CS49K Programming Languages

Chapter 10: Data Abstraction and Object-Orientation

LECTURE 13: CLASSES AND OBJECTS

DR. ERIC CHOU

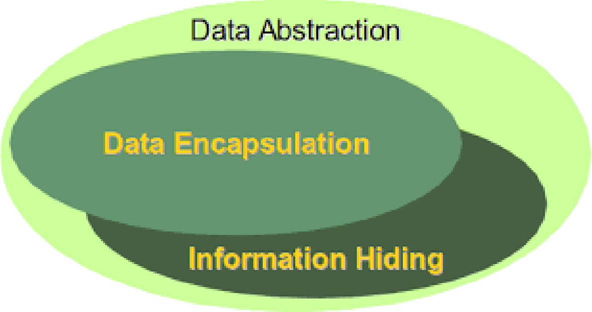
IEEE SENIOR MEMBER

Objectives

- The principles of Object-Oriented Programming
- Class Design
- Derived Class
- Class to Class Relationship
- Encapsulation
- Inheritance
- Instantiation/Finalization

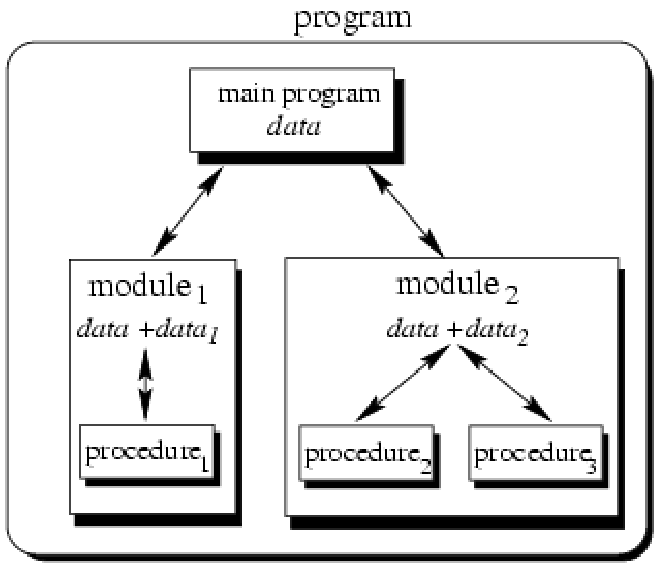
Overview of Object-Oriented Programming

SECTION 1

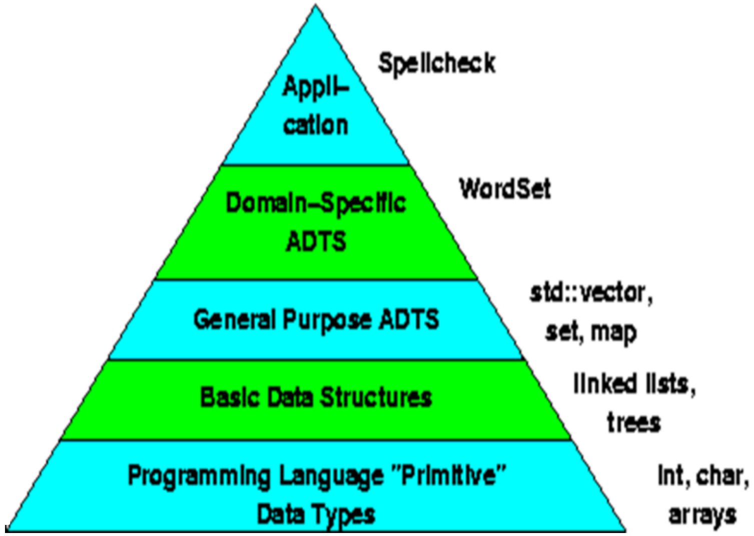


Abstraction

Abstraction (from the Latin ***abs***, meaning ***away*** from and ***trahere***, meaning to ***draw***) is the process of taking away or removing characteristics from something in order to reduce it to a set of essential characteristics.



Data Abstraction in C Language



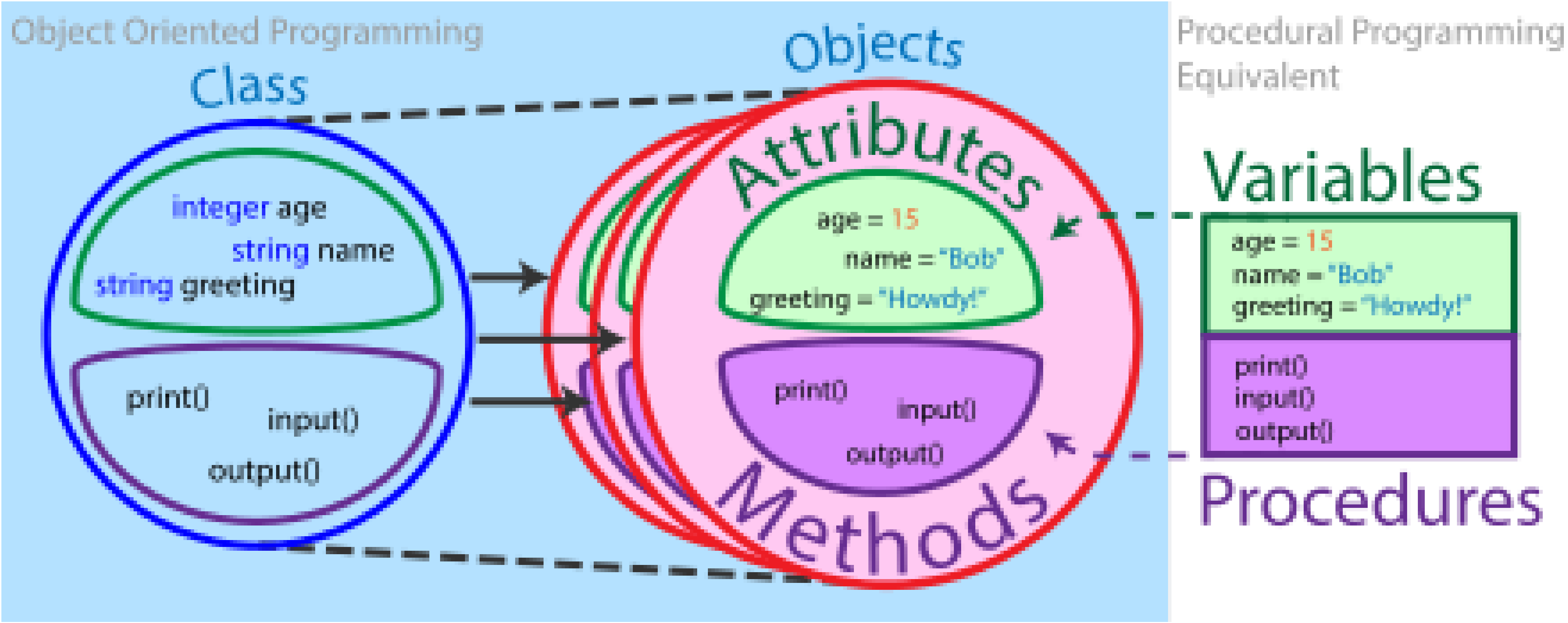
Object-Oriented Programming

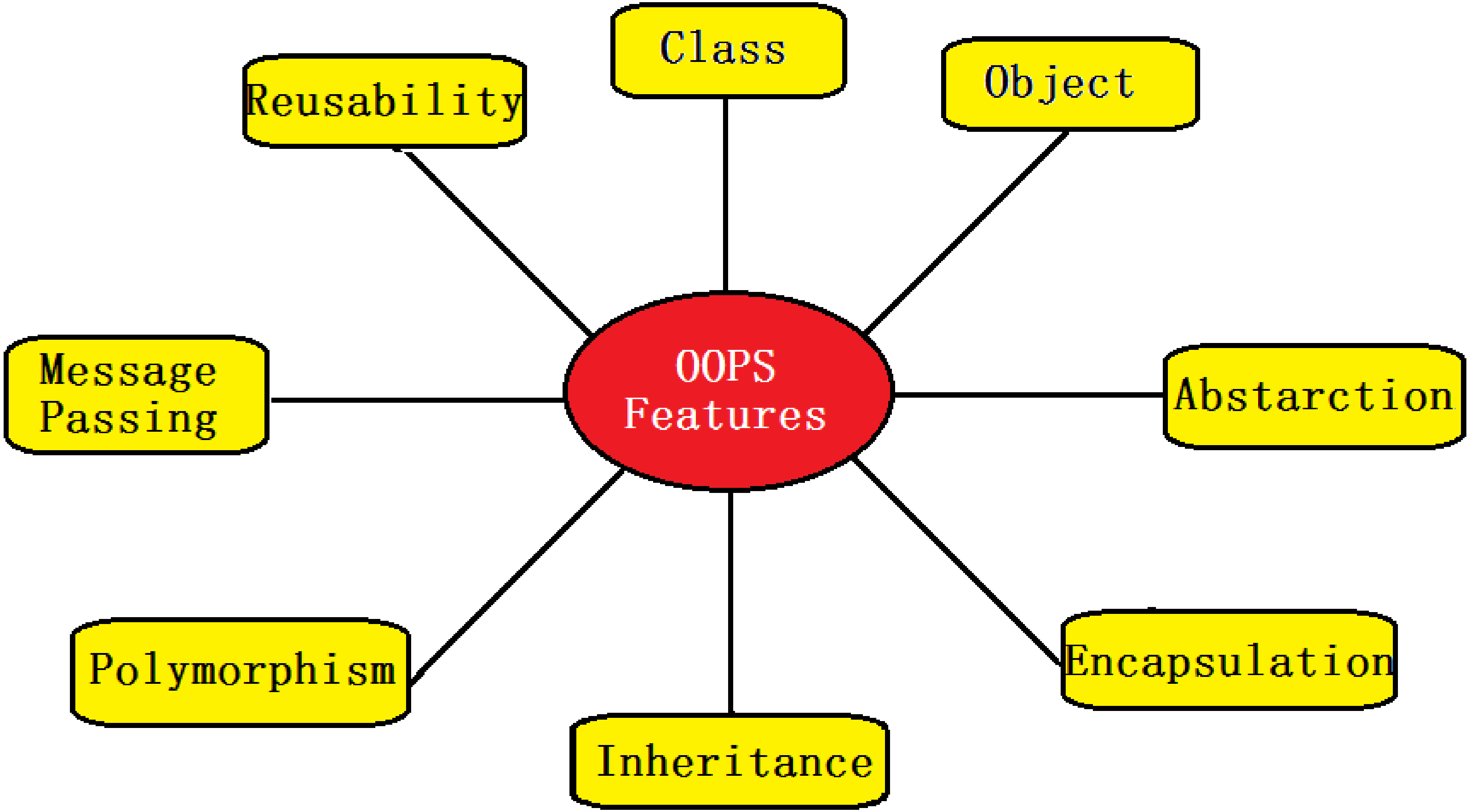
- Control or **PROCESS** abstraction is a very old idea (subroutines!), though few languages provide it in a truly general form (Scheme comes close)
- Data abstraction is somewhat newer, though its roots can be found in Simula67
  - An Abstract Data Type is one that is defined in terms of the operations that it supports (i.e., that can be performed upon it) rather than in terms of its structure or implementation

Object-Oriented Programming

- Why abstractions?
  - easier to think about - hide what doesn't matter to programmer
  - protection - prevent access to things you shouldn't see
  - plug compatibility
    - replacement of pieces, often without recompilation, definitely without rewriting libraries
    - division of labor in software projects

Object-Oriented Programming





OOPS Features

Class

Object

Abstarction

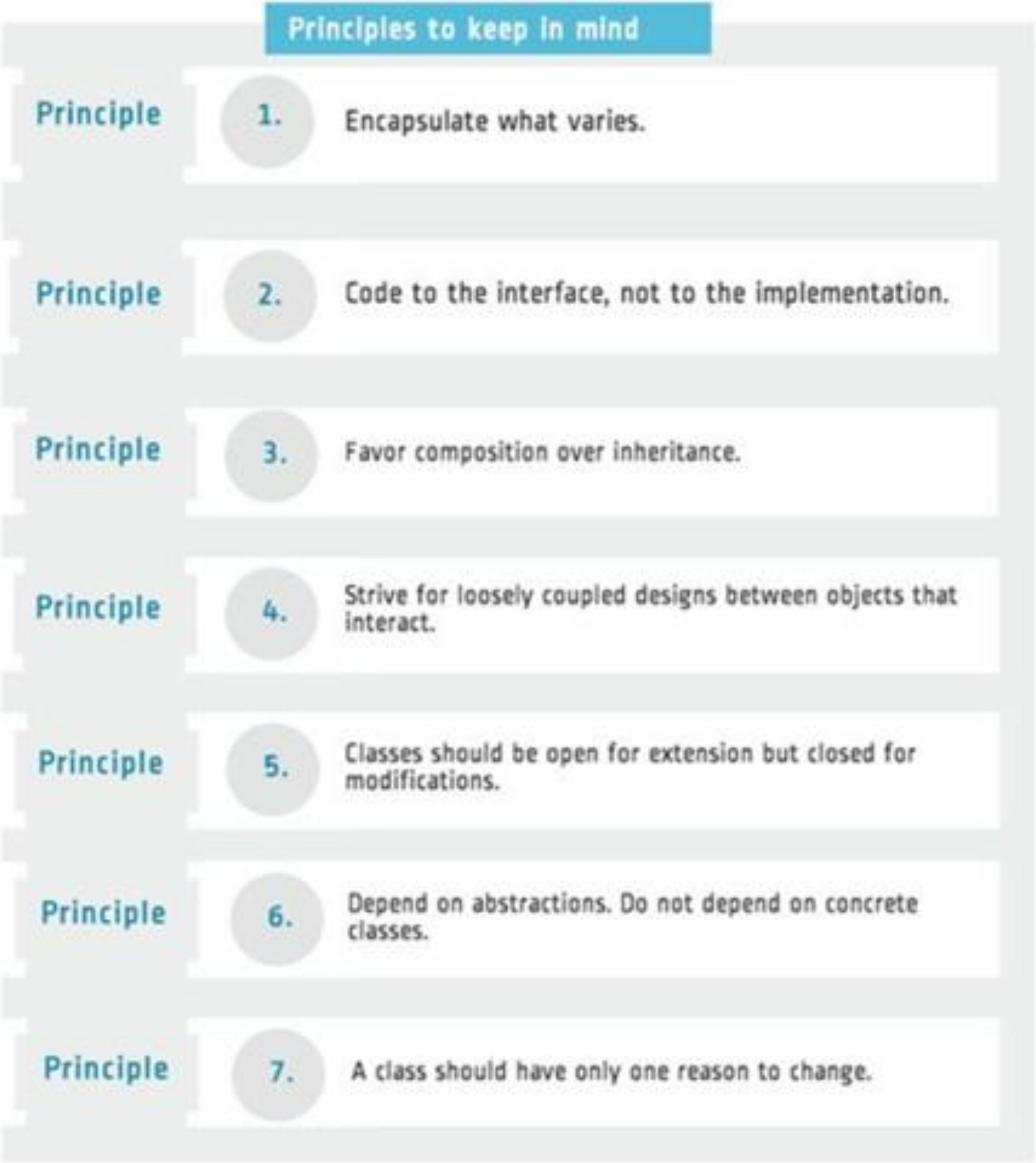
Encapsulation

Inheritance

Polymorphism

Message Passing

Reusability



Principles to keep in mind

Principle 1. Encapsulate what varies.

Principle 2. Code to the interface, not to the implementation.

Principle 3. Favor composition over inheritance.

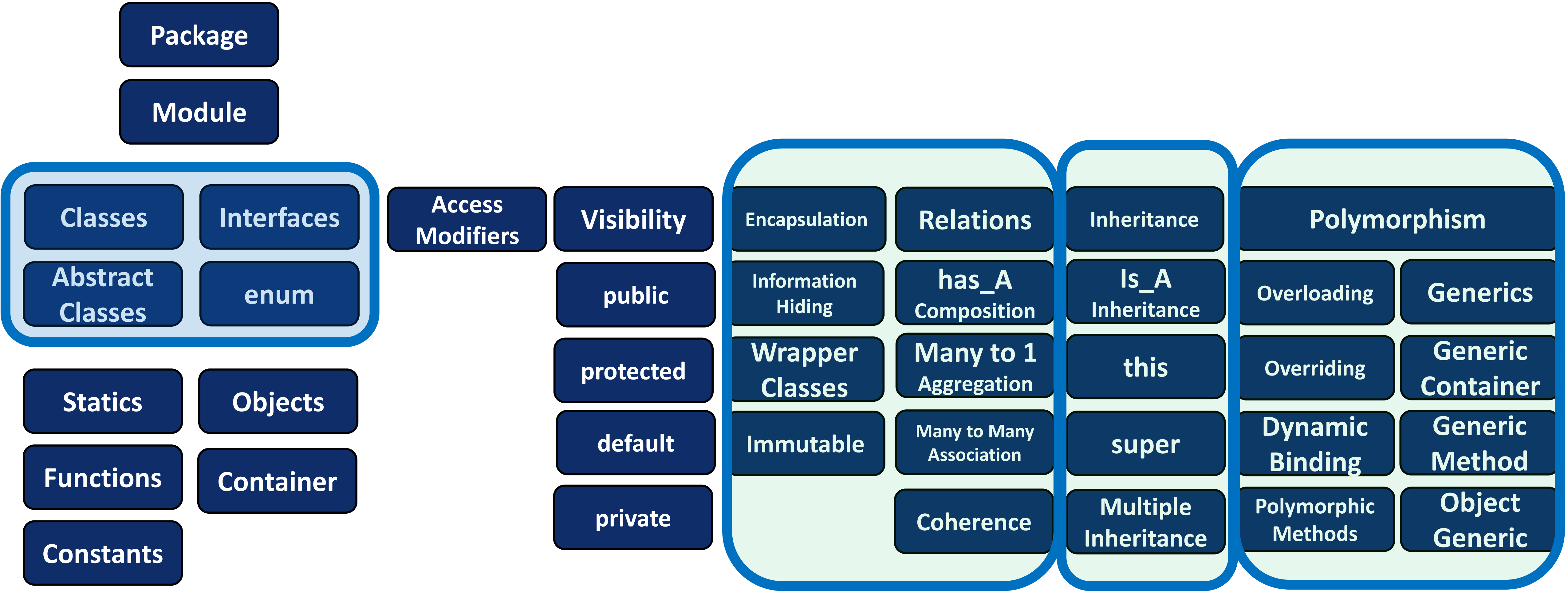
Principle 4. Strive for loosely coupled designs between objects that interact.

Principle 5. Classes should be open for extension but closed for modifications.

Principle 6. Depend on abstractions. Do not depend on concrete classes.

Principle 7. A class should have only one reason to change.

Object-Oriented Programming



Package

Module

Classes

Interfaces

Abstract Classes

enum

Statics

Objects

Functions

Container

Constants

Access Modifiers

Visibility

public

protected

default

private

Encapsulation

Information Hiding

Wrapper Classes

Immutable

Relations

has_A Composition

Many to 1 Aggregation

Many to Many Association

Coherence

Inheritance

Is_A Inheritance

this

super

Multiple Inheritance

Polymorphism

Overloading

Generics

Overriding

Generic Container

Dynamic Binding

Generic Method

Polymorphic Methods

Object Generic

Object-Oriented Programming

- We talked about data abstraction some back in the unit on naming and scoping (Ch. 3)
- Recall that we traced the historical development of abstraction mechanisms

• Static set of variables	**Basic**
• Locals	**Fortran**
• Statics	**Fortran, Algol 60, C**
• Modules	**Modula-2, Ada 83**
• Module types	**Euclid**
• Objects	**Smalltalk, C++, Eiffel, Java** **Oberon, Modula-3, Ada 95**

Object-Oriented Programming

- **Statics** allow a subroutine to retain values from one invocation to the next, while hiding the name in-between
- **Modules** allow a collection of subroutines to share some statics, still with hiding
  - If you want to build an abstract data type, though, you have to make the module a manager

Object-Oriented Programming

- **Module types** allow the module to **be** the abstract data type - you can declare a bunch of them
  - This is generally more intuitive
    - It avoids explicit object parameters to many operations
    - One minor drawback: If you have an operation that needs to look at the innards of two different types, you'd define both types in the same manager module in Modula-2
    - In C++ you need to make one of the classes (or some of its members) "friends" of the other class

Object-Oriented Programming

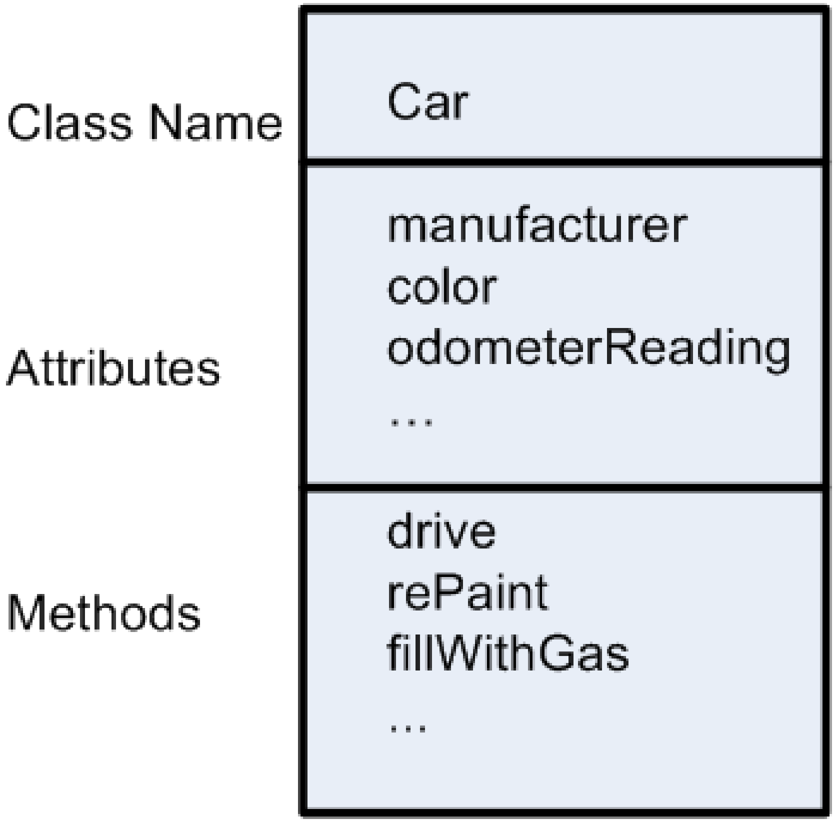
- Objects add inheritance and dynamic method binding
- Simula 67 introduced these, but didn't have data hiding
- The three key factors in OO programming
  - **Encapsulation (data hiding)**
  - **Inheritance**
  - **Polymorphism (Dynamic method binding)**

Classes and Objects I

Basic Definition

SECTION 2

Class



Class Name

Car

Attributes

manufacturer
color
odometerReading
...

Methods

drive
rePaint
fillWithGas
...

Members (Inherited)

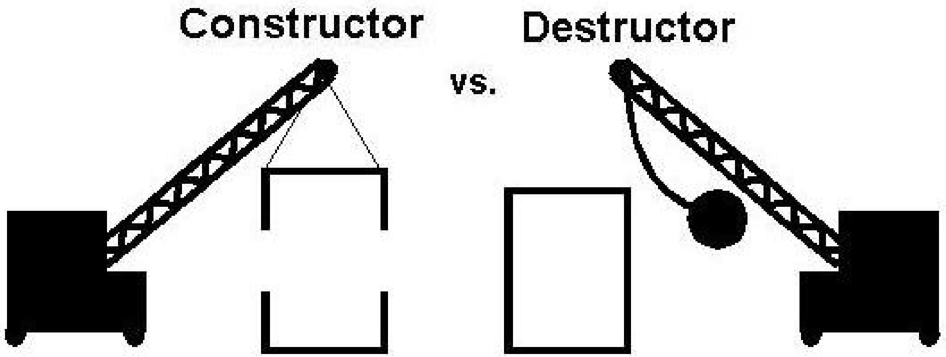
Data Members
Data Fields

Member Methods
Methods
Subroutines

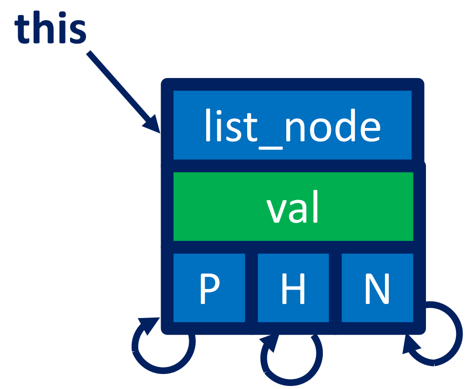
Non-Members (Not Inherited)

Constructors
A::A(){}

Destructors
A::~A(){}



A list_node After Constuction



Constructor:

list_node();

Destructor:

~list_node();

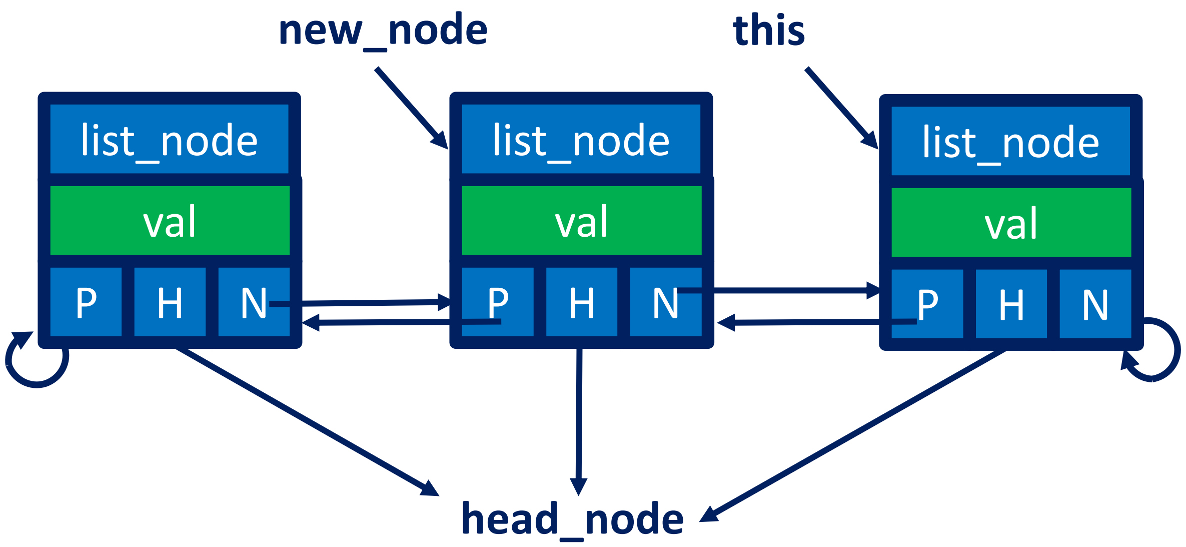
Pointer:

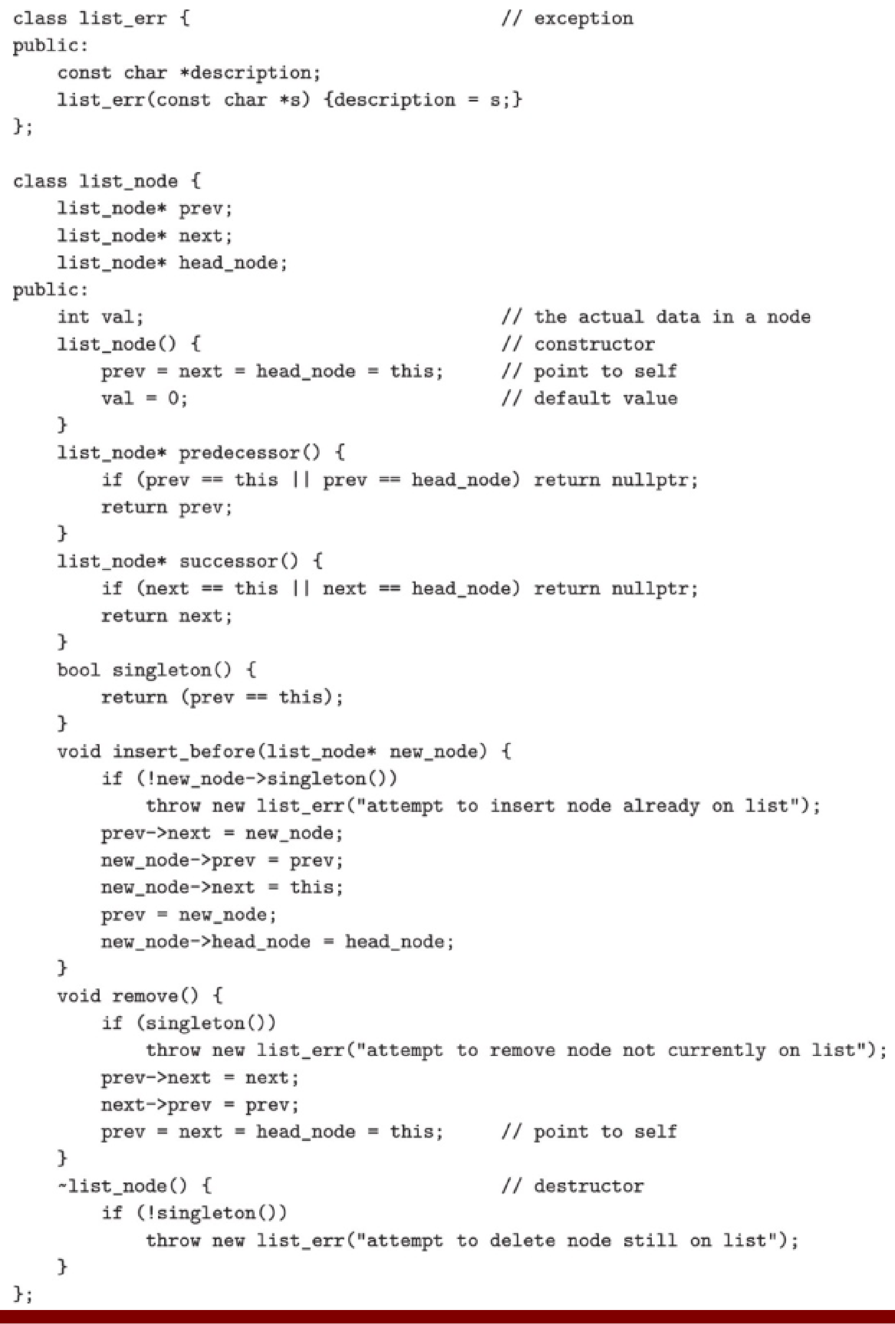
list_node* elem;

Reference:

list_node &ref;

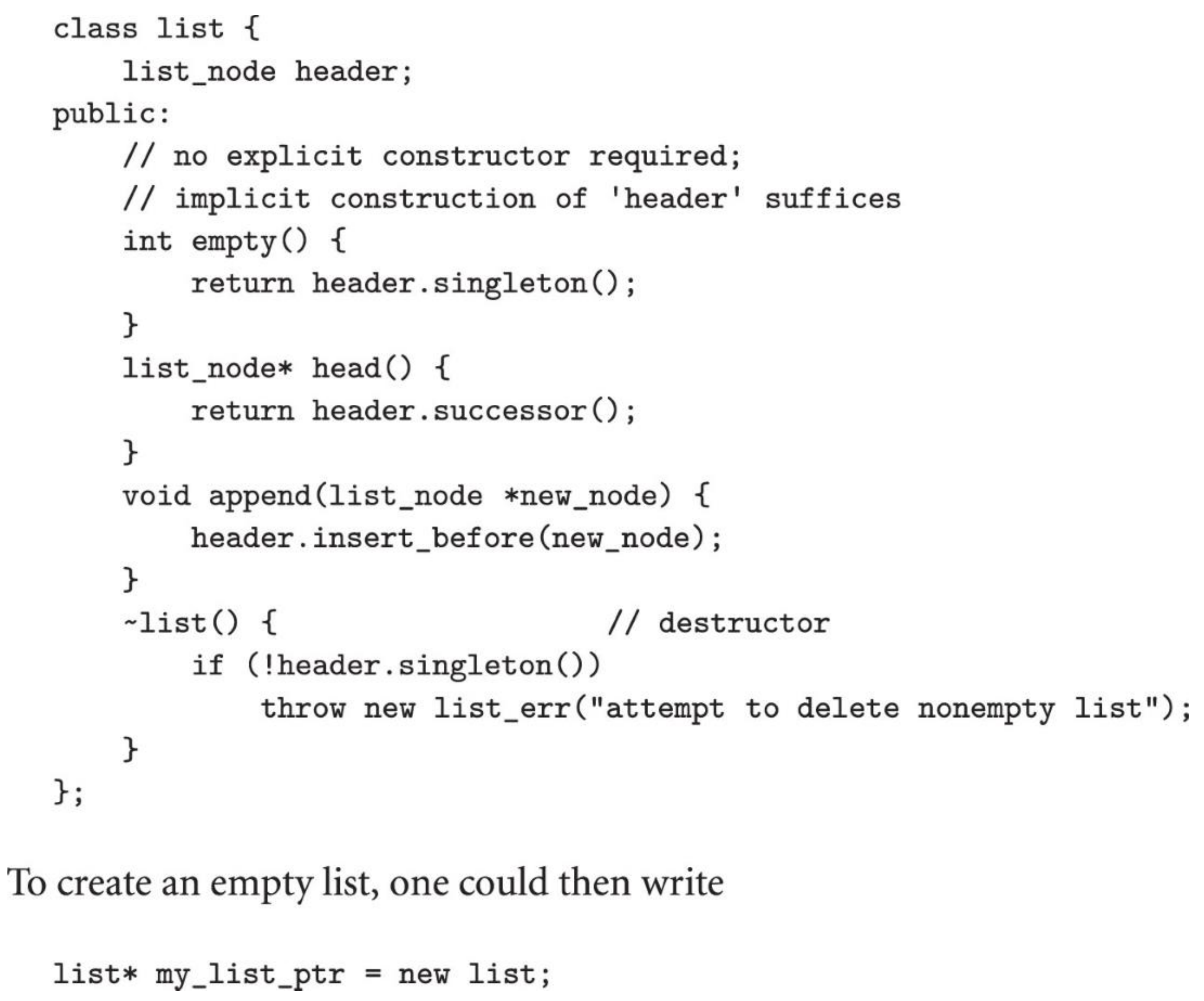
Insert Before:





```
class list_err {                              // exception
public:
    const char *description;
    list_err(const char *s) {description = s;}
};

class list_node {
    list_node* prev;
    list_node* next;
    list_node* head_node;
public:
    int val;                                  // the actual data in a node
    list_node() {                             // constructor
        prev = next = head_node = this;       // point to self
        val = 0;                              // default value
    }
    list_node* predecessor() {
        if (prev == this || prev == head_node) return nullptr;
        return prev;
    }
    list_node* successor() {
        if (next == this || next == head_node) return nullptr;
        return next;
    }
    bool singleton() {
        return (prev == this);
    }
    void insert_before(list_node* new_node) {
        if (!new_node->singleton())
            throw new list_err("attempt to insert node already on list");
        prev->next = new_node;
        new_node->prev = prev;
        new_node->next = this;
        prev = new_node;
        new_node->head_node = head_node;
    }
    void remove() {
        if (singleton())
            throw new list_err("attempt to remove node not currently on list");
        prev->next = next;
        next->prev = prev;
        prev = next = head_node = this;       // point to self
    }
    ~list_node() {                            // destructor
        if (!singleton())
            throw new list_err("attempt to delete node still on list");
    }
};
```

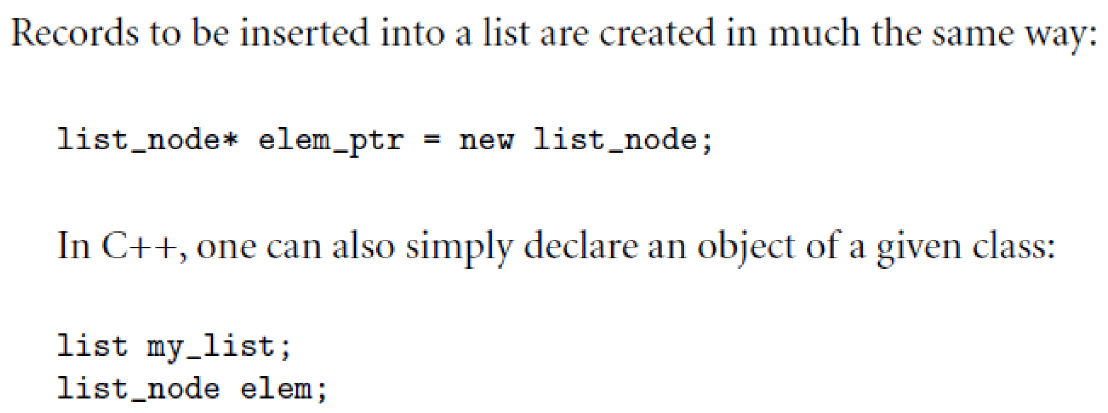
List Class that Used list_node



```
class list {
    list_node header;
public:
    // no explicit constructor required;
    // implicit construction of 'header' suffices
    int empty() {
        return header.singleton();
    }
    list_node* head() {
        return header.successor();
    }
    void append(list_node *new_node) {
        header.insert_before(new_node);
    }
    ~list() {                    // destructor
        if (!header.singleton())
            throw new list_err("attempt to delete nonempty list");
    }
};
```

To create an empty list, one could then write

```
list* my_list_ptr = new list;
```

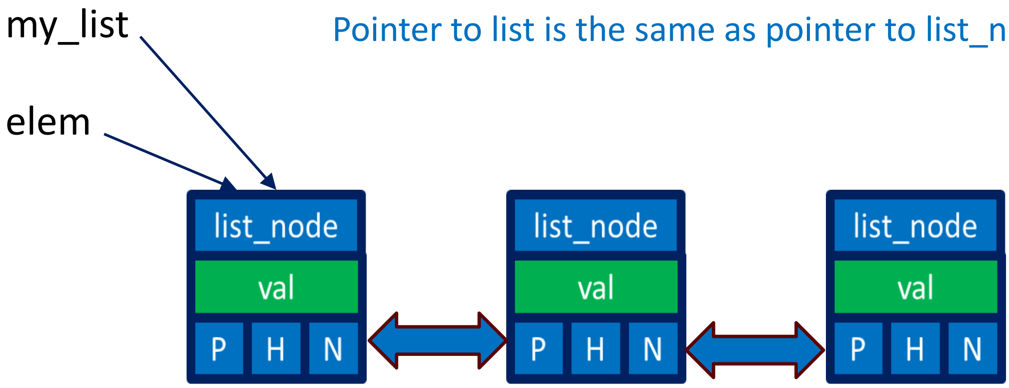


Records to be inserted into a list are created in much the same way:

```
list_node* elem_ptr = new list_node;
```

In C++, one can also simply declare an object of a given class:

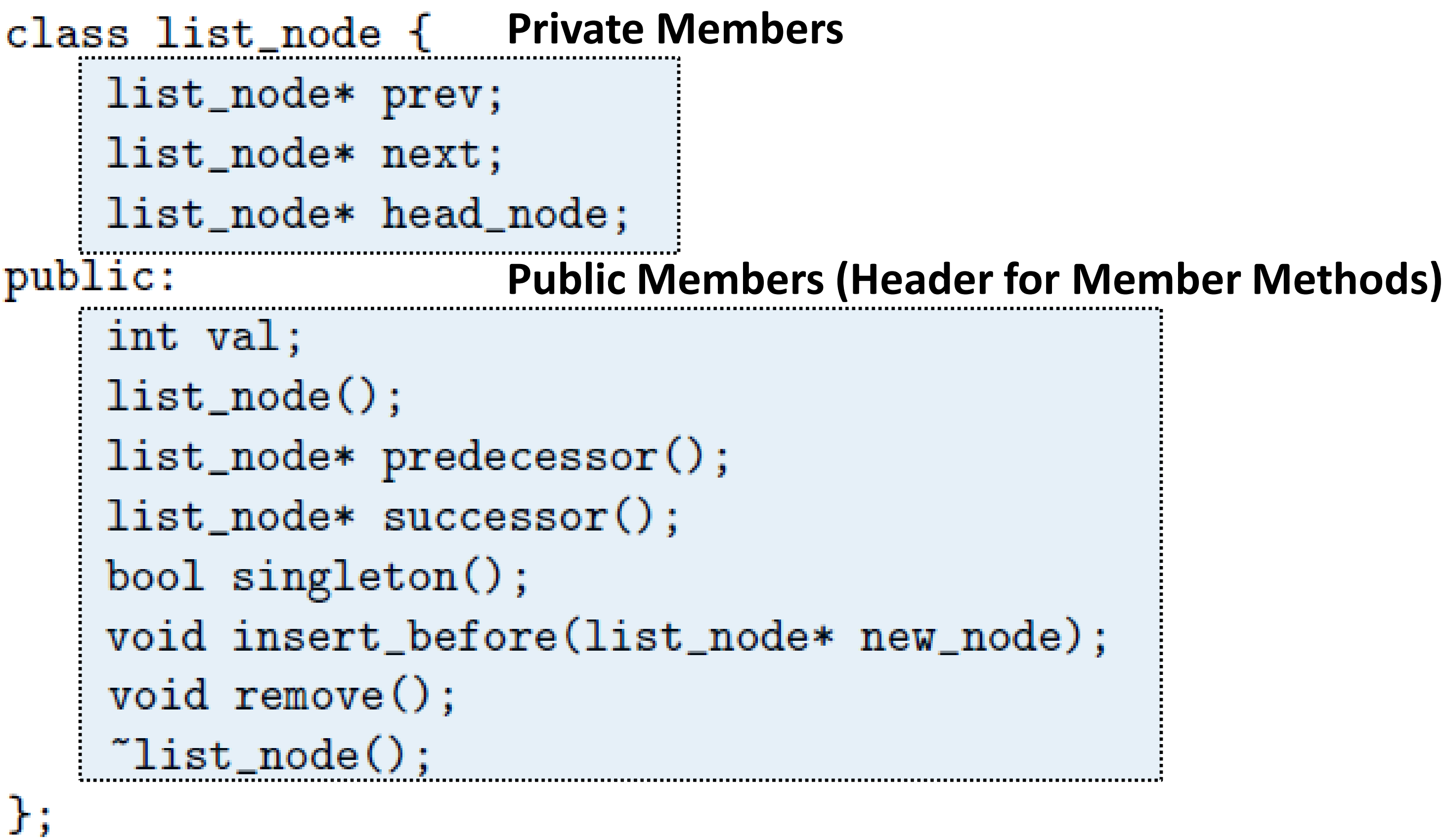
```
list my_list;
list_node elem;
```



Note:
Pointer to list is the same as pointer to list_node.

Public and Private Member

Visibility [.h file]



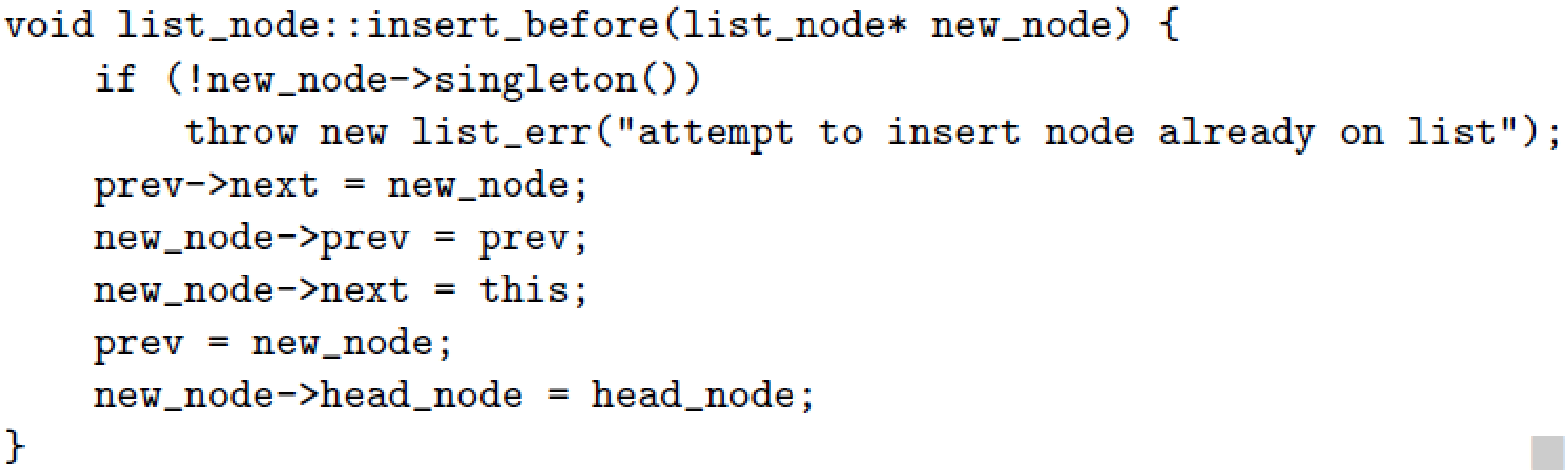
Private Members

Public Members (Header for Member Methods)

```
class list_node {
    list_node* prev;
    list_node* next;
    list_node* head_node;
public:
    int val;
    list_node();
    list_node* predecessor();
    list_node* successor();
    bool singleton();
    void insert_before(list_node* new_node);
    void remove();
    ~list_node();
};
```

Separate Method Definition

using a :: **scope** resolution operator [.cc file .cpp file]



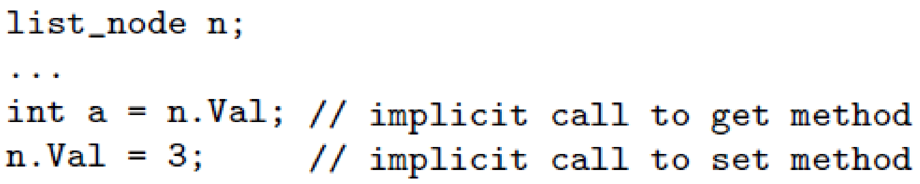
```
void list_node::insert_before(list_node* new_node) {
    if (!new_node->singleton())
        throw new list_err("attempt to insert node already on list");
    prev->next = new_node;
    new_node->prev = prev;
    new_node->next = this;
    prev = new_node;
    new_node->head_node = head_node;
}
```

Tiny Subroutines

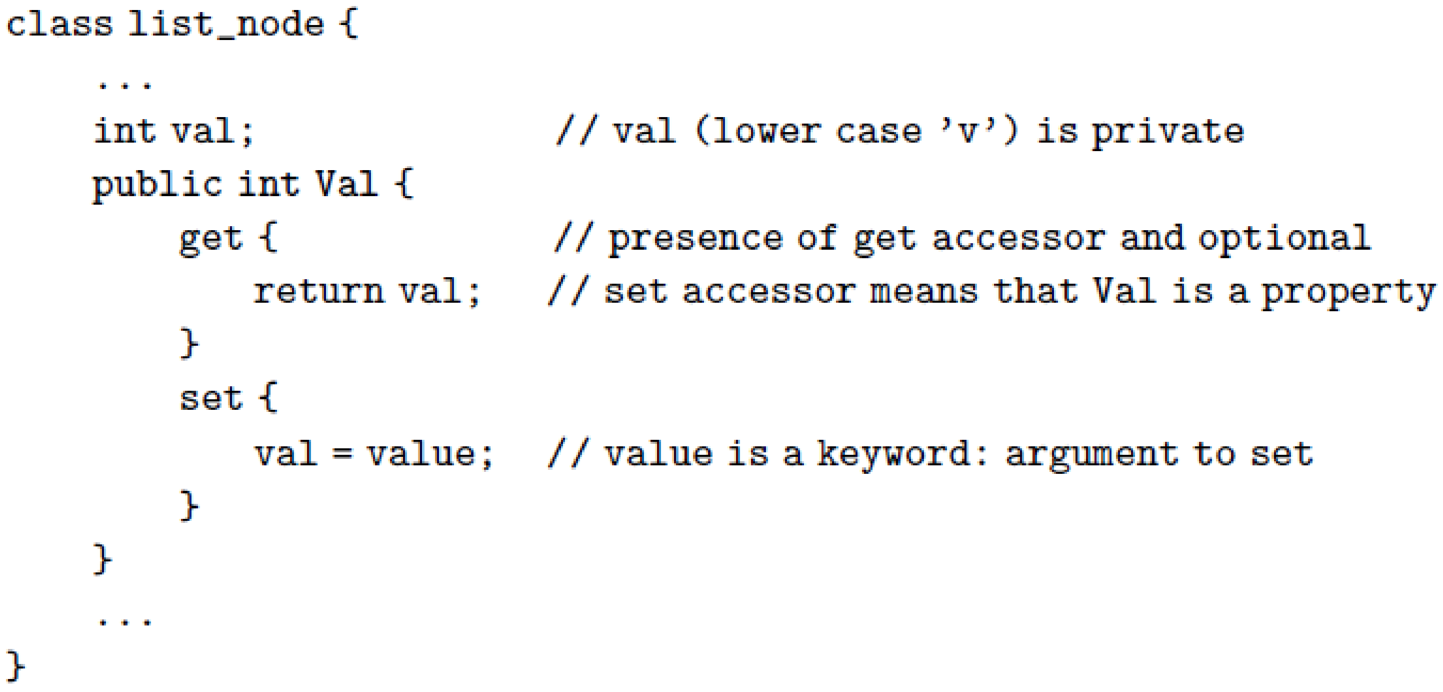
Accessor/Mutator

Property and indexer methods in C#:

C# provides a property mechanism specifically designed to facilitate the declaration of methods (called accessors) to “get” and “set” values.



```
list_node n;
...
int a = n.Val; // implicit call to get method
n.Val = 3;     // implicit call to set method
```



```
class list_node {
    ...
    int val;                 // val (lower case 'v') is private
    public int Val {
        get {                // presence of get accessor and optional
            return val;      // set accessor means that Val is a property
        }
        set {
            val = value;     // value is a keyword: argument to set
        }
    }
    ...
}
```

Classes and Objects II

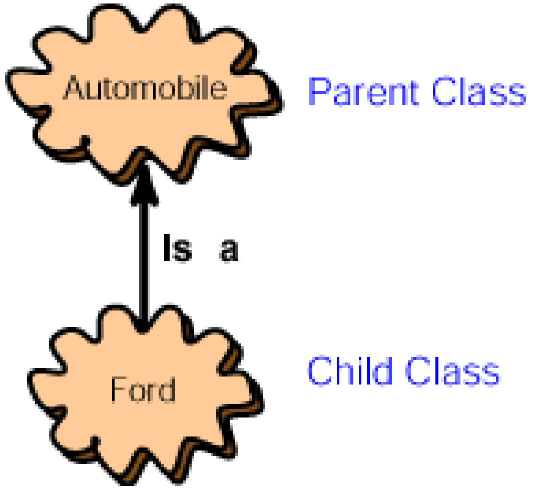
Derived Class

SECTION 3

Derived Classes

Sub-class (IS_A relationship, child class)

```
class queue : public list {                    // derive from list
public:
    // no specialized constructor or destructor required
    void enqueue(list_node* new_node) {
        append(new_node);
    }
    list_node* dequeue() {
        if (empty())
            throw new list_err("attempt to dequeue from empty queue");
        list_node* p = head();
        p->remove();
        return p;
    }
};
```



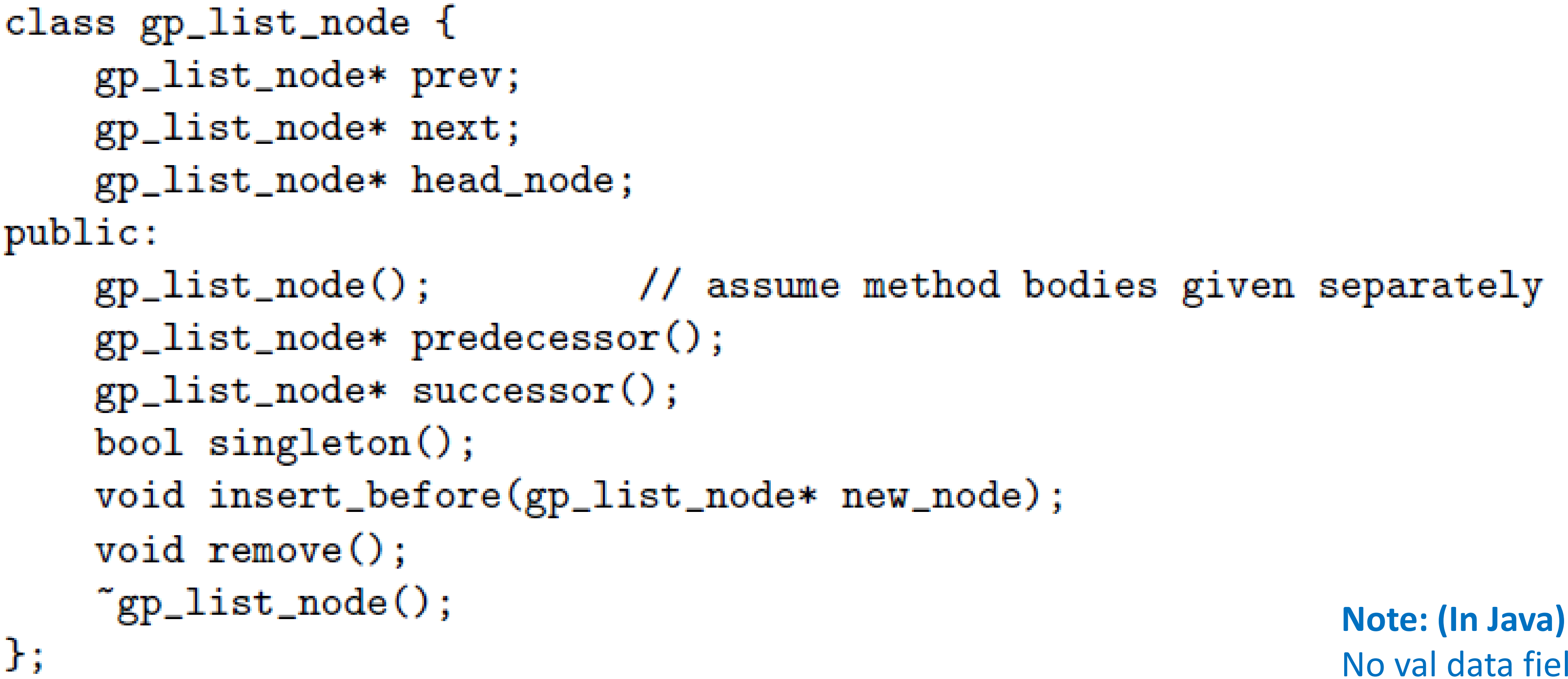
Note: (In Java)

class Queue extends List { .. }

- All member fields and member methods are inherited.

General-Purpose Base Classes

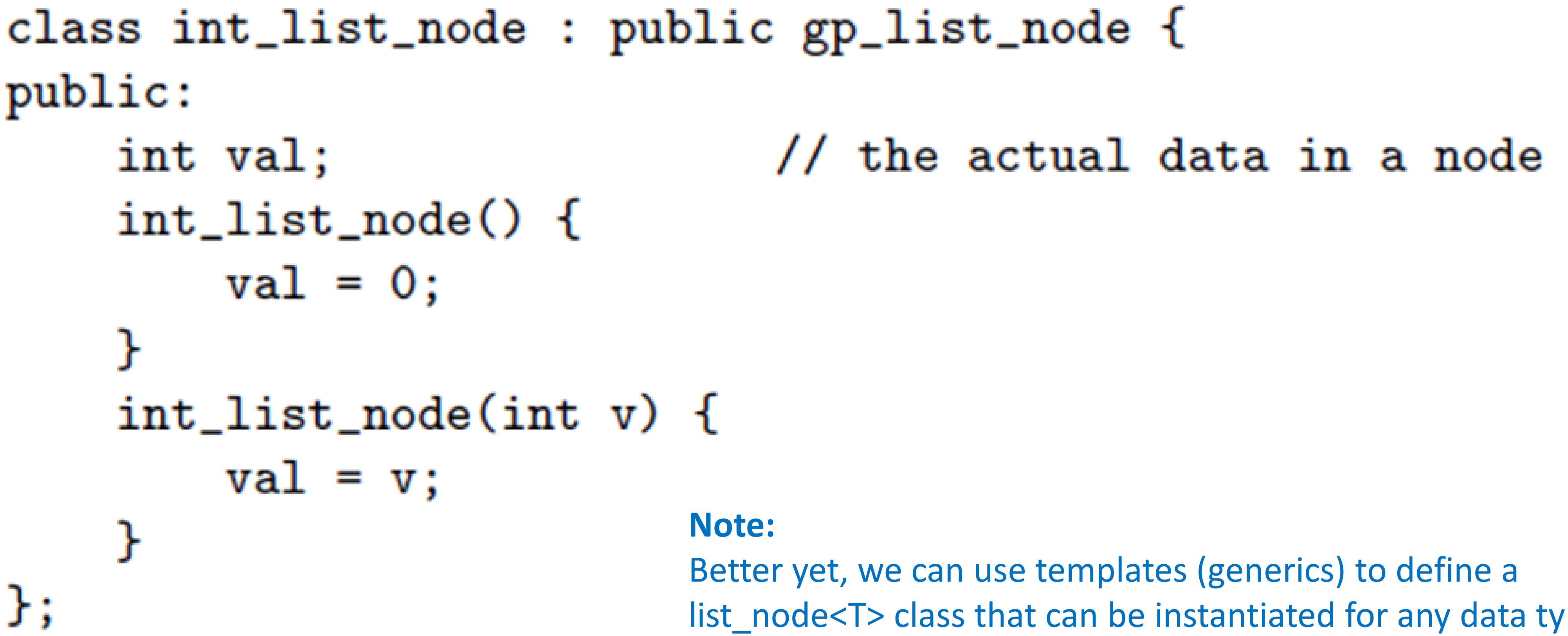
Higher Level Class (Less Contents)



```
class gp_list_node {
    gp_list_node* prev;
    gp_list_node* next;
    gp_list_node* head_node;
public:
    gp_list_node();            // assume method bodies given separately
    gp_list_node* predecessor();
    gp_list_node* successor();
    bool singleton();
    void insert_before(gp_list_node* new_node);
    void remove();
    ~gp_list_node();
};
```

Note: (In Java)
No val data field

Specific Sub-Class



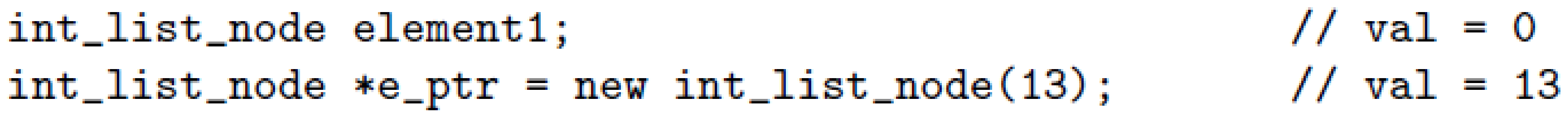
```
class int_list_node : public gp_list_node {
public:
    int val;                        // the actual data in a node
    int_list_node() {
        val = 0;
    }
    int_list_node(int v) {
        val = v;
    }
};
```

Note:
Better yet, we can use templates (generics) to define a list_node<T> class that can be instantiated for any data type T, without the need to use inheritance.

Overloaded Constructors

Constructors Taking Different Parameters

We have overloaded the constructor in **int_list_node**, providing two alternative implementations. One takes an argument, the other does not. Now the programmer can create **int_list_node**s with or without specifying an initial value:



```
int_list_node element1;                             // val = 0
int_list_node *e_ptr = new int_list_node(13);       // val = 13
```

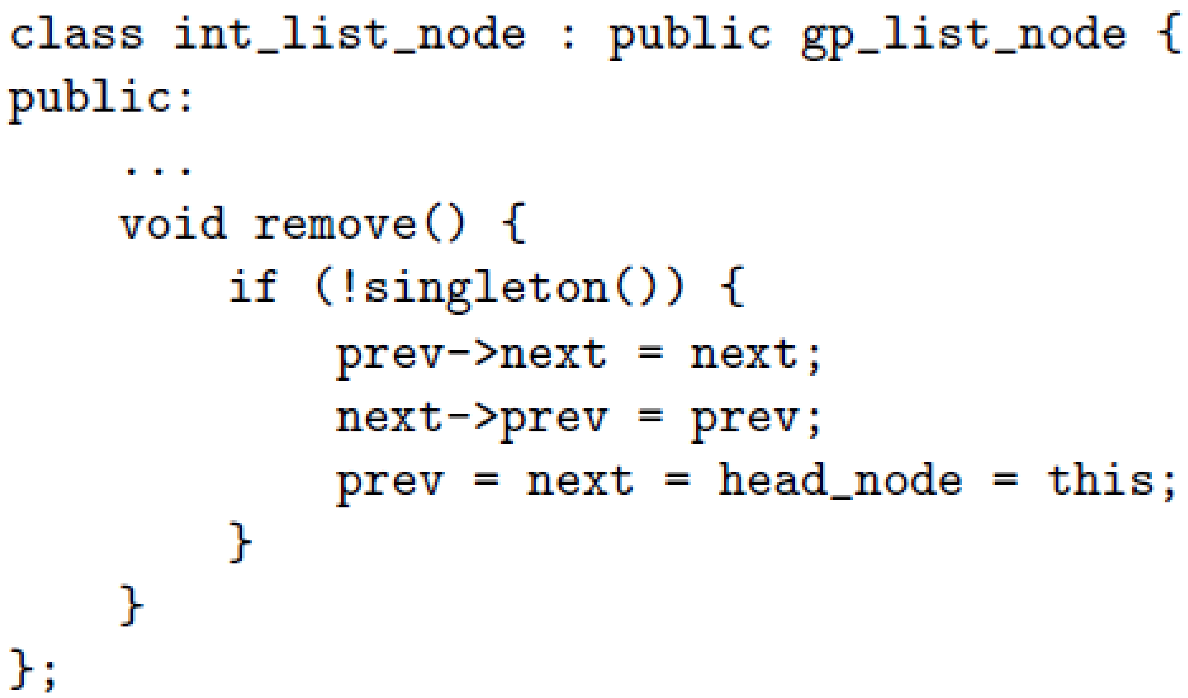
In C++, the compiler ensures that constructors for base classes are executed before those of derived classes. In our example, the constructor for **gp_list_node** will be executed first, followed by the constructor for **int_list_node**.

Note: default constructors

Overriding Methods

Modifying Base Class Methods

- To redefine a method of a base class, a derived class simply declares a new version.
- If written as in Figure 10.1, **gp_list_node::remove** will throw a **list_err** exception if the node to be removed is not currently on a list.
- If we want **int_list_node::remove** simply to return without doing anything in this situation, we can declare it that way explicitly:



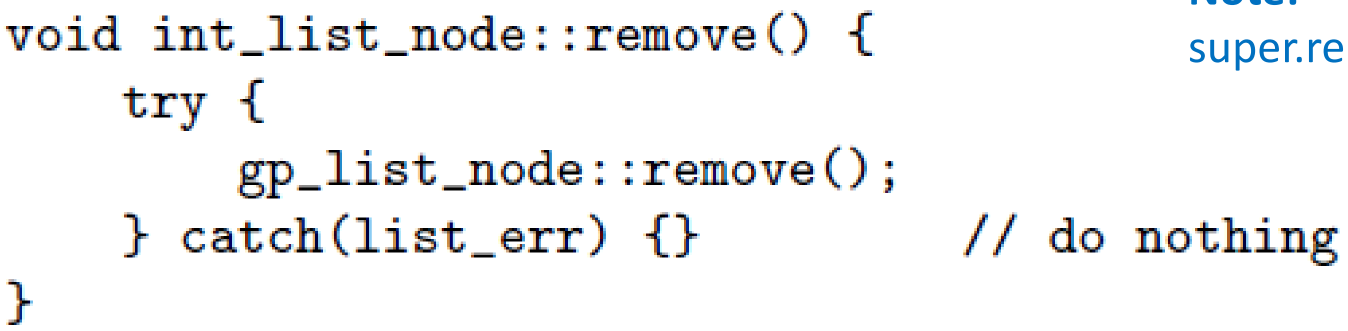
```
class int_list_node : public gp_list_node {
public:
    ...
    void remove() {
        if (!singleton()) {
            prev->next = next;
            next->prev = prev;
            prev = next = head_node = this;
        }
    }
};
```

Overriding Methods

Modifying Base Class Methods

A better approach is to leave the implementation details to the base class and simply catch the exception if it arises:

Note:
super.remove(); // Java



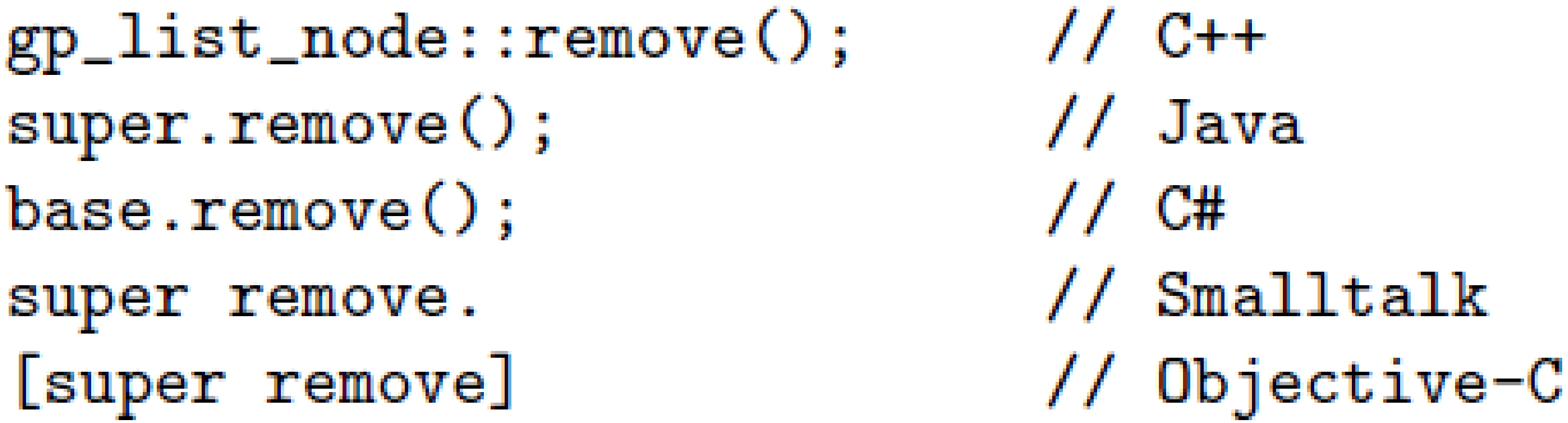
```
void int_list_node::remove() {
    try {
        gp_list_node::remove();
    } catch(list_err) {}          // do nothing
}
```

This version of the code may be slightly slower than the previous one, depending on how try blocks are implemented, but it does a better job of maintaining abstraction. This version will get updated if gp_list_mode's remove() is updated.

Note that the scope resolution operator (**::**) allows us to access the remove method of the base class explicitly, even though we have redefined it for **int_list_node**.

Accessing Base Class Members

Other object-oriented languages provide other means of accessing the members of a base class. In **Smalltalk**, **Objective-C**, **Java**, and **C#**, one uses the keyword base or super:



```
gp_list_node::remove();      // C++
super.remove();              // Java
base.remove();               // C#
super remove.                // Smalltalk
[super remove]               // Objective-C
```

Hiding Members of a Base Class

Arrange Member Visibility

- In C++, public members of a base class are always visible inside the methods of a derived class.
- They are visible to users of the derived class only if the base class name is preceded with the keyword public in the first line of the derived class's declaration.

- If we want to keep base list nodes hidden, we must prevent the user from accessing the head and append methods of class list. We can do so by making list a private base class instead:

```
class queue : private list { …
```

- To make the empty method visible again, we can call it out explicitly:

```
public;
        using list::empty;
```

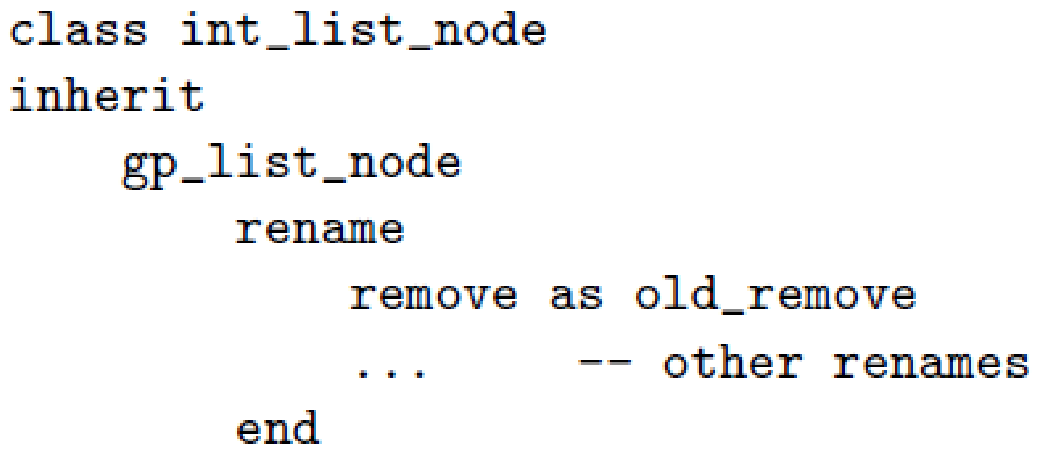
Visibility in C++

```
class Base {
    public: // public members go here
    protected: // protected members go here
    private: // private members go here
};
```

- A **public** member is accessible from anywhere outside the class but within a program.
- A **protected** member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes.
- A default (no specifier) member is accessible from anywhere in the same package.
- A **private** member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members.

Renaming methods in Eiffel

In Eiffel, one must explicitly rename methods inherited from a base class, in order to make them accessible:



```
class int_list_node
inherit
    gp_list_node
        rename
            remove as old_remove
            ...     -- other renames
        end
```

Within methods of **int_list_node**, the remove method of **gp_list_node** can be invoked as **old_remove**. **C++** and **Eiffel** cannot use the keyword **super**, because it would be ambiguous in the presence of multiple inheritance.

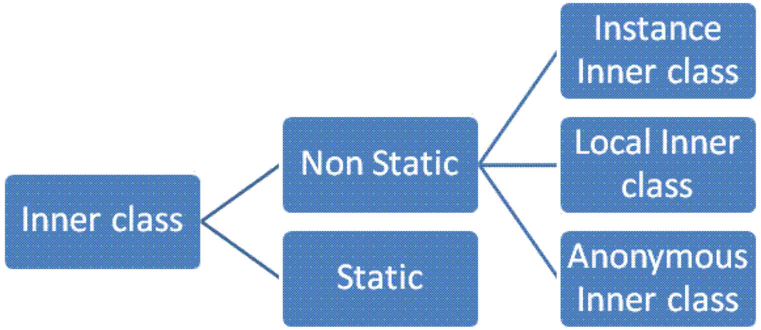
Classes and Objects III

Class Relationship

SECTION 5

Objects as Fields of Other Objects

Nested Class (Has_A relationship, Use, Composition, Association, Aggregation)



Inner class

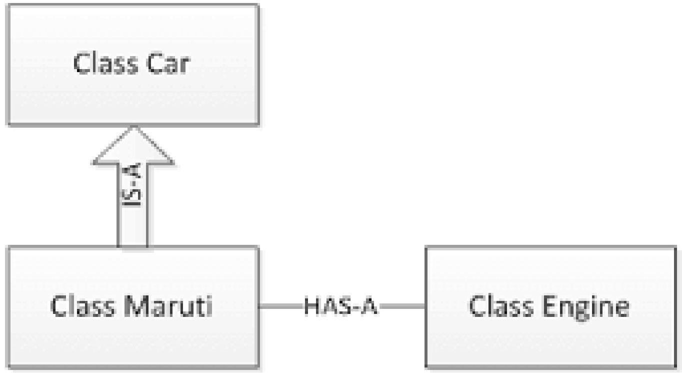
Non Static

Static

Instance Inner class

Local Inner class

Anonymous Inner class



Class Car

IS-A

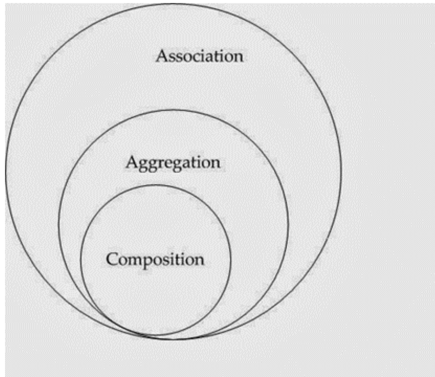
Class Maruti

HAS-A

Class Engine

```
class queue {   // queue can be implemented in IS_A or HAS_A ways
   list contents;
public:
   bool empty(){
     return contents.empty();
   }
   void enqueue(const int v){
         contents.append(new ist_node(v));
   }
   int dequeue(){
     if (empty()) throw new list_err(“attempt to dequeue from empty queue”);
     list_node* p = contents.head();
     p->remove();
     int v = p->val;
     delete p;
     return v;
   }
}
```

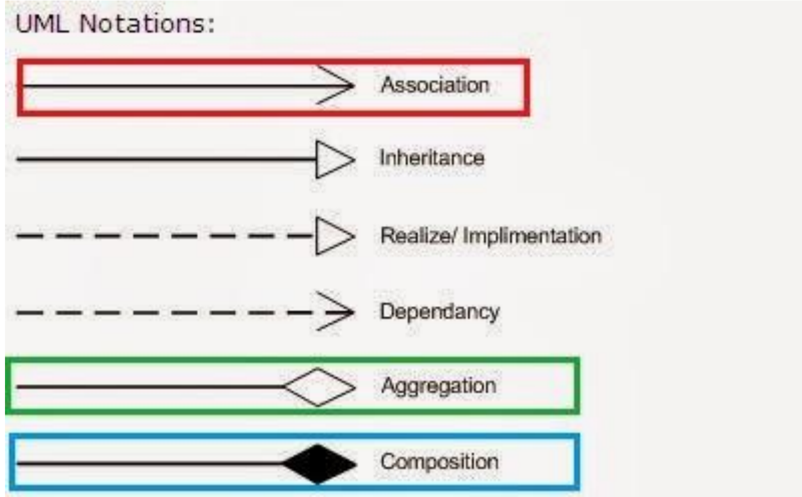
Types of Class-To-Class Relationships



Association

Aggregation

Composition



UML Notations:

Association

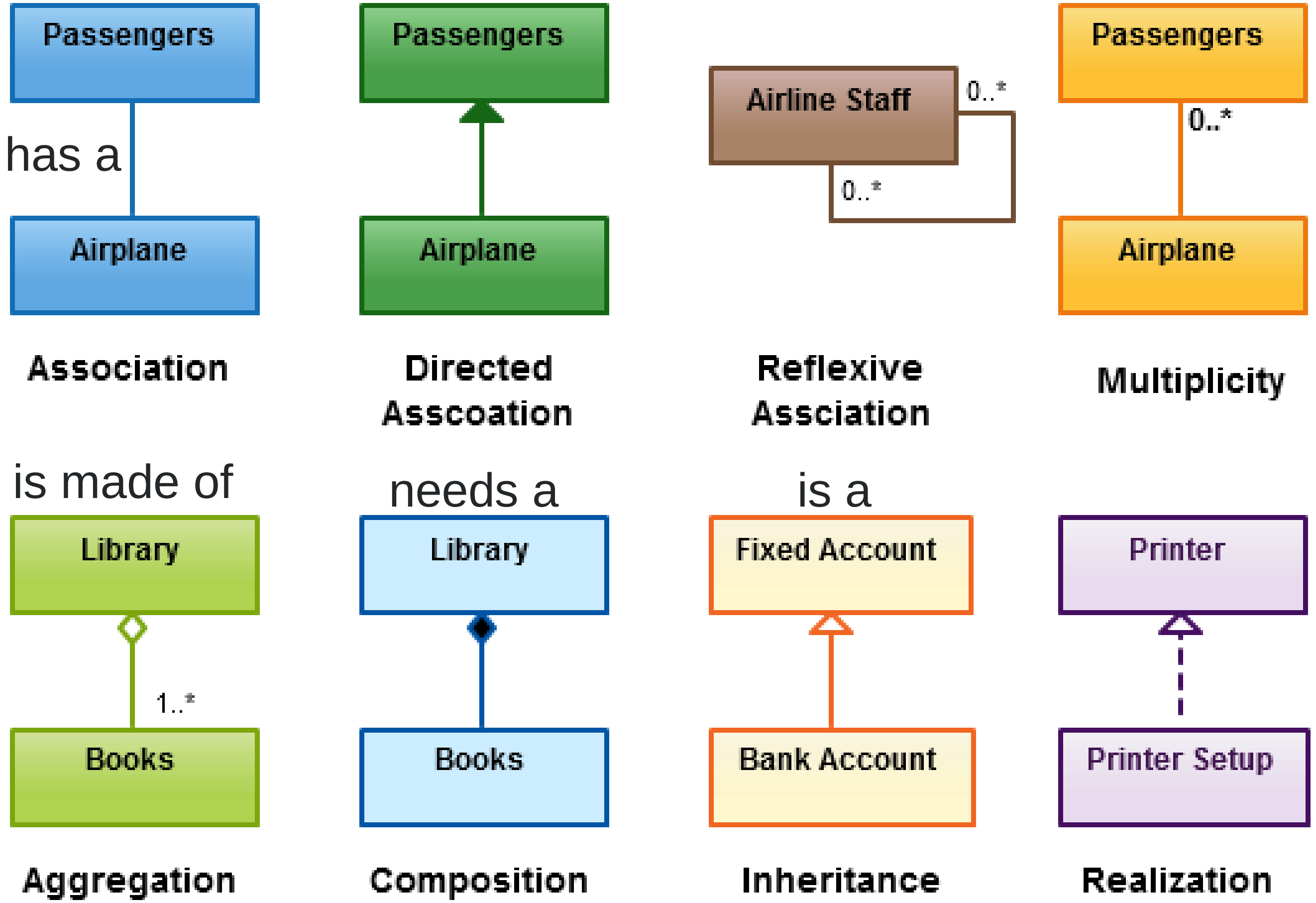
Inheritance

Realize/ Implimentation

Dependancy

Aggregation

Composition



Passengers

has a

Airplane

Association

Passengers

Airplane

Directed Asscoation

Airline Staff

0..*

0..*

Reflexive Assciation

Passengers

0..*

Airplane

Multiplicity

is made of

Library

1..*

Books

Aggregation

needs a

Library

Books

Composition

is a

Fixed Account

Bank Account

Inheritance

Printer

Printer Setup

Realization

Classes and Generics

Base Class for General Purpose Lists

```
class gp_list_node {
   gp_list_node* prev;
   gp_list_node* next;
   gp_list_node* head_node;

public:
   gp_list_node();
   gp_list_node* predecessor();
   gp_list_node* successor();
   bool singleton();
   void insert_before(gp_list_node*, new_node);
   void remoe();
   ~gp_list_node();
}
```

- In a statically typed language like **C++**, it is tempting t create a general-purpose **list_node** class that has no val field, and then derive sub-classes (e.g. **int_list_node**) that add the value.
- **Techniques:** subclass, abstract class, and class with generic types.

```
class int_list_node: public gp_list_node{
public:
   int val;
   int_list_node()        { val = 0;  }
   int_list_node(int v) { val = v;  }
…
```

Classes and Generics

Base Type problem with Type-specific Extensions

- **int_list_node** can inherit *list_node* class.
- But what about successor and predecessor? If we leave the methods unhanged. It won't compile.

```
int_list_node* p = …                // whatever
int v = p-> successor().val;        // won't compile;
```

- As far as the compiler is concerned, the successor of an **int_list_node** will have no **val** field.

```
int_list_node* predecessor(){
    return static_cast<int_list_node*>(gp_list_node::predecessor());
}
int_list_node* successor(){
    return static_cast<int_list_node*>(gp_list_node::successor());
}
```

Generic Lists in C++

The Correct Answer to the Need for Parametric Polymorphism

```
template<typename V>
class list_node{
   list_node<V>* prev;
   list_node<V>* next;
   list_node<V>* head_node;
public:
   V val;
   list_node<V>* predecessor(){ …
   list_node<V>* successor(){ …
   oid insert_before(list_node<V>* new_node){ …
   …
};
template<typename V>
class list{
   list_node<V> header;
public:
   list_node<V>* head(){ …
   void append(list_node<V>* new_node) { …
};
```

```
template<typename V>
class queue : private list<V>{ …
public:
   using list<V>::empty;
   void enqueue(const V v){ …
   V dequeue(){ …
   V head(){ …
};
```

USE:

```
typedef list_node<int> int_list_node;
typedef list_node<string> string_list_node;
typedef list int_list;   /* polymorphic list */
int_list_node n(3);
string_int_node s(“boo!”):
int_list L;
L.append(&n);
L.append(&s);
```

Generic Containers

In object-oriented programming, an abstraction that holds a collection of objects of some given class is often called a container. Common containers include sorted and unsorted sets, stacks, queues, and dictionaries, implemented as list, trees, hash tables, and various other concrete data structures.

All of the major object-oriented languages include extensive container libraries. A few of the issues involved in their creation have been hinted at in this section:

- Which classes are derived from which others?
- When do we say that “X is a Y”? Instead of “X contains/uses a Y”?
- Which operations supported, and what is their time complexity?
- How much “memory churn” does each operation incur?
- Is everything type safe? How extensive is the use of generics?
- How easy is it to iterate over the contents of a container?

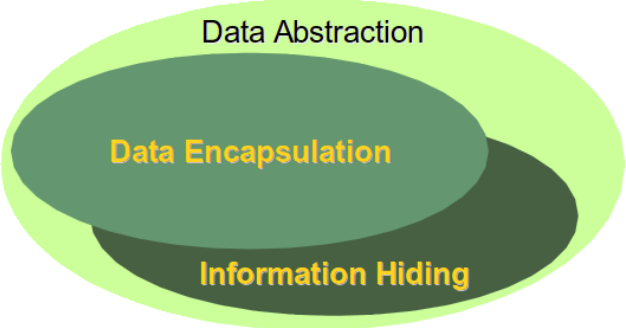
Encapsulation and Inheritance I

Encapsulation

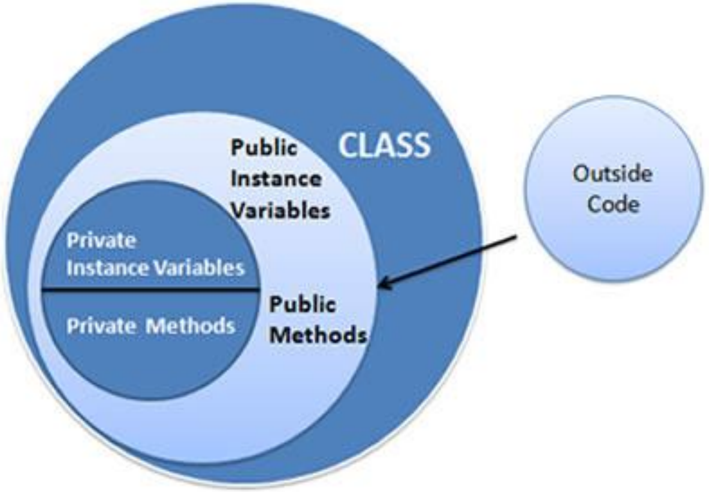
SECTION 6

Encapsulation

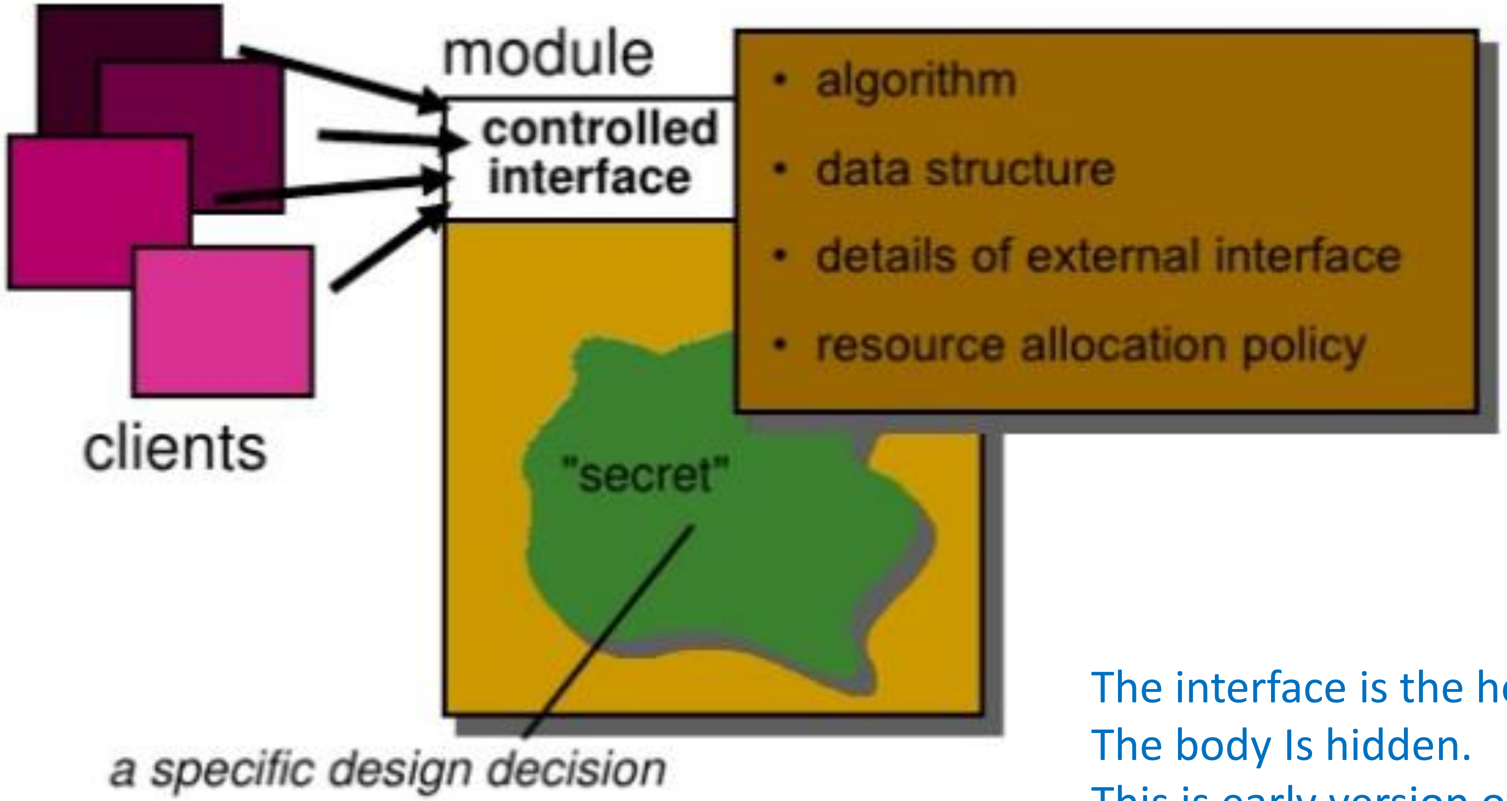
Information Hiding



- Encapsulation mechanisms enable the programmer to group data and the subroutines that operate on them together in one place, and to hide irrelevant details from the users of an abstraction.
- Topics for Encapsulation:
  - Data-hiding mechanisms of modules in non–object-oriented languages.
  - The new data-hiding issues, in OOP, that arise when we add inheritance to modules to make classes.
  - Module-as-manager approach, and show how several languages, including Ada 95 and Fortran 2003, add inheritance to records, allowing (static) modules to continue to provide data hiding.



Information Hiding by Module

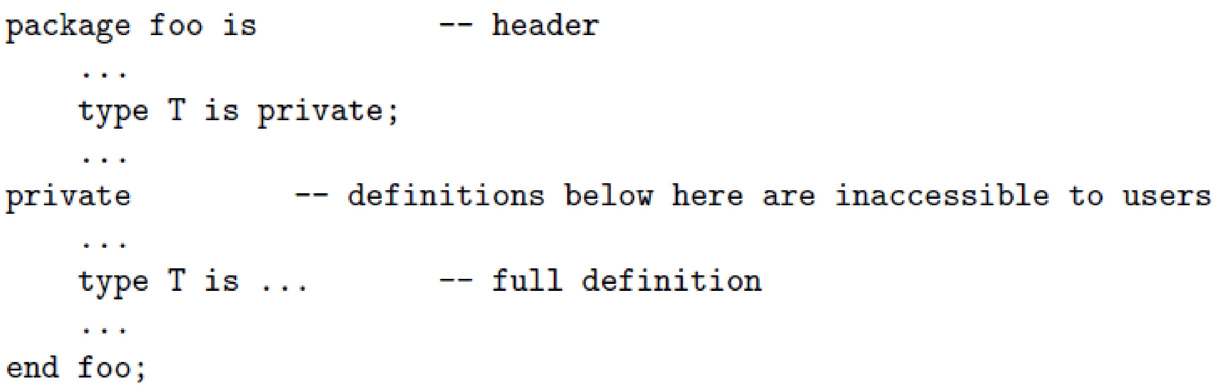

module
controlled interface
algorithm
data structure
details of external interface
resource allocation policy
clients
"secret"
a specific design decision

The interface is the header.
The body Is hidden.
This is early version of data encapsulation.

Modules

Data Hiding in Ada (Header and Body called Module)

- Ada allows the header of a package to be divided into public and private parts.
- Details of an exported type can be made ***opaque*** by putting them in the private part of the header and simply naming the type in the public part:



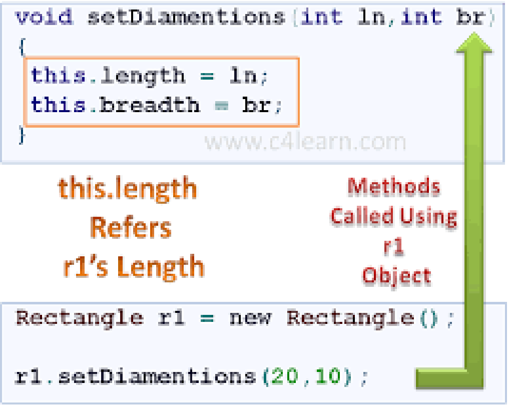
```
package foo is          -- header
    ...
    type T is private;
    ...
private         -- definitions below here are inaccessible to users
    ...
    type T is ...       -- full definition
    ...
end foo;
```

- Module with private section provide mechanism for data hiding.
- The private part provides the information the compiler needs to allocate objects “in line.”
- A change to the body of a module never forces recompilation of any of the uses of the module.
- A change to the private part of the module header may force recompilation, but it **never** requires changes to the source code of the users.
- Affect only the visibility of names, static, manager-style modules introduce no special code generation issues.
- If the module appears in a global scope, then its data ca be allocated statically.
- If a module appears within a subroutine, then its data can be allocated in call stack.

“this” Parameter

When a instance make a call, itself is implicitly passed as this reference.

- One more issue arise when a subroutine inside a module.
- How do you know which variable to use? (The private part may have a variable of the same name with the global or parameter variable.)
- Technique: to create a single instance of each module subroutine, and to pass that instance, at run time, the address of the storage of the appropriate module instance. This address takes the form of an extra, hidden first parameter for every module routine.



```
void setDiamentions(int ln,int br)
{
  this.length = ln;
  this.breadth = br;
}
```

```
Rectangle r1 = new Rectangle();

r1.setDiamentions(20,10);
```

www.c4learn.com
this.length Refers r1's Length
Methods Called Using r1 Object

An Euclid call of the form

```
my_stack.push(x)
```

is translated as if it were really

```
push(my_stack, x)
```

this object reference

Where my_stack is passed by reference. The same translation occurs in object-oriented languages.

Encapsulation and Inheritance II

Inheritance and Visibility

SECTION 7

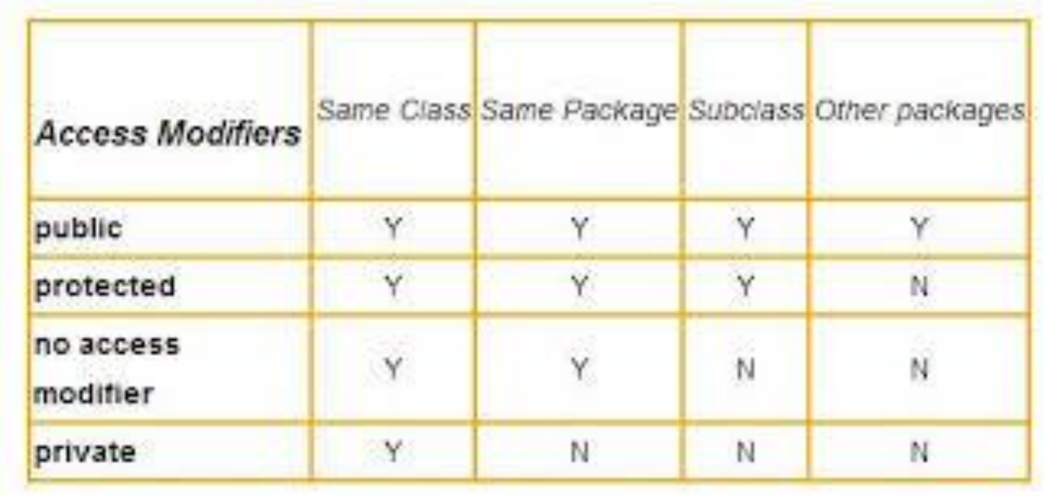
Visibility Rules

- Public and Private parts of an object declaration/definition
- 2 reasons to put things in the declaration
  - so programmers can get at them
  - so the compiler can understand them
- At the very least the compiler needs to know the size of an object, even though the programmer isn't allowed to get at many or most of the fields (members) that contribute to that size
  - That's why private fields have to be in declaration

C++

- C++ distinguishes among
  - public class members
    - accessible to anybody
  - protected class members
    - accessible to members of this or derived classes
  - private
    - accessible just to members of this class
- A C++ structure (*struct*) is simply a class whose members are public by default
- C++ base classes can also be public, private, or protected

Java Visibility



Access Modifiers	*Same Class*	*Same Package*	*Subclass*	*Other packages*
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no access modifier	Y	Y	N	N
private	Y	N	N	N

C# Visibility

visibility **keyword**	Containing Classes	Derived Classes	Containing Assembly	Anywhere outside the containing assembly
public	yes	yes	yes	yes
protected internal	yes	yes	yes	no
protected	yes	yes	no	no
private	yes	no	no	no
internal	yes	no	yes	no

Note:

For using object-oriented language, it is very essential to find out this visibility matrix.

C++Inheritance and Visibility

- Example:

```
class circle : public shape { ...
```

anybody can convert (assign) a circle* into a shape*

```
class circle : protected shape { ...
```

only members and friends of circle or its derived classes can convert (assign) a circle* into a shape*

```
class circle : private shape { ...
```

only members and friends of circle can convert (assign) a circle* into a shape*

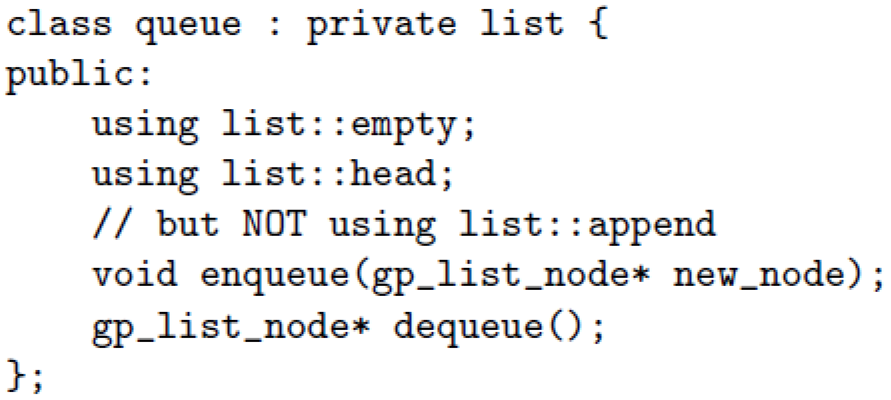
Class

Visibility

- With the introduction of inheritance, object-oriented languages must supplement the scope rules of module-based languages to cover additional issues.
- For example,
  - should private members of a base class be visible to methods of a derived class?
  - Should public members of a base class always be public members of a derived class (i.e., be visible to users of the derived class)?
  - How much control should a base class exercise over the visibility of its members in derived classes?

Default Hiding:

In C++, the definition of class queue can specify that its base (list) class is to be private:



```
class queue : private list {
public:
    using list::empty;
    using list::head;
    // but NOT using list::append
    void enqueue(gp_list_node* new_node);
    gp_list_node* dequeue();
};
```

Note:

Sharing is an exception.

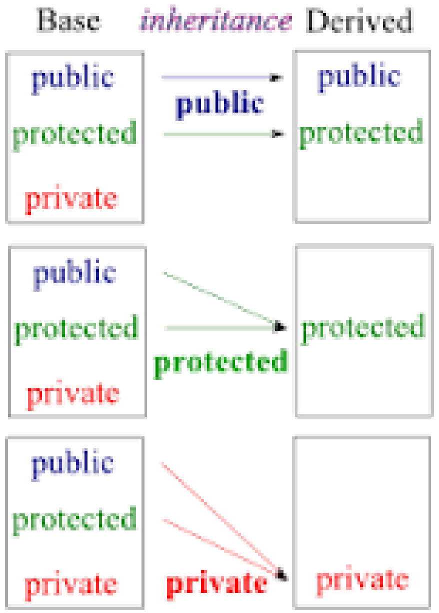
Class

Visibility

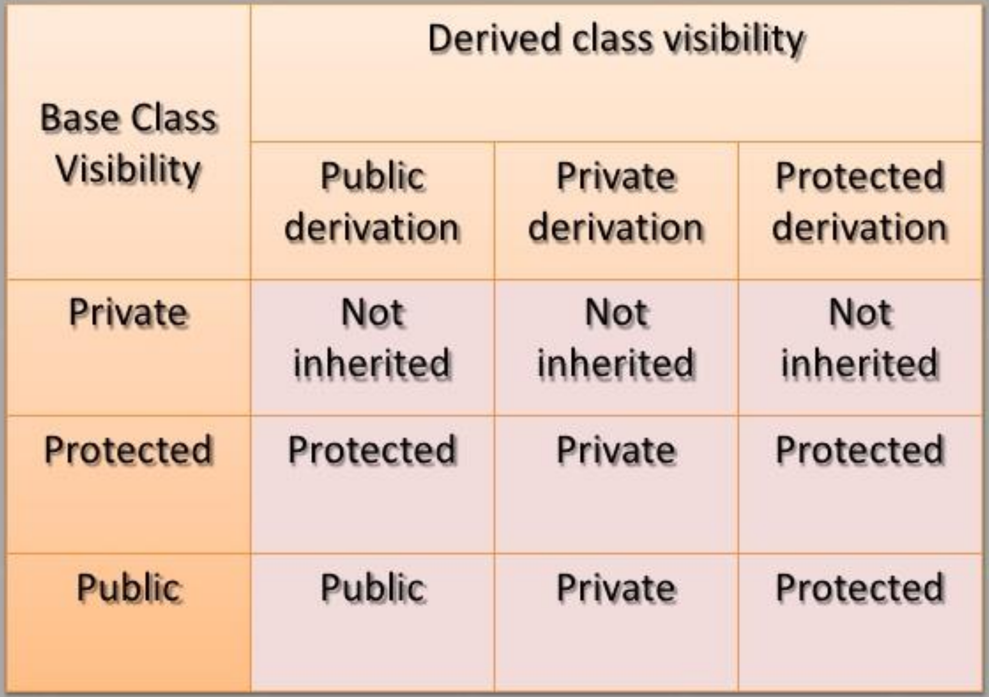
Inhibiting by delete:

```
class queue : public list {
  …
  void append(list_node * new_node) = delete;
}
```

Note:
Hiding is exceptional in this case.



C++ Visibility



Base Class Visibility	Derived class visibility		
	Public derivation	Private derivation	Protected derivation
Private	Not inherited	Not inherited	Not inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

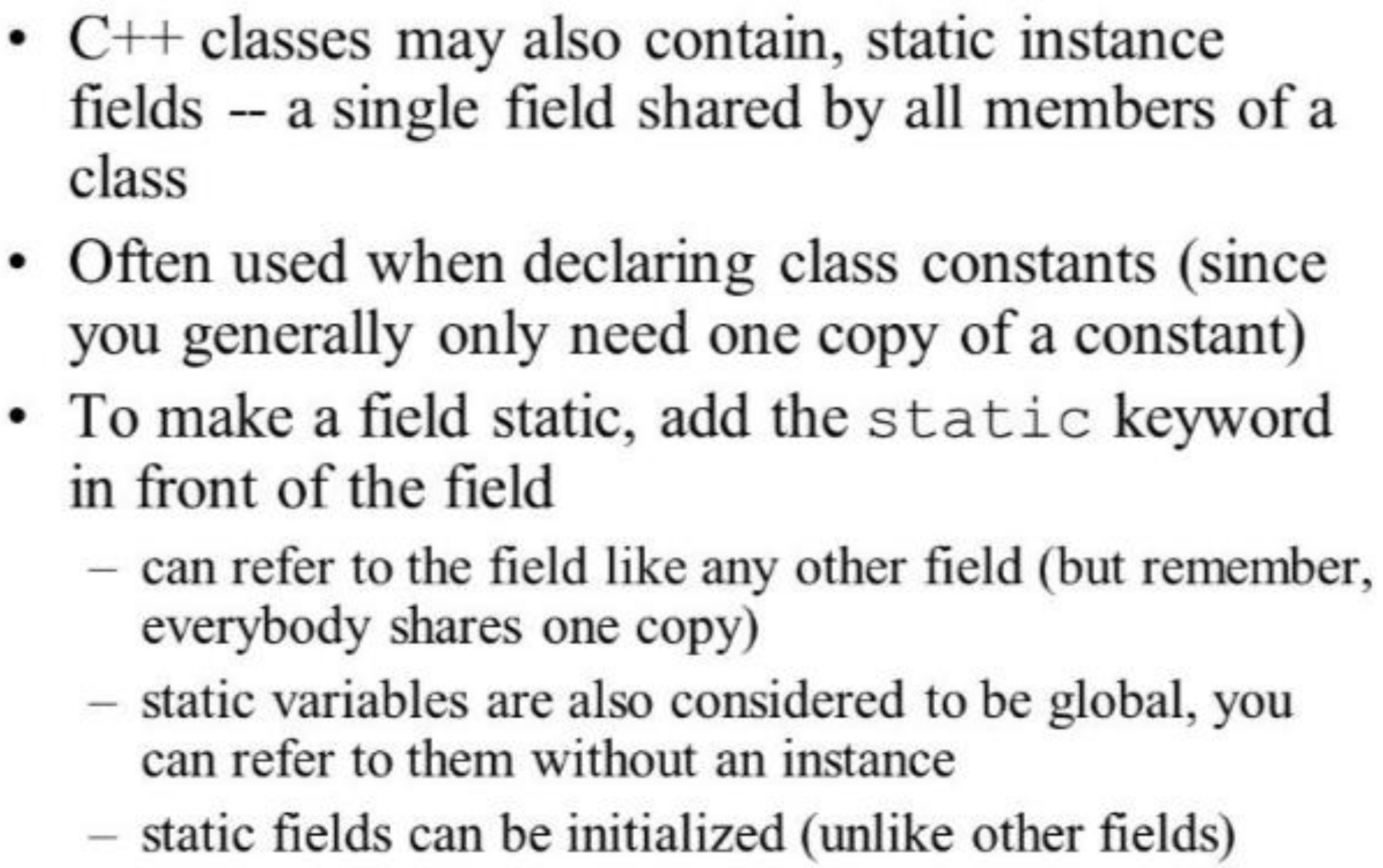
Encapsulation and Inheritance III

Other Topics

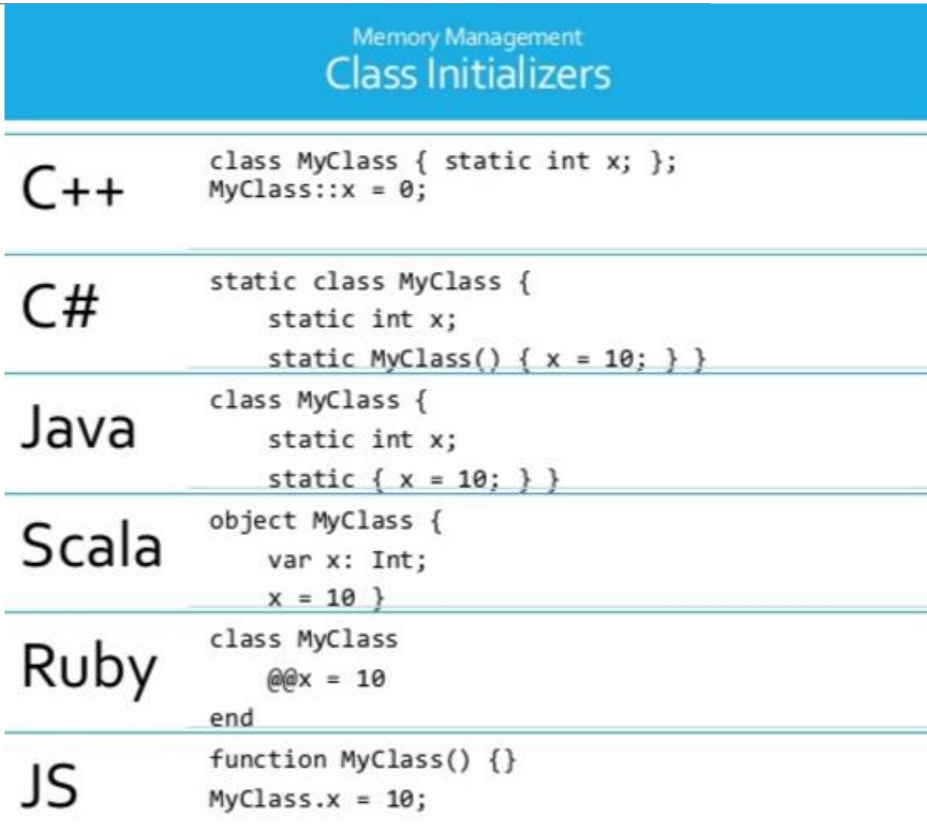
SECTION 8

Static Fields and Methods

Static method can only access static data. Instance method can access both static/instance data



- C++ classes may also contain, static instance fields -- a single field shared by all members of a class
- Often used when declaring class constants (since you generally only need one copy of a constant)
- To make a field static, add the `static` keyword in front of the field
  - can refer to the field like any other field (but remember, everybody shares one copy)
  - static variables are also considered to be global, you can refer to them without an instance
  - static fields can be initialized (unlike other fields)



Memory Management

Class Initializers

C++	`class MyClass { static int x; };`
	`MyClass::x = 0;`

C#	`static class MyClass {`
	`    static int x;`
	`    static MyClass() { x = 10; } }`

Java	`class MyClass {`
	`    static int x;`
	`    static { x = 10; } }`

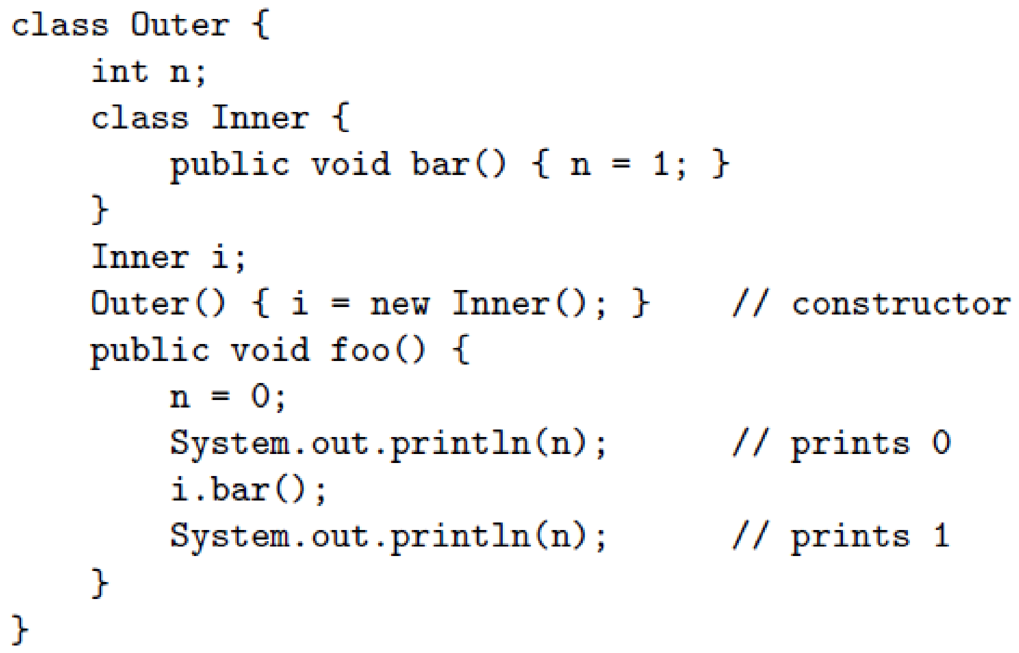
Scala	`object MyClass {`
	`    var x: Int;`
	`    x = 10 }`

Ruby	`class MyClass`
	`    @@x = 10`
	`end`

JS	`function MyClass() {}`
	`MyClass.x = 10;`

Nested (Inner Class)

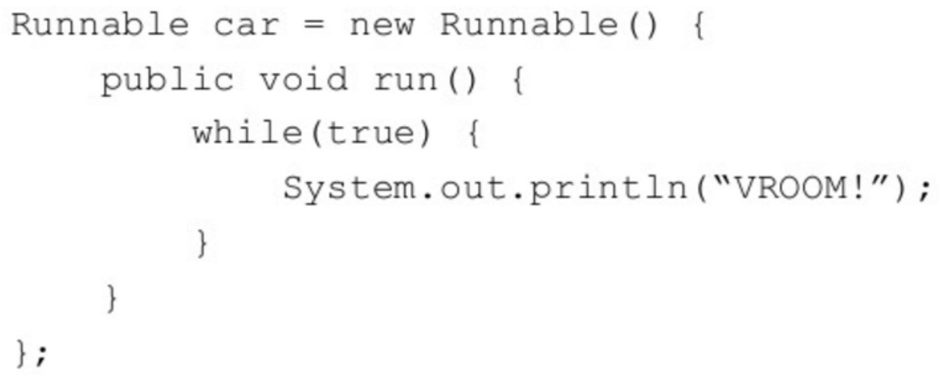
Inner Classes in Java:



```
class Outer {
    int n;
    class Inner {
        public void bar() { n = 1; }
    }
    Inner i;
    Outer() { i = new Inner(); }    // constructor
    public void foo() {
        n = 0;
        System.out.println(n);      // prints 0
        i.bar();
        System.out.println(n);      // prints 1
    }
}
```

Note: If there are multiple instances of Outer, each instance will have a different n, and calls to **Inner.bar** will access the appropriate n. To make this work, each instance of Inner must contain a hidden pointer (this) to the instance of Outer to which it belongs.

Anonymous Classes in Java:



```
Runnable car = new Runnable() {
    public void run() {
        while(true) {
            System.out.println("VROOM!");
        }
    }
};
```

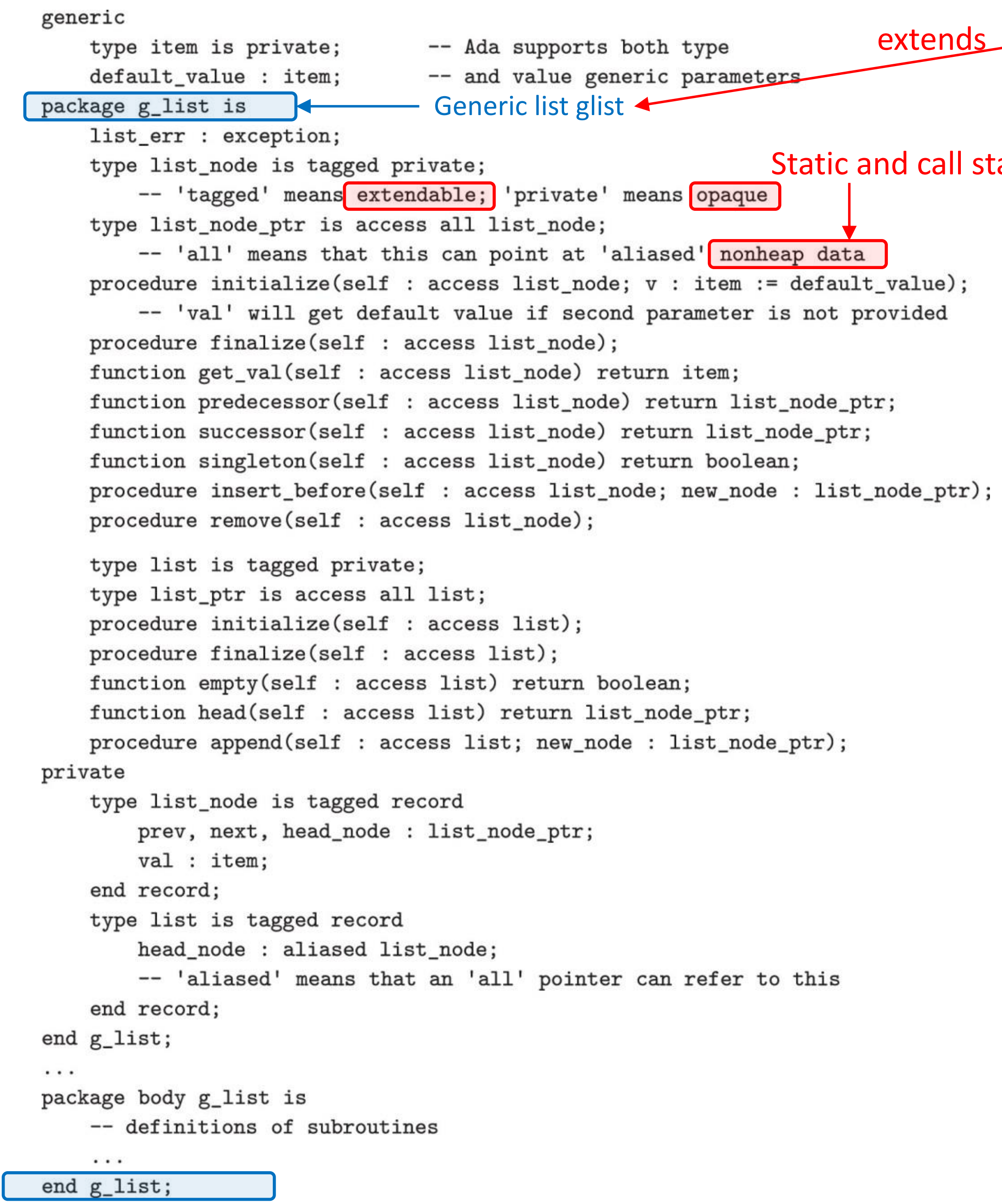
Note:
Inner and local classes in Java are widely used to create object closures, as described in Section 3.6.3. In Section 9.6.2 we used them as handlers for events.

Type Extensions

- **Smalltalk**, **Objective-C**, **Eiffel**, **C++**, **Java**, and **C#** were all designed from the outset as object-oriented languages, either starting from scratch or from an existing language without a strong encapsulation mechanism. They all support a **module-as-type** approach to abstraction, in which a single mechanism (**the class**) provides both **encapsulation** and **inheritance**.

- Several other languages, including **Modula-3**, **Ada 95**,**Oberon**, **CLOS**, and **Fortran 2003**, can be characterized as object-oriented extensions to languages in which modules already provide encapsulation. Rather than alter the existing module mechanism, these languages provide inheritance and dynamic method binding through a mechanism for extending records. [Note: separate data and method extension mechanisms. Not the main stream.]

List and queue abstractions in Ada 2005

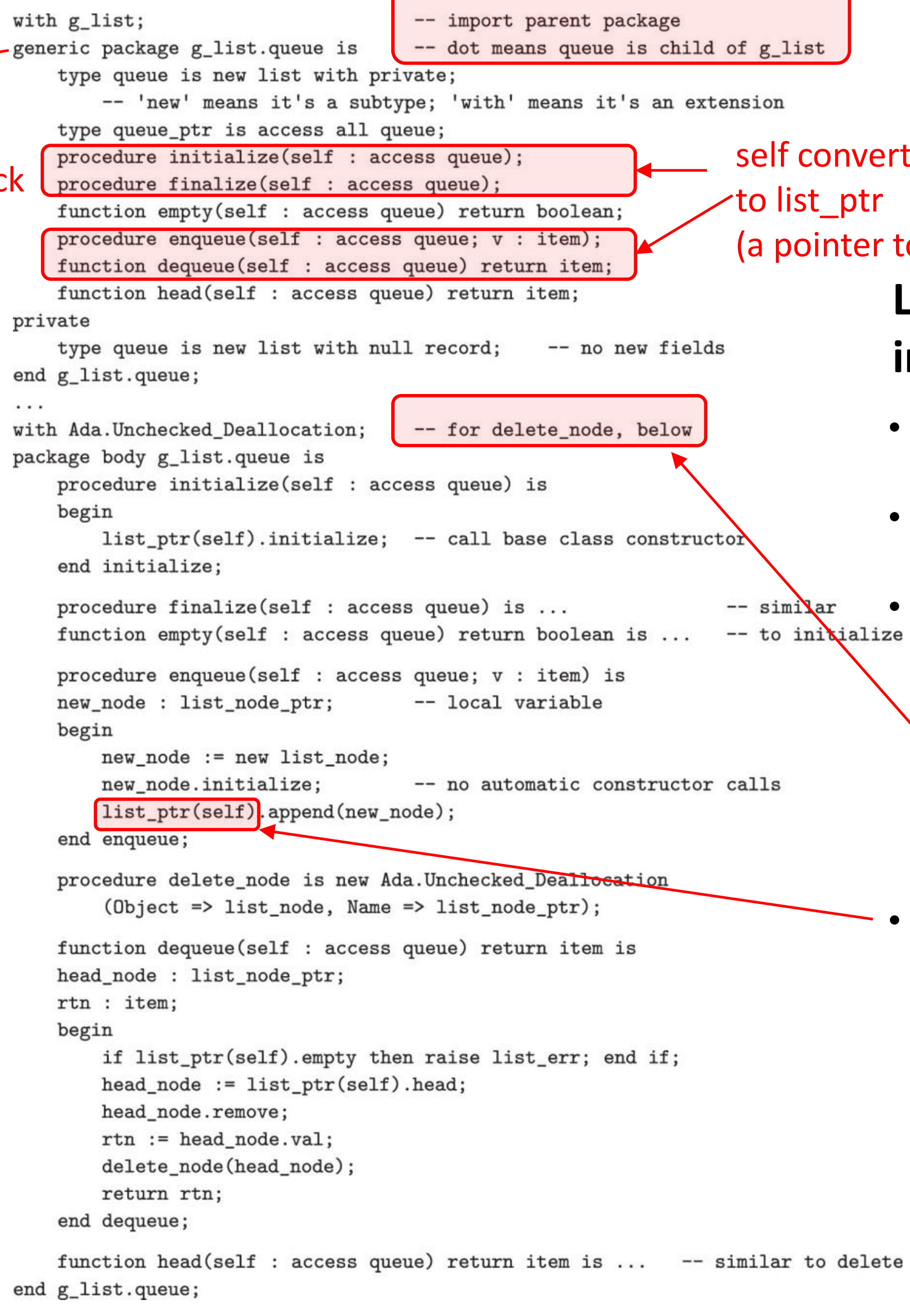


```
generic
    type item is private;         -- Ada supports both type
    default_value : item;         -- and value generic parameters
package g_list is
    list_err : exception;
    type list_node is tagged private;
        -- 'tagged' means extendable; 'private' means opaque
    type list_node_ptr is access all list_node;
        -- 'all' means that this can point at 'aliased' nonheap data
    procedure initialize(self : access list_node; v : item := default_value);
        -- 'val' will get default value if second parameter is not provided
    procedure finalize(self : access list_node);
    function get_val(self : access list_node) return item;
    function predecessor(self : access list_node) return list_node_ptr;
    function successor(self : access list_node) return list_node_ptr;
    function singleton(self : access list_node) return boolean;
    procedure insert_before(self : access list_node; new_node : list_node_ptr);
    procedure remove(self : access list_node);

    type list is tagged private;
    type list_ptr is access all list;
    procedure initialize(self : access list);
    procedure finalize(self : access list);
    function empty(self : access list) return boolean;
    function head(self : access list) return list_node_ptr;
    procedure append(self : access list; new_node : list_node_ptr);
private
    type list_node is tagged record
        prev, next, head_node : list_node_ptr;
        val : item;
    end record;
    type list is tagged record
        head_node : aliased list_node;
        -- 'aliased' means that an 'all' pointer can refer to this
    end record;
end g_list;
...
package body g_list is
    -- definitions of subroutines
    ...
end g_list;
```

Generic list glist

Static and call stack

extends



```
with g_list;                                -- import parent package
generic package g_list.queue is             -- dot means queue is child of g_list
    type queue is new list with private;
        -- 'new' means it's a subtype; 'with' means it's an extension
    type queue_ptr is access all queue;
    procedure initialize(self : access queue);
    procedure finalize(self : access queue);
    function empty(self : access queue) return boolean;
    procedure enqueue(self : access queue; v : item);
    function dequeue(self : access queue) return item;
    function head(self : access queue) return item;
private
    type queue is new list with null record;    -- no new fields
end g_list.queue;
...
with Ada.Unchecked_Deallocation;            -- for delete_node, below
package body g_list.queue is
    procedure initialize(self : access queue) is
    begin
        list_ptr(self).initialize;   -- call base class constructor
    end initialize;

    procedure finalize(self : access queue) is ...            -- similar
    function empty(self : access queue) return boolean is ...  -- to initialize

    procedure enqueue(self : access queue; v : item) is
    new_node : list_node_ptr;        -- local variable
    begin
        new_node := new list_node;
        new_node.initialize;         -- no automatic constructor calls
        list_ptr(self).append(new_node);
    end enqueue;

    procedure delete_node is new Ada.Unchecked_Deallocation
        (Object => list_node, Name => list_node_ptr);

    function dequeue(self : access queue) return item is
    head_node : list_node_ptr;
    rtn : item;
    begin
        if list_ptr(self).empty then raise list_err; end if;
        head_node := list_ptr(self).head;
        head_node.remove;
        rtn := head_node.val;
        delete_node(head_node);
        return rtn;
    end dequeue;

    function head(self : access queue) return item is ...   -- similar to delete
end g_list.queue;
```

self converted to list_ptr (a pointer to list)

- **self** in many languages work as **this**.
- Compiler has to handle conversion of pointer to **list_ptr**.
- If **ptr** refers to an object a tagged type. Function **delete_node(next_page)** uses the **Unchecked_Deallocation** library package to create a type-specific routine for memory reclamation.
- The expression list_ptr(self) is a (type-safe) cast.

Extending Without Inheritance

- C# 3.0 provides extension methods, which give the appearance of extending an existing class:

```
static class AddToString {
    public static int toInt(this string s) {
        return int.Parse(s);
    }
}
```

Static method works instance method.

- An extension method must be static, and must be declared in a static class. Its first parameter must be prefixed with the keyword **this**. The method can then be invoked as if it were a member of the class of which this is an instance:

```
int n = myString.toInt();
```

- Together, the method declaration and use are syntactic sugar for

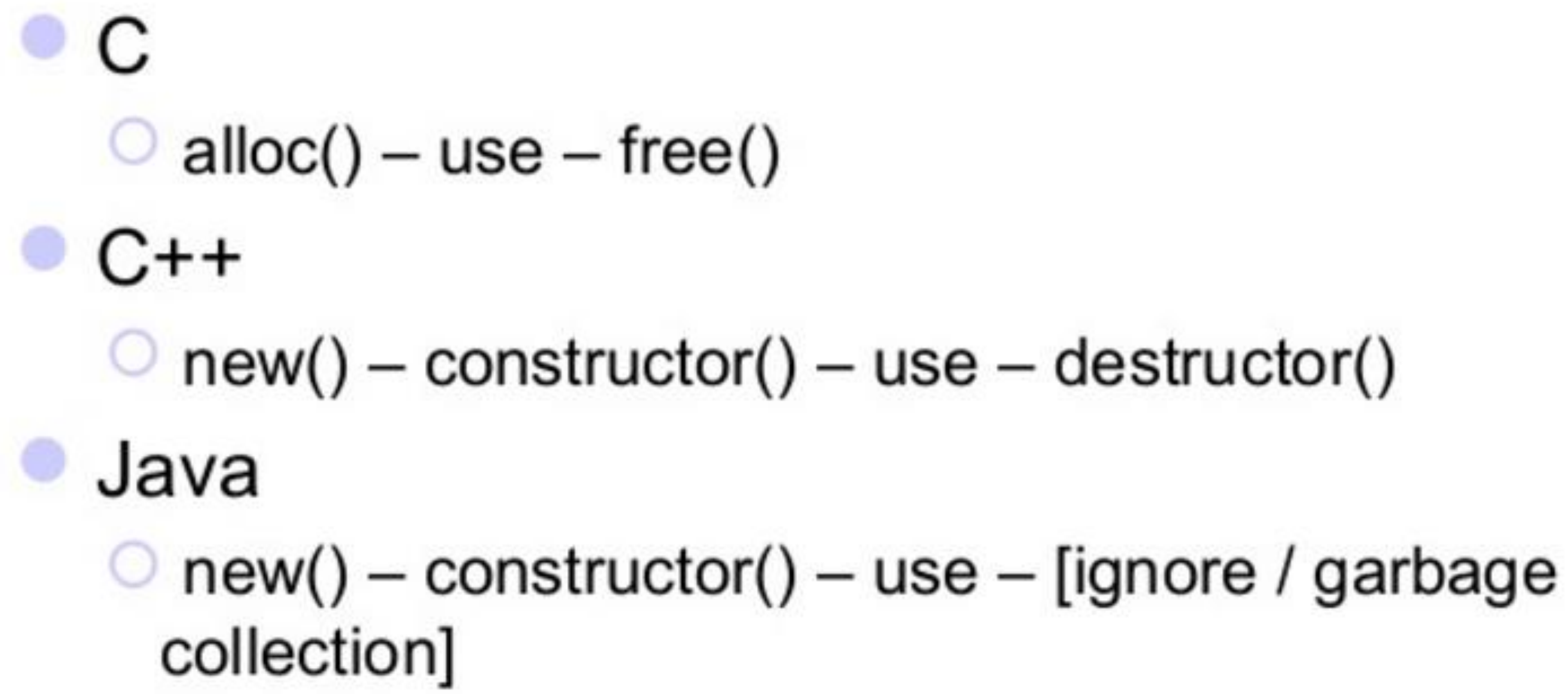
```
static class AddToString {
    public static int toInt(string s) {         // no 'this'
        return int.Parse(s);
    }
}
...
int n = AddToString.toInt(myString);
```

Initialization and Finalization I

Constructor/Reference

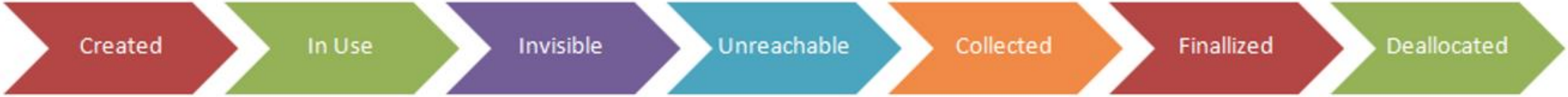
SECTION 9

Three Types of Object Life-Cycle

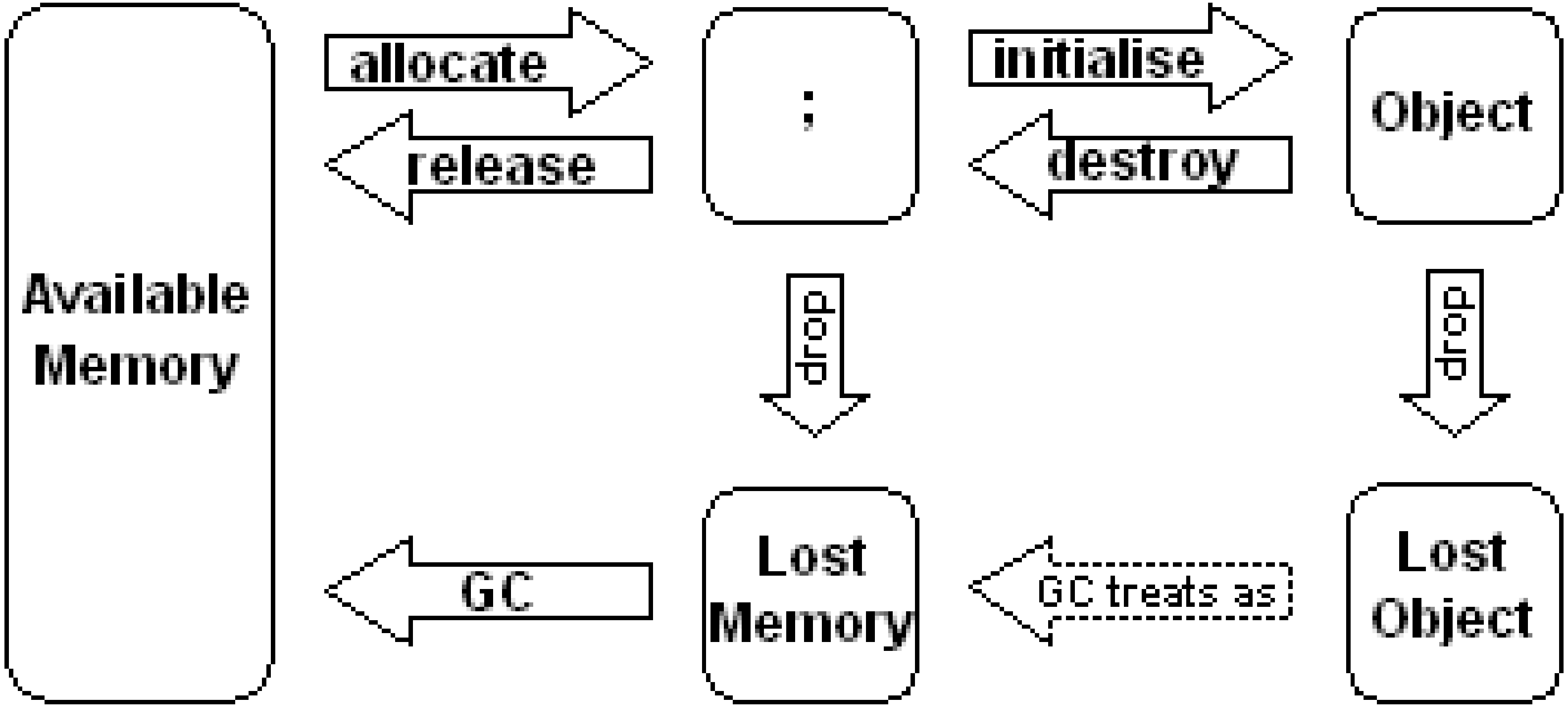


- C
  - alloc() – use – free()
- C++
  - new() – constructor() – use – destructor()
- Java
  - new() – constructor() – use – [ignore / garbage collection]

Java Object Life Cycle



Created → In Use → Invisible → Unreachable → Collected → Finallized → Deallocated



Initialization and Finalization

In Section 3.2, we defined the lifetime of an object to be the interval during which it occupies space and can hold data

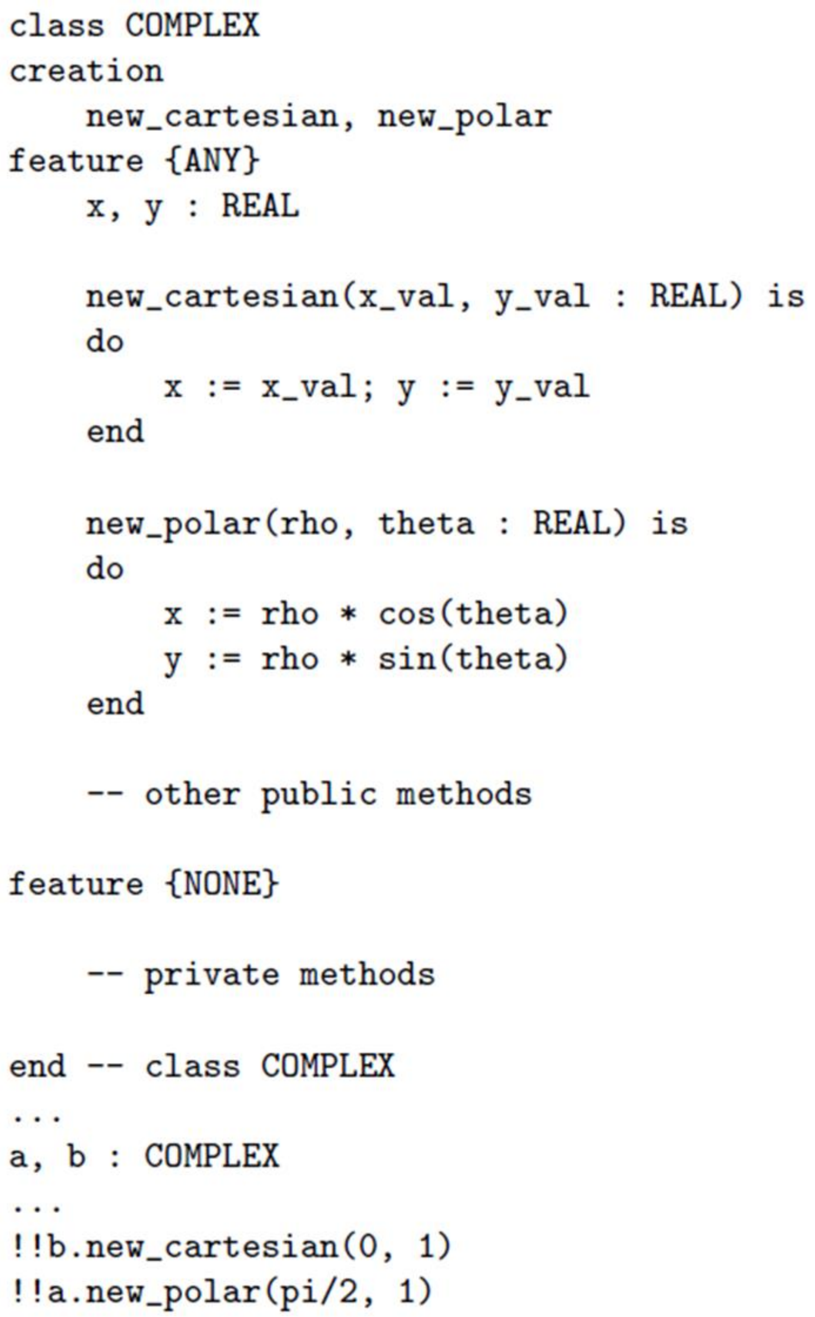
- Most object-oriented languages provide some sort of special mechanism to **initialize** an object automatically at the beginning of its lifetime
  - When written in the form of a subroutine, this mechanism is known as a **constructor**
  - A constructor does not allocate space
- A few languages provide a similar **destructor** mechanism to **finalize** an object automatically at the end of its lifetime

Initialization and Finalization

1. Choosing a constructor
2. References and values
   - If variables are references, then every object must be created explicitly - appropriate constructor is called
   - If variables are values, then object creation can happen implicitly as a result of elaboration
3. Execution order
   - When an object of a derived class is created in C++, the constructors for any base classes will be executed before the constructor for the derived class
4. Garbage collection

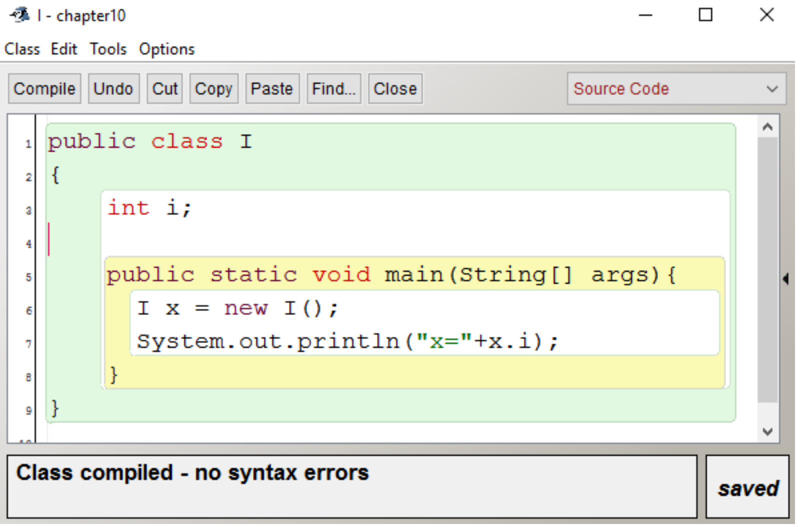
Choosing a Constructor (I)

- Smalltalk, Eiffel, C++, Java, and C# all allow the programmer to specify more than one constructor for a given class.
- In C++, Java, and C#, the constructors behave like overloaded subroutines: they must be distinguished by their numbers and types of arguments.
- The !! operator is Eiffel’s equivalent of new.
- Smalltalk resembles Eiffel in the use of multiple named constructors, but it distinguishes more sharply between operations that pertain to an individual object and operations that pertain to a class of objects.



```
class COMPLEX
creation
    new_cartesian, new_polar
feature {ANY}
    x, y : REAL

    new_cartesian(x_val, y_val : REAL) is
    do
        x := x_val; y := y_val
    end

    new_polar(rho, theta : REAL) is
    do
        x := rho * cos(theta)
        y := rho * sin(theta)
    end

    -- other public methods

feature {NONE}

    -- private methods

end -- class COMPLEX
...
a, b : COMPLEX
...
!!b.new_cartesian(0, 1)
!!a.new_polar(pi/2, 1)
```

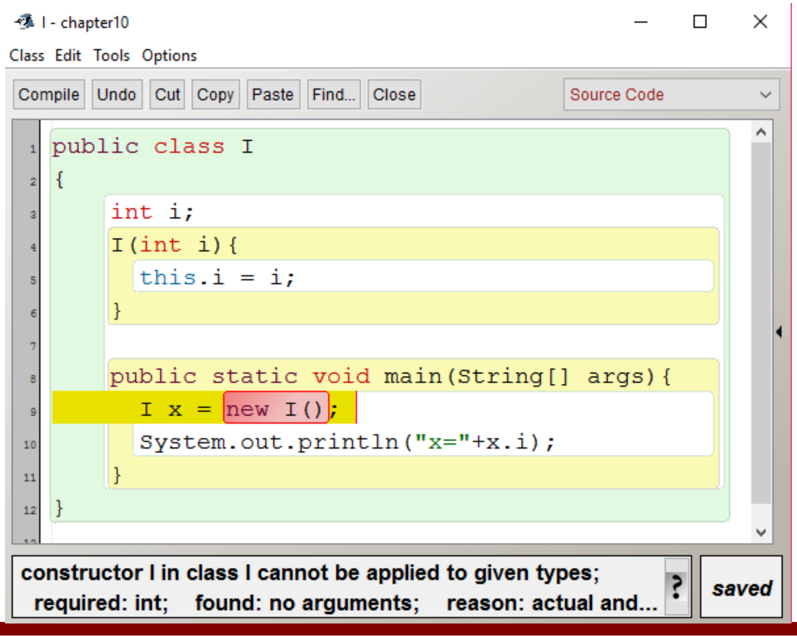
Java Example for Default, Overloaded, Multiple Constructors, and **this** Reference for Constructor



```java
public class I
{
    int i;

    public static void main(String[] args){
        I x = new I();
        System.out.println("x="+x.i);
    }
}
```

Class compiled - no syntax errors

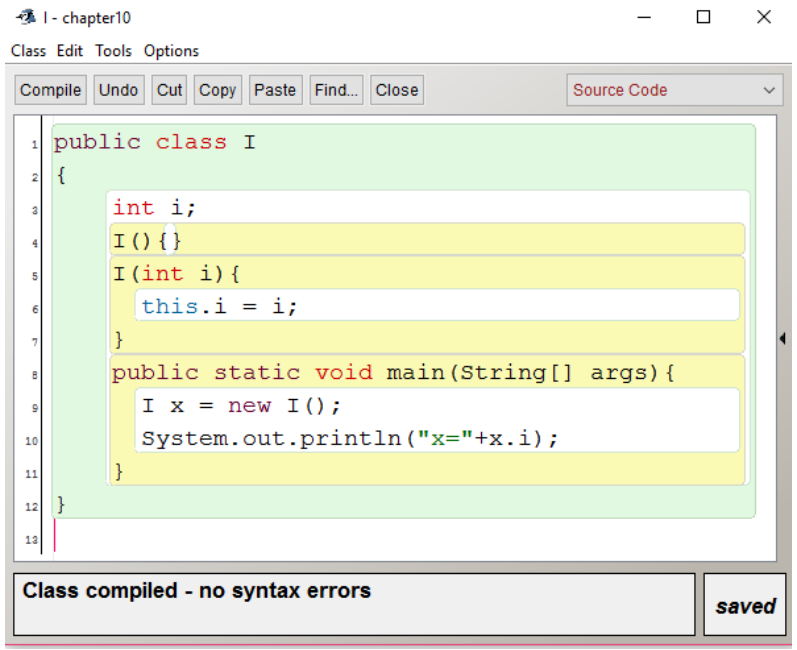
saved



```java
public class I
{
    int i;
    I(int i){
        this.i = i;
    }

    public static void main(String[] args){
        I x = new I();
        System.out.println("x="+x.i);
    }
}
```

constructor I in class I cannot be applied to given types; required: int; found: no arguments; reason: actual and...

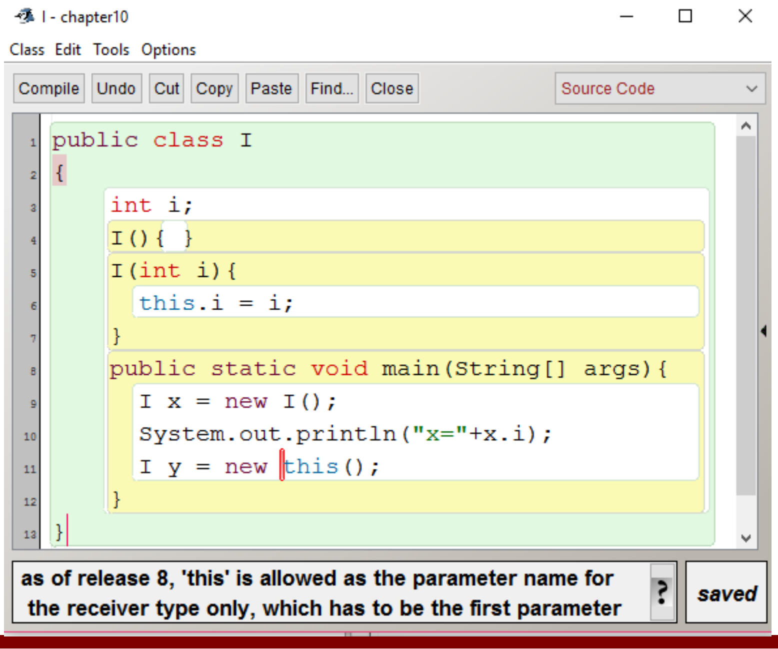
saved



```java
public class I
{
    int i;
    I(){}
    I(int i){
        this.i = i;
    }
    public static void main(String[] args){
        I x = new I();
        System.out.println("x="+x.i);
    }
}
```

Class compiled - no syntax errors

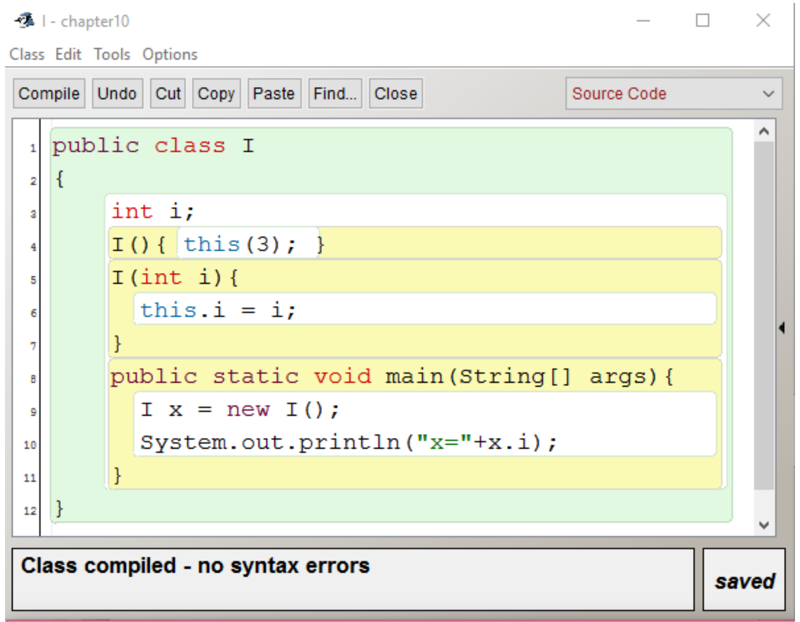
saved



```java
public class I
{
    int i;
    I(){ }
    I(int i){
        this.i = i;
    }
    public static void main(String[] args){
        I x = new I();
        System.out.println("x="+x.i);
        I y = new this();
    }
}
```

as of release 8, 'this' is allowed as the parameter name for the receiver type only, which has to be the first parameter

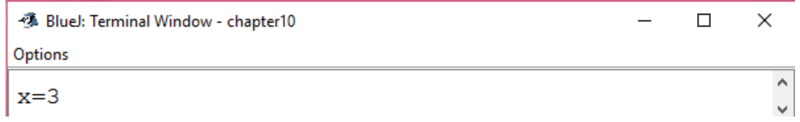
saved



```java
public class I
{
    int i;
    I(){ this(3); }
    I(int i){
        this.i = i;
    }
    public static void main(String[] args){
        I x = new I();
        System.out.println("x="+x.i);
    }
}
```

Class compiled - no syntax errors

saved



```
x=3
```

References and Values

- Several object-oriented languages, including **Simula**, **Smalltalk**, **Python**, **Ruby**, and **Java**, use a programming model in which variables refer to objects.
- Other languages, including **C++**, **Modula-3**, **Ada 95**, and **Oberon**, allow a variable to have a value that is an object.
- **C#** and **Swift** uses struct to define types whose variables are values, and class to define types whose variables are references.
- With a reference model for variables every object is created explicitly, and it is easy to ensure that an appropriate constructor (Overloaded) is called.
- In C++, the compiler ensures that an appropriate constructor is called for every elaborated object, but the rules it uses to identify constructors and their arguments can sometimes be confusing.

Declarations and Constructors in C++

Declarations and Constructors in C++:

```
foo   b;        // calls foo::foo()
```

If a C++ variable of class type foo is declared with no initial value, then the compiler will call foo's zero-argument constructor.

Declarations and Constructors with parameters in C++:

```
foo   b(10, 'x');       // calls foo::foo(int, char)
foo   c{10, 'x'};       // alternative syntax in C++11
```

Declarations and Constructors with Reference in C++:

```
foo   a;
…
foo   b(a);    // calls  foo::foo(foo&)
foo  c{a} ;    // alternative syntax
```

Copying of Objects:

```
foo   a;        // calls foo::foo()
…
foo   b = a;  // calls  foo::foo(foo&)
```

In recognition of this intent, a single-argument constructor in C++ is called a **copy constructor**. It is important to realize here that the equals sign (=) in these declarations indicates initialization, not assignment. The effect is not the same as that of the similar code fragment.

Assignment:

```
foo   a, b;        // calls foo::foo() twice
…
b = a;        // calls  foo::operator=(foo&)
```

This is assignment not initialization.

Declarations and Constructors in C++

Temporary Objects

In **C++**, the requirement that every object be constructed (and likewise destructed) applies not only to objects with names but also to temporary objects. The following, for example, entails a call to both the **string(const char*)** constructor ad the **~string()** destructor:

```
cout<< string(“Hi, Mom”).length();
```

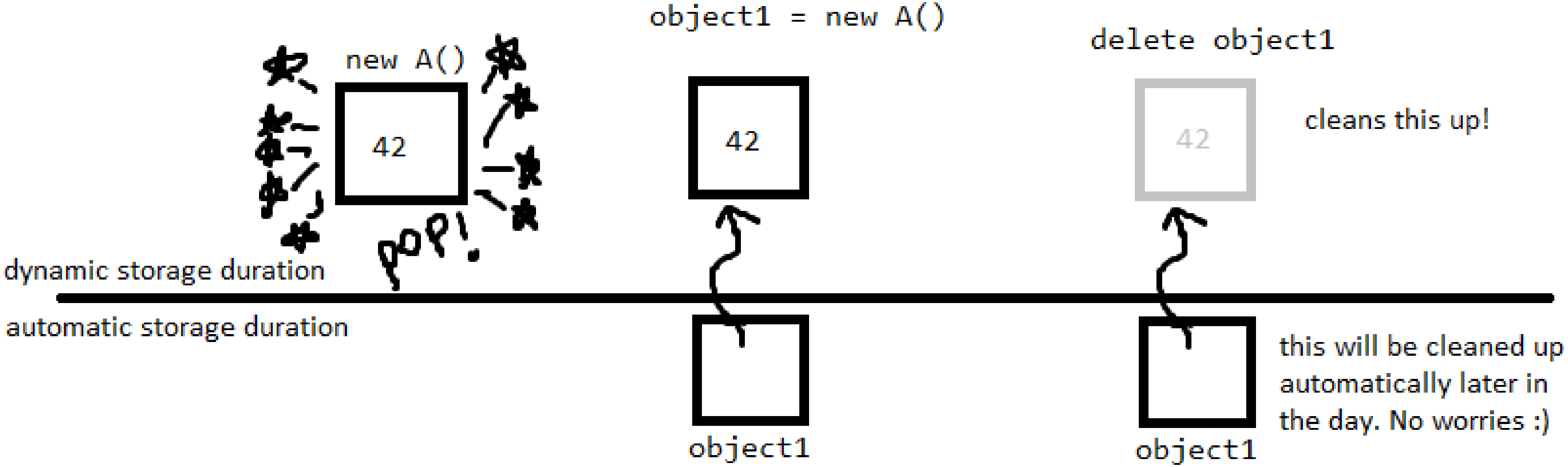
The destructor called at the end of the output statement: the temporary objects behaves as if its sop were just the line shown here.

The following entails not only two calls to the default string constructor and a call to string::operator(), but also constructor call to initialize the temporary object returned by operator() – the object whose length is then queried by the caller:

```
string a, b;
…
(a+b).length();
```

new Operator:

Object Created by memory allocation in heap.



Return Value Optimization

- **f** is a function returning a value of class type **foo**.
- If instance of **foo** are too big to fit in a register, the compiler will arrange for **f**’s caller to pass an extra, hidden parameter that specifies the locations into which **f** should construct the return value. **(a mail box for return value)**
- If the return statement itself creates a temporary object -

```
return foo(args)
```

- **f**’s source looks more like this:

```
foo rtn;
…
return rtn;
```

Note: This option is known as Return value optimization.

Return Value Optimization

- In other programs the compiler may need to invoke a copy constructor after a function returns:

```
foo c;
…
c = foo(args);
```

- The location of **c** cannot be passed to the hidden parameter to unless the compiler is able to prove that **c**'s value will not be used during the call.
- **The bottom line:** returning an object from a function in C++ may entail zero, one or two invocations of return type's copy. Constructor, depending on whether the compiler is able to optimize either or both of the return statement and the subsequent use in the caller.

Initialization and Finalization II

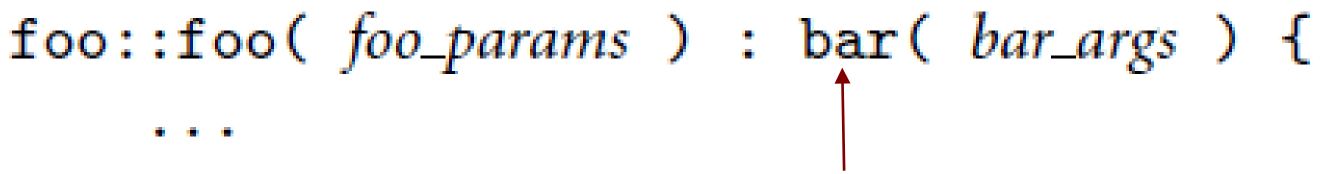
Execution Order

SECTION 10

Execution Order

Super Class Before this class

- C++ insists that every object be initialized before it can be used.
- If the object's class (call it B) is derived from some other class (call it A), C++ insists on calling an A constructor before calling a B constructor, so that the derived class is guaranteed never to see its inherited fields in an inconsistent state. [Super Class Constructor First]



```
foo::foo( foo_params ) : bar( bar_args ) {
    ...
```

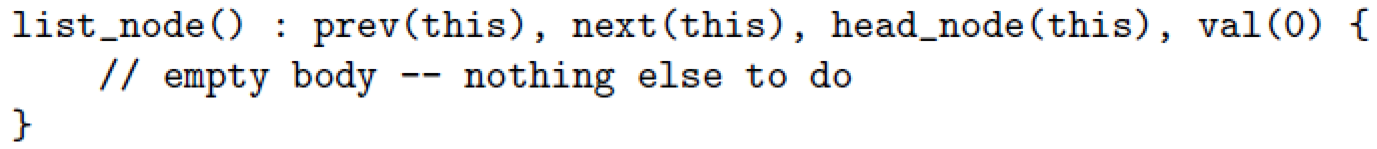
Super class

- **bar(bar_args)** is run first with its own parameter. Then the foo(foo_params) constructor.

Execution Order

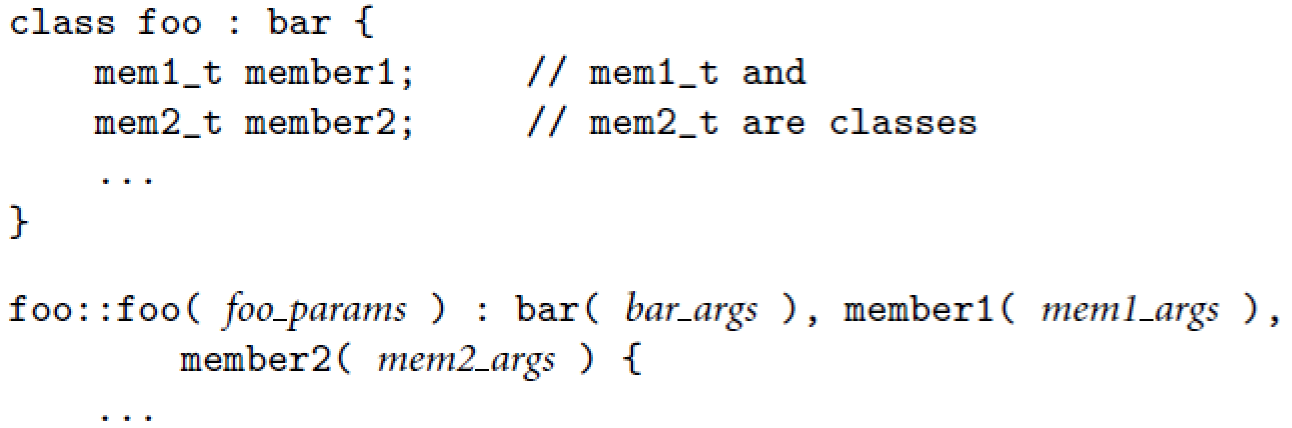
Member Constructors

C++ Allow Parameterized Member Objects: (Example for Passing Simple Value)



```
list_node() : prev(this), next(this), head_node(this), val(0) {
    // empty body -- nothing else to do
}
```

C++ Allow Parameterized Member Objects: (Example for Parameterized Constructors)



```
class foo : bar {
    mem1_t member1;     // mem1_t and
    mem2_t member2;     // mem2_t are classes
    ...
}

foo::foo( foo_params ) : bar( bar_args ), member1( mem1_args ),
        member2( mem2_args ) {
    ...
```

Note: The compiler call the copy constructors for member objects, rather than calling the default constructors, followed by operator= within the body of the constructor. Both semantics and performance may be different as a result.

Execution Order

Constructor Forwarding (call a() constructor is calling a(1) constructor)

Constructor Forwarding:

```
class list_node{
    ...
    list_node() : list_node(0) {}
```

- In Java, if A extends B

```
class B { int i; } // int = 0, float = 0.0, boolan = false, ref = null
class A { }
```

- Without explicit declaration, both A, B class have their own A() and B() constructor as default constructor.
- Without explicit declaration, A() will call super() which is B().
- If you want to explicitly define the A() constructor, you must write:

```
A(){
    super(args);  // B(args), this statement must be run before
                  // any other statements.
}
```

Garbage Collection

Reclaiming Space with Destructors

C++: (Using destructor)

- destructor for the derived class is called first, followed by those of the base class(es), in reverse order of derivation.

```
~queue(){
     while (!empty()){
         list_nod* p = contents.head();
         p-> remove();
         delete p;
     }
}
```

- Since **dequeue()** has already been designed to delete the node that contained the dequeued element:

```
~queue(){
    while( !empty()){
             int v = dequeue();
       }
}
```

Garbage Collection

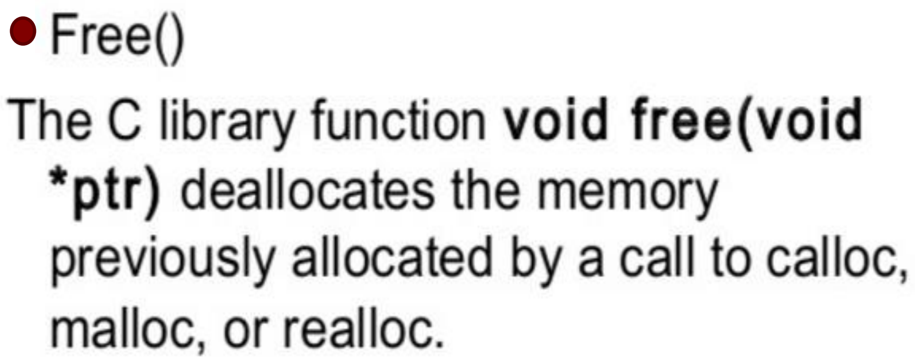
Reclaiming Space with Destructors

Java: (parking to null)

Return memory by explicit assignment:

```
  A a = new A();
  ….
  a = null;  // a's original object will be dangling
             // Then, the object body will be re-claimed
```

C: (no garbage collection, C uses a volunteering recycling)



- Free()

The C library function **void free(void *ptr)** deallocates the memory previously allocated by a call to calloc, malloc, or realloc.