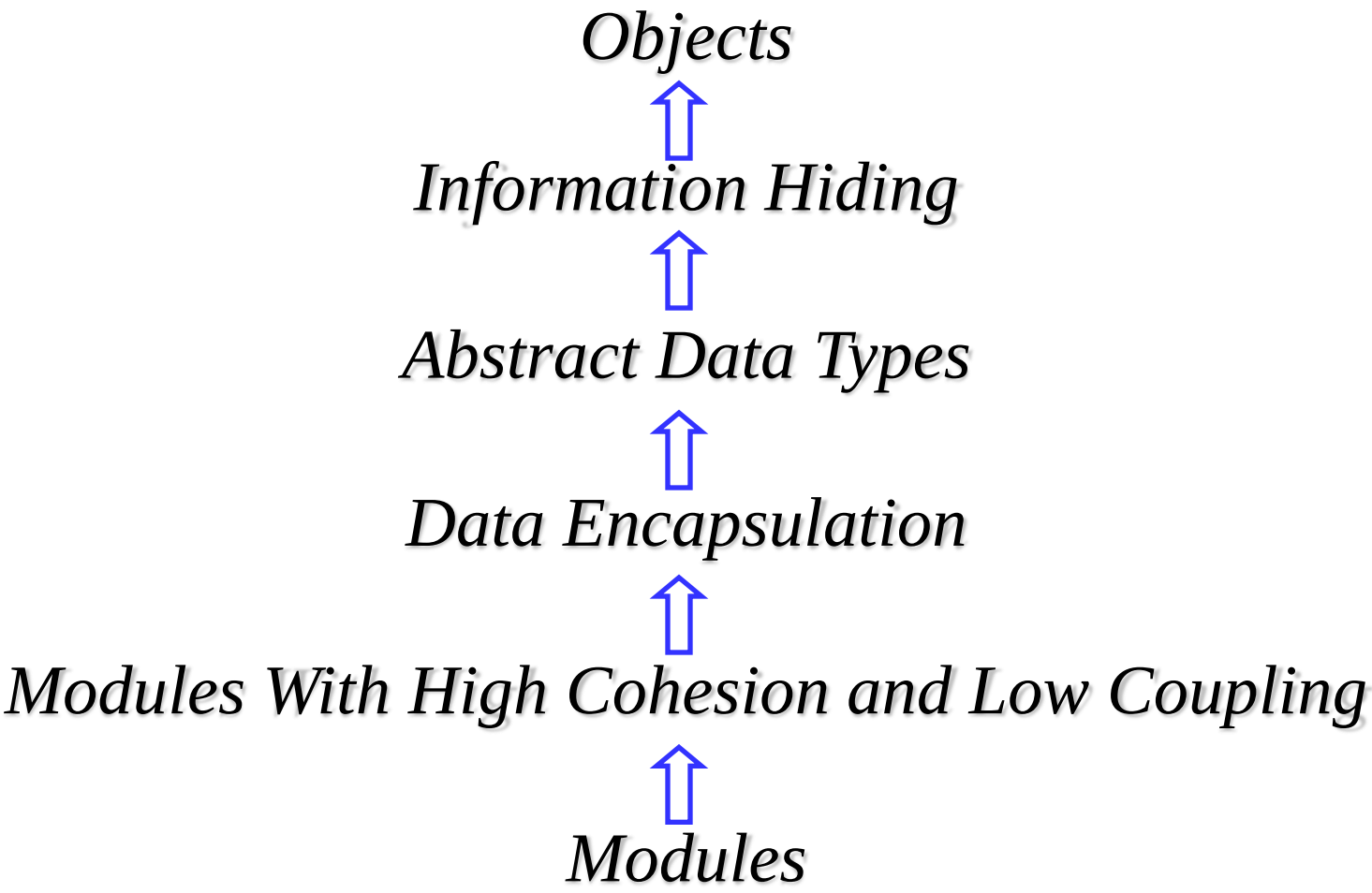
Lessons 34, 35, 36, 37– Effective Module Design

October 19, 21, 26, 28, 2015

*Chap7 (Schach) From Modules to Objects

The road to Object-Oriented programming:

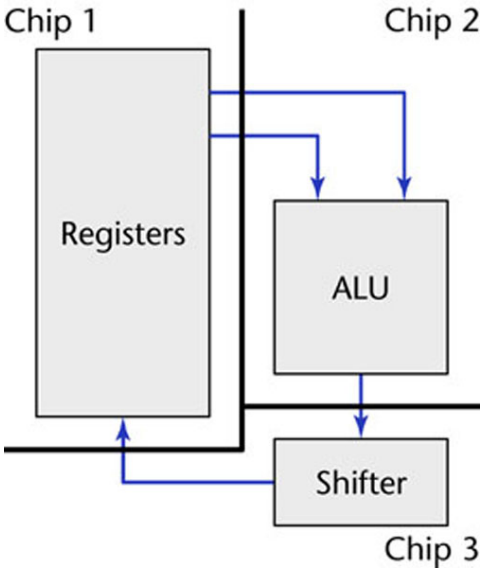


*All-In-One vs. Modular Program Construction

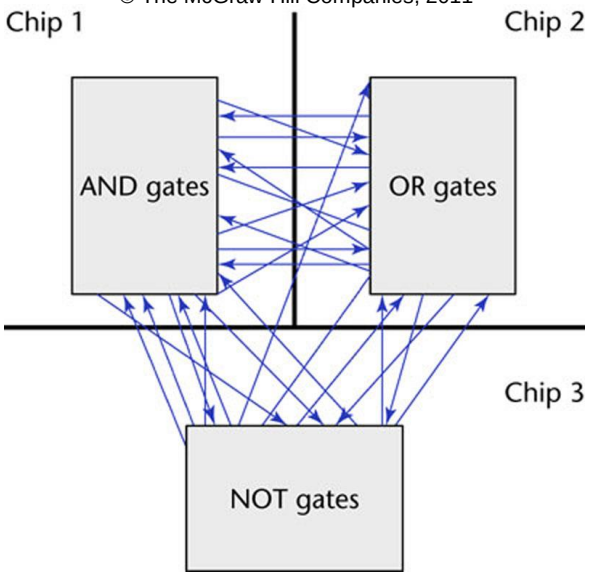
- Which implementation of a large software product is better?
  - a single monolithic block of code
  - a number of inter-related blocks or modules

Are all Modules Created Equally?

- Which chip design for a new computer is best?



OR



- Design containing 3 chips, 1 chip containing Register logic, 1 containing ALU logic, and 1 containing Shifter logic

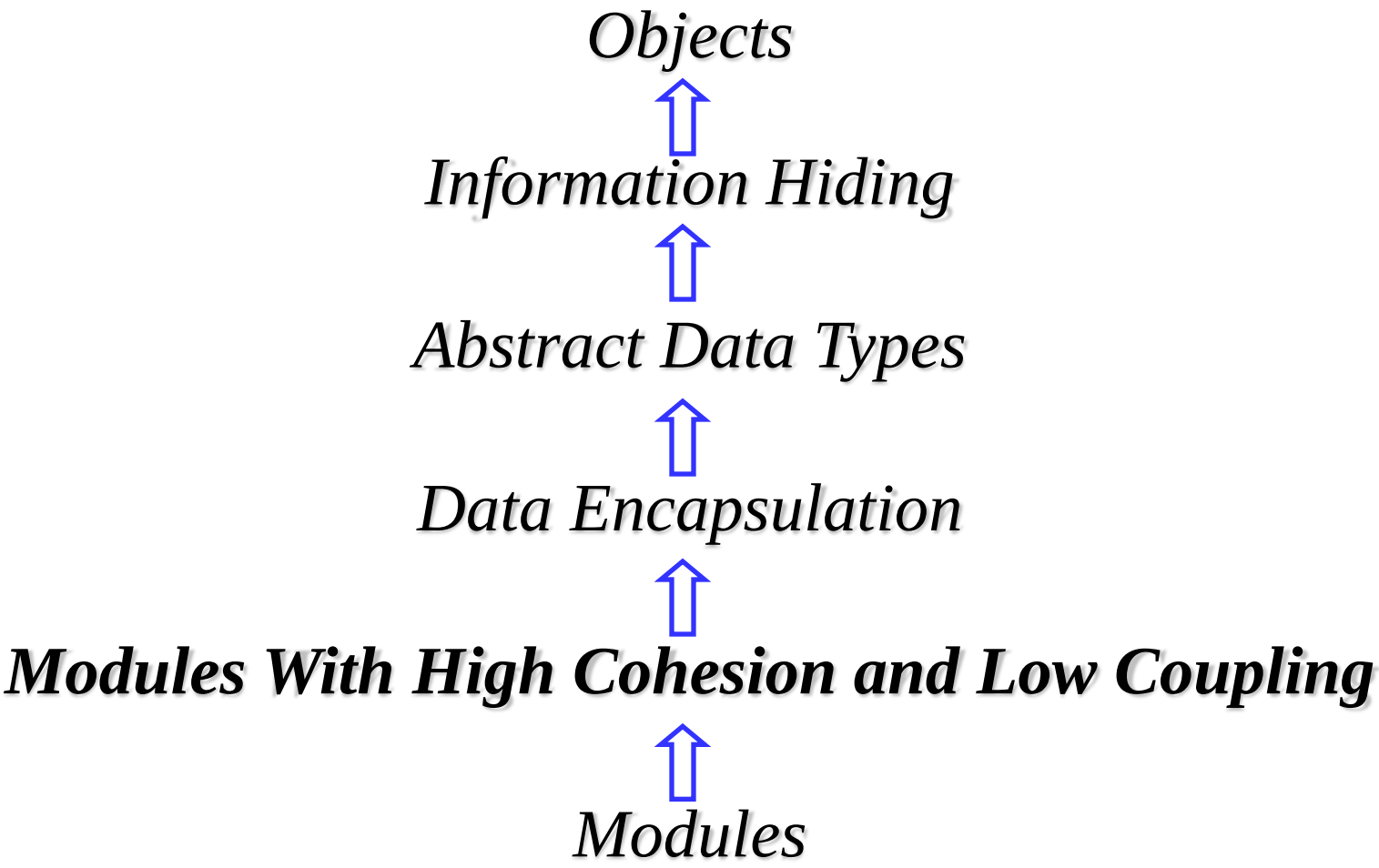
- Design containing 3 chips, 1 chip containing all of the AND gates, 1 containing all the OR gates, and 1 containing all the NOT gates

*Characteristic of a *Good* Module

- Modules should be:

*Chap7 (Schach) From Modules to Objects

The road to Object-Oriented programming:



*Relationships Within vs. Between Modules

- The "goodness" of a module can be defined in terms of two characteristics- cohesion and coupling
  - Module *cohesion* - degree of interaction within the module
  - Module *coupling* - degree of interaction between modules
- What kind of relationships do we want both within and between modules?

*Relationships Within vs. Between Modules

- What are some advantages of having strong relationships between components *within* modules?

- What are some advantages of having minimal relationships *between* modules:

*Cohesion: Why are these components together?

- *Ideal:* Module implements a single logical function or entity.
- All parts of module should contribute to this logical function.

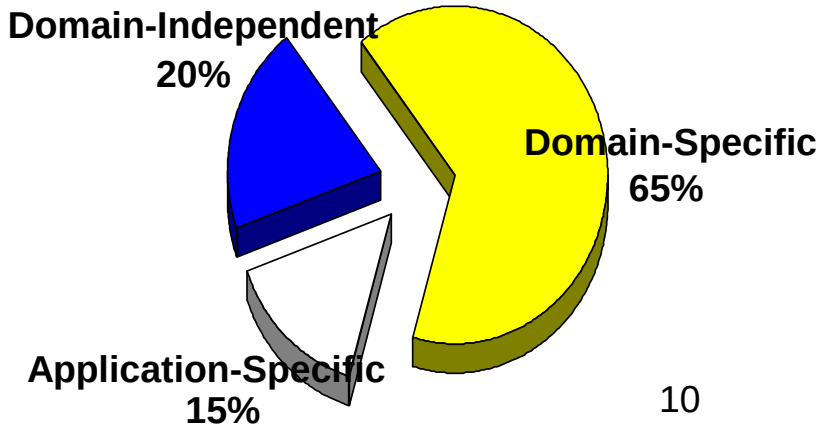
  - ***Low cohesion*** - module includes parts that are not directly related to its logical function
  - ***High cohesion*** - unit represents single part of problem solution
  - Why do we care about a module's cohesion?

7.	Informational cohesion	(Good)
6.	Functional cohesion	
5.	Communicational cohesion	
4.	Procedural cohesion	
3.	Temporal cohesion	
2.	Logical cohesion	
1.	Coincidental cohesion	(Bad)

What Impacts Software Reuse?

- Reuse is the process of taking components from one product, and using them to facilitate development of a different product
- **Goal:** Spend less on software production, but get increased quality.
- What (exactly) is reused?

- Typically, Software can be broken down into three areas that impact reuse:



*Seven Levels of Cohesion (starting from least desirable)

- ***(1) Coincidental*** cohesion
  - Parts of component are not related, just happen to be bundled into a single module:
  - Arises from rules like: "Every module will consist of between 35 and 50 statements"

```
int printNextLine(String inStr, int x, int y) {
        System.out.println(inStr);
         resetScreen();
         return x+y;
 }
```

- Can you tell what printNextLine() is intended to do from its name?
- Why is Coincidental Cohesion bad?

*(2) Logical Cohesion

- Module contains related actions, caller selects which one to use
  - Example 1:

```
int mySwitch(char opCode, int x, int y) {
   switch (opCode){
     case '+' : return (x+y);
     case '-' : return (x-y);
     case '*' : return (x*y);
     case '/' : return (x/y);
     default: return 0;}
}
```

  - Example 2: One Module performs all input and output

- Why is logical cohesion bad?

*(3) Temporal Cohesion

- Module performs series of actions related in time
  - Example: init( ) performs all functions that are necessary to be done before all other processing.

```
void init( ) {
    // code to open Account DB
    // code to open Transaction DB
    // code to reset Transaction Count
    println("Update in progress");
    return;
}
```

- Why is temporal Cohesion bad?

*(4) Procedural Cohesion

- Elements in module make up a single, *related* control sequence.
  - Example: sumOfTransactions( ) calculates the total debit and credit transaction amounts in one go.

```
void sumOfTransactions() {
    DB[0]=0; CR[0]=0;
    for (int i=1; i<= noAcct; i++){
        DB[0] += DB[i];   // DB is Debit transactions array
        CR[0] += CR[i];   // CR is Credit transactions array
    }
  return;
}
```

- Why is Procedural Cohesion bad?

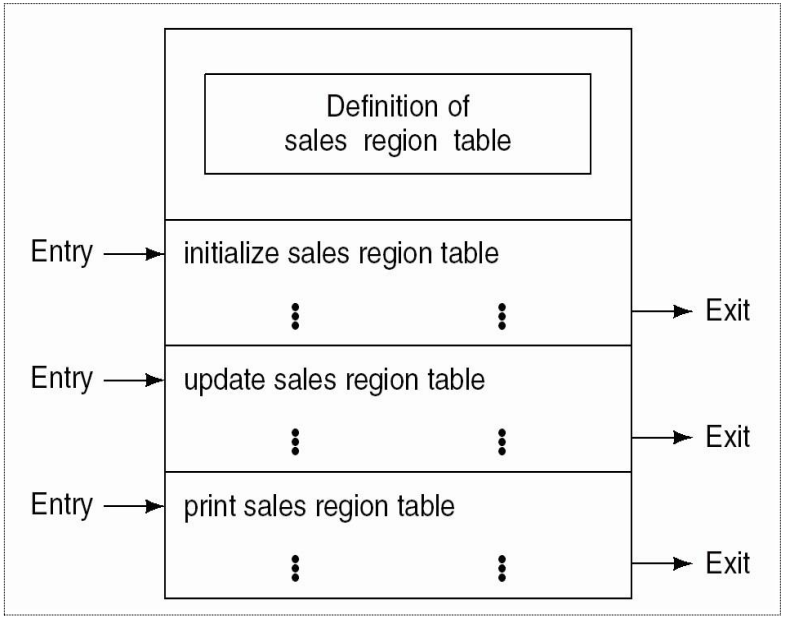
*(5) Communicational Cohesion

- Elements of module perform series of actions, and *all* operate on the same input data
  - Example 1
    - update record in database *and* write *it* to audit trail
  - Example 2
    - write error message to screen, then to error log file
- Why is Communicational Cohesion bad?

*Good forms of cohesion

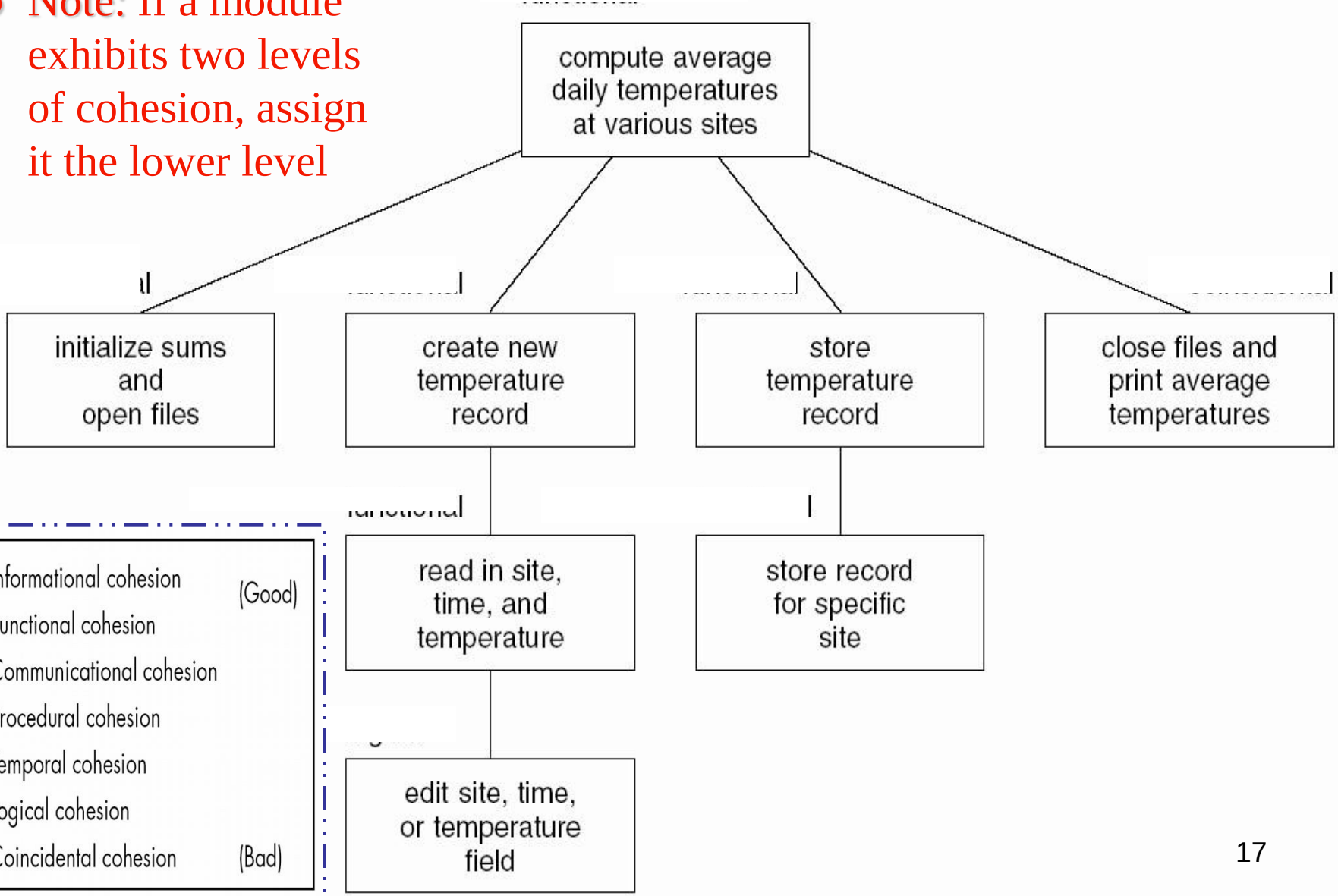
- **(6) Functional:** Each component is necessary for execution of ***one (and only one)*** function. Great for isolating faults.

- **(7) Informational:** Module performs actions, each with its own entry point, with independent code for each action, all performed on the same data structure
  - How is this different from Communicational cohesion?



~*ICE: Determine Cohesiveness based on Description

- Note: If a module exhibits two levels of cohesion, assign it the lower level



7.	Informational cohesion	(Good)
	Functional cohesion	
5.	Communicational cohesion	
4.	Procedural cohesion	
3.	Temporal cohesion	
2.	Logical cohesion	
1.	Coincidental cohesion	(Bad)

Does good cohesion come from just using OO?

- What form of cohesion *is* a feature of *properly designed* object-oriented systems?

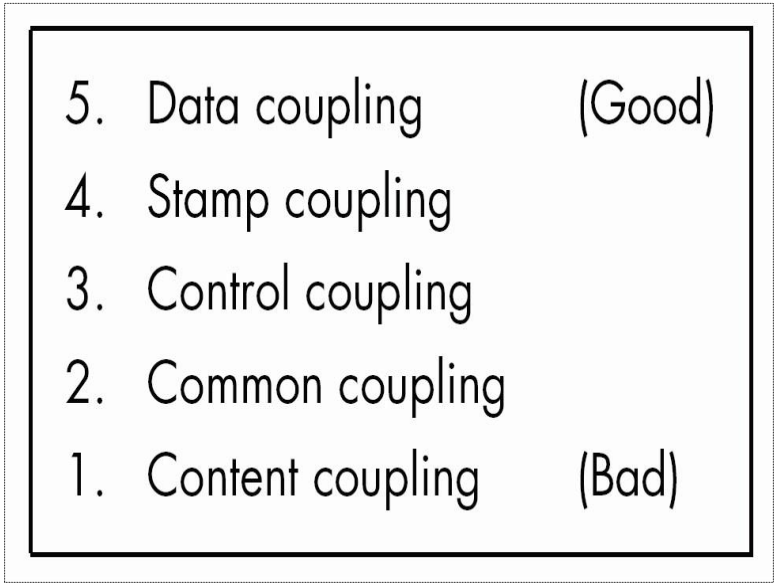
- How should we design OO classes in order to increase the cohesiveness of our classes?

Homework: Project Class's Cohesiveness

- Each person shall pick one module (class) from your course project (Project Manager and Team Leaders to coordinate). Include the detailed class diagram for the class and explain the level of cohesion exhibited by the class.

Coupling

- Degree of interaction *between* modules
- Five levels of coupling:



*(1) Content Coupling

- One module directly references the contents of the other.

```
public class Multiplier {
    public static int factor = 1;
    public static int productOf(int x) {
        return (x* factor); }}

public class MultiplierUser {
    int user() {
      Multiplier.factor=3;
      return Multiplier.productOf(2));}}
```

- Other examples?

*(2) Common coupling

- Modules directly access same data. Make use of shared variables and read/write to the shared variables.

```
int DB[25], CR[25];
void sumOfTran() {
    DB[0]=0; CR[0]=0;
    for (int i=1; i<= numAcct; i++){
        DB[0] += DB[i]; CR[0] += CR[i];
    }
    return;
}
```

Other examples:

```
while (global variable == 0)
{
    if (argument xyz > 25)
        module 3 ();
    else
        module 4 ();
}
```

*(3) Control Coupling

- One module tells another module what to do via the information it passes to the called module.

```
int mySwitch(char opCode, int x, int y) {
  switch (opCode){
    case '+' : return (x+y);
    case '-' : return (x-y);
    case '*' : return (x*y);
    case '/' : return (x/y);
    default: return 0;}
}
```

- The caller of mySwitch() passes a control flag and is therefore control-coupled with mySwitch().
- Why is this bad?

*(4) Stamp Coupling

- A data structure is passed as an argument, but the called function operates only on part(s) of that data structure. Example,

```
void updateName(BankAccount thisAccount, String newName)
{
   thisAccout.name = newName;
}
```

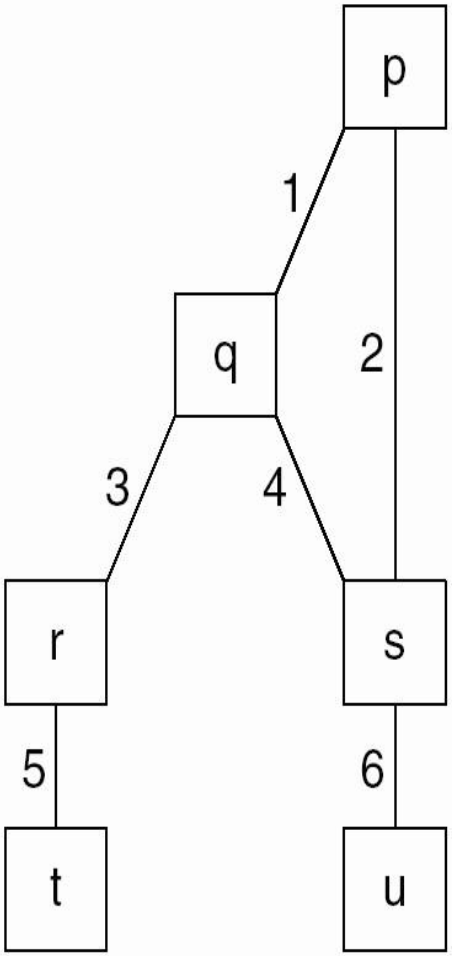
*(5) Data Coupling

- All arguments are either simple argument or a data structure in which **all** elements are used by the called module.

```
int sumOfArray(int number[]){
    int result = 0;
    for (i=0; i< number.len; i++)
        result += number[i];
    return result;
}
```

- Is data coupling desirable for OO Design? Why?

~*ICE: Determine Couplings given Interface descriptions



Example: 1. When p calls q's interface, p passes one argument, an aircraft type, and q passes back a "status flag".

p, t, and u access the same database in update mode

Number	In	Out
1	aircraft type	status flag
2	list of aircraft parts	—
3	function code	—
4	list of aircraft parts	—
5	part number	part manufacturer
6	part number	part name

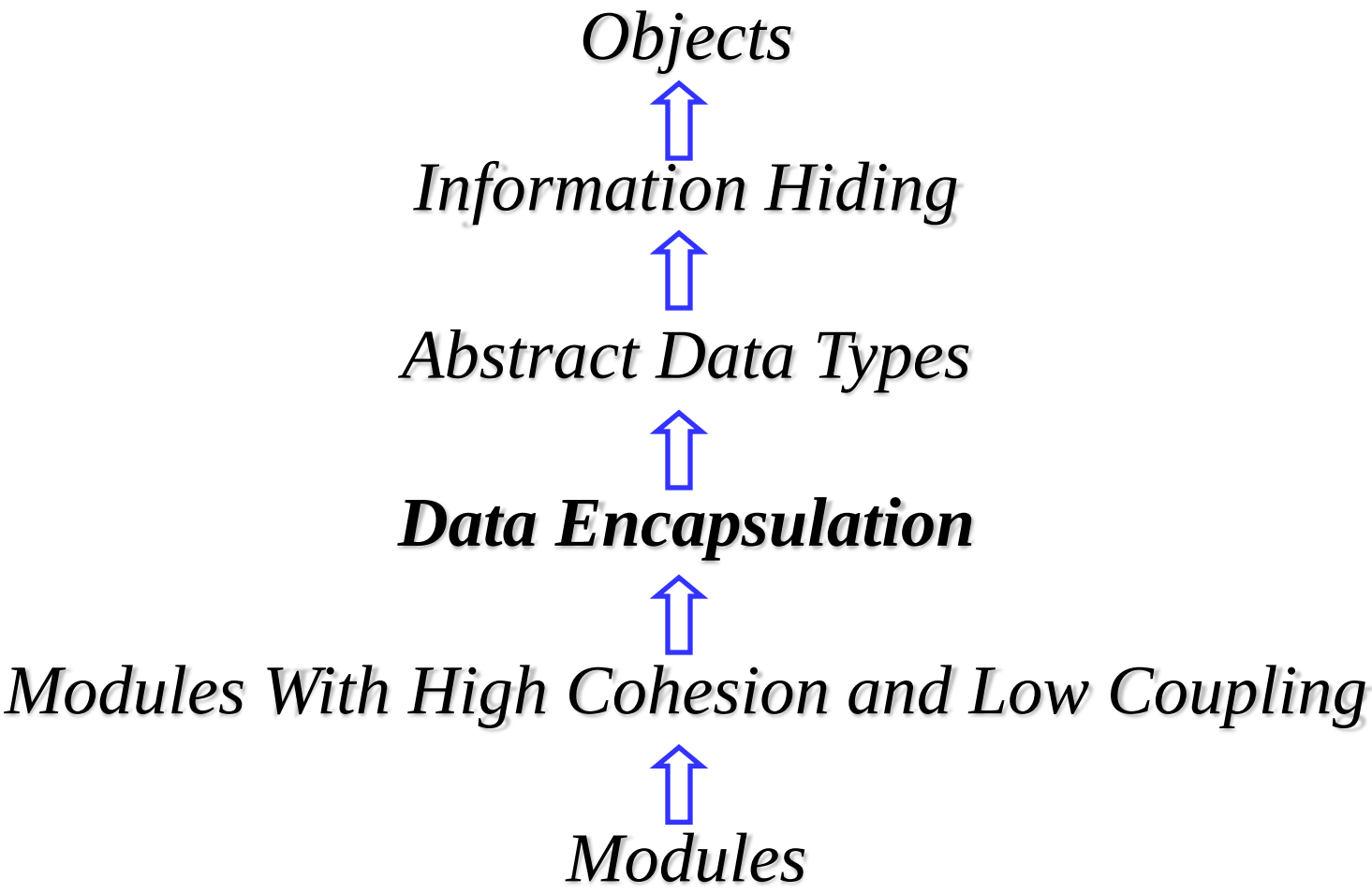
5. Data coupling (Good)
4. Stamp coupling
3. Control coupling
2. Common coupling
1. Content coupling (Bad)

Homework: Project Class Coupling

- Using your interaction diagrams as a guide, each person shall pick two modules (classes) from your course project that interact (Project Manager / Team Leaders to coordinate).
- Submit detailed class diagrams for the classes and explain the level of coupling exhibited between the methods of the two classes, justifying why you chose the indicated level of coupling.

*Chap7 (Schach) From Modules to Objects

The road to Object-Oriented programming:



Data Encapsulation

- Example
  - Design an operating system for a large mainframe computer. Batch jobs submitted to the computer will be classified as high priority, medium priority, or low priority. There must be three queues for incoming batch jobs, one for each job type. When a job is submitted by a user, the job is added to the appropriate queue, and when the operating system decides that a job is ready to be run, it is removed from its queue and memory is allocated to it.

Data Encapsulation — Design 1

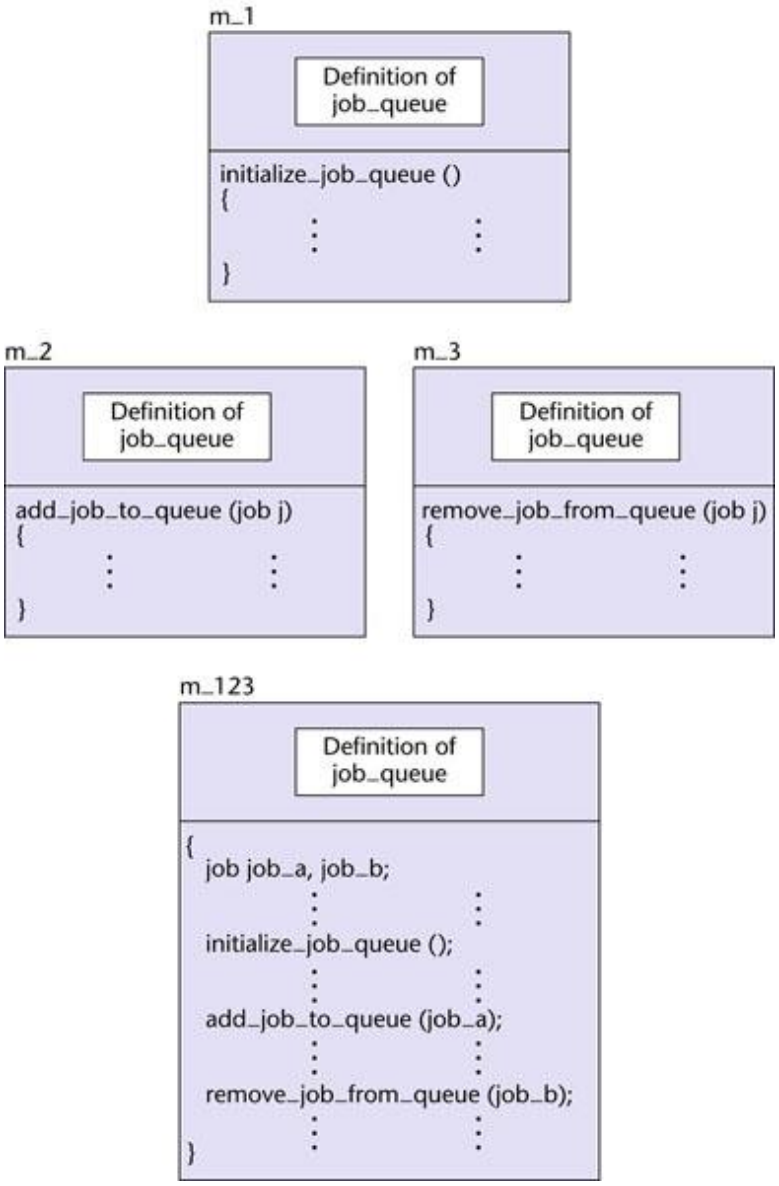


Figure 7.15

Data Encapsulation — Design 2

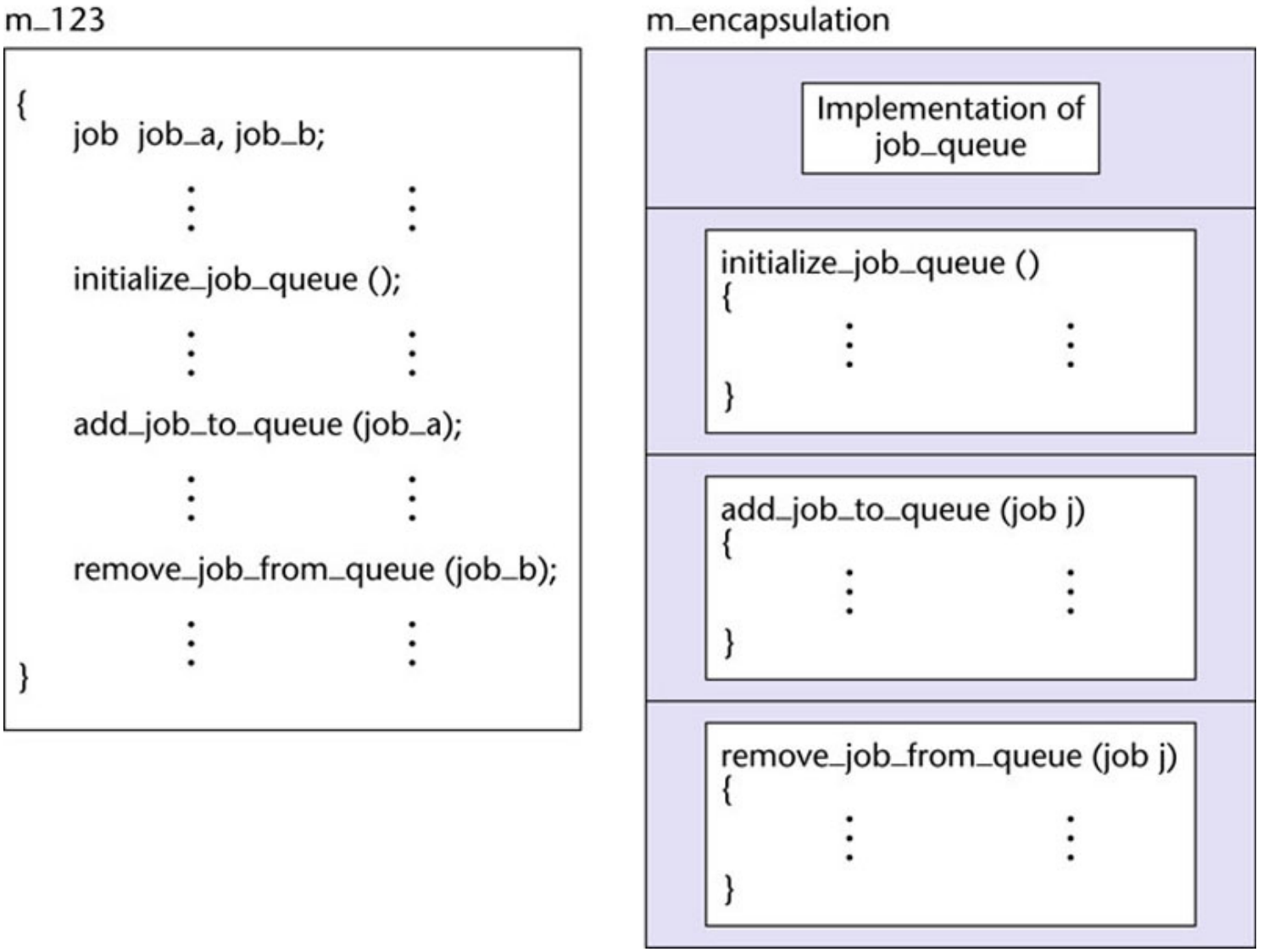


Figure 7.16

Data Encapsulation (cont.)

- `m_encapsulation` has informational cohesion
- `m_encapsulation` is an implementation of data encapsulation
- Advantages
  - Development
  - Maintenance

Data Encapsulation and Development

- Data encapsulation is an example of *abstraction*
- Job queue example:
  - Data structure
    - `job_queue`
  - Three new functions
    - `initialize_job_queue`
    - `add_job_to_queue`
    - `remove_job_from_queue`

Data Encapsulation and Development

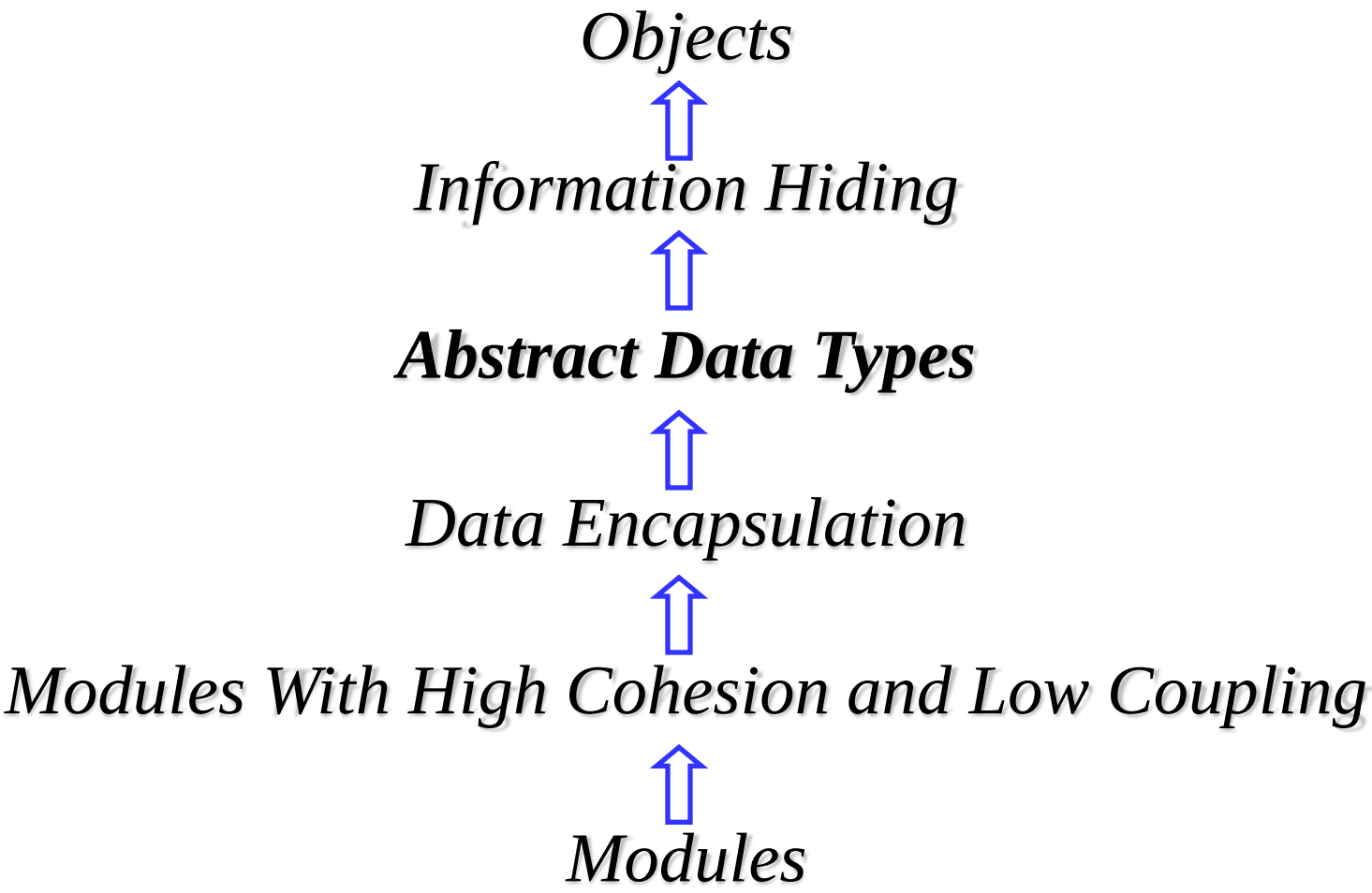
- Abstraction
  - Conceptualize problem at a higher level
    - Job queues and operations on job queues
  - Not a lower level
    - Records or arrays

Data Encapsulation and Maintenance

- Data encapsulation allows you to design the product to minimize the effects of future changes
  - Need for job queues not likely to change
    - Job queue with operations initialize, add and delete
  - Implementation of job queue may change
    - arrays vs. linked list
  - Data encapsulation enables you to change method of implementation without effecting modules that use job queue

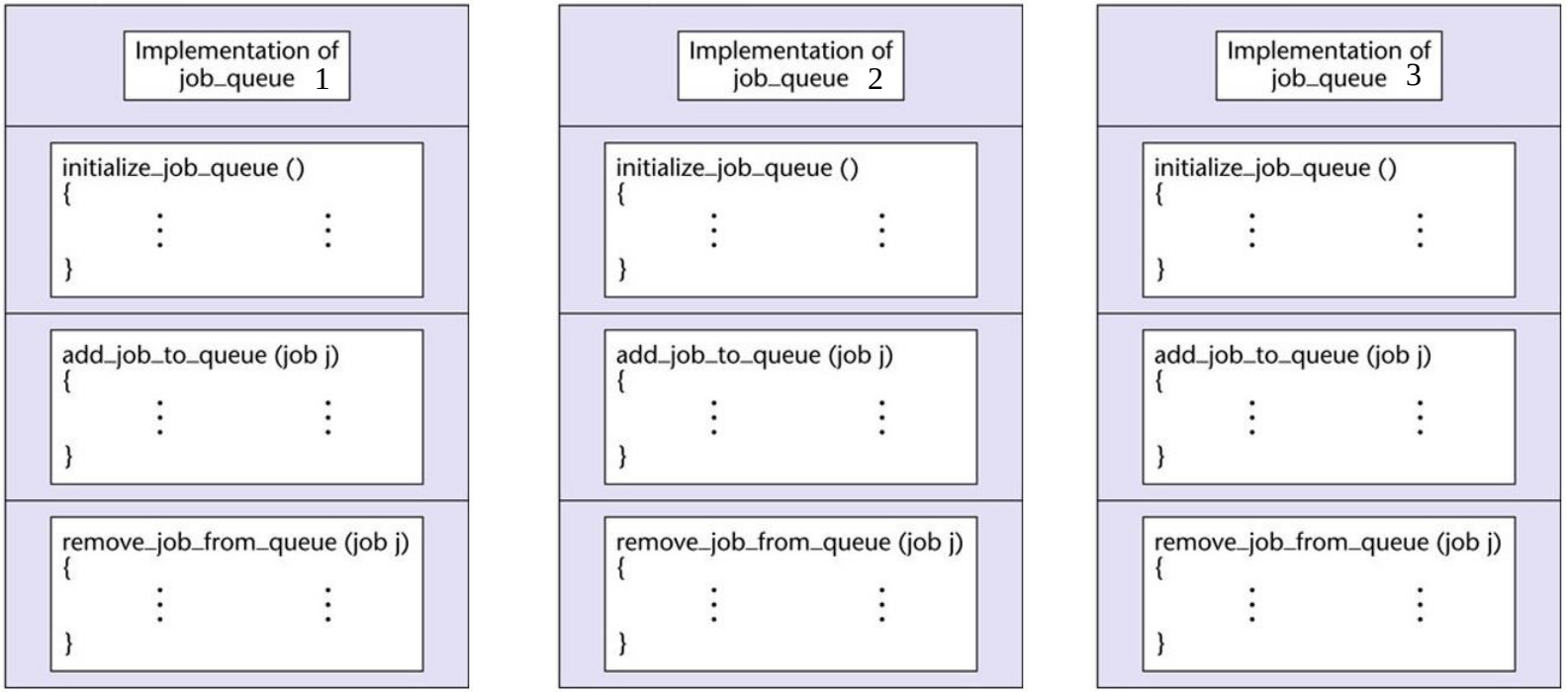
*Chap7 (Schach) From Modules to Objects

The road to Object-Oriented programming:



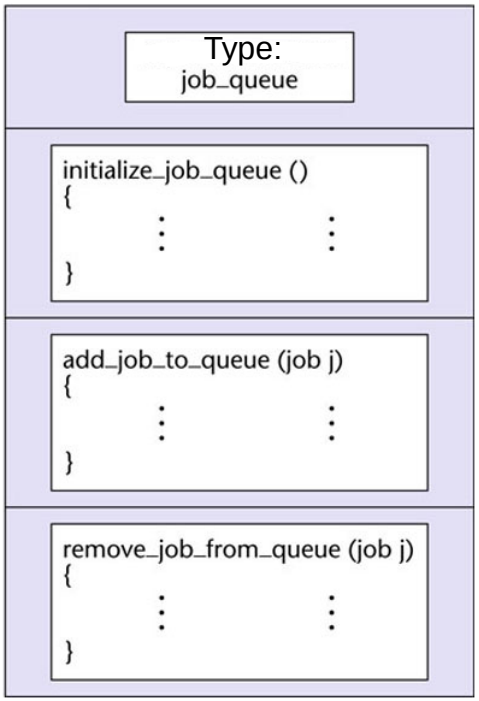
Abstract Data Types

- What if we need three queues, and not just one?
  - `m_encapsulation` provides a data structure and operations for a single `job_queue`



Abstract Data Types

- We need:
  - to define data type for `job_queue` that contains data structure + operations performed on that data structure that we can instantiate to provide multiple occurrences of `job_queue`




```
job_queue queue1, queue2, queue3;
```

Type **Instances**

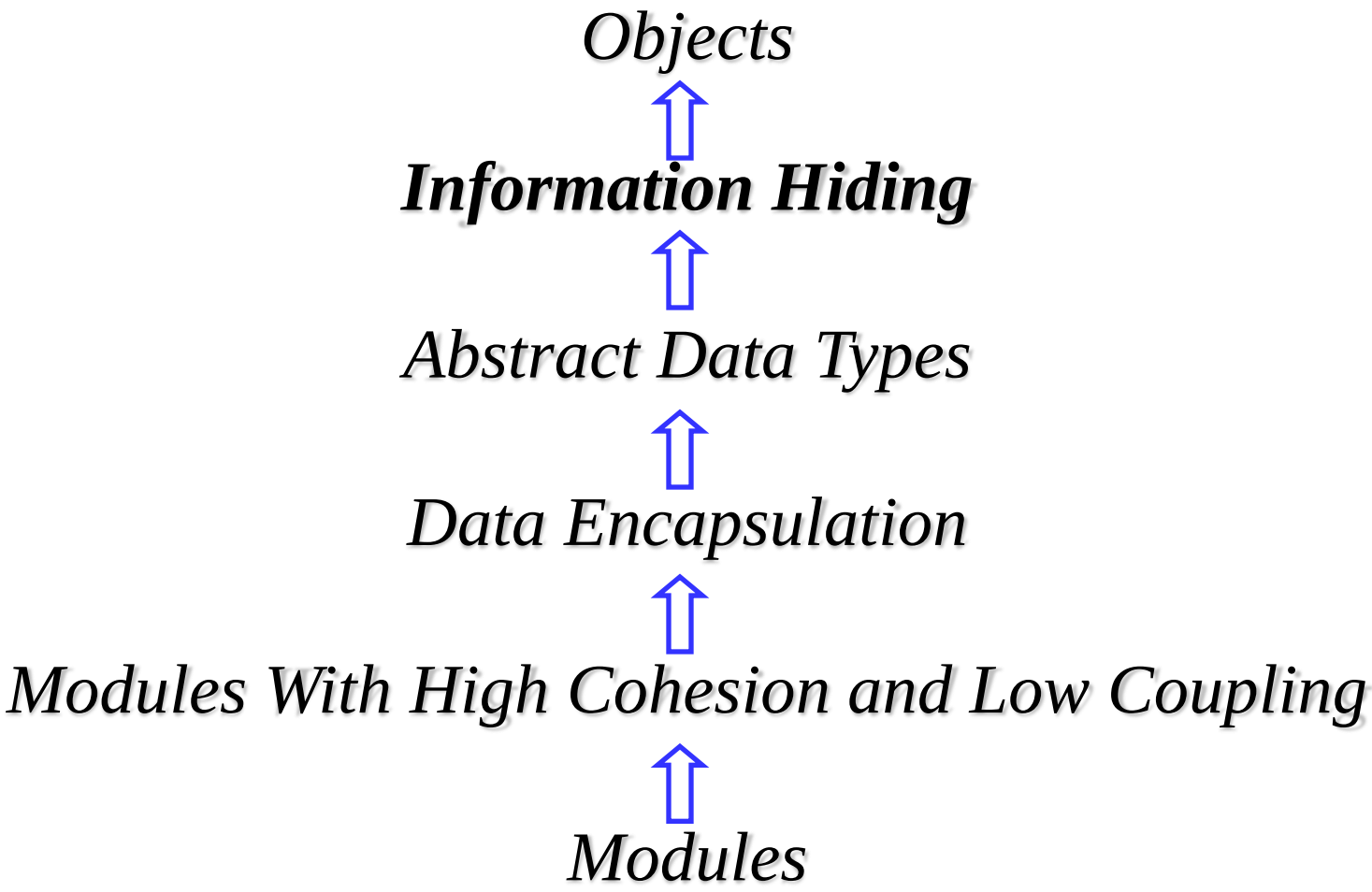
Abstract Data Type Example

- Java abstract data type implementation of job queue

```
class JobQueue
{
   // attributes

   public int queueLength;  // length of job queue
   public int queue[ ] = new int[25];
                                  //  queue can contain up to 25 jobs

   // methods
   public void initialize jobQueue()
    {
        // details hidden
    }

   public void addjobTo Queue(int jobNumber)
     {
         // details hidden
     }

   public int removejobFromQueue()
     {
         // details hidden
     }
}// class JobQueue
```

*Chap7 (Schach) From Modules to Objects

The road to Object-Oriented programming:



Information Hiding

- Data abstraction
  - The designer thinks at level of an ADT
- Procedural abstraction
  - Define a procedure — extend the language
- Both are instances of a more general design concept, *information hiding*

Information Hiding (cont.)

- Java abstract data type implementation with information hiding

```
class JobQueue
{

   // attributes
   private int queueLength;  // length of job queue
   private int queue[ ] = new int[25];
                                  // queue can contain up to 25 jobs

   // methods
   public void initialize jobQueue()
    {
        // details hidden
    }

   public void addjobTo Queue(int jobNumber)
     {
         // details hidden
     }

   public int removejobFromQueue()
     {
         // details hidden
     }
}// class JobQueue
```

Information Hiding (cont.)

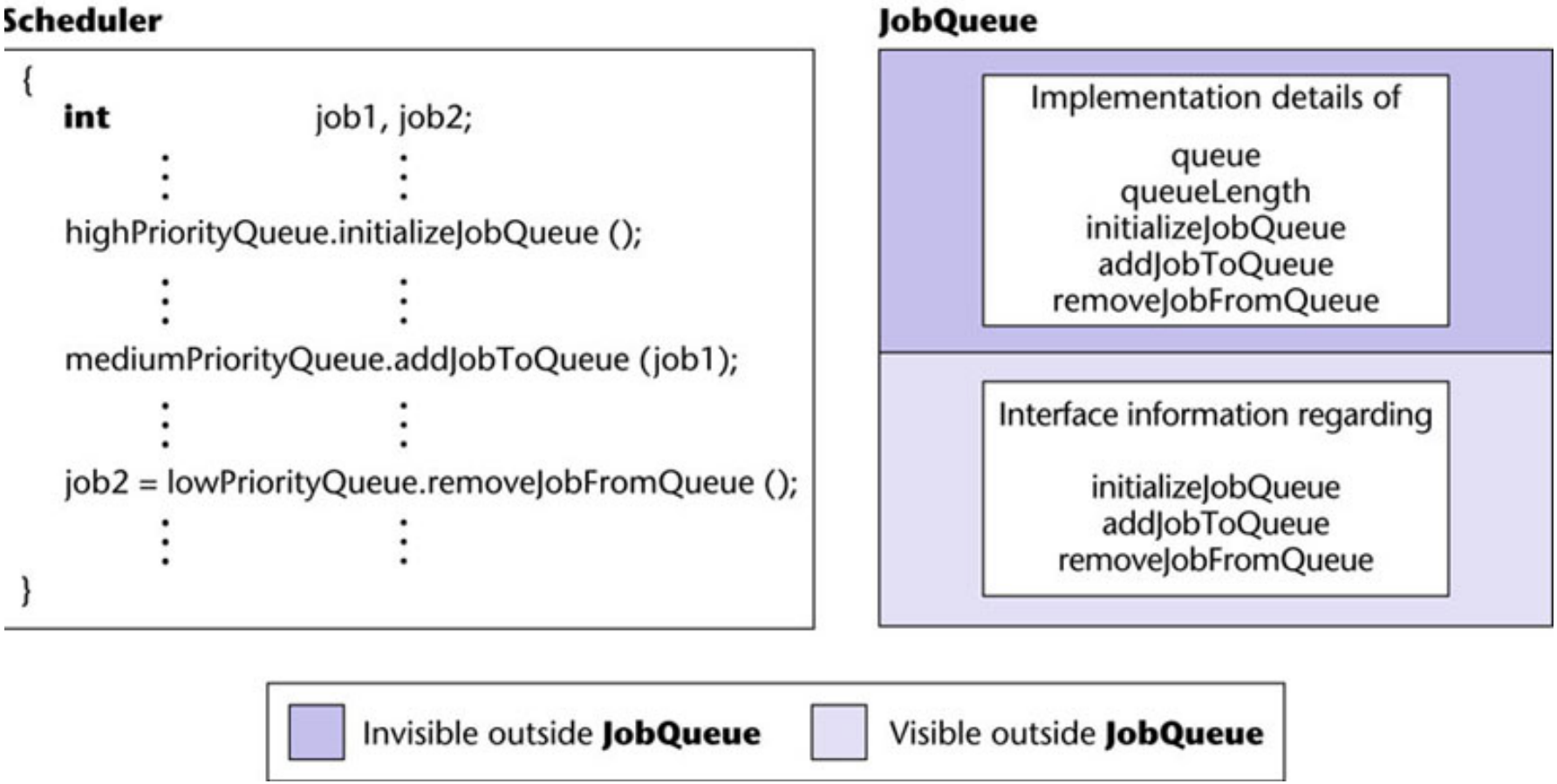
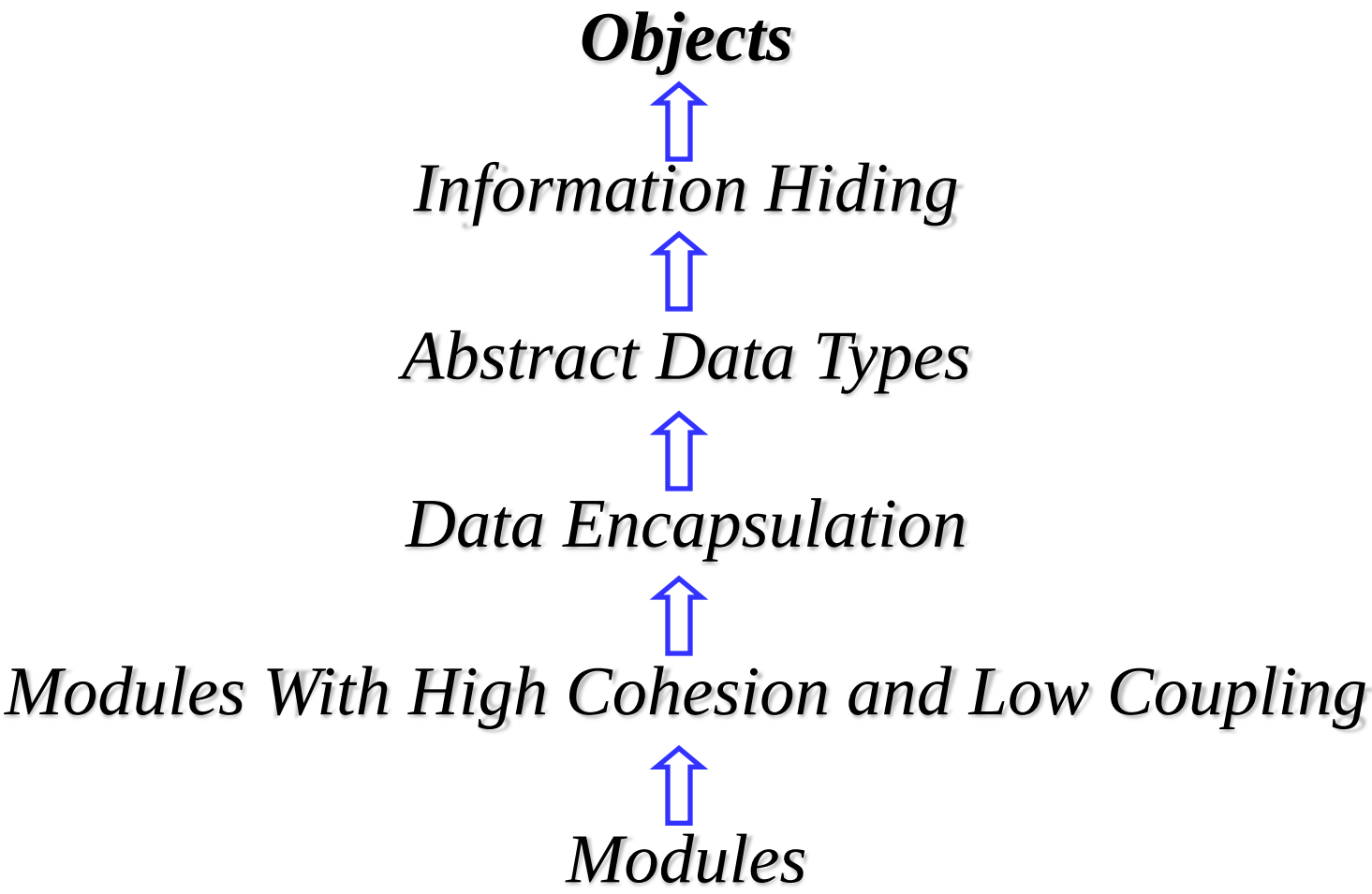


Figure 7.27

- Effect of information hiding via **`private`** attributes

*Chap7 (Schach) From Modules to Objects

The road to Object-Oriented programming:



Objects

- First refinement
  - Product designed in terms of abstract data types with information hiding
  - Variables ("objects") are instantiations of abstract data types
- Second refinement
  - Class: abstract data type with information hiding that supports *inheritance*
  - Objects are instantiations of classes

Inheritance

- Define **`HumanBeing`** to be a *class*
  - A **`HumanBeing`** has *attributes*, such as
    - `age, height, gender`
  - Assign values to the attributes when describing an object

- Define **`Parent`** to be a *subclass* of **`HumanBeing`**
  - A **`Parent`** has all the attributes of a **`HumanBeing`**, plus attributes of his/her own
    - `nameOfOldestChild, numberOfChildren`
  - A **`Parent`** inherits all attributes of a **`HumanBeing`**

Inheritance (cont.)

- The property of inheritance is an essential feature of all object-oriented languages
  - Such as Smalltalk, C++, Ada 95, Java
- But not of classical languages
  - Such as C, COBOL or FORTRAN

Inheritance (cont.)

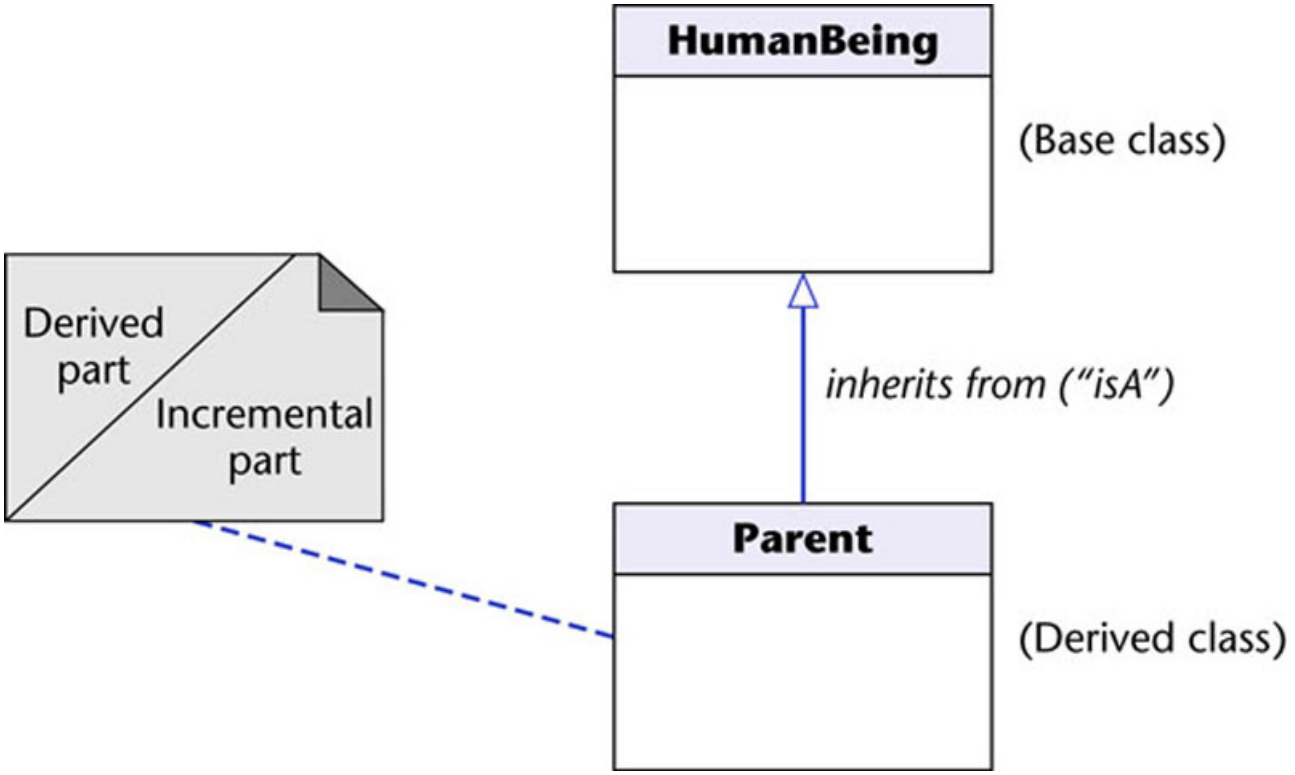


Figure 7.29

- UML notation
  - Inheritance is represented by a large open triangle

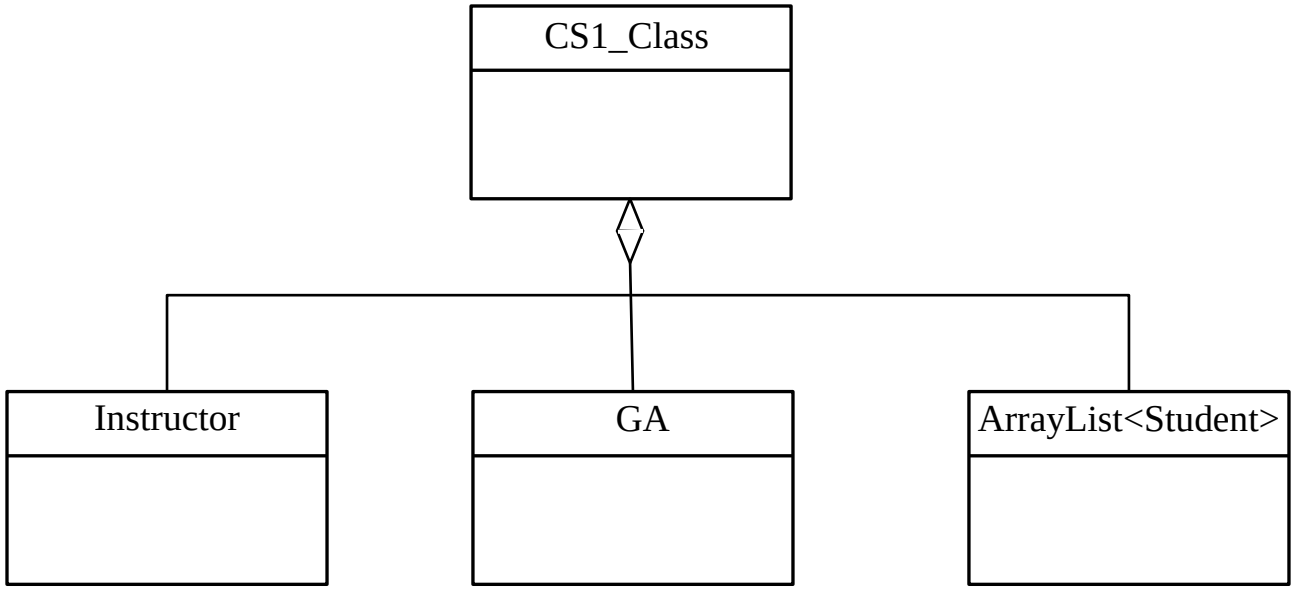
Java Implementation

```
class HumanBeing
{
   private int      age;
   private float    height;

   // public declarations of operations on HumanBeing

}// class HumanBeing


class Parent extends HumanBeing
{
   private String   nameOfOldestChild;
   private int      numberOfChildren;

   // public declarations of operations on Parent

}// class Parent
```

Figure 7.30

Aggregation

```
public class CS1_Class {
        private Instructor teacher;
        private GA gradAssist;
        private ArrayList<Student> studentArray;
}
```



- UML notation for aggregation — open diamond

Association

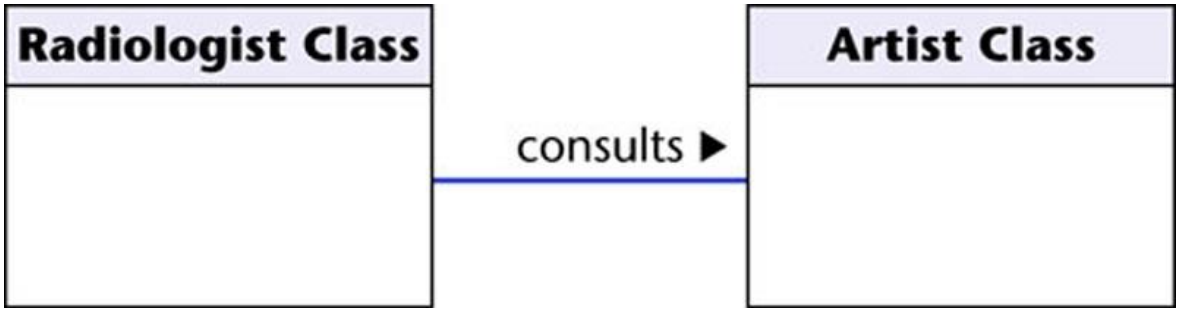


Figure 7.32

- UML notation for association — line
  - Optional navigation triangle

Inheritance, Polymorphism and Dynamic Binding

Figure 7.33a

- Classical paradigm
  - We must explicitly invoke the appropriate version

Inheritance, Polymorphism and Dynamic Binding (cont.)

- Classical code to open a file
  - The correct method is explicitly selected

```
switch (file_type)
{
    case 1:
        open_disk_file ( );                    // file_type 1 corresponds to a disk
        break;
    case 2:
        open_tape_file ( );                    // file_type 2 corresponds to a tape
        break;
    case 3:
        open_diskette_file ( );                // file_type 3 corresponds to a disk
        break;
}
```

Figure 7.34(a)

Inheritance, Polymorphism and Dynamic Binding (cont.)

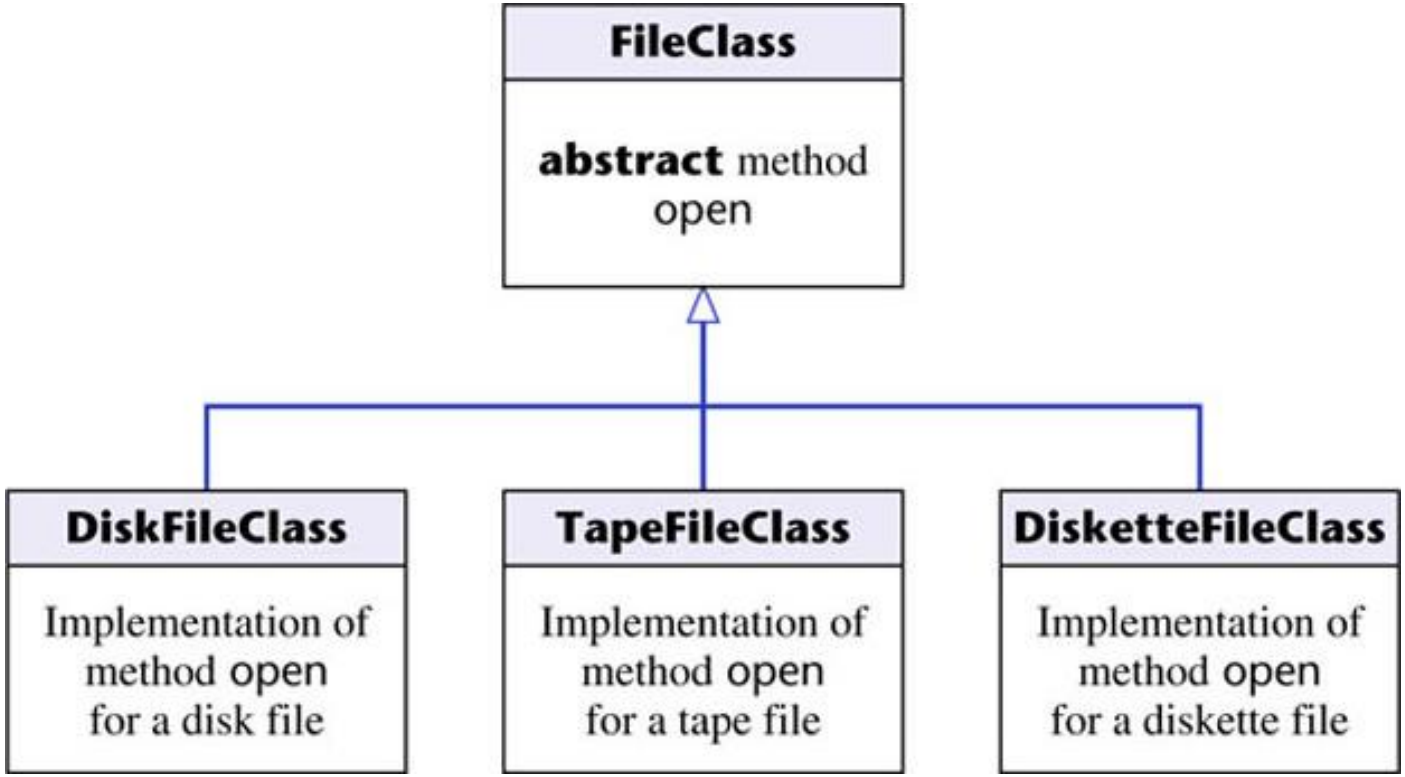


Figure 7.33(b)

- Object-oriented paradigm

Inheritance, Polymorphism and Dynamic Binding (cont.)

- Object-oriented code to open a file
  - Correct method invoked at run-time (dynamically)

```
DiskFileClass myFile;  // or TapeFileClass or DisketteFileClass
myMethod(myFile);  // Call to myMethod, passing myFile as parm.
        .
        .
        .
public void myMethod( FileClass thisFile) {
    thisFile.open();  // dependent on type of thisFile
 }
```

- Method `open` can be applied to objects of different classes
  - "Polymorphic"

Inheritance, Polymorphism and Dynamic Binding (cont.)

- **Polymorphism:**

- **Dynamic Binding:**

- **C++**: through explicit use of "virtual member functions".
- **Java**: all methods are *implicitly* dynamically bound, unless keyword “final” is used.

Inheritance, Polymorphism and Dynamic Binding (cont.)

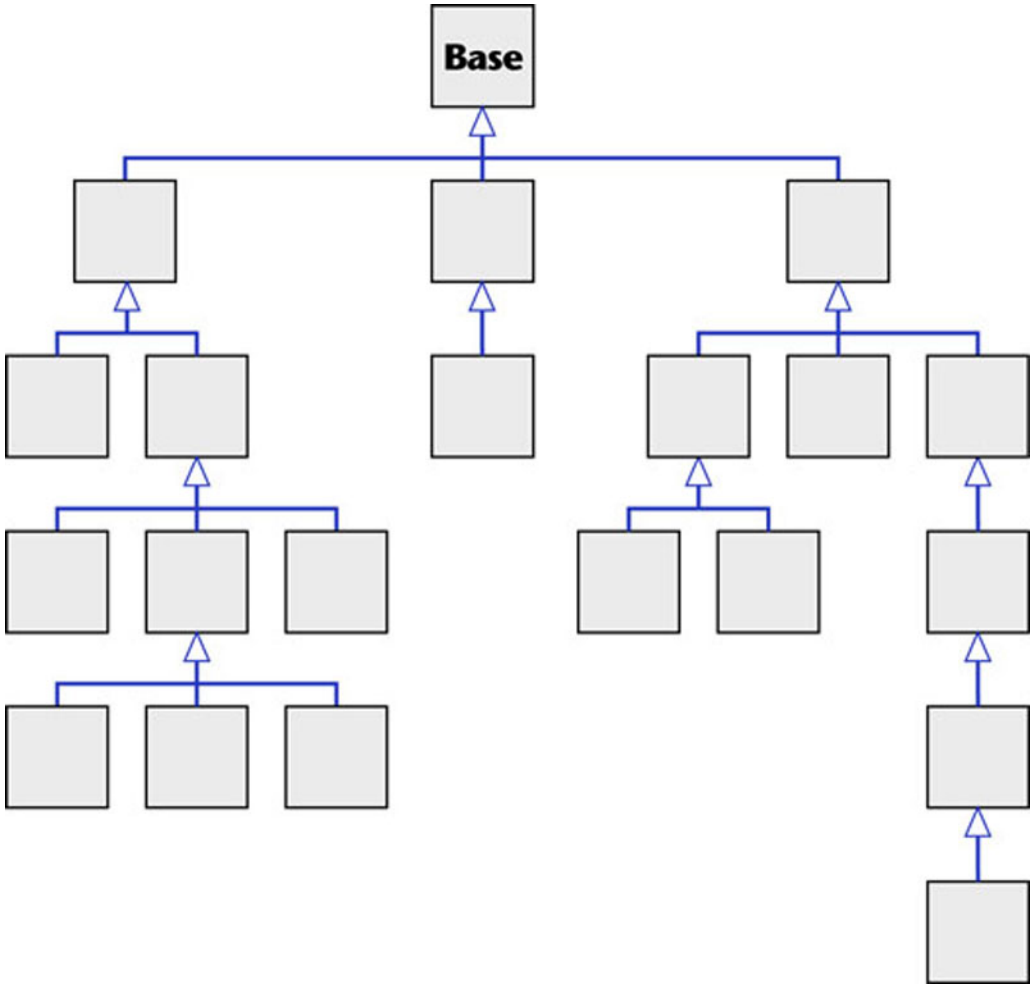


Figure 7.35

- Method `checkOrder (b : Base)` can be applied to objects of any subclass of **`Base`**

*Why Bother with Dynamic Binding?

- Useful when a child class overrides a method of parent, and class of each object is not obvious at compile time.
  - Provides mechanism for selecting between alternatives; more robust than explicit selection by alternatives.
  - Dynamic binding involves overhead:

- How does dynamic binding effect reusability?

```
public void myMethod( FileClass thisFile) {
   thisFile.open();  // which method called?
 }
```

*ICE: What is output by the following code?

```
class Q {
  protected int data;
  public int aMethod( Q q) {
     return q.data+ 1;
  }
  public Q( int x) {
     data= x;
  }
} // end Q

class R extends Q {
  public int aMethod( R r) {
    return r.data- 1;
  }
  public R( int x) {
    super( x);
  }
} // end R
```

```
public class Test {
 public static void main( String[] s) {
  Q q= new Q( 1);
  R r= new R( 10);
  System.out.println( r.aMethod( q ) );  // 1
  System.out.println( r.aMethod( r ) );  // 2
  System.out.println( q.aMethod( q ) );  // 3
  System.out.println( q.aMethod( r ) );  // 4
  } // end main
} // end test
```

*Impact of Polymorphism & Dynamic Binding

- What is the impact of Polymorphism and Dynamic Binding on Maintenance?

*Revisit Cohesion and Coupling as Related to OO

- Do we need expanded levels of Cohesion and Coupling to Account for OO's Inheritance, Polymorphism, and Dynamic Binding?
- What is OO's impact:
  - On Cohesion?
  - On Coupling?

- **Cohesion** - How well related are a class's methods.

- **Coupling** - Relationships *between* classes.
  - Inheritance: attributes/behaviors of parent class are present in child class, may be tailored via polymorphism.

*Conditions for software development with reuse

- Must be possible to find appropriate reusable components.
  - Re-user must have confidence that the components will behave as specified and will be reliable.
  - Components must have associated documentation to help re-user understand and adapt them
- What are possible adverse affects of inheritance on reuse?