Interfaces

Madhavan Mukund

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

Programming Concepts using Java

Week 4

Interfaces

- An interface is a purely abstract class
  - All methods are abstract
- A class implements an interface
  - Provide concrete code for each abstract function
- Classes can implement multiple interfaces
  - Abstract functions, so no contradictory inheritance
- Interfaces describe relevant aspects of a class
  - Abstract functions describe a specific “slice” of capabilities
  - Another class only needs to know about these capabilities

Exposing limited capabilities

- Generic `quicksort` for any datatype that supports comparisons

Exposing limited capabilities

- Generic `quicksort` for any datatype that supports comparisons
- Express this capability by making the argument type `Comparable[]`
  - Only information that `quicksort` needs about the underlying type
  - All other aspects are irrelevant

```
public class SortFunctions{
  public static void quicksort(Comparable[] a){
    ...
    // Usual code for quicksort, except that
    // to compare a[i] and a[j] we use
    //  a[i].cmp(a[j])
  }
}
```

Exposing limited capabilities

- Generic `quicksort` for any datatype that supports comparisons
- Express this capability by making the argument type `Comparable[]`
  - Only information that `quicksort` needs about the underlying type
  - All other aspects are irrelevant
- Describe the relevant functions supported by `Comparable` objects through an interface

```
public class SortFunctions{
  public static void quicksort(Comparable[] a){
    ...
    // Usual code for quicksort, except that
    // to compare a[i] and a[j] we use
    //  a[i].cmp(a[j])
  }
}
```

```
public interface Comparable{
  public abstract int cmp(Comparable s);
    // return -1 if this < s,
    //         0 if this == 0,
    //        +1 if this > s
}
```

Exposing limited capabilities

- Generic `quicksort` for any datatype that supports comparisons
- Express this capability by making the argument type `Comparable[]`
  - Only information that `quicksort` needs about the underlying type
  - All other aspects are irrelevant
- Describe the relevant functions supported by `Comparable` objects through an interface
- However, we cannot express the intended behaviour of `cmp` explicitly

```
public class SortFunctions{
  public static void quicksort(Comparable[] a){
    ...
    // Usual code for quicksort, except that
    // to compare a[i] and a[j] we use
    //  a[i].cmp(a[j])
  }
}
```

```
public interface Comparable{
  public abstract int cmp(Comparable s);
    // return -1 if this < s,
    //         0 if this == 0,
    //        +1 if this > s
}
```

Adding methods to interfaces

- Java interfaces extended to allow functions to be added

Adding methods to interfaces

- Java interfaces extended to allow functions to be added
- Static functions
  - Cannot access instance variables
  - Invoke directly or using interface name: `Comparable.cmpdoc()`

```
public interface Comparable{
  public static String cmpdoc(){
    String s;
    s = "Return -1 if this < s, ";
    s = s + "0 if this == s, ";
    s = s + "+1 if this >  s.";
    return(s);
}
```

Adding methods to interfaces

- Java interfaces extended to allow functions to be added
- Static functions
  - Cannot access instance variables
  - Invoke directly or using interface name: `Comparable.cmpdoc()`
- Default functions
  - Provide a default implementation for some functions
  - Class can override these
  - Invoke like normal method, using object name: `a[i].cmp(a[j])`

```
public interface Comparable{
  public static String cmpdoc(){
    String s;
    s = "Return -1 if this < s, ";
    s = s + "0 if this == s, ";
    s = s + "+1 if this >  s.";
    return(s);
}
```

```
public interface Comparable{
  public default int cmp(Comparable s) {
    return(0);
  }
}
```

Dealing with conflicts

- Old problem of multiple inheritance returns
  - Conflict between static/default methods

```
public interface Person{
  public default String getName() {
    return("No name");
  }
}

public interface Designation{
  public default String getName() {
    return("No designation");
  }
}

public class Employee
  implements Person, Designation {...}
```

Dealing with conflicts

- Old problem of multiple inheritance returns
  - Conflict between static/default methods
- Subclass must provide a fresh implementation

```
public interface Person{
  public default String getName() {
    return("No name");
  }
}

public interface Designation{
  public default String getName() {
    return("No designation");
  }
}

public class Employee
  implements Person, Designation {
  ...

  public String getName(){
    ...
  }
```

Dealing with conflicts

- Old problem of multiple inheritance returns
  - Conflict between static/default methods
- Subclass must provide a fresh implementation
- Conflict could be between a class and an interface
  - `Employee` inherits from class `Person` and implements `Designation`
  - Method inherited from the class "wins"
  - Motivated by reverse compatibility

```
public class Person{
  public String getName() {
    return("No name");
  }
}

public interface Designation{
  public default String getName() {
    return("No designation");
  }
}

public class Employee
  extends Person implements Designation {
  ...
}
```

Summary

- Interfaces express abstract capabilities
  - Capabilities are expressed in terms of methods that must be present
  - Cannot specify the intended behaviour of these functions
- Java later allowed concrete functions to be added to interfaces
  - Static functions — cannot access instance variables
  - Default functions — may be overridden
- Reintroduces conflicts in multiple inheritance
  - Subclass must resolve the conflict by providing a fresh implementation
  - Special "class wins" rule for conflict between superclass and interface
- Pitfalls of extending a language and maintaining compatibility