Array operations

- Adding elements
- Removing/deleting elements
- Finding elements
- Increasing the size of an array
- Sorting

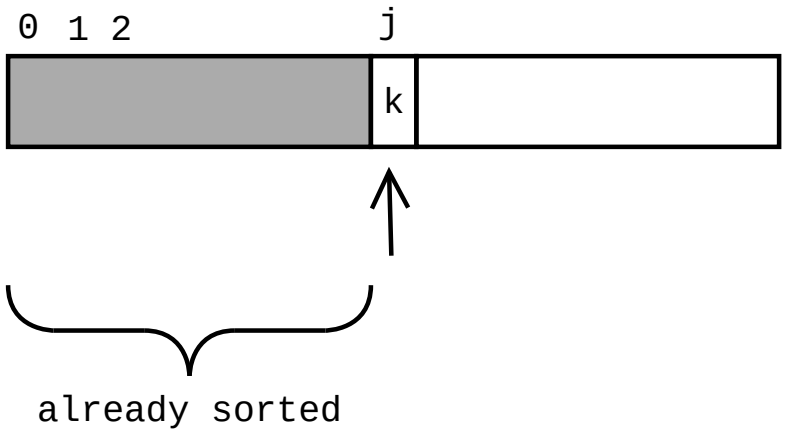
Sorting

- To sort an array of objects we need:
  - each object to have a *key*
  - a way to *compare* keys

Insertion sort

- Notation (not Java!): `a[i..j]` is the part of the array from the `i`-th index to the `j`-th index.

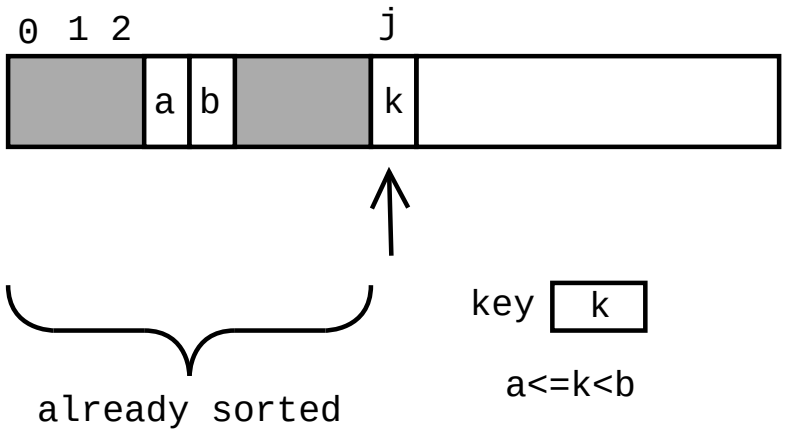
- Idea: sorting a set of cards can be done by inserting a card in the subset of the cards which are already sorted.



Insertion sort

- Notation (not Java!): `a[i..j]` is the part of the array from the `i`-th index to the `j`-th index.

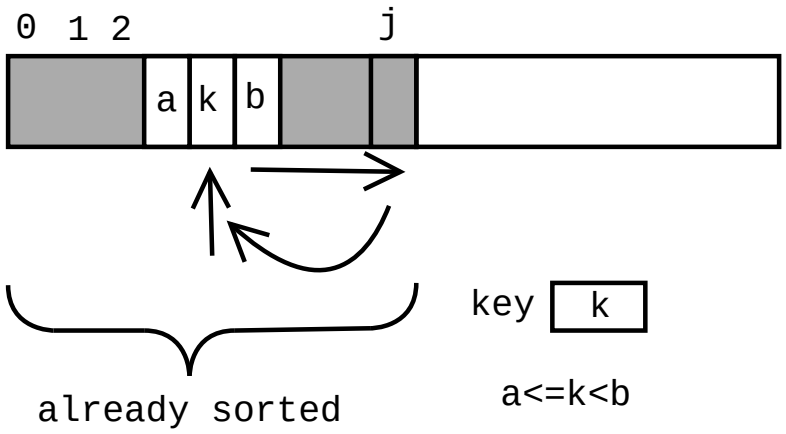
- Idea: sorting a set of cards can be done by inserting a card in the subset of the cards which are already sorted.



Insertion sort

- Notation (not Java!): `a[i..j]` is the part of the array from the `i`-th index to the `j`-th index.

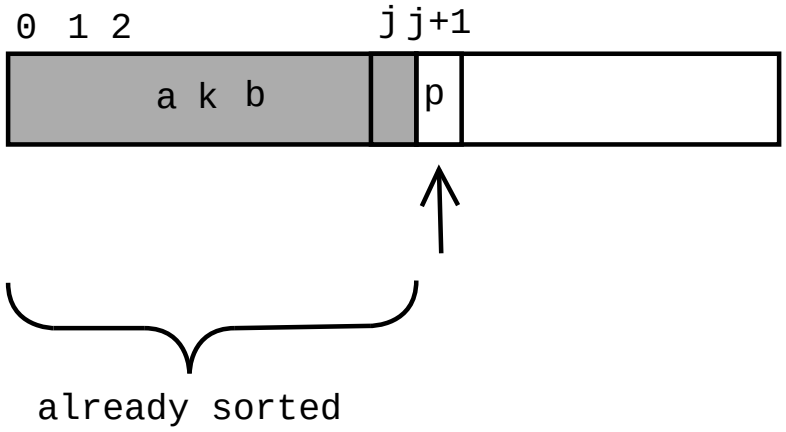
- Idea: sorting a set of cards can be done by inserting a card in the subset of the cards which are already sorted.



Insertion sort

- Notation (not Java!): `a[i..j]` is the part of the array from the `i`-th index to the `j`-th index.

- Idea: sorting a set of cards can be done by inserting a card in the subset of the cards which are already sorted.



Insertion sort

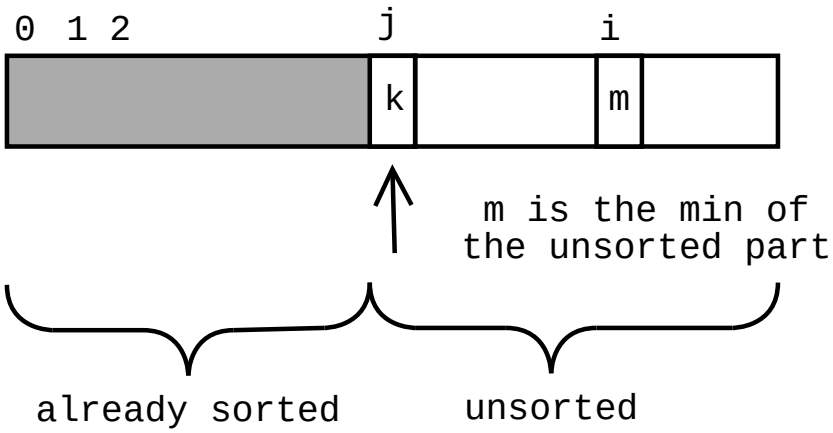
- Algorithm refined:

1. For each `j` from 1 to the length of `a-1`
   (a) Insert `a[j]` into the sorted subarray `a[0..j-1]`

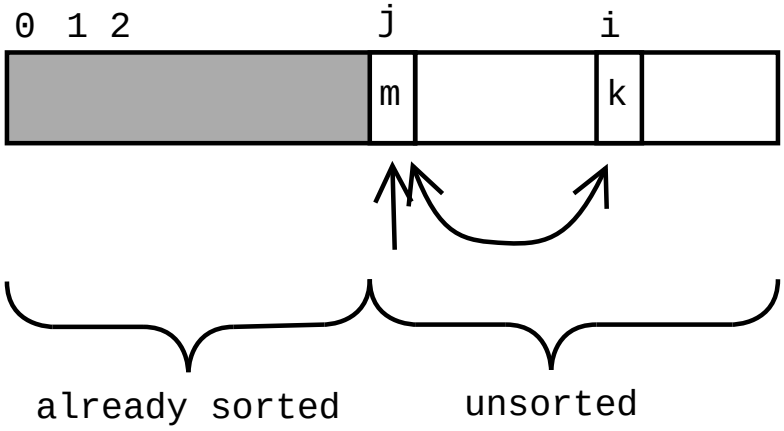
Insertion sort

```
void insertion_sort() // In class Library
{
  int i, j;
  String key;
  Book focus;
  j = 1;
  while (j < book_list.length)
  {
    focus = book_list[j];
    key = focus.title();
    i = j - 1;
    while (i >= 0 &&
      key.compareTo(book_list[i].title()) < 0 )
    {
      book_list[i+1] = book_list[i];
      i--;
    }
    book_list[i+1] = focus;
    j++;
  }
}
```

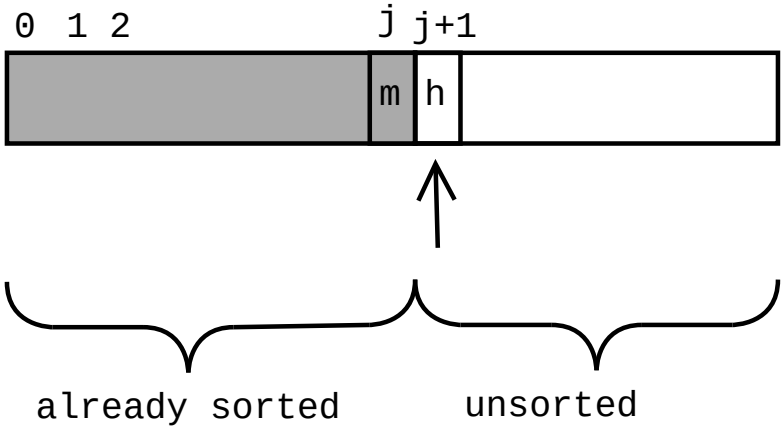
Selection sort



Selection sort



Selection sort



Selection sort

- Algorithm

1. For each `j` from 0 to length `a - 2` do
   (a) Let `min_index` to be the index of the minimum in `a[j+1..length a-1]`
   (b) Swap `a[min_index]` and `a[j]`

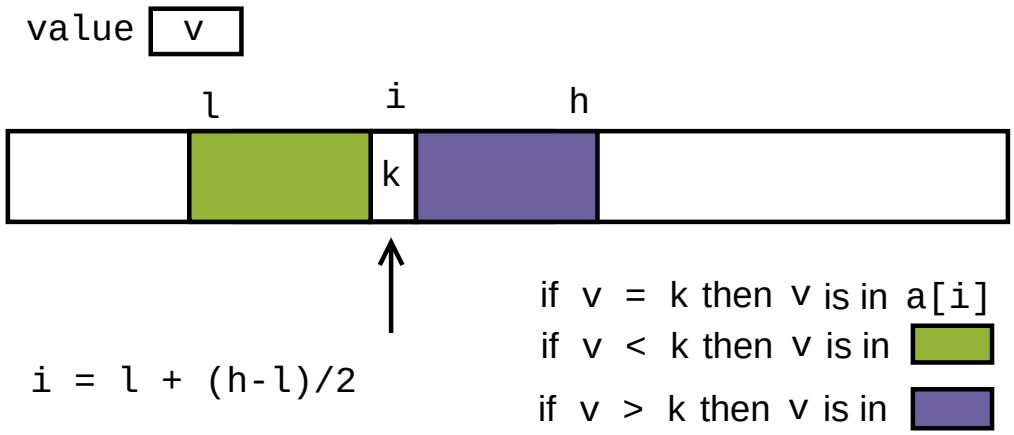
Selection sort

```
void selection_sort() {
  int min_index, j = 0, i;
  String minimum;
  Book temp;
  while (j <= a.length - 2) {
    minimum = book_list[j].title();
    min_index = j;
    i = j + 1;
    while (i <= book_list.length - 1) {
      String current_key = book_list[i].title();
      if (current_key.compareTo(minimum) < 0) {
        minimum = current_key;
        min_index = i;
      }
      i++;
    }
    temp = book_list[j];
    book_list[j] = book_list[min_index];
    book_list[min_index] = temp;
    j++;
  }
}
```

Binary search

- But if we know that the array is sorted, we can improve the speed of searching by ignoring parts which do not need to look at.

- If we are looking for a value `v` in an array `a`, and we have already narrowed down the search space to `a[l..h]`, then



Binary search

```
int binary_search(String title)
{
  int lower = 0, higher = book_list.length - 1;
  int index;
  String current_title;
  int comparison;
  while (lower <= higher) {
    index = lower + (higher - lower) / 2;
    current_title = book_list[index].title();
    comparison = title.compareTo(current_title);
    if (comparison == 0) {
      return index;
    }
    else if (comparison < 0) {
      higher = index - 1;
    }
    else {  // comparison > 0
      lower = index + 1;
    }
  }
  return -1;  // Not found
}
```

Array operations

```
class Library
{
  private Book[] book_list;
  private int next_available;
  private boolean sorted;
  public Library(int max_capacity) { ... }

  public int number_of_books() { ... } // Accessor

  public void add_book(Book m) { ... } // Mutator
  private void grow_array(int n) { ... } // Mutato
  public int book_index(String title) { ... } // A
  public Book find_book(String title) { ... } // A
  public void delete_book(String title) { ... } //

  public void sort() { ... } // Mutator
  public int linear_search(String title) { ... } /
  public int binary_search(String title) { ... } /
} // End of Library
```

Array operations

```
public void sort()
{
  if (!sorted)
  {
    // Code for insertion sort or selection sort
  }
  sorted = true;
}
```

Array operations

```
public int book_index(String title)
{
  if (sorted)
  {
    return binary_search(title);
  }
  return linear_search(title);
}
```

Array operations

```
public int linear_search(String title)
{
  int i;
  i = 0;
  while (i < book_list.length)
  {
    Book m = book_list[i];
    if (m != null)
    {
      String s = m.title();
      if (s.equals(title))
      {
        return i;
      }
    }
    i++;
  }
  return -1;
}
```

Array operations

- Sorting takes too much time
- Alternative design:
  - Whenever we add a book, preserve the ordering
  - Whenever we delete a book, preserve the ordering
- Possible solution: sort every time we add or delete (bad idea.)

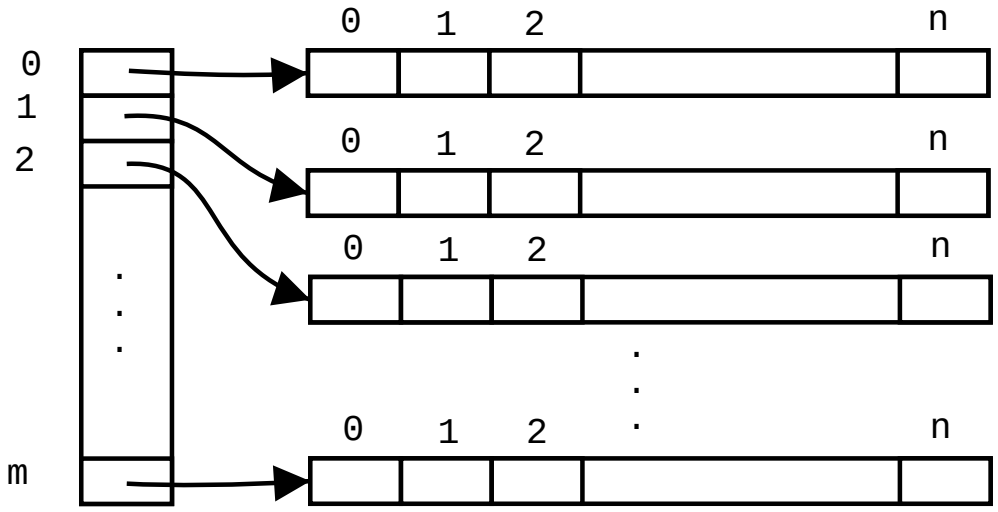
Array operations

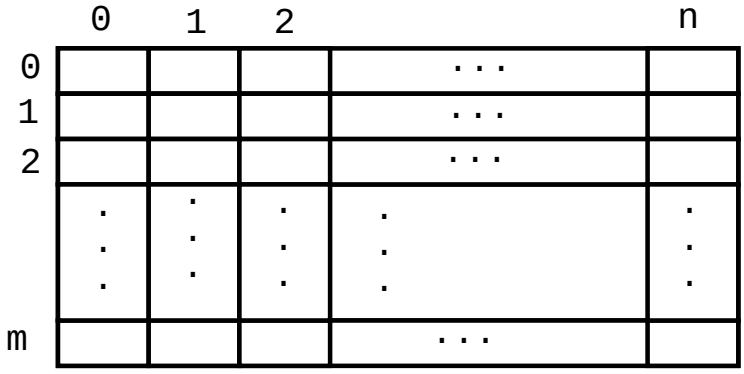
```
public void add_book(Book m)
{
  if (next_available >= book_list.length)
  {
    int l = book_list.length;
    grow_array( (int)(l * 0.10) + 10 );
  }
  Book focus = book_list[next_available];
  String key = focus.title();
  int i = next_available - 1;
  while (i >= 0 &&
    key.compareTo(book_list[i].title()) < 0 )
  {
    book_list[i+1] = book_list[i];
    i--;
  }
  book_list[i+1] = focus;
  next_available++;
}
```

Array operations

```
public void delete_book(String title)
{
  int i = book_index(title);
  if (i != -1)
  {
    book_list[i] = null;
    int j = i;
    while (j <= next_available - 2)
    {
      book_list[j] = book_list[j+1];
    }
    book_list[next_available - 1] = null;
    next_available--;
  }
}
```

Multidimensional arrays





Multidimensional arrays

- A two-dimensional array is an array of arrays.

```
int[][] table = new int[5][10];

for (int row=0; row < table.length; row++)
  for (int col=0; col < table[row].length; col++)
    table[row][col] = row * 10 + col;
```

- A multidimensional array is an n-dimensional array, i.e. an array of arrays of arrays of ...

- Processing nested arrays is commonly done with nested loops.

Efficiency

- Linear search: $O(n)$

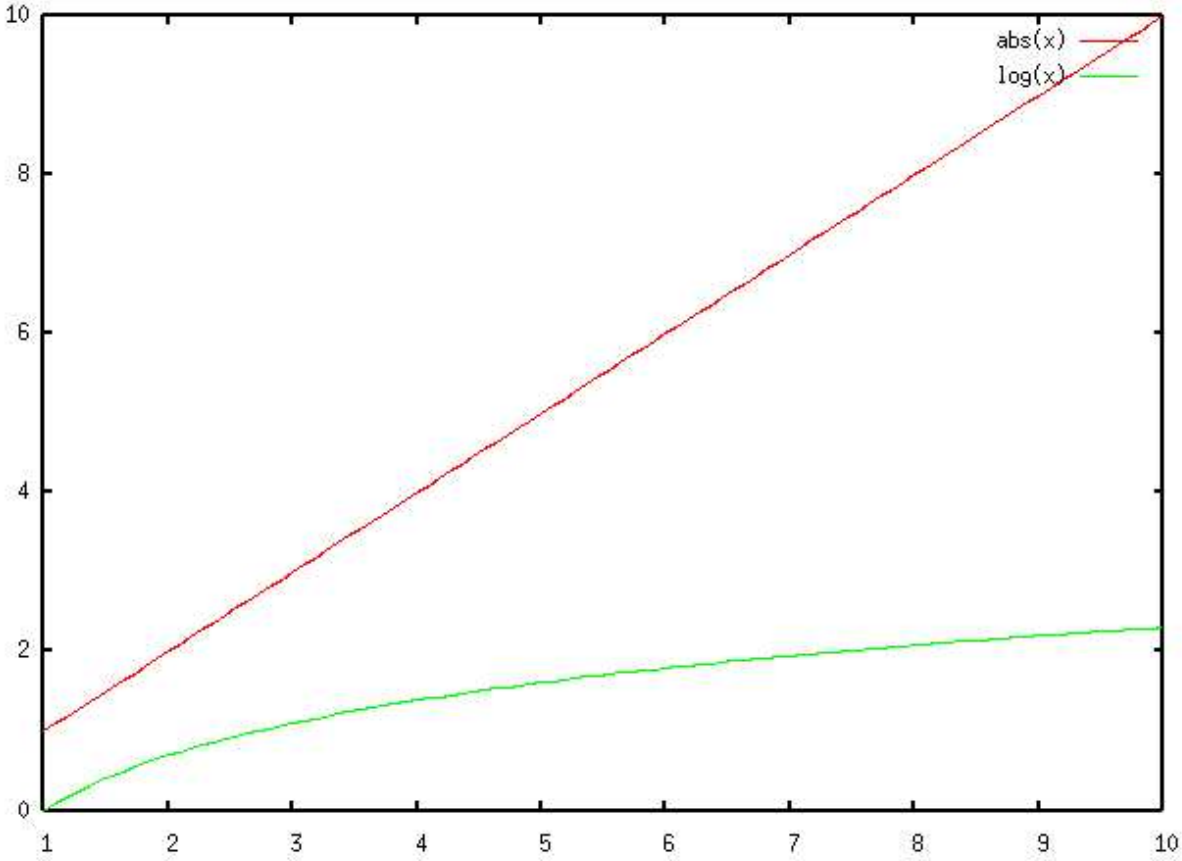
n	# of comparisons
1	1
2	2
3	3
4	4
⋮	⋮
1000	1000
⋮	⋮
10000000	10000000
⋮	⋮
k	k

Efficiency

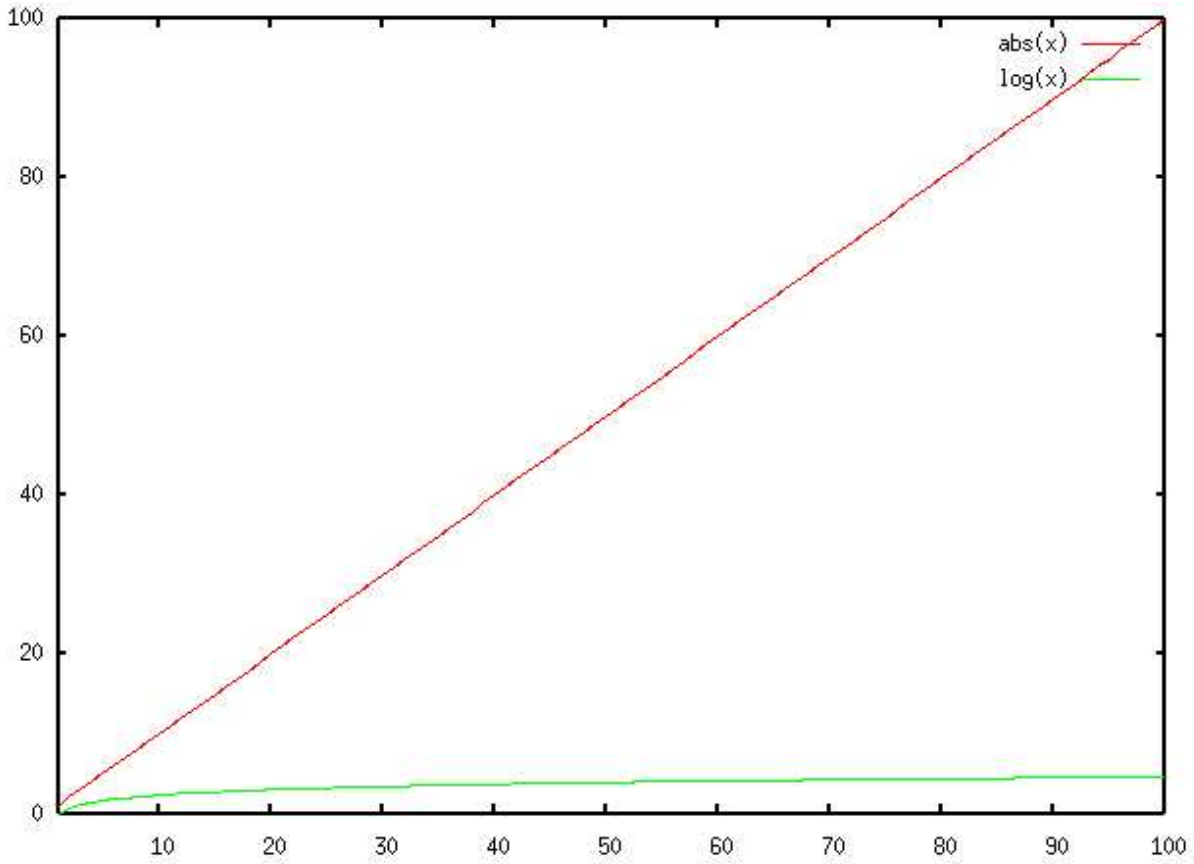
- Binary search: $O(log_2 n)$

n	# of comparisons
1	0
2	2
3	3
⋮	⋮
10	4
100	7
1000	10
10000	14
100000	17
1000000	20
10000000	24
100000000	27
10^9	30
10^{10}	33

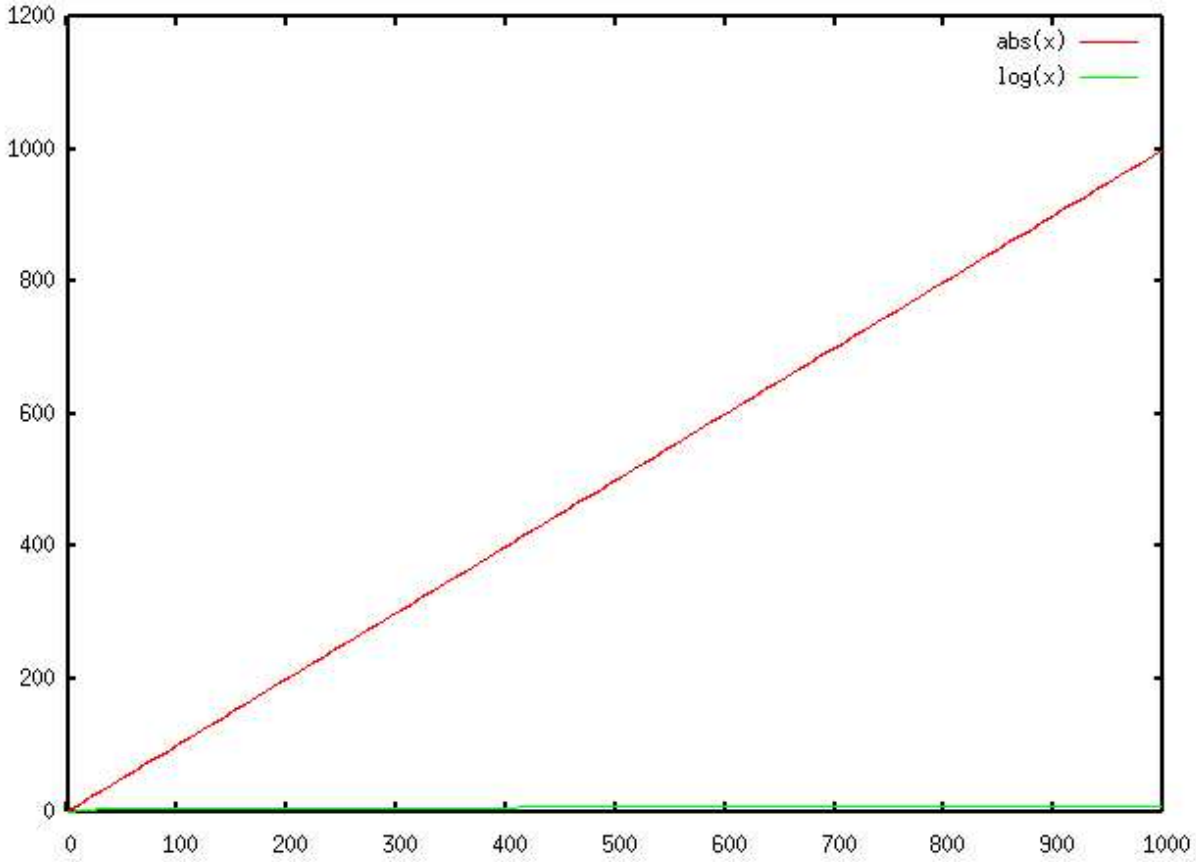
Efficiency



Efficiency



Efficiency


abs(x)
log(x)
1200
1000
800
600
400
200
0
0
100
200
300
400
500
600
700
800
900
1000

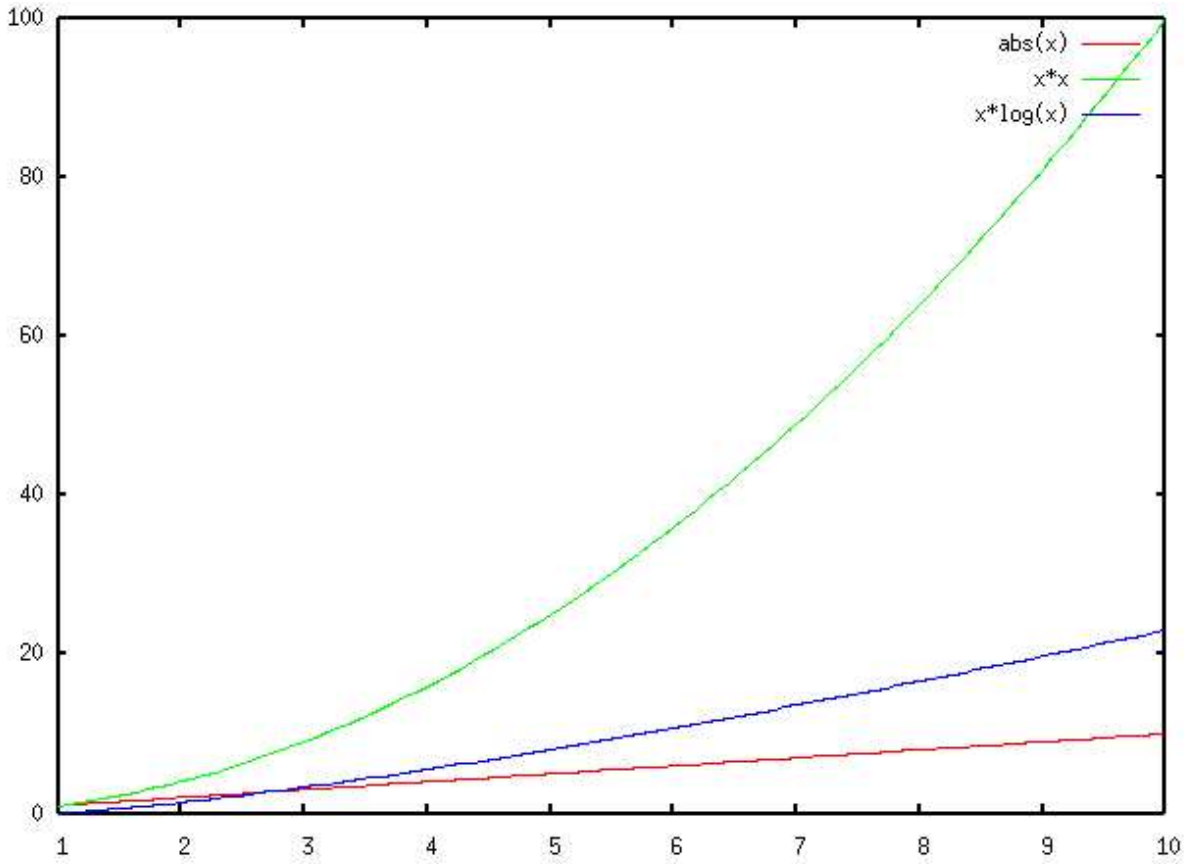
Sorting algorithms

- Insertion sort
- Selection sort
- Bubble sort
- Heap sort
- Merge sort
- Quick sort
- Bucket sort
- Counting sort
- Radix sort
- Sorting networks

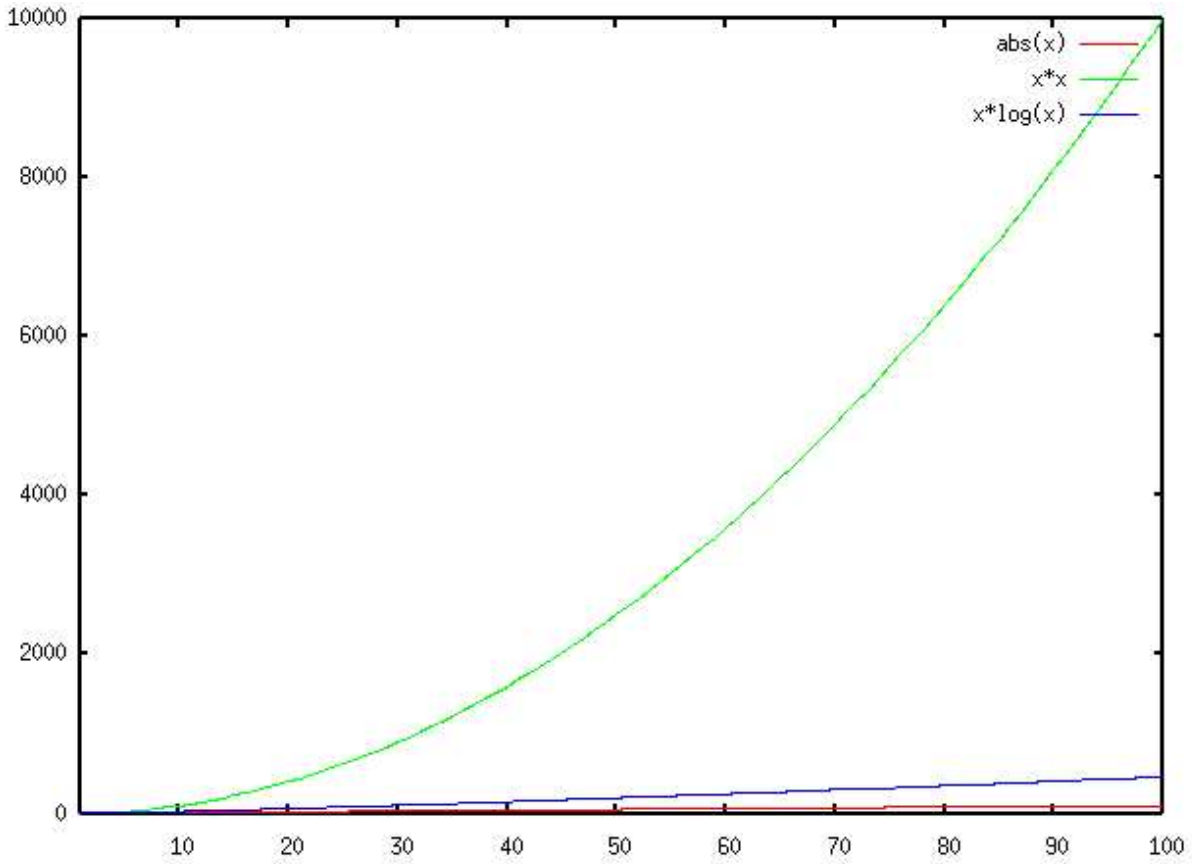
Sorting algorithms

Algorithm	Complexity	$n = 1000$	$n = 10^6$
Insertion sort	$O(n^2)$	10^6	10^{12}
Selection sort	$O(n^2)$	10^6	10^{12}
Bubble sort	$O(n^2)$	10^6	10^{12}
Heap sort	$O(n \log_2 n)$	≈ 10000	$\approx 20 \times 10^7$
Merge sort	$O(n \log_2 n)$	≈ 10000	$\approx 20 \times 10^7$
Quick sort	$O(n^2)$ in the worst case, but $O(n \log_2 n)$ on average		
Bucket sort	$O(n)$ but with restrictions	1000	10^6
Counting sort	$O(n)$ but with restrictions	1000	10^6
Radix sort	$O(n)$ but with restrictions	1000	10^6
Sorting networks	$O(n)$ but with restrictions	1000	10^6

Efficiency

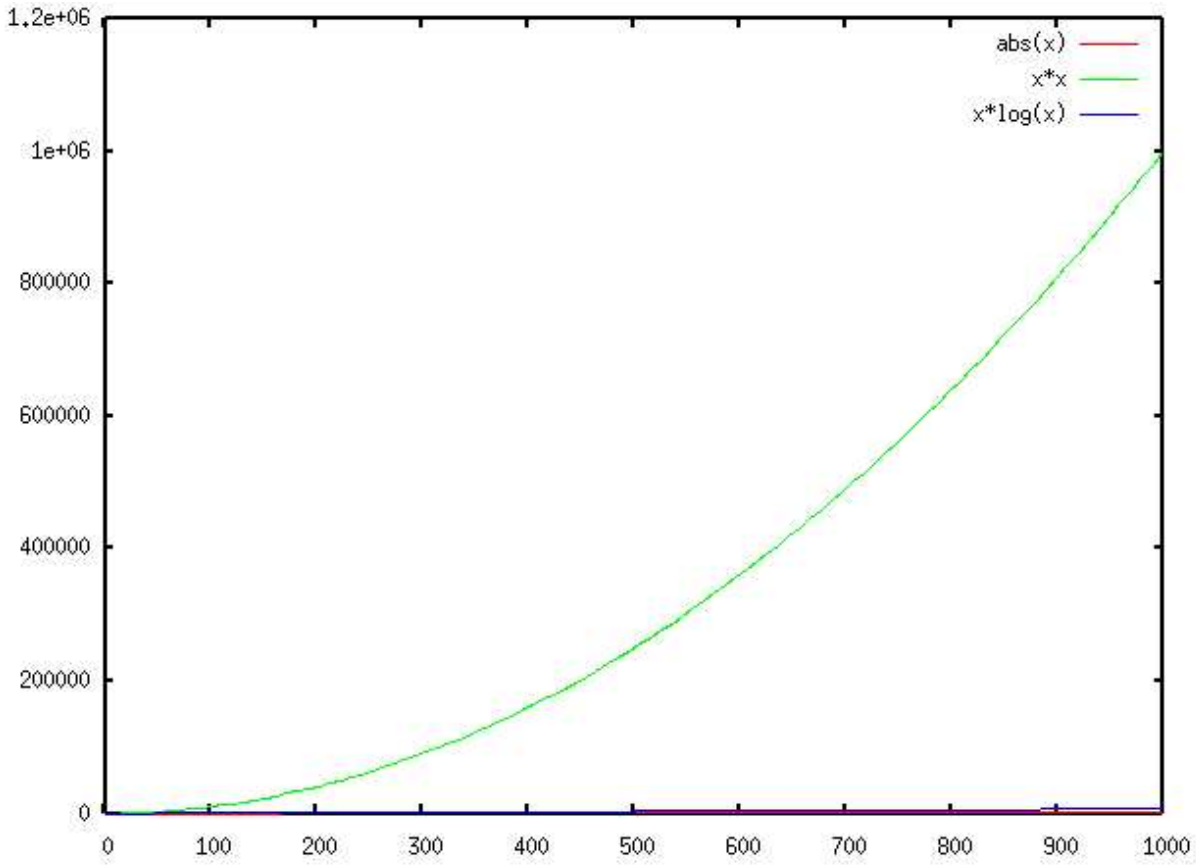


Efficiency



McGill

Efficiency


1.2e+06
1e+06
800000
600000
400000
200000
0
0
100
200
300
400
500
600
700
800
900
1000
abs(x)
x*x
x*log(x)

Object Oriented Programming

- The execution of an OO program consists of
  - Creation of objects
  - Interaction between objects (message-passing)
- Defining features of an OO language:
  - Class definitions (describing the types of objects and their structure,)
  - Object instantiation (creation,)
  - Message-passing (invoking methods,)
  - Aggregation (object structure, *has-a* relationships)
  - Encapsulation (objects as abstract units, hiding,)
  - Inheritance,
  - Polymorphism

Inheritance

- A class represents a set of objects which share the same structure (attributes) and capabilities (methods)

- Sometimes it is useful to identify specific subsets within a set (e.g. the set of savings accounts is a subset of the set of bank accounts, the set of art students is a subset of the set of students, the set of dogs is a subset of the set of animals. etc.)

- The elements of a subset A of a set B are more *specialized* than those of B. This is, they may have additional characteristics and capabilities.

Inheritance

- Inheritance is the mechanism that allows us to describe this *specialization* relationship between classes.

```
class B { ... }
class A extends B { ... }
```

- A is a *subclass* of B, or equivalently, A is *derived from* B, A is a *child of* B, or B is a *superclass of* A, or B is a *parent of* A.

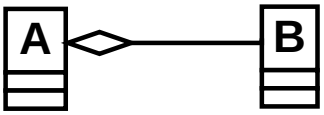
- Means that the set of A objects is a subset of the set of B objects.

```
class Labrador extends Dog { ... }
```

- Inheritance represents the "is-a" relationship

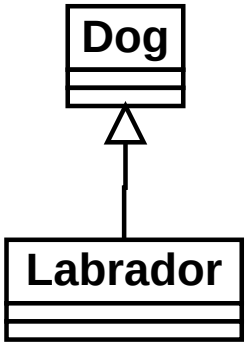
Inheritance

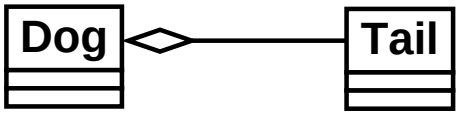
represents:
"every A is a B"
(inheritance)



represents:
"every A has a B"
(aggregation)

For example:





Inheritance

- A class is like a blueprint
- Objects are particular instances of that blueprint
- A subclass A of a class B is an extension to the original blueprint of B
- A subclass adds additional features (attributes *and* methods)
- We say that the subclass *inherits* all of its parent's attributes and methods
- An instance of the subclass has the attributes and methods of the parent in addition to the subclass's own attributes and methods.

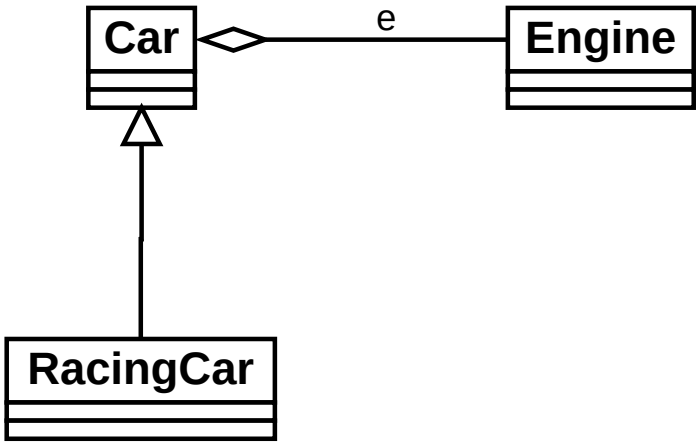
Inheritance

```
class C { ... }
class B
{
  C v;
  // ...
}
class A extends B
{
  // Has an implicit C v;
  // ...
}
```

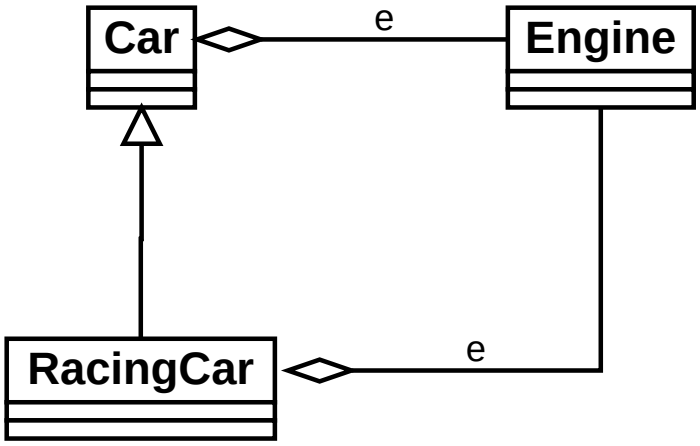
Inheritance

```
class Engine {
  // ...
}

class Car {
  Engine engine;
  // ...
}

class RacingCar extends Car {
  // It implicitly has Engine e;
  // ...
}

// In some client
RacingCar r = new RacingCar();
Engine e1 = r.engine;  // engine is inherited from
```

Inheritance



is the same as



Inheritance

- Inheritance also represents specialization

```
class Engine {
  // ...
}
class Car {
  Engine engine;
  Car() { engine = new Engine(); }
  // ...
}
class RacingCar extends Car {
  Aerofoil aero;
  TurboCharger turbo;
}

// In some client
RacingCar r = new RacingCar();
Engine e1 = r.engine;  // e is inherited from Car
TurboCharger t1 = r.turbo;
Car c = new Car();
Engine e2 = c.engine;
TurboCharger t2 = c.turbo; // Error
```

Inheritance

- Inheritance serves as a tool for reusability:
- We can write

```
class RacingCar extends Car {
  Aerofoil aero;
  TurboCharger turbo;
}
```

instead of

```
class RacingCar {
  Engine engine;
  Aerofoil aero;
  TurboCharger turbo;
}
```

Inheritance

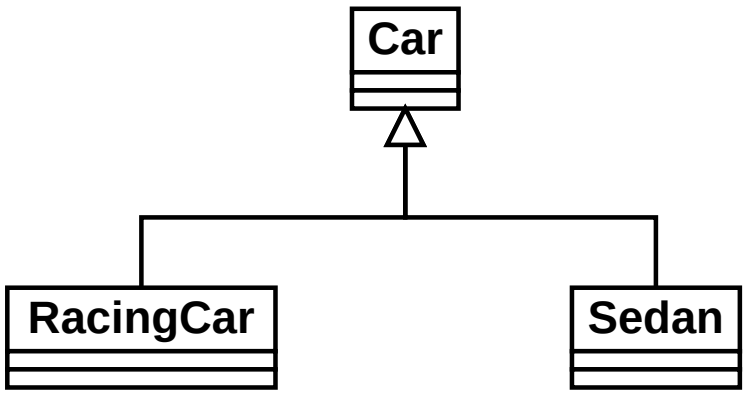
- Methods are inherited too:

```
class Engine {
  void start() { ... }
}
class Car {
  Engine engine;
  double speed;
  Car() { engine = new Engine(); speed = 0.0; }
  void turn_on()
  {
    engine.start();
  }
}
class RacingCar extends Car {
  Aerofoil aero;
  TurboCharger turbo;
}
// In some client
RacingCar r = new RacingCar();
r.turn_on(); // Inherited from Car
```

Inheritance

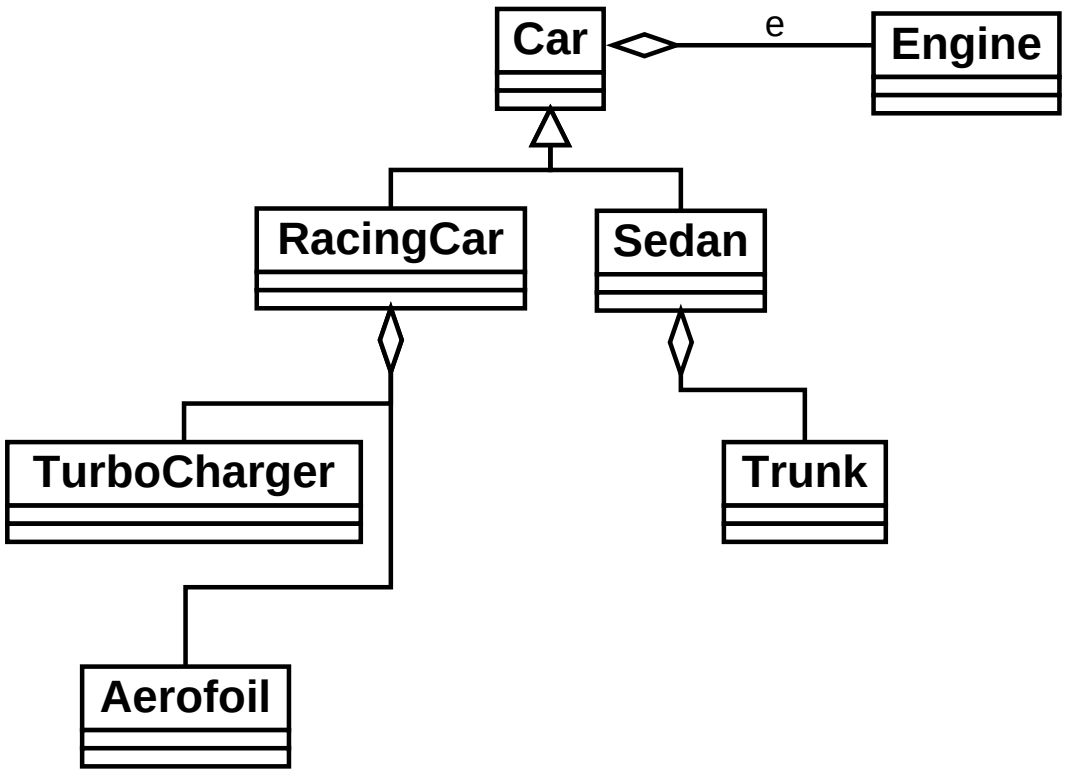
- Classes can have many subclasses

```
class Sedan extends Car {
  Trunk t;
  PassengerSeats[] ps;
}
// In some client
Sedan s = new Sedan();
s.turn_on();
```



Inheritance

- Attributes in a class are shared between its subclasses (but not the values of those attributes!)



Inheritance

- Inheritance is a transitive relation: if every A is a B and every B is a C, then every A is a C

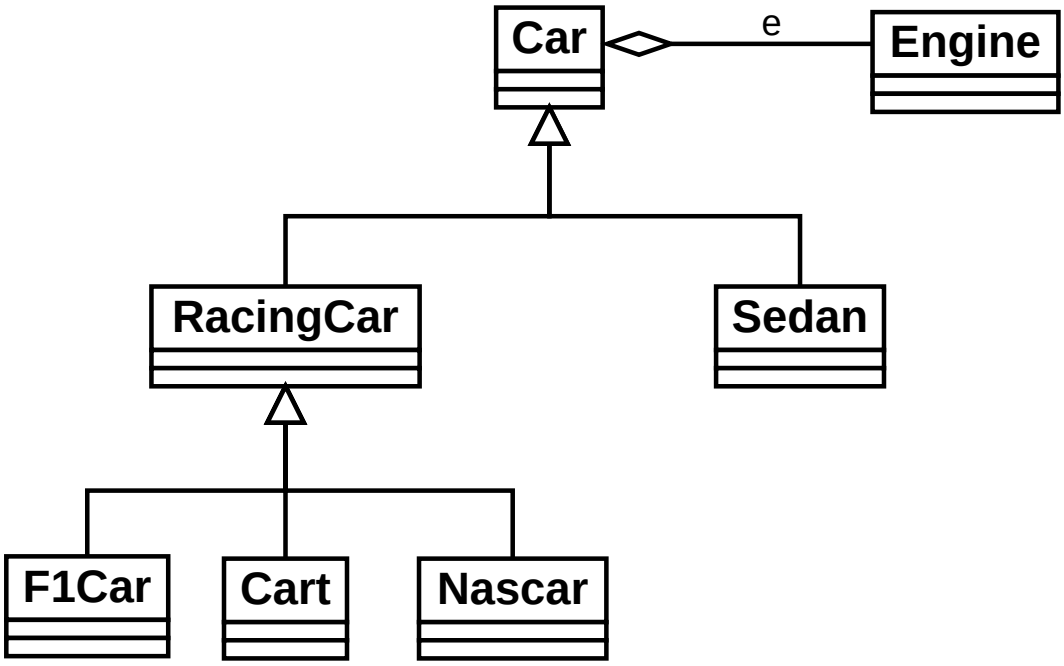
```
class F1Car extends RacingCar {
  SpeedControlSystem scs;
}
```

- instead of

```
class F1Car {
  Engine engine;
  Aerofoil aero;
  TurboCharger turbo;
  SpeedControlSystem scs;
}
```

Inheritance

- Class hierarchy:



Inheritance

- A closer look at inheritance as specialization

```
class Animal {
  boolean tired, hungry;
  void eat()
  {
    get_food();
    hungry = false;
  }
  void get_food() { ... }
  void sleep()
  {
    System.out.println("zzz...");
    tired = false;
  }
}
```

Inheritance

```
class Dog extends Animal {
  Legs[] l;
  Tail t;
  void run()
  {
    tired = true; // From class Animal
    hungry = true;
  }
  void bark()
  {
    System.out.println("Woof, Woof!");
  }
}
class Labrador extends Dog {
  void say_hello()
  {
    t.wiggle(); // t from class Dog
  }
}
```

Inheritance

```
public class ZooTest {
  public static void main(String[] args)
  {
    Labrador l = new Labrador();
    l.say_hello(); // Will call l.t.wiggle();
    l.run();
    if (l.hungry)
      l.eat(); // from class Animal
    if (l.tired)
      l.sleep();
  }
}
```

Inheritance

- Inheritance represents also a spectrum of possibilities or alternatives, given by the subclasses of a class

- If every B is an A and every C is an A, and nothing else is an A, then an A is either a B or a C
  - (e.g. if every racing car is a car, and every sedan is a car, and nothing else is a car, then a car is either a racing car or a sedan.)

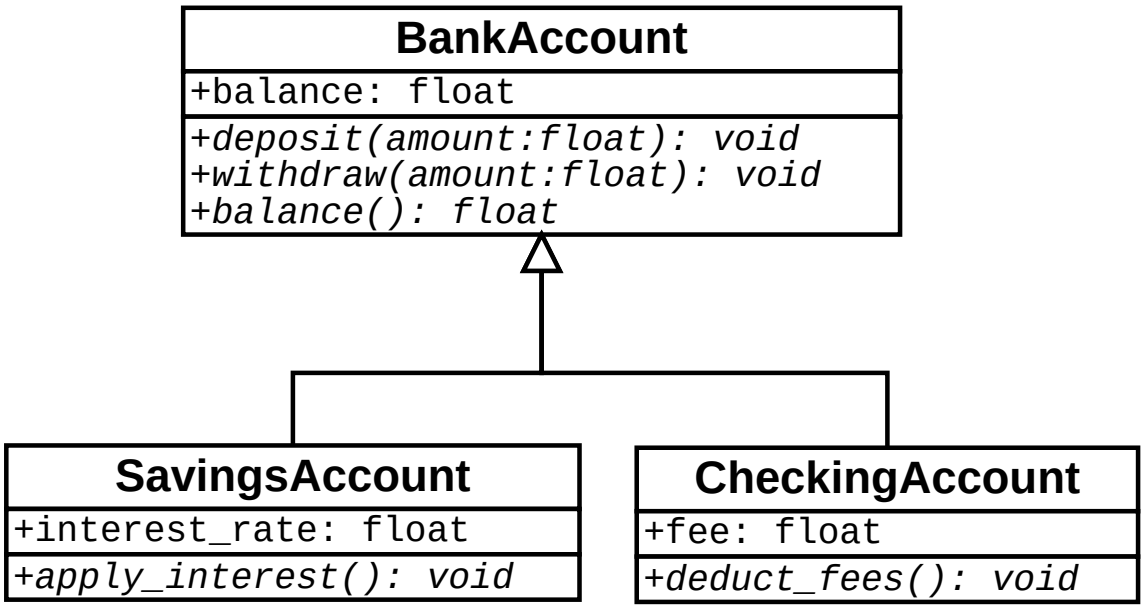
```
class Animal { ... }
class Dog extends Animal { ... }
class Cat extends Animal { ... }
class Bird extends Animal { ... }

// In some client
Animal a1 = new Dog();
Animal a2 = new Cat();
Animal a3 = new Bird();
Dog d = new Animal(); // Wrong!
```

Inheritance

- Classes as sets of objects:
  - "is-a" between an object and a class is the same as $\in$
  - "is-a" between two classes is the same as $\subseteq$
- Let A, B, C be sets
  - If $A \subseteq B$ and $x \in A$ then $x \in B$
  - If $A \subseteq B$ and $B \subseteq C$ then $A \subseteq B$
  - If $B \subseteq A$ and $C \subseteq A$, and there is no other set D such that $D \subseteq A$ then $A = B \cup C$

Inheritance

- A bank account is either a savings account or a checking account, then a savings account is a kind of bank account, and a checking account is a kind of bank account.



Inheritance

```
class BankAccount {
  private float balance;
  public BankAccount(float initial_balance)
  {
    balance = initial_balance;
  }
  public void deposit(float amount)
  {
    balance = balance + amount;
  }
  public void withdraw(float amount)
  {
    balance = balance - amount;
  }
  public float balance() { return balance; }
}
```

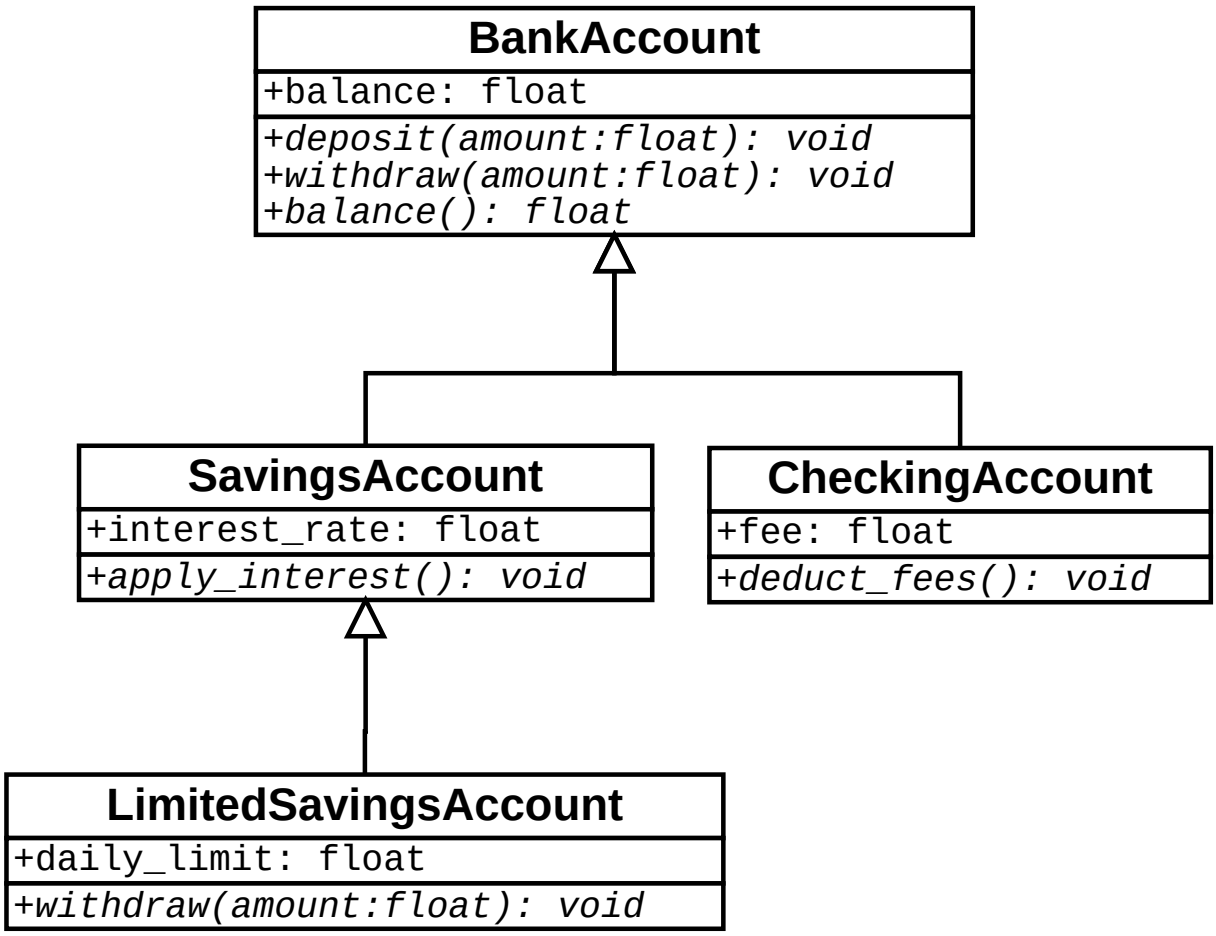
Inheritance

```
class SavingsAccount extends BankAccount {
  private float interest_rate;
  public SavingsAccount(float initial_balance,
                        float rate)
  {
    super(initial_balance); // Calls superclass
                            // constructor
    interest_rate = rate;
  }
  public void apply_interest()
  {
    balance = balance
            + balance * interest_rate/100.0;
  }
}
```

Inheritance

```
class CheckingAccount extends BankAccount {
  private float fee;
  public SavingsAccount(float initial_balance,
                        float fee)
  {
    super(initial_balance);
    this.fee = fee;
  }
  public void deduct_fee()
  {
    balance = balance - fee;
  }
}
```

Overriding methods



Overriding methods

```
class LimitedSavingsAccount
extends SavingsAccount {
  private float daily_limit;
  public LimitedAccount(float initial_balance,
                        float rate, float limit)
  {
    super(initial_balance, rate);
    daily_limit = limit;
  }
  public void withdraw(float amount)
  {
    if (amount < daily_limit)
      balance = balance - amount;
  }
}
```

Overriding methods

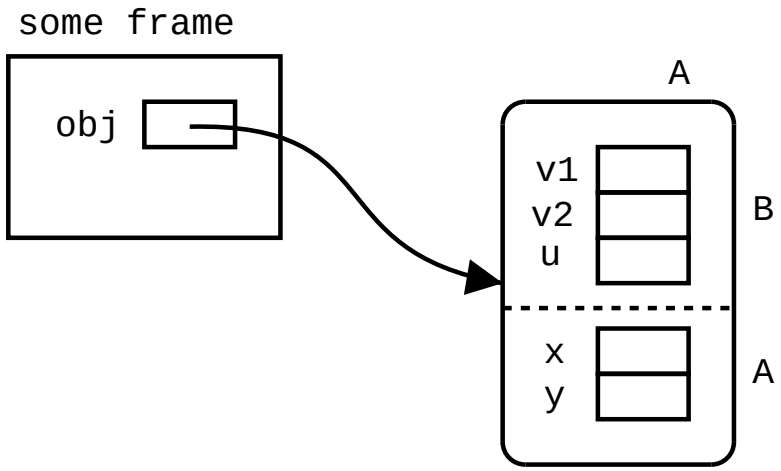
```
public class BankApplication {
  public static void main(String[] args)
  {
    LimitedSavingsAccount a1;
    CheckingAccount a2;
    a1 = new LimitedSavingsAccount(1000.0, 0.2, 20
    a2 = new CheckingAccount(300.0, 3.50);
    a1.withdraw(400.0);
    a1.apply_interest();
    a1.deposit(200.0);
    a2.deduct_fee();
    a2.withdraw(400.0);
  }
}
```

Inheritance

```
class C { ... }
class D { ... }
class E { ... }
class B {
  C v1, v2;
  D u;
  void m() { ... }
}
class A extends B {
  E x;
  C y;
  void p() { ... }
  void s() { ... }
}
```

Inheritance

```
// In some client
A obj = new A();
obj.p();
obj.m();
// We can refer to ... obj.x ... obj.y ...
// ... obj.u ... obj.v1 ... obj.v2 ...
```



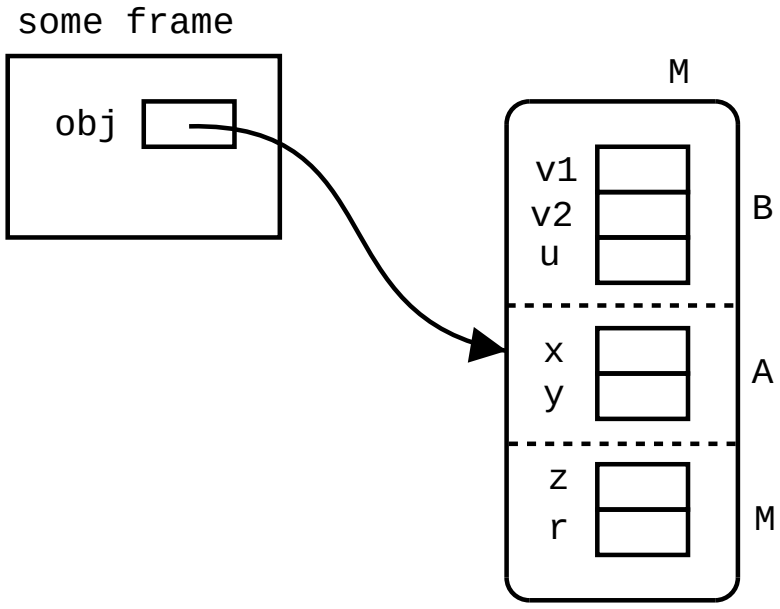
Inheritance

- A method in a subclass can access the attributes and methods of its super class.

```
class C { ... }
class D { ... }
class E { ... }
class B {
  C v1, v2;
  D u;
  void m() { ... v1 ... v2 ... u ... m() ... }
}
class A extends B {
  E x;
  C y;
  void p()
  {
    ... x ... y ... p() ... v1 ...
    ... v2 ... u ... m() ...
  }
  void s() { ... }
}
```

Inheritance

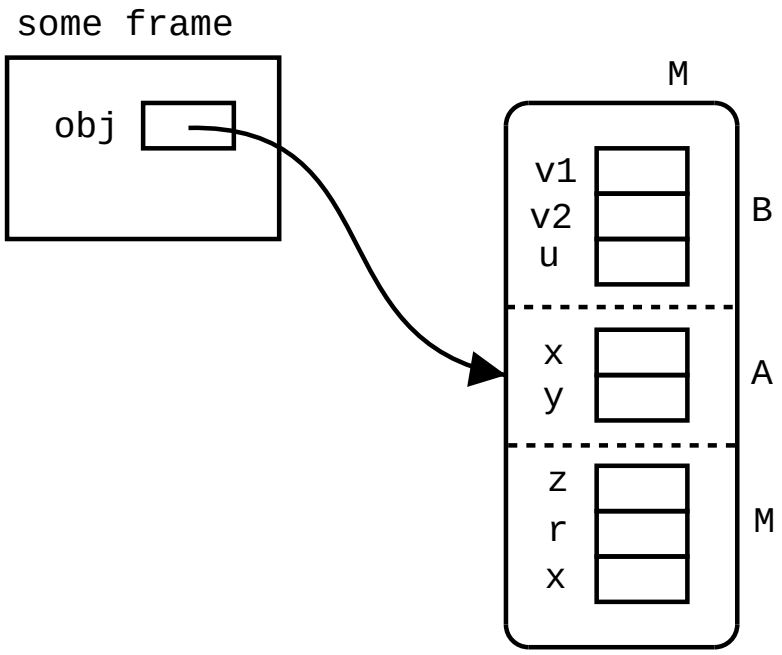
```
class M extends A {
  E z;
  D r;
  void q() { ... }
}
// Somewhere else
M obj2 = new M();
```



Shadowing variables

- An attribute or instance variable can be redefined in a subclass. In this case we say that the variable in the subclass *shadows* the variable in the parent class.

```
class M extends A {
  E z;
  D r, x;
  void q() { ... }
}
```



Accessing variables from the super class

- The `super` reference is used to access an attribute or method in a parent class.

```
class M extends A {
  E z;
  D r, x;
  void q()
  {
    ... this.x ... super.x ...
  }
}
```

Overriding methods

- A method can be redefined in a subclass. This is called *overriding* the method.

```
class M extends A {
  E z;
  D r, x;
  void q()
  {
    ... this.x ... super.x ...
  }
  void p()
  {
    ...
  }
}
```

Inheritance

- A method in a subclass can access the attributes and methods of its super class.

```
class C { ... }
class D { ... }
class E { ... }
class B {
  C v1, v2;
  D u;
  void m() { ... v1 ... v2 ... u ... m() ... }
}
class A extends B {
  E x;
  C y;
  void p()
  {
    ... x ... y ... p() ... v1 ...
    ... v2 ... u ... m() ...
  }
  void s() { ... }
}
```

Accessing a method or attribute

- When we try to access a method or attribute of an object, it is looked up by the Java runtime system in the class of the object first. If it is not found there, it is looked up in the parent class. If it is not found there, it is looked up in the grand-parent, etc...

```
M obj3 = new M();
obj3.q();  // From class M
obj3.m();  // From class B
obj3.p();  // From class M
obj3.s();  // From class A
```

- Attributes and methods declared as `private` cannot be accessed directly by the subclasses, even though they are present in the object. They can be accessed only indirectly by public accessor methods in the class that declared them as private.

Accessing a method or attribute

```
class A extends B {
  private E x, y;
  void p() { }
  void s() { }
  public E get_x() { return x; }
}
class M extends A {
  E z;
  D r, x;
  void q()
  {
    ... this.x ...
    // instead of super.x ...
    ... get_x() ... or ... super.get_x() ...
  }
}
```

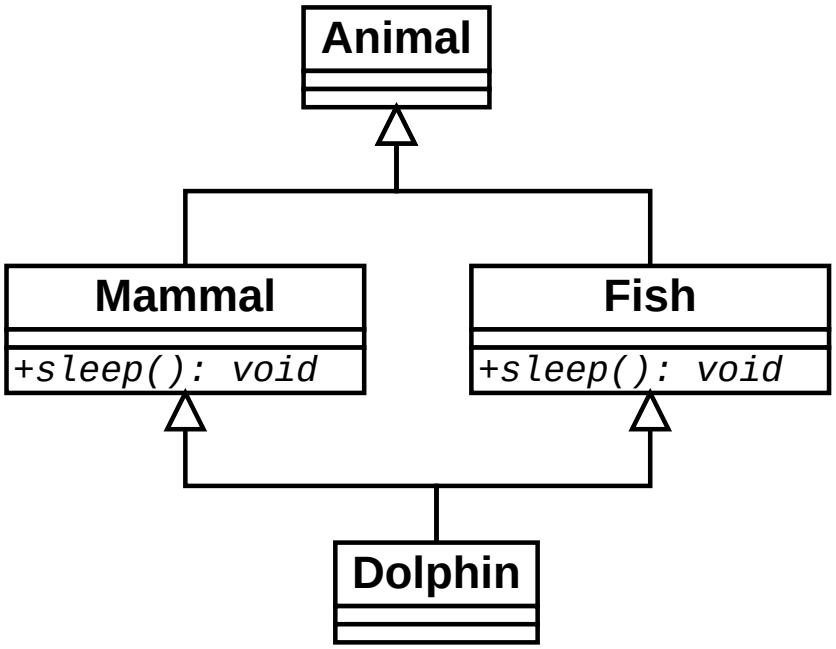
Accessing a method or attribute

- An attribute or method declared as `protected` can be accessed by any subclass, even if it is in a different package.
- An attribute or method declared as `final`, is not inherited at all, i.e. it forbids overriding.
- A class declared as `final`, cannot have subclasses.

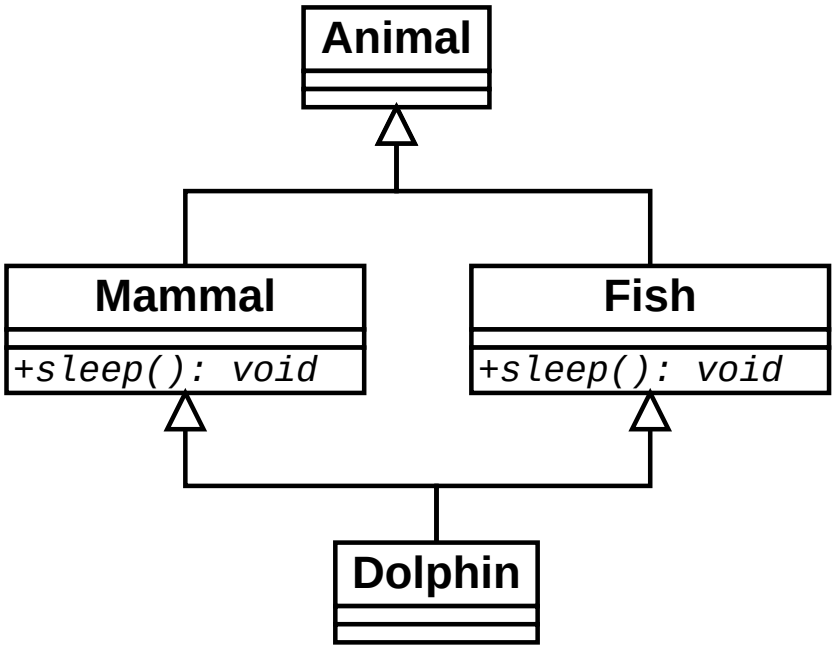
Multiple inheritance

- Multiple inheritance: a class with more than one superclass

Multiple inheritance



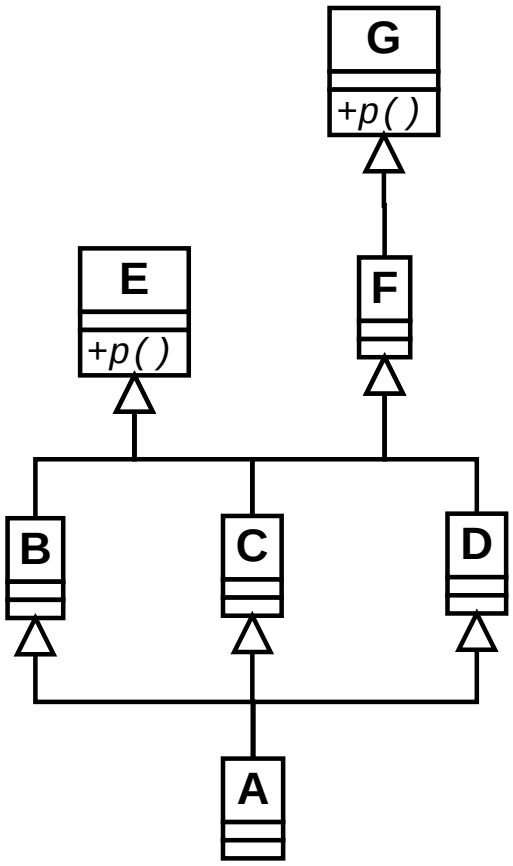
Multiple inheritance




```
class A extends B, C { ... } // Error
```

- Java does not support multiple inheritance

Multiple inheritance



The end