length() - s2.length());
```

```
String[] strarr = new ...;
Arrays.sort(strarr,
            (s1, s2) ->
               s1.length() - s2.length());
```

Lambda expressions

- More complicated function body can be defined as a block

```
(String s1, String s2) -> {
     if s1.length() < s2.length()
       return -1;
     else  if s1.length() > s2.length()
       return 1;
     else
       return 0;
     }
```

Lambda expressions

- More complicated function body can be defined as a block
- Note that the function is anonymous only for the caller

```
(String s1, String s2) -> {
     if s1.length() < s2.length()
       return -1;
     else  if s1.length() > s2.length()
       return 1;
     else
       return 0;
     }
```

Lambda expressions

- More complicated function body can be defined as a block
- Note that the function is anonymous only for the caller
- The function that receives the lambda expression still needs to use a functional interface for the parameter type

```
public static <T> void
   Arrays.sort(T[] a, Comparator<T> c)}
```

  - Inside `Arrays.sort()`, refer to the function by the name `compare()` defined in the `Comparator` interface

```
(String s1, String s2) -> {
     if s1.length() < s2.length()
       return -1;
     else  if s1.length() > s2.length()
       return 1;
     else
       return 0;
     }
```

Passing named functions

- If the lambda expression consists of a single function call, we can pass that function by name
  - Method reference

Passing named functions

- If the lambda expression consists of a single function call, we can pass that function by name
  - Method reference
- We saw an example with adding entries to a Map object
  - Here sum is a static method in Integer

```
Map<String, Integer> scores = ...;
scores.merge(bat,newscore,Integer::sum);
```

Passing named functions

- If the lambda expression consists of a single function call, we can pass that function by name
  - Method reference
- We saw an example with adding entries to a `Map` object
  - Here `sum` is a static method in `Integer`

```
Map<String, Integer> scores = ...;
scores.merge(bat,newscore,Integer::sum);
```

- Here is the corresponding expression, assuming type inference

```
(i,j) -> Integer::sum(i,j)
```

Passing named functions

- If the lambda expression consists of a single function call, we can pass that function by name
  - Method reference
- We saw an example with adding entries to a `Map` object
  - Here `sum` is a static method in `Integer`

```
Map<String, Integer> scores = ...;
scores.merge(bat,newscore,Integer::sum);
```

- Here is the corresponding expression, assuming type inference

```
(i,j) -> Integer::sum(i,j)
```

- Expression should call a function, and nothing else — this expression cannot be replaced by a method reference

```
(i,j) -> Integer::sum(i,j) > 0
```

Method references

- `ClassName::StaticMethod`
  - Method reference is `C::f`
  - Corresponding expression with as many arguments as `f` has

```
(x1,x2,..,xk) -> f(x1,x2,...,xk)
```

Method references

- `ClassName::StaticMethod`
  - Method reference is `C::f`
  - Corresponding expression with as many arguments as `f` has

```
(x1,x2,..,xk) -> f(x1,x2,...,xk)
```

- `ClassName::InstanceMethod`
  - Method reference is `C::f`
  - Called with respect to an object that becomes implicit parameter

```
(o,x1,x2,...,xk) -> o.f(x1,x2,...,xk)
```

Method references

- `ClassName::StaticMethod`
  - Method reference is `C::f`
  - Corresponding expression with as many arguments as `f` has

```
(x1,x2,..,xk) -> f(x1,x2,...,xk)
```

- `ClassName::InstanceMethod`
  - Method reference is `C::f`
  - Called with respect to an object that becomes implicit parameter

```
(o,x1,x2,...,xk) -> o.f(x1,x2,...,xk)
```

- `object::InstanceMethod`
  - Method reference is `o::f`
  - Arguments are passed to `o.f`

```
(x1,x2,..,xk) -> o.f(x1,x2,...,xk)
```

Method references

- `ClassName::StaticMethod`
  - Method reference is `C::f`
  - Corresponding expression with as many arguments as `f` has

```
(x1,x2,..,xk) -> f(x1,x2,...,xk)
```

- `ClassName::InstanceMethod`
  - Method reference is `C::f`
  - Called with respect to an object that becomes implicit parameter

```
(o,x1,x2,...,xk) -> o.f(x1,x2,...,xk)
```

- `object::InstanceMethod`
  - Method reference is `o::f`
  - Arguments are passed to `o.f`

```
(x1,x2,..,xk) -> o.f(x1,x2,...,xk)
```

- Can also pass references to constructors

Summary

- Many languages support higher-order functions
  - Passing a function as an argument to another function
- In object-oriented programming, this is achieved using interfaces
  - Encapsulate the function to be passed as an object
- Java allows functions to be passed directly in place of functional interfaces
  - Interface consists of a single function
- Lambda expressions describe anonymous functions
  - Cannot pass lambda expressions in general
  - Only when the argument is a functional interface
- Can pass a method reference if the lambda expression consists of a single function call