Java: classes, interfaces

Madhavan Mukund, S P Suresh

Programming Language Concepts

Lecture 5, 23 January 2024

Classes and subclasses

- A class can extend another one — subclass
  - Subclass inherits fields and methods
  - Can add new instance variables and methods
  - Call parent constructor to set up hidden parts
  - Use `super` to refer to parent class
- Subclasses are subtypes
  - `Employee e = new Manager(...);`
- Dynamic dispatch — runtime polymorphism
  - `e.bonus()` refers to `Manager.bonus()`
- Static typechecking, casting
  - `e.getSecretary()` generates an error
  - `((Manager) e).getSecretary()` works

```
public class Employee{
  private String name;
  private double salary;
  // Some Constructors ...
  // Some methods ...
  public boolean setName(String s){ ...
  ...
  public double bonus(float percent){ ..
}

public class Manager extends Employee{
  private String secretary;

  // New methods ...
  public boolean setSecretary(name s){ .
  public String getSecretary(){ ... }
  // Overridden methods ...
  public double bonus(float percent){ ..
}
```

Modifiers in Java

- Java uses many modifiers in declarations, to cover different features of object-oriented programming

Modifiers in Java

- Java uses many modifiers in declarations, to cover different features of object-oriented programming
- `public` vs `private` to support encapsulation of data

Modifiers in Java

- Java uses many modifiers in declarations, to cover different features of object-oriented programming

- `public` vs `private` to support encapsulation of data

- `static`, for entities defined inside classes that exist without creating objects of the class

Modifiers in Java

- Java uses many modifiers in declarations, to cover different features of object-oriented programming
- `public` vs `private` to support encapsulation of data
- `static`, for entities defined inside classes that exist without creating objects of the class
- `final`, for values that cannot be changed

Modifiers in Java

- Java uses many modifiers in declarations, to cover different features of object-oriented programming
- `public` vs `private` to support encapsulation of data
- `static`, for entities defined inside classes that exist without creating objects of the class
- `final`, for values that cannot be changed
- These modifiers can be applied to classes, instance variables and methods

Modifiers in Java

- Java uses many modifiers in declarations, to cover different features of object-oriented programming
- `public` vs `private` to support encapsulation of data
- `static`, for entities defined inside classes that exist without creating objects of the class
- `final`, for values that cannot be changed
- These modifiers can be applied to classes, instance variables and methods
- Let's look at some examples of situations where different combinations make sense

public vs private

- Faithful implementation of encapsulation necessitates modifiers `public` and `private`
  - Typically, instance variables are `private`
  - Methods to query (accessor) and update (mutator) the state are `public`

public vs private

- Faithful implementation of encapsulation necessitates modifiers `public` and `private`
  - Typically, instance variables are `private`
  - Methods to query (accessor) and update (mutator) the state are `public`
- Can `private` methods make sense?

public vs private

- Faithful implementation of encapsulation necessitates modifiers public and private
  - Typically, instance variables are private
  - Methods to query (accessor) and update (mutator) the state are public
- Can private methods make sense?
- Example: a Stack class
  - Data stored in a private array
  - Public methods to push, pop, query if empty

```
public class Stack {
  private int[] values; // array of values
  private int tos;      // top of stack
  private int size;     // values.length

  /* Constructors to set up values array */

  public void push (int i){
    ....
  }

  public int pop (){
    ...
  }

  public boolean is_empty (){
    return (tos == 0);
  }
}
```

private methods

- Example: a Stack class
  - Data stored in a private array
  - Public methods to push, pop, query if empty

```
public class Stack {
  private int[] values; // array of values
  private int tos;      // top of stack
  private int size;     // values.length

  /* Constructors to set up values array */

  public void push (int i){
    ....
  }

  public int pop (){
    ...
  }

  public boolean is_empty (){
    return (tos == 0);
  }
}
```

private methods

- Example: a Stack class
  - Data stored in a private array
  - Public methods to push, pop, query if empty
- push() needs to check if stack has space

```
public class Stack {
  ...
  public void push (int i){
    if (tos < size){
      values[tos] = i;
      tos = tos+1;
    }else{
      // Deal with stack overflow
    }
    ...
  }
  ...
}
```

private methods

- Example: a Stack class
  - Data stored in a private array
  - Public methods to push, pop, query if empty
- push() needs to check if stack has space
- Deal gracefully with stack overflow
  - private methods invoked from within push() to check if stack is full and expand storage

```
public class Stack {
  ...
  public void push (int i){
    if (stack_full()){
      extend_stack();
    }
    ... // Usual push operations
  }
  ...
  private boolean stack_full(){
    return(tos == size);
  }

  private void extend_stack(){
    /* Allocate additional space,
       reset size etc */
  }
}
```

Accessor and mutator methods

- Public methods to query and update private instance variables

Accessor and mutator methods

- Public methods to query and update private instance variables
- Date class
  - Private instance variables day, month, year
  - One public accessor/mutator method per instance variable

```
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) {...}
}
```

Accessor and mutator methods

- Public methods to query and update private instance variables
- Date class
  - Private instance variables day, month, year
  - One public accessor/mutator method per instance variable
- Inconsistent updates are now possible
  - Separately set invalid combinations of day and month

```
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) {...}
}
```

Accessor and mutator methods

- Public methods to query and update private instance variables
- Date class
  - Private instance variables day, month, year
  - One public accessor/mutator method per instance variable
- Inconsistent updates are now possible
  - Separately set invalid combinations of day and month
- Instead, allow only combined update

```
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
  }

}
```

`static` components

- Use `static` for components that exist without creating objects
  - Library functions, `main()`, ...
  - Useful constants like `Math.PI`, `Integer.MAX_VALUE`

static components

- Use static for components that exist without creating objects
  - Library functions, main(), ...
  - Useful constants like Math.PI, Integer.MAX_VALUE
- These static components are also public

static components

- Use static for components that exist without creating objects
  - Library functions, main(), . . .
  - Useful constants like Math.PI, Integer.MAX_VALUE
- These static components are also public
- Do private static components make sense?

static components

- Use static for components that exist without creating objects
  - Library functions, main(), ...
  - Useful constants like Math.PI, Integer.MAX_VALUE
- These static components are also public
- Do private static components make sense?
- Internal constants for bookkeeping
  - Constructor sets unique id for each order

```
public class Order {
  private static int lastorderid = 0;

  private int orderid;
  ....

  public Order(...) {
    lastorderid++;
    orderid = lastorderid;
    ...
  }
```

Only one copy across all Order objects

static components

- Use static for components that exist without creating objects
  - Library functions, main(), ...
  - Useful constants like Math.PI, Integer.MAX_VALUE
- These static components are also public
- Do private static components make sense?
- Internal constants for bookkeeping
  - Constructor sets unique id for each order

```
public class Order {
  private static int lastorderid = 0;

  private int orderid;
  ....

  public Order(...) {
    lastorderid++;
    orderid = lastorderid;
    ...
  }
```

- lastorderid is private static field

static components

- Use static for components that exist without creating objects
  - Library functions, main(), . . .
  - Useful constants like Math.PI, Integer.MAX_VALUE
- These static components are also public
- Do private static components make sense?
- Internal constants for bookkeeping
  - Constructor sets unique id for each order

```
public class Order {
  private static int lastorderid = 0;

  private int orderid;
  ....

  public Order(...) {
    lastorderid++;
    orderid = lastorderid;
    ...
  }
```

- lastorderid is private static field
- Common to all objects in the class

static components

- Use static for components that exist without creating objects
  - Library functions, main(), ...
  - Useful constants like Math.PI, Integer.MAX_VALUE
- These static components are also public
- Do private static components make sense?
- Internal constants for bookkeeping
  - Constructor sets unique id for each order

```
public class Order {
  private static int lastorderid = 0;

  private int orderid;
  ....

  public Order(...) {
    lastorderid++;
    orderid = lastorderid;
    ...
  }
```

- lastorderid is private static field
- Common to all objects in the class
- Be careful about concurrent updates!

final components

- final denotes that a value cannot be updated

`final` components

- `final` denotes that a value cannot be updated
- Usually used for constants (`public` and `static` instance variables)
    - `Math.PI`, `Integer.MAX_VALUE`

final components

- final denotes that a value cannot be updated
- Usually used for constants (public and static instance variables)
  - Math.PI, Integer.MAX_VALUE
- What would final mean for a method?
  - Cannot redefine functions at run-time, unlike Python!

f = body

def f(..):
≡

def f(.):
≡

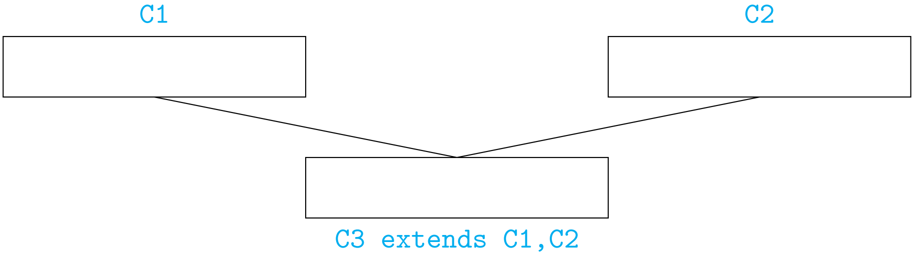
`final` components

- `final` denotes that a value cannot be updated
- Usually used for constants (`public` and `static` instance variables)
  - `Math.PI`, `Integer.MAX_VALUE`
- What would `final` mean for a method?
  - Cannot redefine functions at run-time, unlike Python!
- Recall overriding
  - Subclass redefines a method available with the same signature in the parent class

final components

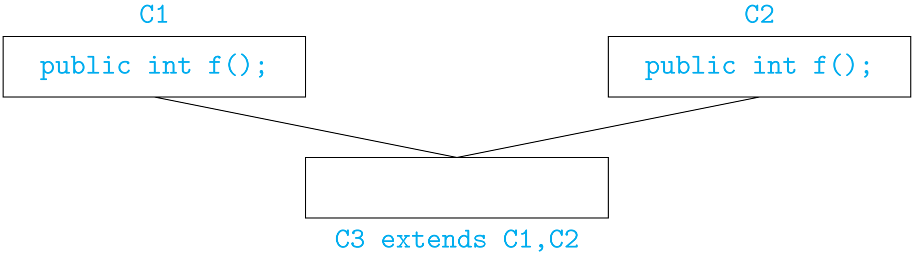
- final denotes that a value cannot be updated
- Usually used for constants (public and static instance variables)
  - Math.PI, Integer.MAX_VALUE
- What would final mean for a method?
  - Cannot redefine functions at run-time, unlike Python!
- Recall overriding
  - Subclass redefines a method available with the same signature in the parent class
- A final method cannot be overridden

Multiple inheritance



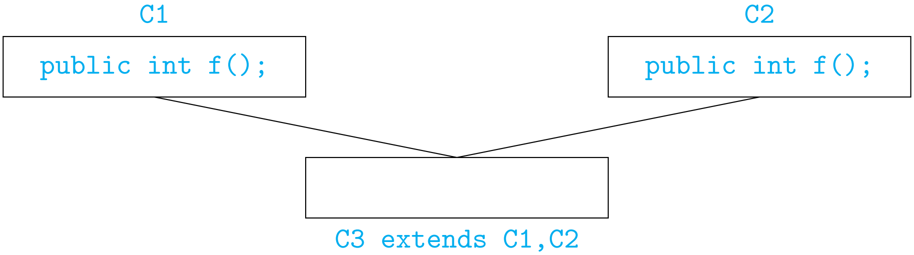
- Can a subclass extend multiple parent classes?

Multiple inheritance



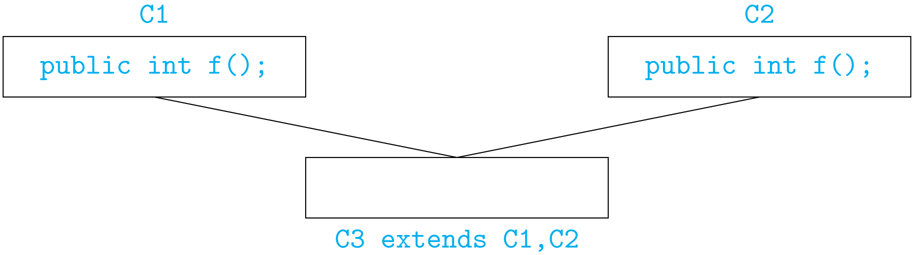
- Can a subclass extend multiple parent classes?
- If `f()` is not overridden, which `f()` do we use in `C3`?

Multiple inheritance



- Can a subclass extend multiple parent classes?
- If `f()` is not overridden, which `f()` do we use in `C3`?
- Java does not allow multiple inheritance

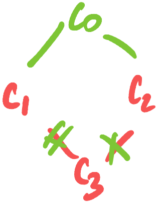
Multiple inheritance



- Can a subclass extend multiple parent classes?
- If `f()` is not overridden, which `f()` do we use in `C3`?
- Java does not allow multiple inheritance
- C++ allows this if `C1` and `C2` have no conflict

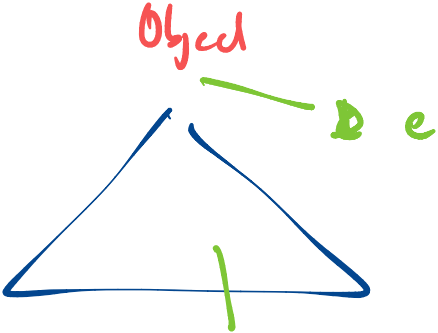
Java class hierarchy

- No multiple inheritance — tree-like



Java class hierarchy

- No multiple inheritance — tree-like
- In fact, there is a universal superclass `Object`



Java class hierarchy

- No multiple inheritance — tree-like
- In fact, there is a universal superclass `Object`
- Useful methods defined in `Object`

```
public boolean equals(Object o)  // defaults to reference (pointer) equality

public String toString()         // converts the values of the
                                 // instance variables to String
```

Java class hierarchy

- No multiple inheritance — tree-like
- In fact, there is a universal superclass `Object`
- Useful methods defined in `Object`

```
public boolean equals(Object o)  // defaults to reference (pointer) equality

public String toString()         // converts the values of the
                                 // instance variables to String
```

- For Java objects `x` and `y`, `x == y` invokes `x.equals(y)`

Java class hierarchy

- No multiple inheritance — tree-like
- In fact, there is a universal superclass `Object`
- Useful methods defined in `Object`

```
public boolean equals(Object o)  // defaults to reference (pointer) equality

public String toString()         // converts the values of the
                                 // instance variables to String
```

- For Java objects `x` and `y`, `x == y` invokes `x.equals(y)`
- To print `o`, use `System.out.println(o+"");` coerces o to String
  - Implicitly invokes `o.toString()`

Java class hierarchy

- Can exploit the tree structure to write generic functions
  - Example: search for an element in an array

```
public int find (Object[] objarr, Object o){
  int i;
  for (i = 0; i < objarr.length(); i++){
      if (objarr[i] == o) {return i};
  }
  return (-1);
}
```

Can be different types when invoked

Java class hierarchy

- Can exploit the tree structure to write generic functions
  - Example: search for an element in an array

```
public int find (Object[] objarr, Object o){
  int i;
  for (i = 0; i < objarr.length(); i++){
      if (objarr[i] == o) {return i};
  }
  return (-1);
}
```

- Recall that == is pointer equality, by default

Java class hierarchy

- Can exploit the tree structure to write generic functions
  - Example: search for an element in an array

```
public int find (Object[] objarr, Object o){
  int i;
  for (i = 0; i < objarr.length(); i++){
      if (objarr[i] == o) {return i};
  }
  return (-1);
}
```

- Recall that `==` is pointer equality, by default
- If a class overrides `equals()`, dynamic dispatch will use the redefined function instead of `Object.equals()` for `objarr[i] == o`

Functions, signatures and overloading

- Signature of a function is its name and the list of argument types

Functions, signatures and overloading

- Signature of a function is its name and the list of argument types
- Can have different functions with the same name and different signatures
  - For example, multiple constructors

Functions, signatures and overloading

- Signature of a function is its name and the list of argument types
- Can have different functions with the same name and different signatures
  - For example, multiple constructors
- Java class `Arrays` has a method `sort` to sort arbitrary scalar arrays

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
  // sorts contents of darr
Arrays.sort(iarr);
  // sorts contents of iarr
```

Functions, signatures and overloading

- Signature of a function is its name and the list of argument types
- Can have different functions with the same name and different signatures
  - For example, multiple constructors
- Java class `Arrays` has a method `sort` to sort arbitrary scalar arrays
- Made possible by overloaded methods defined in class `Arrays`

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
  // sorts contents of darr
Arrays.sort(iarr);
  // sorts contents of iarr


class Arrays{
  ...
  public static void sort(double[] a){..}
     // sorts arrays of double[]
  public static void sort(int[] a){..}
     // sorts arrays of int[]
  ...
}
```

Functions, signatures and overloading

- Overloading: multiple methods, different signatures, choice is static

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
  // sorts contents of darr
Arrays.sort(iarr);
  // sorts contents of iarr


class Arrays{
  ...
  public static void sort(double[] a){..}
     // sorts arrays of double[]
  public static void sort(int[] a){..}
     // sorts arrays of int[]
  ...
}
```

Functions, signatures and overloading

- Overloading: multiple methods, different signatures, choice is static
- Overriding: multiple methods, same signature, choice is static
  - `Employee.bonus()`
  - `Manager.bonus()`

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
  // sorts contents of darr
Arrays.sort(iarr);
  // sorts contents of iarr


class Arrays{
  ...
  public static void sort(double[] a){..}
     // sorts arrays of double[]
  public static void sort(int[] a){..}
     // sorts arrays of int[]
  ...
}
```

Functions, signatures and overloading

- Overloading: multiple methods, different signatures, choice is static
- Overriding: multiple methods, same signature, choice is static
  - `Employee.bonus()`
  - `Manager.bonus()`
- Dynamic dispatch: multiple methods, same signature, choice made at run-time

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);
  // sorts contents of darr
Arrays.sort(iarr);
  // sorts contents of iarr
```

```
class Arrays{
  ...
  public static void sort(double[] a){..}
     // sorts arrays of double[]
  public static void sort(int[] a){..}
     // sorts arrays of int[]
  ...
}
```

Overriding functions

- For instance, a class `Date` with instance variables `day`, `month` and `year`

Overriding functions

- For instance, a class `Date` with instance variables `day`, `month` and `year`
- May wish to override `equals()` to compare the object state, as follows

```
public boolean equals(Date d){
  return ((this.day == d.day) &&
          (this.month == d.month) &&
          (this.year == d.year));
}
```

Private variables of one Date are visible to another

Overriding functions

- For instance, a class `Date` with instance variables `day`, `month` and `year`

- May wish to override `equals()` to compare the object state, as follows

```
public boolean equals(Date d){
  return ((this.day == d.day) &&
          (this.month == d.month) &&
          (this.year == d.year));
}
```

- Unfortunately,
  `boolean equals(Date d)`
  does not override
  `boolean equals(Object o)`!

Overriding functions

- For instance, a class `Date` with instance variables `day`, `month` and `year`
- May wish to override `equals()` to compare the object state, as follows

```
public boolean equals(Date d){
  return ((this.day == d.day) &&
          (this.month == d.month) &&
          (this.year == d.year));
}
```

- Unfortunately,

  `boolean equals(Date d)`

  does not override

  `boolean equals(Object o)`!

- Should write, instead

```
public boolean equals(Object d){
  if (d instanceof Date){
    Date myd = (Date) d;
    return ((this.day == myd.day) &&
            (this.month == myd.month)
            (this.year == myd.year));
  }
  return(false);
}
```

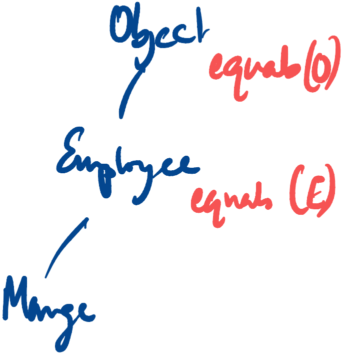
  - Note the run-time type check and the cast

Overriding functions

- Overriding looks for “closest” match

Overriding functions

- Overriding looks for “closest” match
- Suppose we have `public boolean equals(Employee e)` but no `equals()` in `Manager`



Overriding functions

- Overriding looks for "closest" match
- Suppose we have `public boolean equals(Employee e)` but no `equals()` in `Manager`
- Consider

```
Manager m1 = new Manager(...);
Manager m2 = new Manager(...);
...
if (m1.equals(m2)){ ... }
```

Overriding functions

- Overriding looks for "closest" match
- Suppose we have `public boolean equals(Employee e)` but no `equals()` in `Manager`
- Consider

```
Manager m1 = new Manager(...);
Manager m2 = new Manager(...);
...
if (m1.equals(m2)){ ... }
```

- `public boolean equals(Manager m)` is compatible with both `boolean equals(Employee e)` and `boolean equals(Object o)`

Overriding functions

- Overriding looks for “closest” match
- Suppose we have `public boolean equals(Employee e)` but no `equals()` in `Manager`
- Consider

```
Manager m1 = new Manager(...);
Manager m2 = new Manager(...);
...
if (m1.equals(m2)){ ... }
```

- `public boolean equals(Manager m)` is compatible with both `boolean equals(Employee e)` and `boolean equals(Object o)`
- Use `boolean equals(Employee e)`

Subclasses, subtyping and inheritance

- Class hierarchy provides both subtyping and inheritance

Subclasses, subtyping and inheritance

- Class hierarchy provides both subtyping and inheritance
- Subtyping
  - Capabilities of the subtype are a superset of the main type
  - If `B` is a subtype of `A`, wherever we require an object of type `A`, we can use an object of type `B`
  - `Employee e = new Manager(...);` is legal

Subclasses, subtyping and inheritance

- Class hierarchy provides both subtyping and inheritance
- Subtyping
  - Capabilities of the subtype are a superset of the main type
  - If `B` is a subtype of `A`, wherever we require an object of type `A`, we can use an object of type `B`
  - `Employee e = new Manager(...);` is legal
- Inheritance
  - Subtype can reuse code of the main type
  - `B` inherits from `A` if some functions for `B` are written in terms of functions of `A`
  - `Manager.bonus()` uses `Employee.bonus()`

Subclasses, subtyping and inheritance

- Subtyping
  - Compatibility of interfaces.
  - `B` is a subtype of `A` if every function that can be invoked on an object of type `A` can also be invoked on an object of type `B`.

Subclasses, subtyping and inheritance

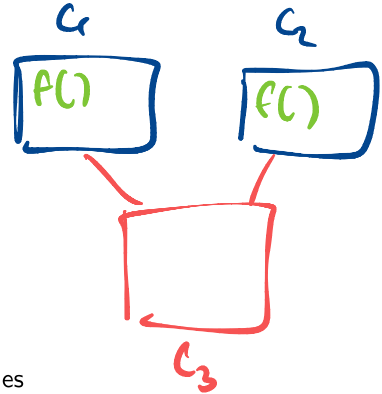
- Subtyping
  - Compatibility of interfaces.
  - `B` is a subtype of `A` if every function that can be invoked on an object of type `A` can also be invoked on an object of type `B`.
- Inheritance
  - Reuse of implementations.
  - `B` inherits from `A` if some functions for `B` are written in terms of functions of `A`.

Subclasses, subtyping and inheritance

- Subtyping
  - Compatibility of interfaces.
  - `B` is a subtype of `A` if every function that can be invoked on an object of type `A` can also be invoked on an object of type `B`.
- Inheritance
  - Reuse of implementations.
  - `B` inherits from `A` if some functions for `B` are written in terms of functions of `A`.
- Using one idea (hierarchy of classes) to implement both concepts blurs the distinction between the two.
  - Recall the example of `Deque`, `Stack` and `Queue`.

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



Interfaces express relevant 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

a.cmp(b)

```
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])
  }
}
```

Ord in Haskell

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

Interactions with state

- Connect database query to 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){
     ...
    }
  }
}
```

Interactions with state . . .

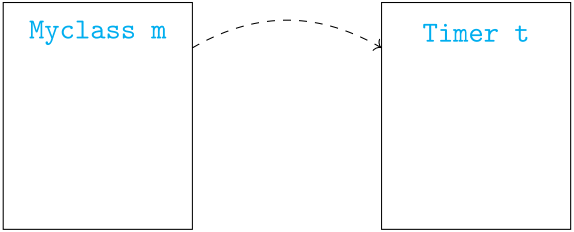
- Query object allows unlimited number of queries
- Limit the number of queries per login?
- Maintain a counter
  - Add instance variables to object returned on login
  - Query object can remember the state of the interaction

```
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
      }
    }
  }
}
```

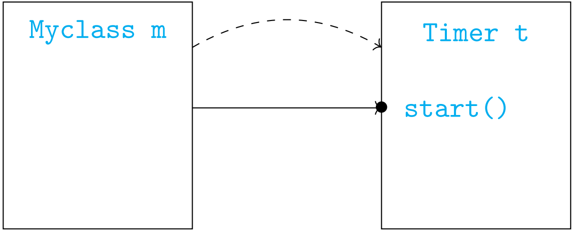
Implementing a call-back facility

- `Myclass m` creates a `Timer t`



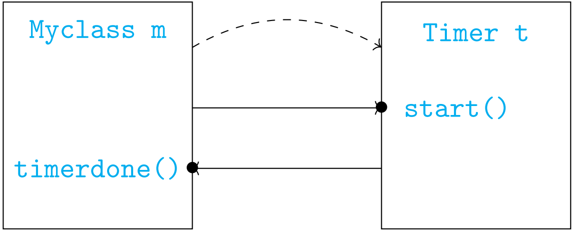
Implementing a call-back facility

- `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!



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

```
public class Myclass{

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

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

Implementing callbacks

- Code for Myclass
- Timer t should know whom to notify
  - Myclass m passes its identity when it creates Timer t

```
public class Myclass{

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

  public void 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

```
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
  }
}
```

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`

```
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
  }
}
```

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
  }
}
```