Week 4

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();
    ```
- 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

```
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 < 2; 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
}
```

- 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){
    ...
  }
}
```

- 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`!
- 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){...}
     ...
}
```

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
- 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
- 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
- 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

Inner class

Nested objects

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

- An object can have nested objects as instance variables
- In some situations, the structure of these nested objects need not be exposed
- Private classes allow an additional degree of data encapsulation
- Combine private classes with interfaces to provide controlled access to the state of an object

```
class Outer {
   Inner o = new Inner();

   class Inner {
      private int i = 0;

      private Inner() {
      }
   }

   public void setI() {
      o.i = 10;
      System.out.println(o.i);
   }

   public static void main(String[] args) {
      Outer ob = new Outer();
      ob.setI();
      }
}
```

Controlled Interaction through Objects

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
  }

}
```

Querying a database

- 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
- 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
- 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
      }
    }
  }
}
```

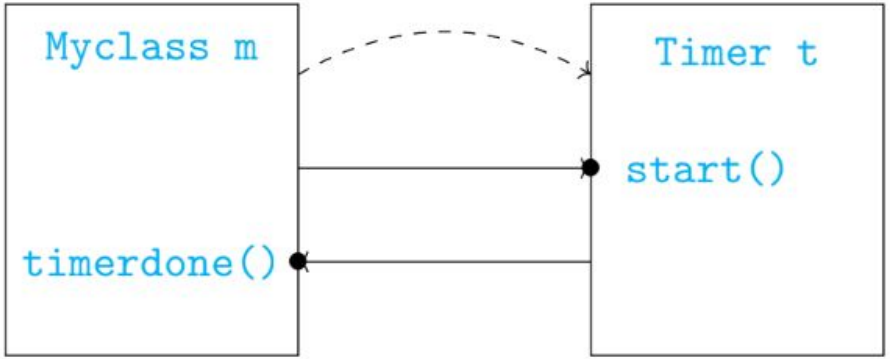
Summary

- Can provide controlled access to an object
- Combine private classes with interfaces
- External interaction is through an object of the private class
- Capabilities of this object are known through a public interface
- Object can maintain instance variables to track the state of the interaction

Callback

Implementing a call-back facility

- `Myclass m` creates a `Timer t`
- Start `t` to run 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 `timerdone()`



Implementing callbacks

- Code for `Myclass`
- `Timer t` should know whom to notify
  - `Myclass m` passes its identity when it creates `Timer t`
- Code for `Timer`
  - Interface `Runnable` indicates that `Timer` can run in parallel
- `Timer` specific to `Myclass`
- Create a generic `Timer`?

```
public class Myclass{

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

  public void timerdone(){...}
}
```

```
public class Timer
      implements Runnable{
  // Timer can be
  // invoked in parallel

  private Myclass owner;

  public Timer(Myclass o){
    owner = o;  // My creator
  }

  public void start(){
    ...
    owner.timerdone();
    // I'm done
  }
}
```

A generic timer

- Use Java class hierarchy
- Parameter of `Timer` constructor of type `Object`
  - Compatible with all caller types
- Need to cast `owner` back to `Myclass`

```
public class Myclass{

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

  public void timerdone(){...}
}
```

```
public class Timer
       implements Runnable{
  // Timer can be
  // invoked in parallel

  private Object owner;

  public Timer(Object o){
    owner = o;  // My creator
  }

  public void start(){
    ...
    ((Myclass) owner).timerdone();
    // I'm done
  }
}
```

Use interfaces

- Define an interface for callback

```
public interface
    Timerowner{

  public abstract
    void timerdone();
}
```

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

```
public class Myclass
   implements Timerowner{

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

  public void timerdone(){...}
}
```

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

  public Timer(Timerowner o){
    owner = o;  // My creator
  }

  public void start(){
    ...
    owner.timerdone();
    // I'm done
  }
}
```

Summary

- Callbacks are useful when we spawn a class in parallel
- Spawned object notifies the owner when it is done
- Can also notify some other object when done
  - `owner` in `Timer` need not be the object that created the `Timer`
- Interfaces allow this callback to be generic
  - `owner` has to have the capability to be notified

Iterator

Linear list

- A generic linear list of objects
- Internal implementation may vary
- An array implementation
- A linked list implementation

```
public 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 (...}
}
```

Iteration

- Want a loop to run through all values in a linear list
- If the list is an array with public access, we write this
- For a linked list with public access, we could write this
- We don't have public access ...
- ...and we don't know which implementation is in use!

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

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

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`?
- 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);
  }
 }
```

- Definition of `Iter` depends on linear list

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!

```
Linearlist l = new Linearlist();
...
Object oi,oj;
Iterator i,j;

i = l.get_iterator();
while (i.has_next()){
  oi = i.get_next();
  j = l.get_iterator();
  while (j.has_next()){
    oj = j.get_next();
    ... // do something with oi, oj
  }
}
...
```

Summary

- Iterators are another example of interaction with state
  - Each iterator needs to remember its position in the list
- Export an object with a prespecified interface to handle the interaction
- The new Java `for` over lists implicitly constructs and uses an iterator

```
for (type x : a)
  do something with x;
}
```