Lecture 5, 22 January 2026
Java: abstract classes, interfaces
Storage allocation

Madhavan Mukund, S P Suresh

Programming Language Concepts
January–April 2026

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

simula
event queue

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`

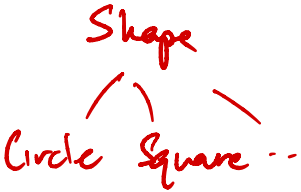
    ```
    public abstract double perimeter();
    ```

- 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
  ...
}
```

← Dynamic dispatch

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])
  }
}
```

any subtype of Comparable

Generic functions . . .

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

Generic functions . . . — "Structural polymorphism"

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

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

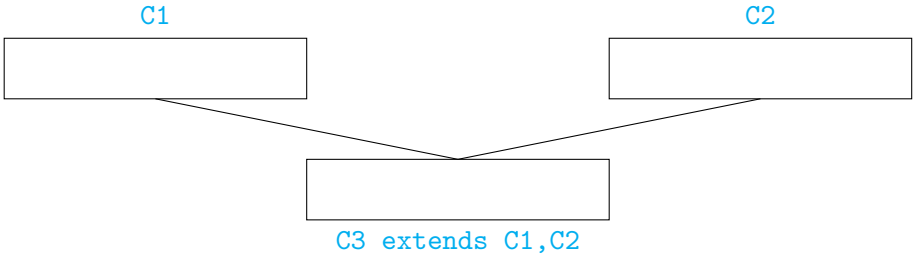
```
public class Myclass extends Comparable{
  private double size;   // quantity used for comparison

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

Multiple inheritance

- Can we sort `Circle` objects using the generic functions in `SortFunctions`?
  - `Circle` already extends `Shape`
  - Need `Circle` to also extend `Comparable`

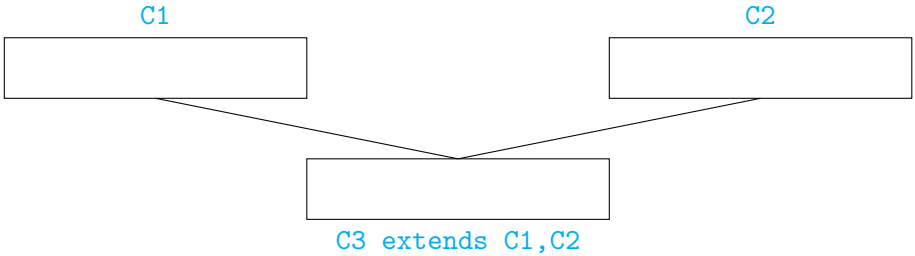
Multiple inheritance

- Can we sort `Circle` objects using the generic functions in `SortFunctions`?
  - `Circle` already extends `Shape`
  - Need `Circle` to also extend `Comparable`
- Can a subclass extend multiple parent classes?



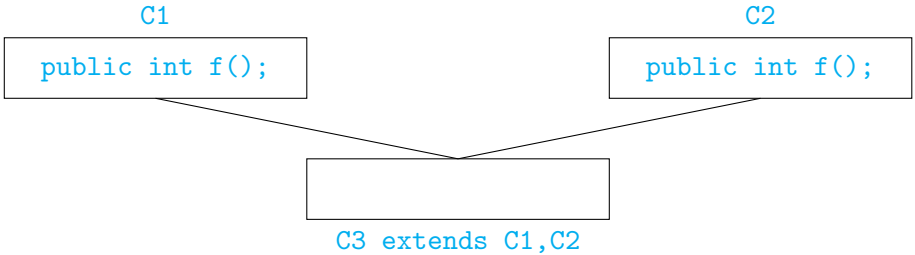
Multiple inheritance

- Can a subclass extend multiple parent classes?



Multiple inheritance

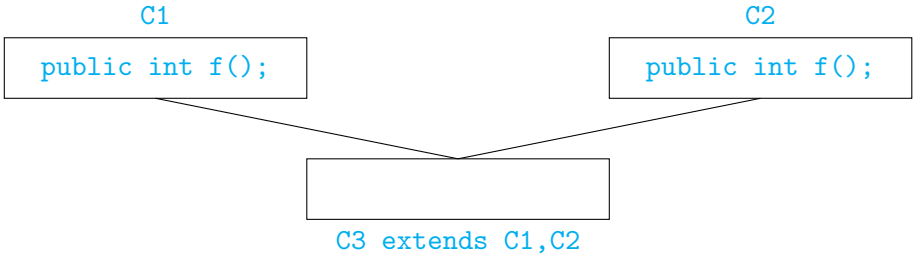
- Can a subclass extend multiple parent classes?



- If `f()` is not overridden, which `f()` do we use in `C3`?

Multiple inheritance

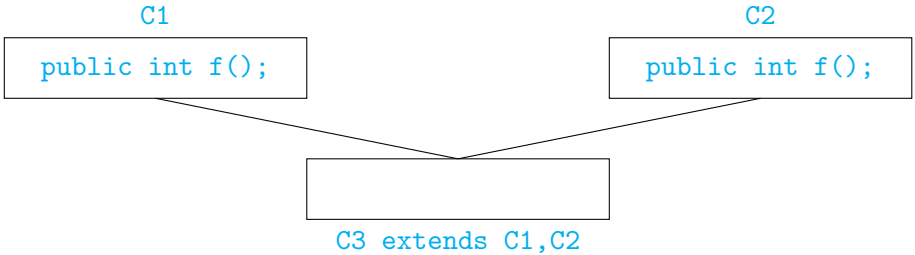
- Can a subclass extend multiple parent classes?



- 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

Interfaces and Multiple inheritance

- An interface is an abstract class with no concrete components

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

Interfaces and Multiple inheritance

- 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 and Multiple inheritance

- 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 and Multiple inheritance

- 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 describe relevant aspects of a class
  - Abstract functions describe a specific "slice" of capabilities
  - Another class only needs to know about these capabilities

But ...

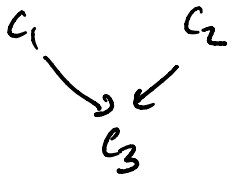
Java now allows concrete fns in interfaces

class A extends B implements C, D, E {

Java class hierarchy

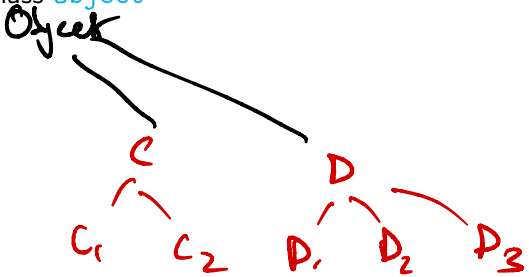
- No multiple inheritance — tree-like

Forest of classes



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 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 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 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+"");`
  - 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);
}
```

Object.equals()

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);
}
```

objarr[i].equals(o)

- 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`

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));
}
```

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)`!

2 fns

- 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);
}
```

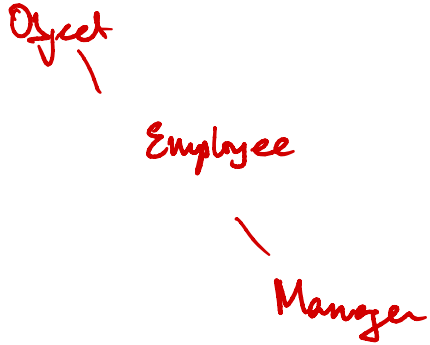
  - Note the run-time type check and the cast

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()`

Subtyping vs inheritance

- Consider the following example
  - `queue`, with methods `insert-rear`, `delete-front`
  - `stack`, with methods `insert-front`, `delete-front`
  - `deque`, with methods `insert-front`, `delete-front`, `insert-rear`, `delete-rear`

Subtyping vs inheritance

- Consider the following example
  - `queue`, with methods `insert-rear`, `delete-front`
  - `stack`, with methods `insert-front`, `delete-front`
  - `deque`, with methods `insert-front`, `delete-front`, `insert-rear`, `delete-rear`
- What are the subtype and inheritance relationships between these classes?

Subtyping vs inheritance

- Consider the following example
  - `queue`, with methods `insert-rear`, `delete-front`
  - `stack`, with methods `insert-front`, `delete-front`
  - `deque`, with methods `insert-front`, `delete-front`, `insert-rear`, `delete-rear`
- What are the subtype and inheritance relationships between these classes?
- Subtyping
  - `deque` has more functionality than `queue` or `stack`
  - `deque` is a subtype of both these types

Subtyping vs inheritance

- Consider the following example
  - `queue`, with methods `insert-rear`, `delete-front`
  - `stack`, with methods `insert-front`, `delete-front`
  - `deque`, with methods `insert-front`, `delete-front`, `insert-rear`, `delete-rear`
- What are the subtype and inheritance relationships between these classes?
- Subtyping
  - `deque` has more functionality than `queue` or `stack`
  - `deque` is a subtype of both these types
- Inheritance
  - Can suppress two functions in a `deque` and use it as a `queue` or `stack`
  - Both `queue` and `stack` inherit from `deque`

Subclasses, subtyping and inheritance

- Class hierarchy represents both 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

Variables, functions and storage

- Variables represent data residing in a memory location
- Compiler creates a map from variables to memory addresses

Variables, functions and storage

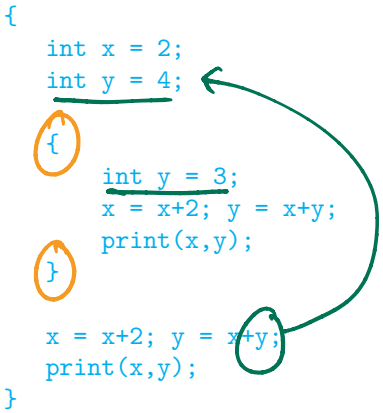
- Variables represent data residing in a memory location
- Compiler creates a map from variables to memory addresses
- Functions represent blocks of (reusable) code
  - Complexities introduced by recursion
  - Many versions of the same local variable active at the same time
  - Need a way to keep track of all copies of a local `x`
  - Figure out which copy of `x` is referred to at any point of the execution

Variables, functions and storage

- Variables represent data residing in a memory location
- Compiler creates a map from variables to memory addresses
- Functions represent blocks of (reusable) code
  - Complexities introduced by recursion
  - Many versions of the same local variable active at the same time
  - Need a way to keep track of all copies of a local `x`
  - Figure out which copy of `x` is referred to at any point of the execution
- Scope and lifetime of variables

Scope

- Consider the following program block



for (int i = 0; i < n; i++){

--

}

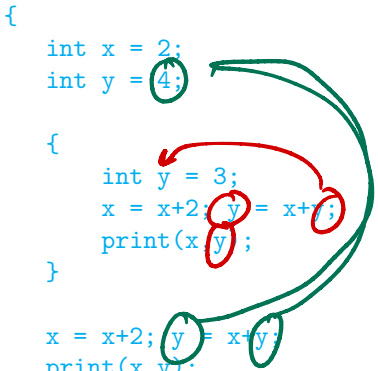
Scope

- Consider the following program block

```
{
    int x = 2;
    int y = 4;

    {
        int y = 3;                  Outer y is hidden.
        x = x+2; y = x+y;           Updated y value is not propagated outside
        print(x,y);                 4, 7
    }

    x = x+2; y = x+y;
    print(x,y);
}
```

Scope

- Consider the following program block



Outer y is hidden.
Updated y value is not propagated outside
4, 7

Outer y value and updated x value
6, 10

Scope and Lifetime

- Scope – Region of text in which a declaration is visible

Scope and Lifetime

- Scope – Region of text in which a declaration is visible
- Lifetime – Duration, at run-time, that a memory location is allocated for a specific declaration

Scope and Lifetime

- Scope – Region of text in which a declaration is visible
- Lifetime – Duration, at run-time, that a memory location is allocated for a specific declaration
- Consider the example below

```
{ int x = ...;
    { int y = ...;
        { int x = ...;
            ...
        }
    }
}
```

Scope and Lifetime

- Scope – Region of text in which a declaration is visible
- Lifetime – Duration, at run-time, that a memory location is allocated for a specific declaration
- Consider the example below

```
{ int x = ...;
  { int y = ...;
    { int x = ...;
      ...
    }
  }
}
```

- Scope of outer `x` is the two outer blocks

Scope and Lifetime

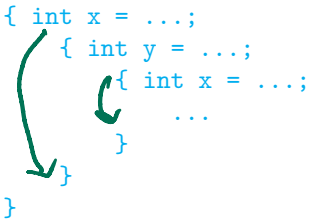
- Scope – Region of text in which a declaration is visible
- Lifetime – Duration, at run-time, that a memory location is allocated for a specific declaration
- Consider the example below

```
{ int x = ...;
    { int y = ...;
        { int x = ...;
            ...
        }
    }
}
```

- Scope of outer `x` is the two outer blocks
- Scope of the inner `x` is the innermost block

Scope and Lifetime

- Scope – Region of text in which a declaration is visible
- Lifetime – Duration, at run-time, that a memory location is allocated for a specific declaration
- Consider the example below



- Scope of outer `x` is the two outer blocks
- Scope of the inner `x` is the innermost block
- Lifetime of inner `x` is the time during which innermost block is active

Scope and Lifetime

- Scope – Region of text in which a declaration is visible
- Lifetime – Duration, at run-time, that a memory location is allocated for a specific declaration
- Consider the example below

```
{ int x = ...;
    { int y = ...;
        { int x = ...;
            ...
        }
    }
}
```

- Scope of outer `x` is the two outer blocks
- Scope of the inner `x` is the innermost block
- Lifetime of inner `x` is the time during which innermost block is active
- Lifetime of outer `x` is the time during which outermost block is active (includes the lifetime of inner `x`)

`static` variables

- `static` variables are associated with a class as a whole
- Do not require instantiation of objects

static variables

- `static` variables are associated with a class as a whole
- Do not require instantiation of objects

```
public class A {
    static int howManyAs = 0;
    int id;
    public A(int id) {
        howManyAs += 1;
        this.id = id;
    }
}
```

`static` variables

- `static` variables are associated with a class as a whole
- Do not require instantiation of objects

```
public class A {
    static int howManyAs = 0;
    int id;
    public A(int id) {
        howManyAs += 1;
        this.id = id;
    }
}
```

- The `static` variable `howManyAs` counts the number of instances of `A` created

`static` variables

- `static` variables are associated with a class as a whole
- Do not require instantiation of objects

```
public class A {
    static int howManyAs = 0;
    int id;
    public A(int id) {
        howManyAs += 1;
        this.id = id;
    }
}
```

- The `static` variable `howManyAs` counts the number of instances of `A` created
- Lifetime of `howManyAs` spans the execution of the entire program

`static` variables

- `static` variables are associated with a class as a whole
- Do not require instantiation of objects

```
public class A {
    static int howManyAs = 0;
    int id;
    public A(int id) {
        howManyAs += 1;
        this.id = id;
    }
}
```

- The `static` variable `howManyAs` counts the number of instances of `A` created
- Lifetime of `howManyAs` spans the execution of the entire program
- Scope of `howManyAs` is limited to the class `A`

Activation Record

- For local variables and function parameters, we need to store one copy for each function invocation (or activation)

Activation Record

- For local variables and function parameters, we need to store one copy for each function invocation (or activation)
- Activation record — collection of all data related to a function invocation

Activation Record

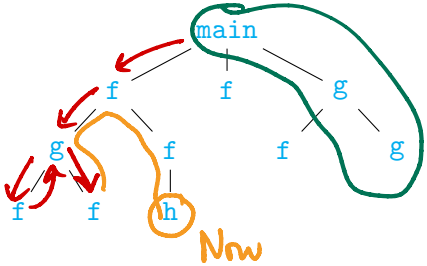
- For local variables and function parameters, we need to store one copy for each function invocation (or activation)
- Activation record — collection of all data related to a function invocation
- Includes space for local variables, parameters, intermediate results, and some pointers

Call graph

- A call graph helps us visualize the function calls during a program execution

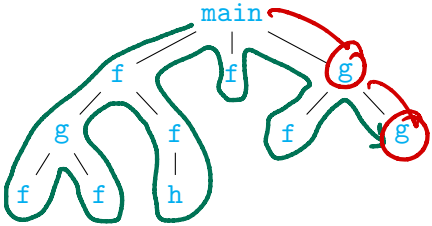
Call graph

- A call graph helps us visualize the function calls during a program execution



Call graph

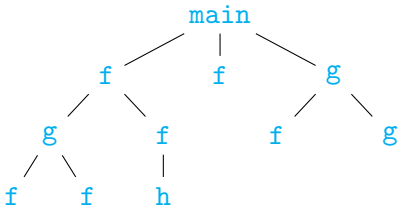
- A call graph helps us visualize the function calls during a program execution



- The set of active function calls at any point of time lies on the path from the root to the right most leaf

Call graph

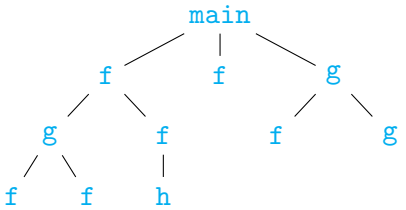
- A call graph helps us visualize the function calls during a program execution



- The set of active function calls at any point of time lies on the path from the root to the right most leaf
- If `f` calls `g`, then `g` is completed before `f`

Call graph

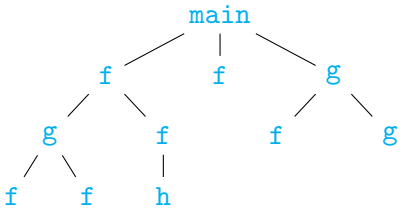
- A call graph helps us visualize the function calls during a program execution



- The set of active function calls at any point of time lies on the path from the root to the right most leaf
- If `f` calls `g`, then `g` is completed before `f`
- Store the activation records on a stack

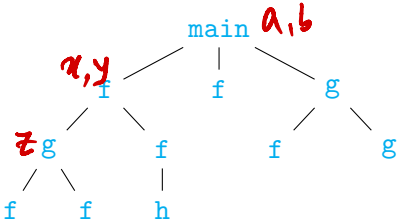
Call graph

- A call graph helps us visualize the function calls during a program execution



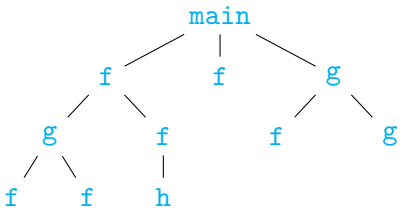
- The set of active function calls at any point of time lies on the path from the root to the right most leaf
- If `f` calls `g`, then `g` is completed before `f`
- Store the activation records on a stack
- Activation record is also called a stack frame

Activation records on stack



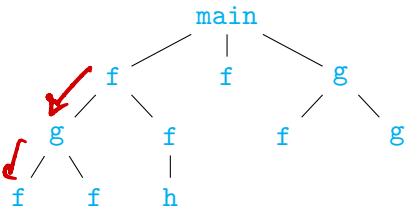
- Assume that `main` has local variables `a` and `b`, `f` has `x` and `y`, and `g` has `z`

Activation records on stack



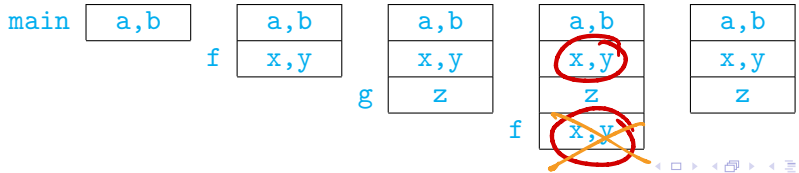
- Assume that `main` has local variables `a` and `b`, `f` has `x` and `y`, and `g` has `z`
- Place activation records on a stack — grows and shrinks as a program executes

Activation records on stack



- Assume that `main` has local variables `a` and `b`, `f` has `x` and `y`, and `g` has `z`
- Place activation records on a stack — grows and shrinks as a program executes

- The stack evolves as follows:



General layout of a program in memory