今天的内容

- Classes and objects – part I
- Evolution of object model
- Foundations of the object model
- Elements of object model
  - abstraction

Class & Objects Part I

齐琦

+

- Scala: hybrid object-functional language; support both object-oriented & functional programming paradigms
- Fields
  - Vals, vars to store data
- Methods
  - Operations to perform
- Creating an object/ creating an instance of an object
  - Create a val or var of a class

- Container or collection
  - Especially useful type of object
  - An object that holds other objects
  - E.g. create a Vector holding Doubles

```
val v1 = Vector(19.2, 88.3, 22.1)
```

  - V1.reverse; v1.sorted; v1.max; v1.min

+
Creating Classes

- A class definition includes:
  - Class keyword
  - A name for the class
  - An optional body; it contains:
    - Field definitions (vals and vars)
    - Method definitions
    - Code that executed during creation of each object
- Use new keyword to create an instance of a class
- Fields can be any type

```
class NoBody
val nb = new NoBody

class SomeBody {
  val name = "Janet Doe"
  println(name + " is SomeBody")
}
val sb = new SomeBody

class EveryBody {
  val all = Vector(new SomeBody,
    new SomeBody, new SomeBody)
}
val eb = new EveryBody
```

+

Class with methods

- Methods have access to members of the class: current, scale
- The val for temp: it prevents reference temp from being reassigned to a new object; not restrict the behavior of the object itself

```
// Temperature.scala
import com.atomicscala.AtomicTest._

class Temperature {
  var current = 0.0
  var scale = "f"
  def setFahrenheit(now:Double):Unit = {
    current = now
    scale = "f"
  }
  def setCelsius(now:Double):Unit = {
    current = now
    scale = "c"
  }
  def getFahrenheit():Double = {
    if(scale == "f")
      current
    else
      current * 9.0/5.0 + 32.0
  }
  def getCelsius():Double = {
    if(scale == "c")
      current
    else
      (current - 32.0) * 5.0/9.0
  }
}

val temp = new Temperature
temp.setFahrenheit(98.6)
temp.getFahrenheit() is 98.6
temp.getCelsius is 37.0
temp.setCelsius(100.0)
temp.getFahrenheit is 212.0
```

Another example: tic-tac-toe game

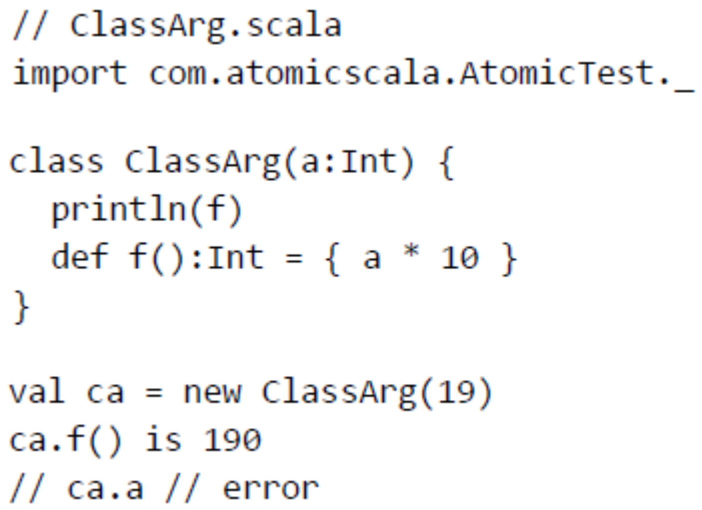
```scala
class Cell {
  var entry = ' '
  def set(e:Char):String = {
    if(entry==' ' && (e=='X' || e=='O')) {
      entry = e
      "successful move"
    } else
      "invalid move"
  }
}

class Grid {
  val cells = Vector(
    Vector(new Cell, new Cell, new Cell),
    Vector(new Cell, new Cell, new Cell),
    Vector(new Cell, new Cell, new Cell)
  )
  def play(e:Char, x:Int, y:Int):String = {
    if(x < 0 || x > 2 || y < 0 || y > 2)
      "invalid move"
    else
      cells(x)(y).set(e)
  }
}
```

```scala
val grid = new Grid
grid.play('X', 1, 1) is "successful move"
grid.play('X', 1, 1) is "invalid move"
grid.play('O', 1, 3) is "invalid move"
```

Class arguments

- When create new object, pass information into that object to initialize it
- Like a method argument list, but placed after the class name
- “new” requires an argument
- Initialization of “a” happens before we
  Enter the class body
- All definitions(values & methods) are
  Initialized before rest of the body is executed
- “a” not accessible outside class body
  - Want it to visible, declare it as var / val in
  Argument list

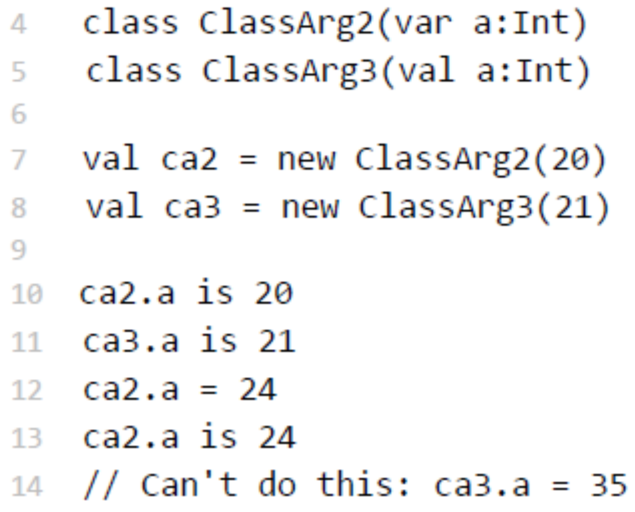


```
// ClassArg.scala
import com.atomicscala.AtomicTest._

class ClassArg(a:Int) {
  println(f)
  def f():Int = { a * 10 }
}

val ca = new ClassArg(19)
ca.f() is 190
// ca.a // error
```

- Class argument with val, cannot be changed outside the class; those with var can
- ca2 is a val, you can change value of "a"
- Val defined ca2, ca3, means you can't point them at other objects; val not control the inside of the object

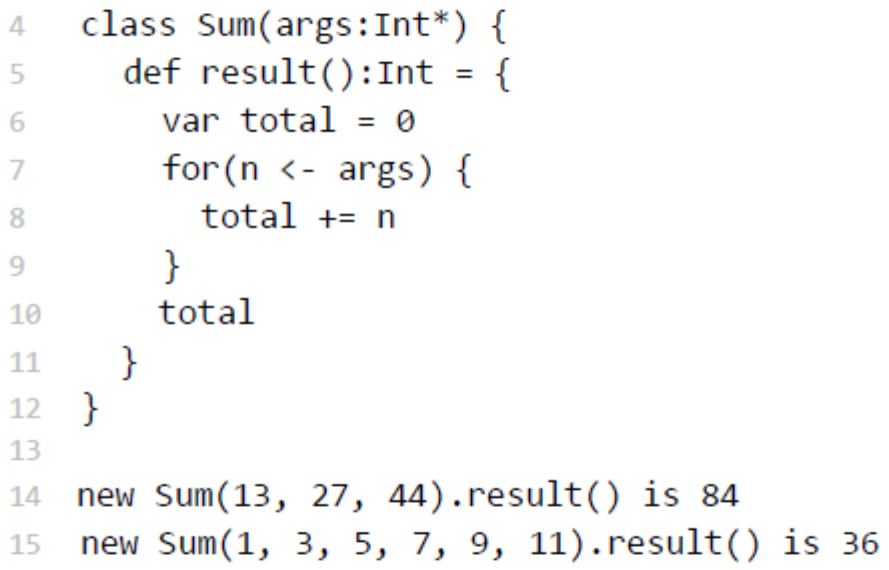


```
class ClassArg2(var a:Int)
class ClassArg3(val a:Int)

val ca2 = new ClassArg2(20)
val ca3 = new ClassArg3(21)

ca2.a is 20
ca3.a is 21
ca2.a = 24
ca2.a is 24
// Can't do this: ca3.a = 35
```

- A class can have many arguments
- Also support any number of arguments using a variable argument list, denoted by a trailing *



```scala
class Sum(args:Int*) {
  def result():Int = {
    var total = 0
    for(n <- args) {
      total += n
    }
    total
  }
}

new Sum(13, 27, 44).result() is 84
new Sum(1, 3, 5, 7, 9, 11).result() is 36
```

+
Named & Default arguments

- When create an instance, you can specify the argument names
- Default values for arguments in class definition
  - Only need specify arguments that different from defaults
- Work with variable argument lists, but the variable argument list must appear last; also variable argument list itself cannot support default arguments

```scala
// NamedArguments.scala

class Color(red:Int, blue:Int, green:Int)
new Color(red = 80, blue = 9, green = 100)
new Color(80, 9, green = 100)
```

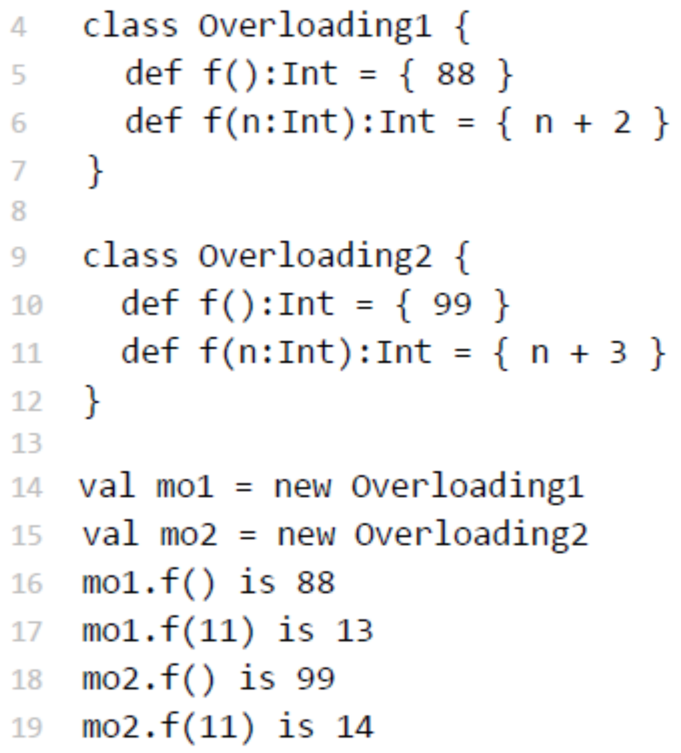
```scala
class Color2(red:Int = 100,
  blue:Int = 100, green:Int = 100)
new Color2(20)
new Color2(20, 17)
new Color2(blue = 20)
new Color2(red = 11, green = 42)
```

+
Overloading

- Same name(“overload” that name) for different methods as long as argument lists differ

- Distinguish by comparing signatures
  - Signature: comprised of name, argument list, return type

- A method signature also includes info about enclosing class
  - Overloading1’s f not clash with overloading2’s f



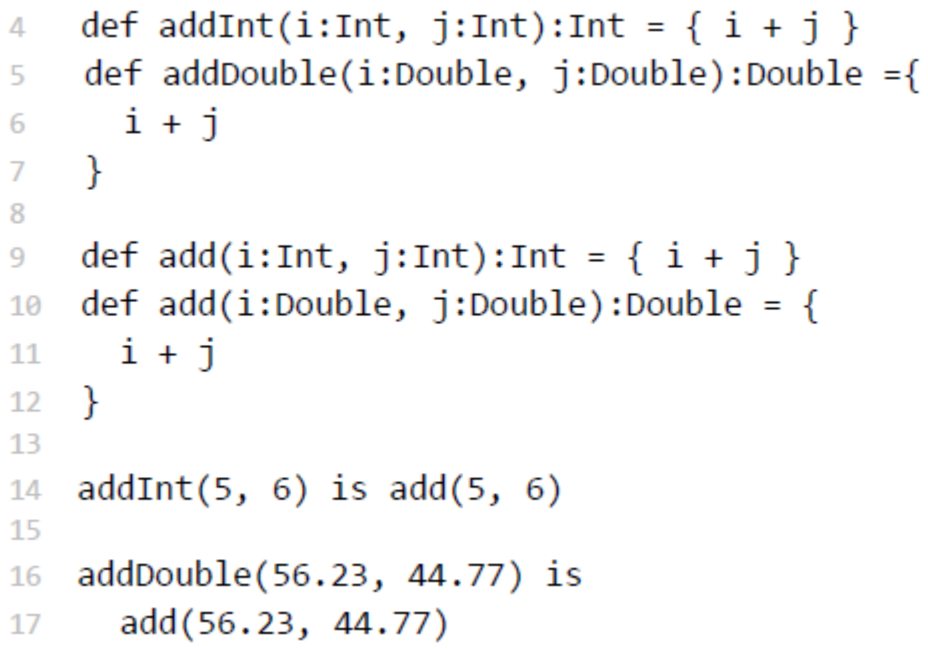
```
class Overloading1 {
  def f():Int = { 88 }
  def f(n:Int):Int = { n + 2 }
}

class Overloading2 {
  def f():Int = { 99 }
  def f(n:Int):Int = { n + 3 }
}

val mo1 = new Overloading1
val mo2 = new Overloading2
mo1.f() is 88
mo1.f(11) is 13
mo2.f() is 99
mo2.f(11) is 14
```

Why overloading useful?

- Allow to express "variations on a theme" more clearly than using different method names



```
def addInt(i:Int, j:Int):Int = { i + j }
def addDouble(i:Double, j:Double):Double ={
  i + j
}

def add(i:Int, j:Int):Int = { i + j }
def add(i:Double, j:Double):Double = {
  i + j
}

addInt(5, 6) is add(5, 6)

addDouble(56.23, 44.77) is
  add(56.23, 44.77)
```

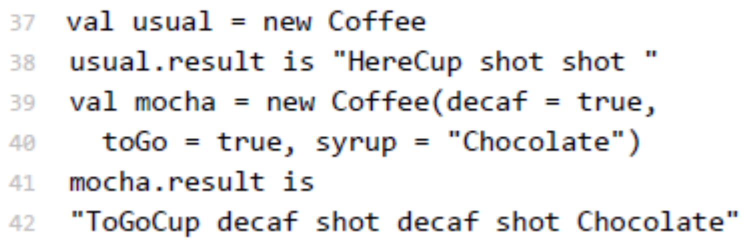
Constructors

- The code that "constructs" a new object
- Combine:
  - Class argument list – initialized before entering class body
  - Class body – statements execute from top to bottom
- Scala already did these:
  - Class arguments – initialization and make them accessible to other objects

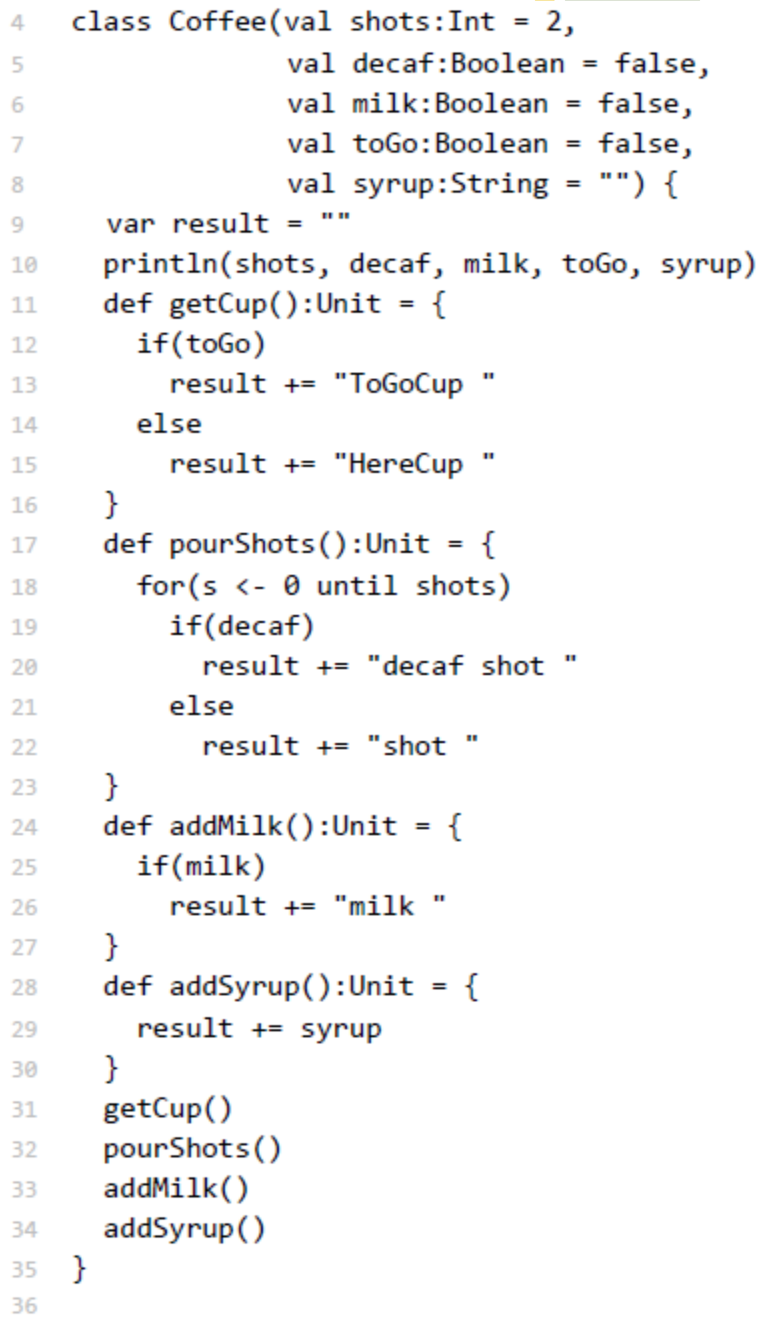
+

Customized constructor

- Coffee constructor completes
  - Class body has run
  - All initialization occurred
  - Result field has the result of all operations



```
val usual = new Coffee
usual.result is "HereCup shot shot "
val mocha = new Coffee(decaf = true,
  toGo = true, syrup = "Chocolate")
mocha.result is
"ToGoCup decaf shot decaf shot Chocolate"
```

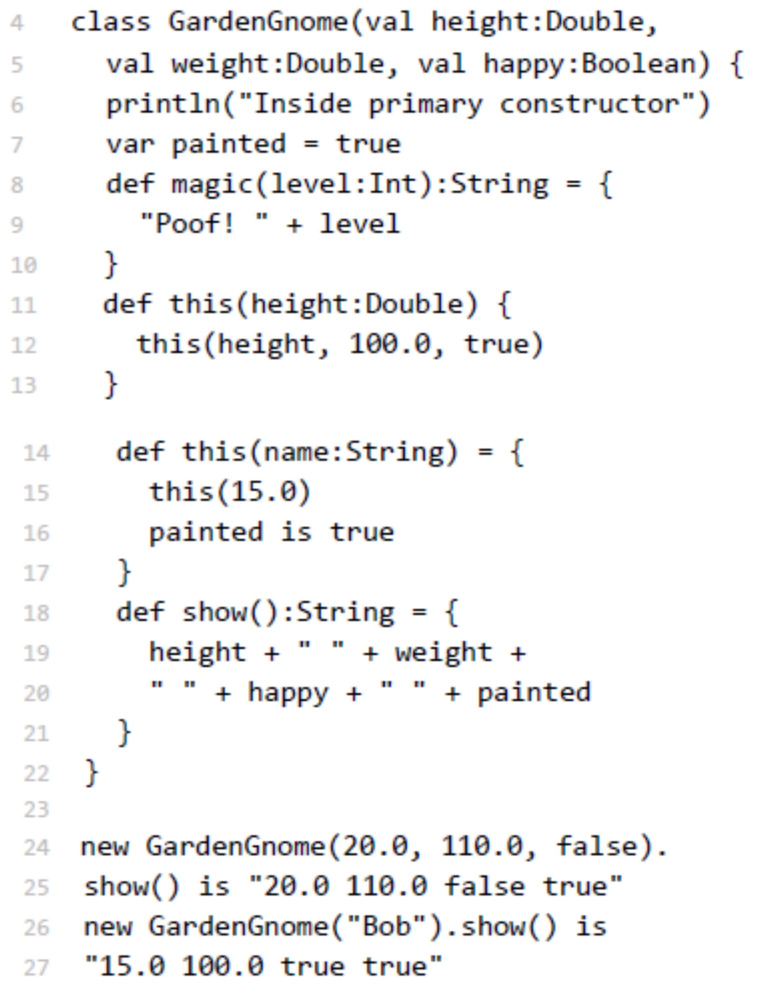


```
class Coffee(val shots:Int = 2,
             val decaf:Boolean = false,
             val milk:Boolean = false,
             val toGo:Boolean = false,
             val syrup:String = "") {
  var result = ""
  println(shots, decaf, milk, toGo, syrup)
  def getCup():Unit = {
    if(toGo)
      result += "ToGoCup "
    else
      result += "HereCup "
  }
  def pourShots():Unit = {
    for(s <- 0 until shots)
      if(decaf)
        result += "decaf shot "
      else
        result += "shot "
  }
  def addMilk():Unit = {
    if(milk)
      result += "milk "
  }
  def addSyrup():Unit = {
    result += syrup
  }
  getCup()
  pourShots()
  addMilk()
  addSyrup()
}
```

+

Auxiliary constructors

- Constructor overloading
  - Different ways to create objects of same class
  - Define a method called “this”(a keyword)
- All auxiliary constructors must first call primary constructor, or another auxiliary constructor
  - Primary constructor – this( )
- Can’t use val or var for auxiliary constructor arguments
- Result of the final expression in a constructor not returned, but ignored



```
class GardenGnome(val height:Double,
  val weight:Double, val happy:Boolean) {
  println("Inside primary constructor")
  var painted = true
  def magic(level:Int):String = {
    "Poof! " + level
  }
  def this(height:Double) {
    this(height, 100.0, true)
  }

  def this(name:String) = {
    this(15.0)
    painted is true
  }
  def show():String = {
    height + " " + weight +
    " " + happy + " " + painted
  }
}

new GardenGnome(20.0, 110.0, false).
show() is "20.0 110.0 false true"
new GardenGnome("Bob").show() is
"15.0 100.0 true true"
```

Brevity / succinct

- For comprehension's use

```
def filterWithYield3(
  v:Vector[Int]):Vector[Int] =
  for {
    n <- v
    if n < 10
    if n % 2 != 0
  } yield n

val v = Vector(1,2,3,5,6,7,8,10,13,14,17)
filterWithYield3(v) is Vector(1,3,5,7)
```

```
// Semicolons allow a single-line for:
def filterWithYield4(
  v:Vector[Int]):Vector[Int] =
  for{n <- v; if n < 10; if n % 2 != 0}
    yield n

val v = Vector(1,2,3,5,6,7,8,10,13,14,17)

filterWithYield4(v) is Vector(1,3,5,7)
```

+

Companion Objects

- For those that "this method or field is about the class, but not about a particular object"
- Object keyword
  - Can't create instances of an object – there's only one.
  - Collect methods and fields logically belong together
  - Create a companion object for a class
    - Same name as the class
- Naming convention
  - Instance names – lower case first letter
  - Class and object names – capitalize first letter
- Creating a case class automatically creates a companion object, that containing apply method (factory method)

Example

```scala
class X {
  def increment() =  { X.n += 1; X.n }
}

object X {
  var n:Int = 0 // Only one of these
}

var a = new X
var b = new X
a.increment() is 1
b.increment() is 2
a.increment() is 3
```

```scala
// FactoryMethod.scala
import com.atomicscala.AtomicTest._

class Car(val make:String) {
  override def toString = s"Car($make)"
}

object Car {
  def apply(make:String) = new Car(make)
}

val myCar = Car("Toyota")
myCar is "Car(Toyota)"
```

```scala
// ObjCounter.scala
import com.atomicscala.AtomicTest._

class Count() {
  val id = Count.id()
  override def toString = s"Count$id"
}

object Count {
  var n = -1
  def id() = { n += 1; n }
}

Vector(new Count, new Count, new Count,
  new Count, new Count) is
"Vector(Count0, Count1, " +
"Count2, Count3, Count4)"
```

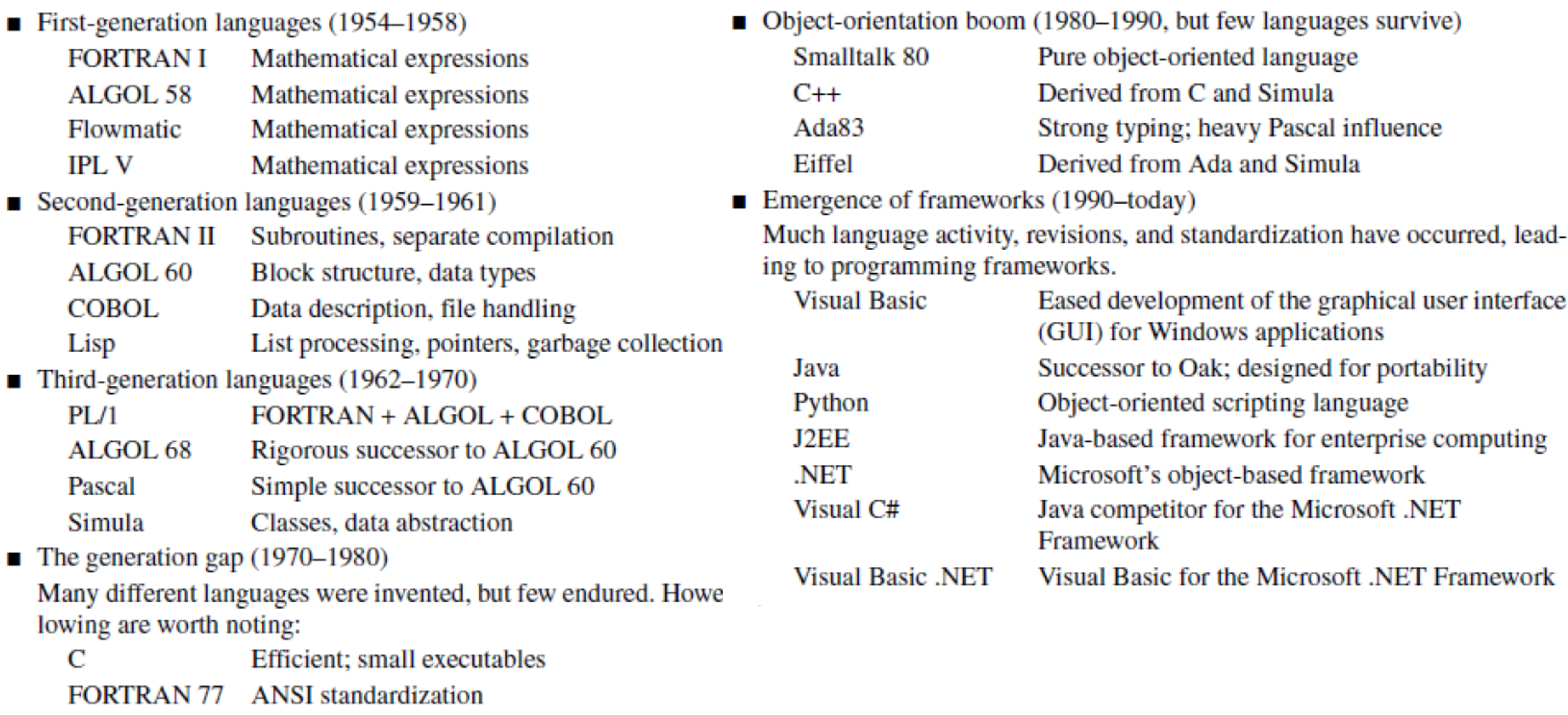
Evolution of the Object Model

- Two sweeping trends

1. The shift in focus from programming-in-the-small to programming-in-the-large
2. The evolution of high-order programming languages

- Complexity in software system prompted applied research in software engineering
  - Decomposition
  - Abstraction
  - Hierarchy
- Needs more expressive programming languages

Generations of programming languages



- First-generation languages (1954–1958)

FORTRAN I	Mathematical expressions
ALGOL 58	Mathematical expressions
Flowmatic	Mathematical expressions
IPL V	Mathematical expressions

- Second-generation languages (1959–1961)

FORTRAN II	Subroutines, separate compilation
ALGOL 60	Block structure, data types
COBOL	Data description, file handling
Lisp	List processing, pointers, garbage collection

- Third-generation languages (1962–1970)

PL/1	FORTRAN + ALGOL + COBOL
ALGOL 68	Rigorous successor to ALGOL 60
Pascal	Simple successor to ALGOL 60
Simula	Classes, data abstraction

- The generation gap (1970–1980)

Many different languages were invented, but few endured. Howe
lowing are worth noting:

C	Efficient; small executables
FORTRAN 77	ANSI standardization

- Object-orientation boom (1980–1990, but few languages survive)

Smalltalk 80	Pure object-oriented language
C++	Derived from C and Simula
Ada83	Strong typing; heavy Pascal influence
Eiffel	Derived from Ada and Simula

- Emergence of frameworks (1990–today)

Much language activity, revisions, and standardization have occurred, leading to programming frameworks.

Visual Basic	Eased development of the graphical user interface (GUI) for Windows applications
Java	Successor to Oak; designed for portability
Python	Object-oriented scripting language
J2EE	Java-based framework for enterprise computing
.NET	Microsoft's object-based framework
Visual C#	Java competitor for the Microsoft .NET Framework
Visual Basic .NET	Visual Basic for the Microsoft .NET Framework

- First-generation
  - Primarily for scientific and engineering apps, vocabulary entirely mathematics; write math formulas, freeing from assembly or machine code.
- Second-generation
  - Machine gets powerful, business application
  - Emphasis on algorithmic abstractions; tell machine what to do
- Third
  - Transistors advent; integrated circuit technology; hardware cost dropped
  - Demands of data manipulation; Support for data abstraction

+

- 70s
  - Thousand of different program languages;
  - Larger programs highlighted inadequacies of earlier languages
  - Few survived; but many concepts introduced adopted by successors
- Object-oriented (from 80s, 90s)
  - Object-oriented decomposition of software
  - Main streams: Java, C++, etc.
  - Emergence of frameworks (e.g. J2EE, .NET)

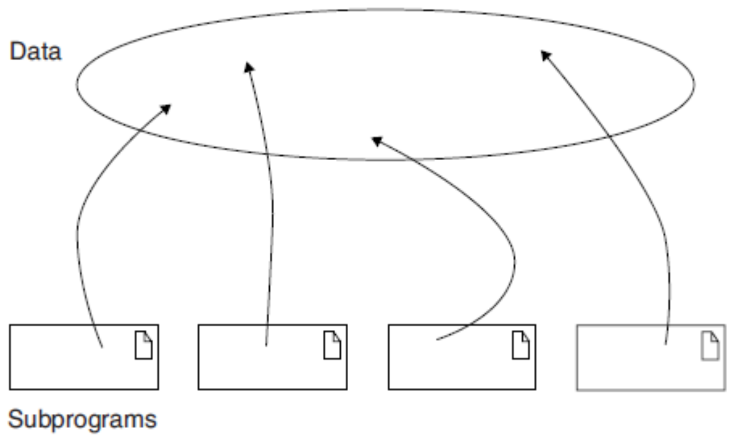


Figure 2–1 The Topology of First- and Early Second-Generation Programming Languages

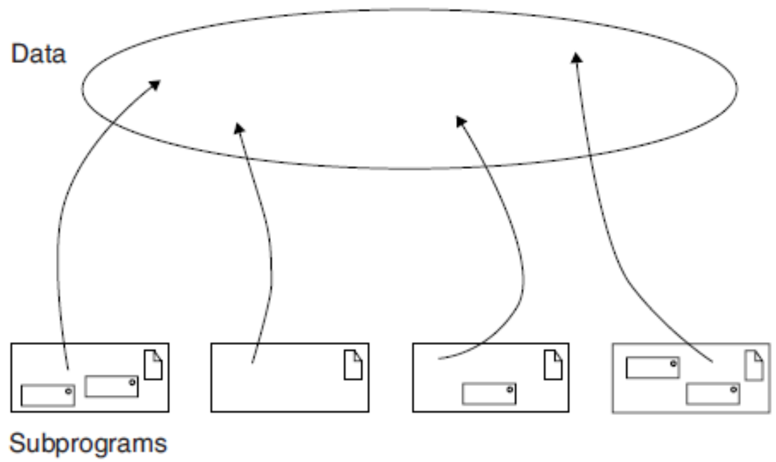


Figure 2–2 The Topology of Late Second- and Early Third-Generation Programming Languages

Subprograms as an abstraction mechanism

+

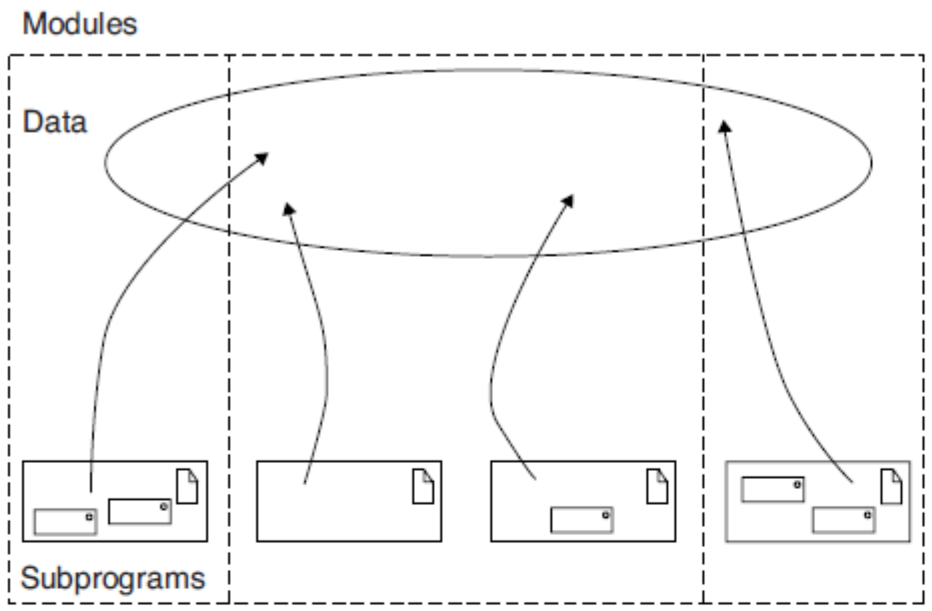


Figure 2–3 The Topology of Late Third-Generation Programming Languages

Modular structure

Most lacked support for data abstraction and strong typing, some errors can only detected during execution of program.

For object-oriented

- Data abstraction important to master complexity of problem.
- Physical building block is module(a logical collection of classes and objects instead of subprograms)

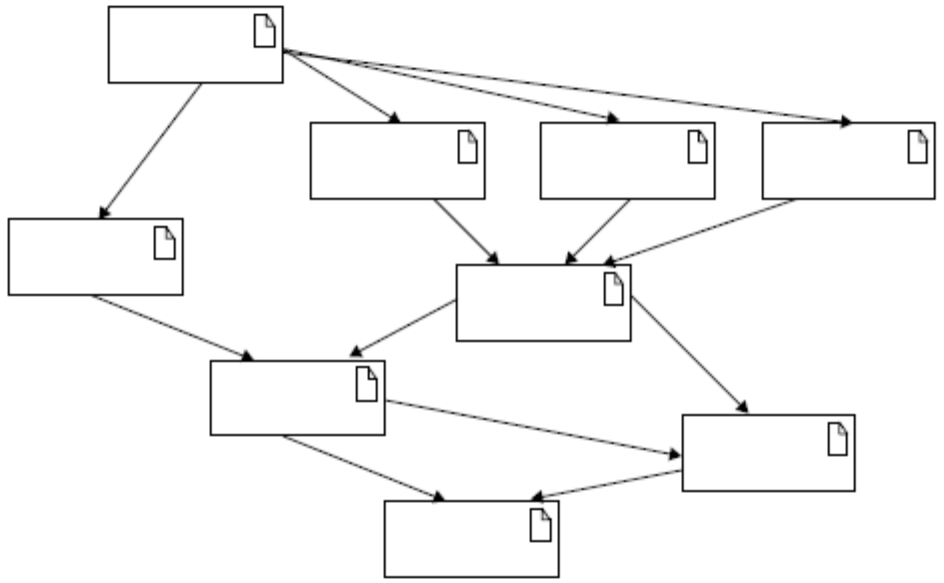
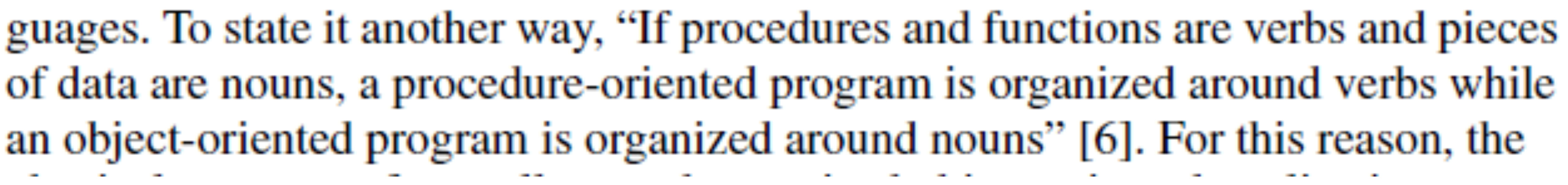


Figure 2–4 The Topology of Small to Moderate-Sized Applications Using Object-Based and Object-Oriented Programming Languages

For object-oriented

- Data and operations are united, that fundamental logical building blocks are no longer algorithms, but classes and objects
- Little or no global data


guages. To state it another way, "If procedures and functions are verbs and pieces of data are nouns, a procedure-oriented program is organized around verbs while an object-oriented program is organized around nouns" [6]. For this reason, the

+

Foundations of Object Model

Object-oriented programming(OOP)

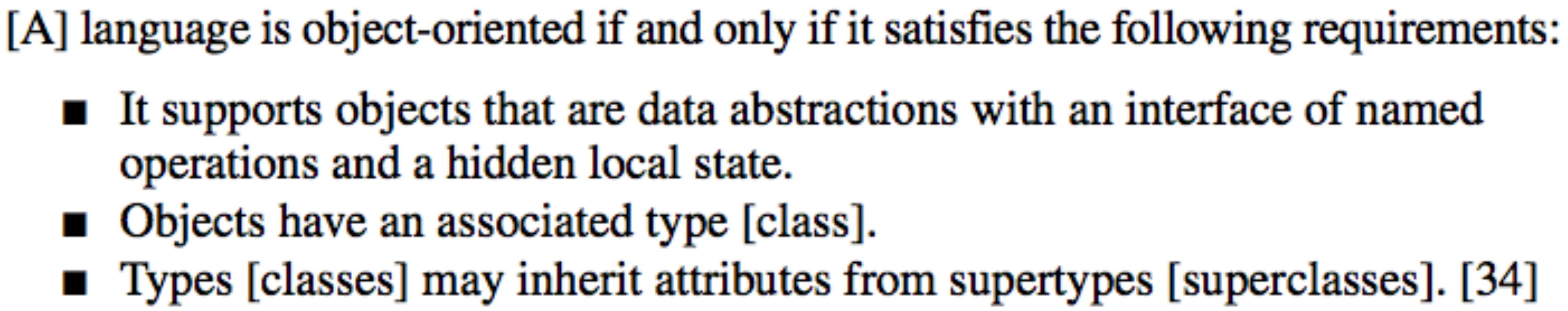
- Object orientation cope with complexity inherent in many different systems
  - Not just to programming languages, user interface design, databases, computer architectures
- OOP

> Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

  - Uses objects as building blocks
  - Each object is an instance of some class
  - Classes relates to one another via inheritance

What’s object-oriented?

- Cardelli and Wegner say:

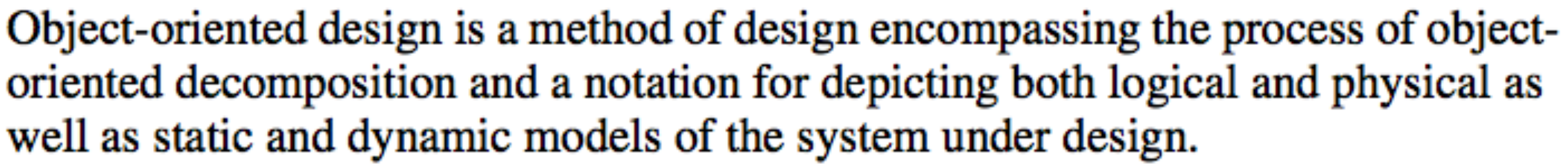


[A] language is object-oriented if and only if it satisfies the following requirements:

- It supports objects that are data abstractions with an interface of named operations and a hidden local state.
- Objects have an associated type [class].
- Types [classes] may inherit attributes from supertypes [superclasses]. [34]

Object-oriented design(OOD)

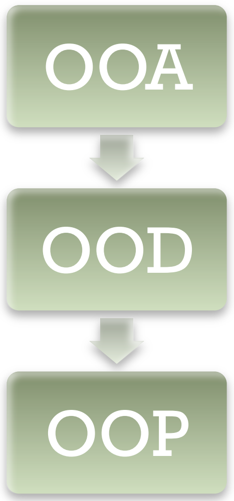
- Leads to an object-oriented decomposition
- Uses different notations to express different models of logical (class and object structure), and physical (module and process architecture) design of a system



Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design.

Object-oriented analysis(OOA)

Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.

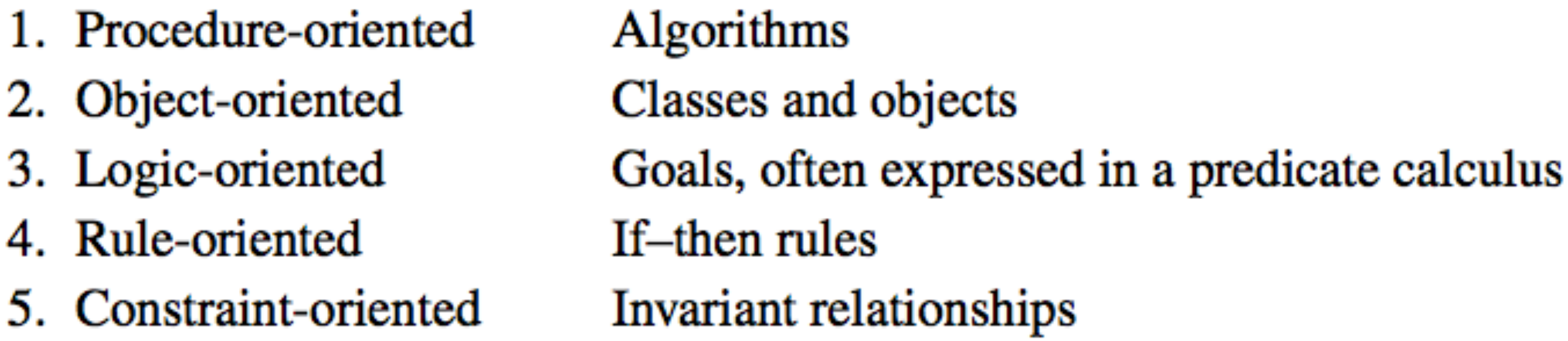


OOA serves OOD; OOD as blueprints for implementing system using OOP methods

Elements of the Object Model

Programming style

- No single one best for all kinds applications.
  - Knowledge base
  - Computation-intense operation
  - Broadest set of applications



1. Procedure-oriented	Algorithms
2. Object-oriented	Classes and objects
3. Logic-oriented	Goals, often expressed in a predicate calculus
4. Rule-oriented	If–then rules
5. Constraint-oriented	Invariant relationships

+

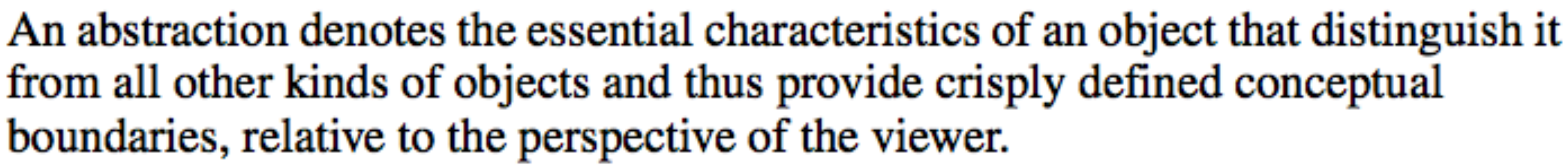
Elements of object model

- Conceptual framework for object-oriented, is the object model
- Four major elements of this model(a model without any one of these is not object-oriented)
  - Abstraction
  - Encapsulation
  - Modularity
  - Hierarchy
- Three minor elements: (useful but not essential)
  - Typing
  - Concurrency
  - Persistence

+

Meaning of Abstraction

- Define abstraction:



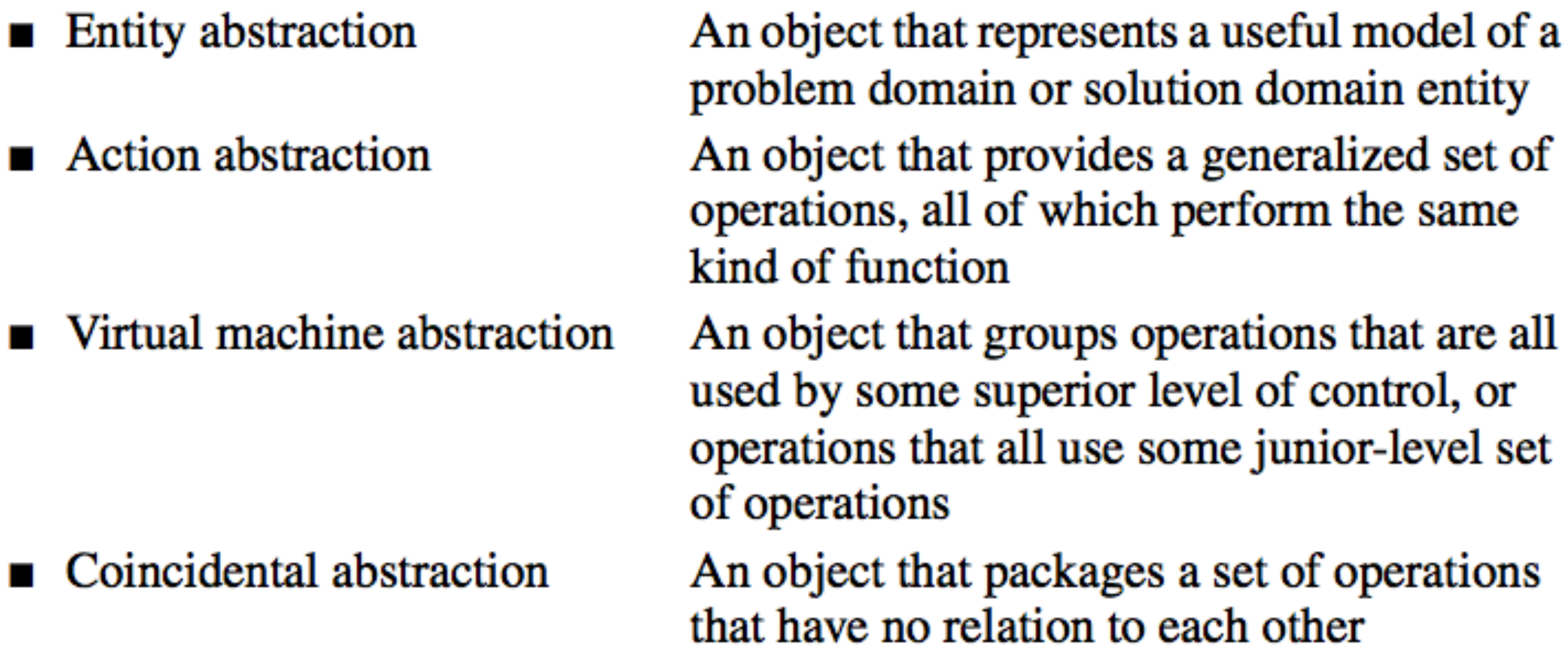
An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.

- Focus on outside view of an object, separate object's essential behavior from its implementation

- Decide right set of abstractions for a given domain, is central problem in OOD

+
Spectrum of abstraction

- From most to least useful:



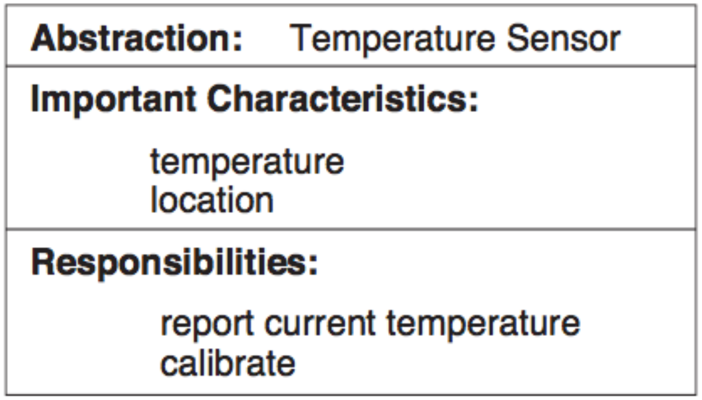
■ Entity abstraction	An object that represents a useful model of a problem domain or solution domain entity
■ Action abstraction	An object that provides a generalized set of operations, all of which perform the same kind of function
■ Virtual machine abstraction	An object that groups operations that are all used by some superior level of control, or operations that all use some junior-level set of operations
■ Coincidental abstraction	An object that packages a set of operations that have no relation to each other

+

- A client is any object that uses resources of another object (known as server).
  - Characterize behavior of an object by considering services it provides to other objects
  - Force to concentrate on outside view of an object, which defines a contract on which other objects may depend, and which must be carried out by inside view
- Protocol: entire set of operations that contributes to the contract, with legal ordering of their invoking
  - Denotes ways that object may act and react, thus constitutes entire outside view of the abstraction.
- Terms: operation, method, member function virtually mean same thing.

Examples of Abstraction

- Farm, maintaining proper greenhouse environment
- A key abstraction is about a sensor
  - A temperature sensor: an object that measures temperature at a location
  - What are responsibilities of a temp sensor? Answers yield different design decisions



Abstraction: Temperature Sensor

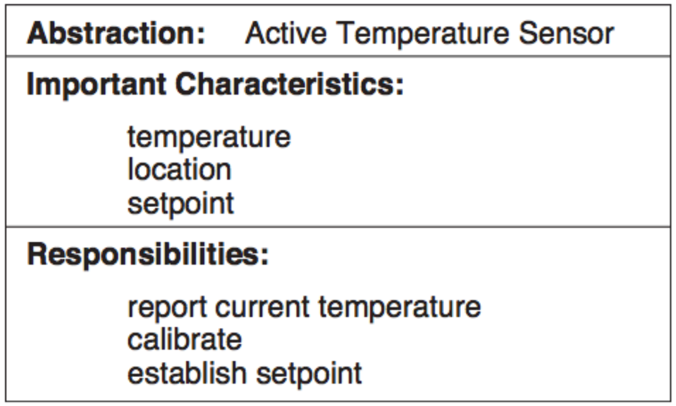
Important Characteristics:

temperature
location

Responsibilities:

report current temperature
calibrate

Figure 2–6 Abstraction of a `Temperature Sensor`



Abstraction: Active Temperature Sensor

Important Characteristics:

temperature
location
setpoint

Responsibilities:

report current temperature
calibrate
establish setpoint

Figure 2–7 Abstraction of an `Active Temperature Sensor`

+

- No objects stands alone; every object collaborates with others to achieve some behavior.
- Design decisions about how they cooperate, define boundaries of each abstraction and the responsibilities and protocol of each object.