Java: abstract classes, interfaces

Madhavan Mukund, S P Suresh

Programming Language Concepts

Lecture 5, 19 January 2023

Grouping together classes

- Sometimes we collect together classes under a common heading

Grouping together classes

- Sometimes we collect together classes under a common heading
- Classes `Circle`, `Square` and `Rectangle` are all shapes

Grouping together classes

- Sometimes we collect together classes under a common heading
- Classes `Circle`, `Square` and `Rectangle` are all shapes
- Create a class `Shape` so that `Circle`, `Square` and `Rectangle` extend `Shape`

Grouping together classes

- Sometimes we collect together classes under a common heading
- Classes `Circle`, `Square` and `Rectangle` are all shapes
- Create a class `Shape` so that `Circle`, `Square` and `Rectangle` extend `Shape`
- We want to force every `Shape` to define a function

  ```
  public double perimeter()
  ```

Grouping together classes

- Sometimes we collect together classes under a common heading
- Classes `Circle`, `Square` and `Rectangle` are all shapes
- Create a class `Shape` so that `Circle`, `Square` and `Rectangle` extend `Shape`
- We want to force every `Shape` to define a function

  ```
  public double perimeter()
  ```

- Could define a function in `Shape` that returns an absurd value

  ```
  public double perimeter() { return(-1.0); }
  ```

Grouping together classes

- Sometimes we collect together classes under a common heading
- Classes `Circle`, `Square` and `Rectangle` are all shapes
- Create a class `Shape` so that `Circle`, `Square` and `Rectangle` extend `Shape`
- We want to force every `Shape` to define a function

  ```
  public double perimeter()
  ```
- Could define a function in `Shape` that returns an absurd value

  ```
  public double perimeter() { return(-1.0); }
  ```
- Rely on the subclass to redefine this function

Grouping together classes

- Sometimes we collect together classes under a common heading
- Classes `Circle`, `Square` and `Rectangle` are all shapes
- Create a class `Shape` so that `Circle`, `Square` and `Rectangle` extend `Shape`
- We want to force every `Shape` to define a function

```
public double perimeter()
```

- Could define a function in `Shape` that returns an absurd value

```
public double perimeter() { return(-1.0); }
```

- Rely on the subclass to redefine this function
- What if this doesn't happen?
  - Should not depend on programmer discipline

Abstract classes

- A better solution
  - Provide an abstract definition in `Shape`

```
public abstract double perimeter();
```

Abstract classes

- A better solution
  - Provide an abstract definition in `Shape`

    ```
    public abstract double perimeter();
    ```

- Forces subclasses to provide a concrete implementation

Abstract classes

- A better solution
  - Provide an abstract definition in `Shape`

    ```
    public abstract double perimeter();
    ```

- Forces subclasses to provide a concrete implementation
- Cannot create objects from a class that has abstract functions

Abstract classes

- A better solution
  - Provide an abstract definition in `Shape`

    ```
    public abstract double perimeter();
    ```

- Forces subclasses to provide a concrete implementation
- Cannot create objects from a class that has abstract functions
- `Shape` must itself be declared to be `abstract`

```
public abstract class Shape{
  ...
  public abstract double perimeter();
  ...
}
```

Abstract classes ...

- Can still declare variables whose type is an abstract class

Abstract classes ...

- Can still declare variables whose type is an abstract class

```
Shape shapearr[] = new Shape[3];
int sizearr[] = new int[3];

shapearr[0] = new Circle(...);
shapearr[1] = new Square(...);
shapearr[2] = new Rectangle(...);

for (i = 0; i < 3; i++){
  sizearr[i] = shapearr[i].perimeter();
     // each shapearr[i] calls the appropriate method
  ...
}
```

Generic functions

- Use abstract classes to specify generic properties

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

Generic functions

- Use abstract classes to specify generic properties

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

- Now we can sort any array of objects that extend `Comparable`

```
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])
  }
}
```

Generic functions ...

```
public class SortFunctions{
  public static void quicksort(Comparable[] a){
    ...
  }
}
```

Generic functions ...

```
public class SortFunctions{
  public static void quicksort(Comparable[] a){
    ...
  }
}
```

- To use this definition of `quicksort`, we write

```
public class Myclass extends Comparable{
  private double size;   // quantity used for comparison

  public int cmp(Comparable s){
    if (s instanceof Myclass){
      // compare this.size and ((Myclass) s).size
      // Note the cast to access s.size
    }
  }
}
```

Mutiple inheritance

- Can we sort `Circle` objects using the generic functions in `SortFunctions`?
  - `Circle` already extends `Shape`
  - Java does not allow `Circle` to also extend `Comparable`!

Mutiple inheritance

- Can we sort `Circle` objects using the generic functions in `SortFunctions`?
  - `Circle` already extends `Shape`
  - Java does not allow `Circle` to also extend `Comparable`!
- An interface is an abstract class with no concrete components

```
public interface Comparable{
  public abstract int cmp(Comparable s);
}
```

Mutiple inheritance

- Can we sort `Circle` objects using the generic functions in `SortFunctions`?
  - `Circle` already extends `Shape`
  - Java does not allow `Circle` to also extend `Comparable`!
- An interface is an abstract class with no concrete components

```
public interface Comparable{
  public abstract int cmp(Comparable s);
}
```

- A class that extends an interface is said to implement it:

```
public class Circle extends Shape implements Comparable{
  public double perimeter(){...}
  public int cmp(Comparable s){...}
      ...
}
```

Mutiple inheritance

- Can we sort `Circle` objects using the generic functions in `SortFunctions`?
  - `Circle` already extends `Shape`
  - Java does not allow `Circle` to also extend `Comparable`!
- An interface is an abstract class with no concrete components

```
public interface Comparable{
  public abstract int cmp(Comparable s);
}
```

- A class that extends an interface is said to implement it:

```
public class Circle extends Shape implements Comparable{
  public double perimeter(){...}
  public int cmp(Comparable s){...}
      ...
}
```

- Can extend only one class, but can implement multiple interfaces

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 {
  ...
}
```

Private classes

- An instance variable can be a user defined type
  - `Employee` uses `Date`

```
public class Employee{
  private String name;
  private double salary;
  private Date joindate;

  ...

}

public class Date {
  private int day, month year;

  ...
}
```

Private classes

- An instance variable can be a user defined type
  - `Employee` uses `Date`
- `Date` is a public class, also available to other classes

```
public class Employee{
  private String name;
  private double salary;
  private Date joindate;

  ...

}

public class Date {
  private int day, month year;

  ...
}
```

Private classes

- An instance variable can be a user defined type
  - `Employee` uses `Date`
- `Date` is a public class, also available to other classes
- When could a private class make sense?

```
public class Employee{
  private String name;
  private double salary;
  private Date joindate;

  ...

}

public class Date {
  private int day, month year;

  ...
}
```

Nested objects

- LinkedList is built using Node

```
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);
  }
}
```

Nested objects

- LinkedList is built using Node
- Why should Node be public?
  - May want to enhance with prev field, doubly linked list
  - Does not affect interface of LinkedList

```
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);
  }
}
```

Nested objects

- LinkedList is built using Node
- Why should Node be public?
  - May want to enhance with prev field, doubly linked list
  - Does not affect interface of LinkedList
- Instead, make Node a private class
  - Nested within LinkedList
  - Also called an inner class

```
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;
    ...
  }
}
```

Nested objects

- LinkedList is built using Node
- Why should Node be public?
  - May want to enhance with prev field, doubly linked list
  - Does not affect interface of LinkedList
- Instead, make Node a private class
  - Nested within LinkedList
  - Also called an inner class
- Objects of private class can see private components of enclosing class

```
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;
    ...
  }
}
```

Manipulating objects

- Encapsulation is a key principle of object oriented programming
  - Internal data is private
  - Access to the data is regulated through public methods
  - Accessor and mutator methods

```
public class Date {
  private int day, month year;

  public void getDay(int d) {...}
  public void getMonth(int m) {...}
  public void getYear(int y) {...}

  public void setDay(int d) {...}
  public void setMonth(int m) {...}
  public void setYear(int y) {...}
}
```

Manipulating objects

- Encapsulation is a key principle of object oriented programming
  - Internal data is private
  - Access to the data is regulated through public methods
  - Accessor and mutator methods
- Can ensure data integrity by regulating access

```
public class Date {
  private int day, month year;

  public void getDay(int d) {...}
  public void getMonth(int m) {...}
  public void getYear(int y) {...}

  public void setDay(int d) {...}
  public void setMonth(int m) {...}
  public void setYear(int y) {...}
}
```

Manipulating objects

- Encapsulation is a key principle of object oriented programming
  - Internal data is private
  - Access to the data is regulated through public methods
  - Accessor and mutator methods
- Can ensure data integrity by regulating access
- Update date as a whole, rather than individual components

```
public class Date {
  private int day, month year;

  public void getDay(int d) {...}
  public void getMonth(int m) {...}
  public void getYear(int y) {...}

  public void setDate(int d, int m, int y) {
    ...
    // Validate d-m-y combination
  }

}
```

Manipulating objects

- Encapsulation is a key principle of object oriented programming
  - Internal data is private
  - Access to the data is regulated through public methods
  - Accessor and mutator methods
- Can ensure data integrity by regulating access
- Update date as a whole, rather than individual components
- Does this provide sufficient control?

```
public class Date {
  private int day, month year;

  public void getDay(int d) {...}
  public void getMonth(int m) {...}
  public void getYear(int y) {...}

  public void setDate(int d, int m, int y) {
    ...
    // Validate d-m-y combination
  }

}
```

Interactions with state

- Object stores train reservation information
  - Can query availability for a given train, date

```
public class RailwayBooking {
  private BookingDB railwaydb;

  public int getStatus(int trainno, Date d) {
    // Return number of seats available
    // on train number trainno on date d
    ...
  }
}
```

Interactions with state

- Object stores train reservation information
  - Can query availability for a given train, date
- To control spamming by bots, require user to log in before querying

```
public class RailwayBooking {
  private BookingDB railwaydb;

  public int getStatus(int trainno, Date d) {
    // Return number of seats available
    // on train number trainno on date d
    ...
  }
}
```

Interactions with state

- Object stores train reservation information
  - Can query availability for a given train, date
- To control spamming by bots, require user to log in before querying
- Need to connect the query to the logged in status of the user

```
public class RailwayBooking {
  private BookingDB railwaydb;

  public int getStatus(int trainno, Date d) {
    // Return number of seats available
    // on train number trainno on date d
    ...
  }
}
```

Interactions with state

- Object stores train reservation information
  - Can query availability for a given train, date
- To control spamming by bots, require user to log in before querying
- Need to connect the query to the logged in status of the user
- "Interaction with state"

```
public class RailwayBooking {
  private BookingDB railwaydb;

  public int getStatus(int trainno, Date d) {
    // Return number of seats available
    // on train number trainno on date d
    ...
  }
}
```

Querying a database

- Need to connect the query to the logged in status of the user

```
public class RailwayBooking {
  private BookingDB railwaydb;

  public int getStatus(int trainno, Date d) {
    // Return number of seats available
    // on train number trainno on date d
    ...
  }
}
```

Querying a database

- Need to connect the query to the logged in status of the user
- Use objects!
  - On log in, user receives an object that can make a query
  - Object is created from private class that can look up `railwaydb`

```
public class RailwayBooking {
  private BookingDB railwaydb;

  public QueryObject login(String u, String p){
    QueryObject qobj;
    if (valid_login(u,p)) {
       qobj = new QueryObject();
       return(qobj);
    }
  }

  private class QueryObject {
    public int getStatus(int trainno, Date d) {
      // Return number of seats available
      // on train number trainno on date d
      ...
    }
  }
}
```

Querying a database

- Need to connect the query to the logged in status of the user
- Use objects!
  - On log in, user receives an object that can make a query
  - Object is created from private class that can look up `railwaydb`
- How does user know the capabilities of private class `QueryObject`?

```
public class RailwayBooking {
  private BookingDB railwaydb;

  public QueryObject login(String u, String p){
    QueryObject qobj;
    if (valid_login(u,p)) {
       qobj = new QueryObject();
       return(qobj);
    }
  }

  private class QueryObject {
    public int getStatus(int trainno, Date d) {
      // Return number of seats available
      // on train number trainno on date d
      ...
    }
  }
}
```

Querying a database

- Need to connect the query to the logged in status of the user
- Use objects!
  - On log in, user receives an object that can make a query
  - Object is created from private class that can look up `railwaydb`
- How does user know the capabilities of private class `QueryObject`?
- Use an interface!
  - Interface describes the capability of the object returned on login

```
public interface QIF{
  public abstract int
    getStatus(int trainno, Date d);
}

public class RailwayBooking {
  private BookingDB railwaydb;
  public QIF login(String u, String p){
    QueryObject qobj;
    if (valid_login(u,p)) {
       qobj = new QueryObject();
       return(qobj);
    }
  }
  private class QueryObject implements QIF {
    public int getStatus(int trainno, Date d){
     ...
    }
  }
}
```

Querying a database

- Query object allows unlimited number of queries

```
public interface QIF{
  public abstract int
    getStatus(int trainno, Date d);
}

public class RailwayBooking {
  private BookingDB railwaydb;
  public QIF login(String u, String p){
    QueryObject qobj;
    if (valid_login(u,p)) {
       qobj = new QueryObject();
       return(qobj);
    }
  }
  private class QueryObject implements QIF {
    public int getStatus(int trainno, Date d){
      ...
    }
  }
}
```

Querying a database

- Query object allows unlimited number of queries
- Limit the number of queries per login?

```
public interface QIF{
  public abstract int
    getStatus(int trainno, Date d);
}

public class RailwayBooking {
  private BookingDB railwaydb;
  public QIF login(String u, String p){
    QueryObject qobj;
    if (valid_login(u,p)) {
       qobj = new QueryObject();
       return(qobj);
    }
  }
  private class QueryObject implements QIF {
    public int getStatus(int trainno, Date d){
     ...
    }
  }
}
```

Querying a database

- Query object allows unlimited number of queries
- Limit the number of queries per login?
- Maintain a counter
  - Add instance variables to object returned on login
  - Query object can remember the state of the interaction

```
public class RailwayBooking {
  private BookingDB railwaydb;
  public QIF login(String u, String p){
    QueryObject qobj;
    if (valid_login(u,p)) {
       qobj = new QueryObject();
       return(qobj);
    }
  }
  private class QueryObject implements QIF {
    private int numqueries;
    private static int QLIM;

    public int getStatus(int trainno, Date d){
      if (numqueries < QLIM){
        // respond, increment numqueries
      }
    }
  }
}
```