Why Object Oriented Programming

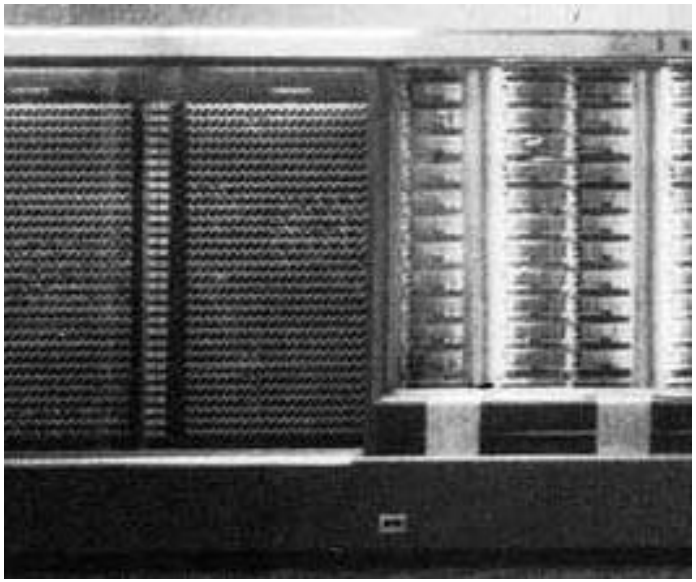
Computer Programming

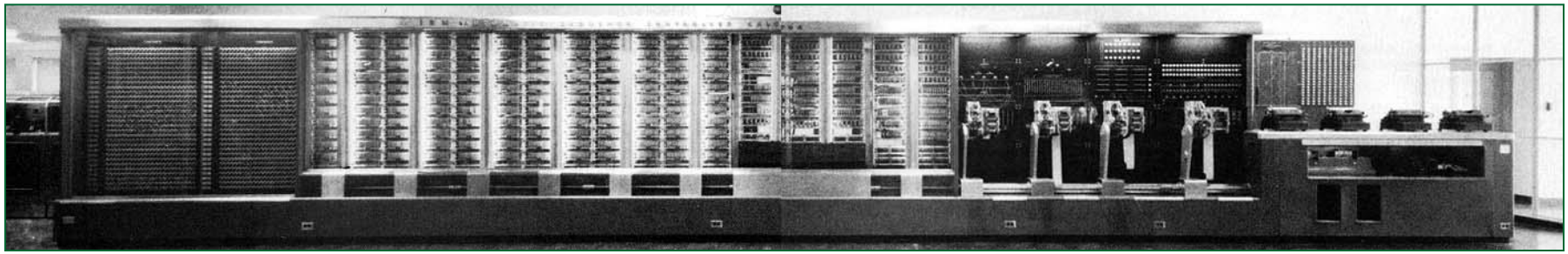
- The history of computer programming is a steady move away from machine-oriented views of programming towards concepts and metaphors that more closely reflect the way in which we ourselves understand the world

Programming progression…

- Programming has progressed through:
  - machine code
  - assembly language
  - machine-independent programming languages
  - procedures & functions
  - objects

Machine language – Mark I





Machine Language

0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111

Assembly Language – PDP-11

Assembly Language – Macro-11

```
GCD:        TST     B
            BEQ     SIMPLE
            MOV     A, R5
            SXT     R4
            DIV     B, R4
            MOV     B, A
            MOV     R5, B
            CALL    GCD
SIMPLE:     RETURN
```

Assembly Language – Macro-11

```
GCD:        TST     B
            BEQ     SIMPLE
            MOV     A, R5
            SXT     R4
            DIV     B, R4
            MOV     B, A
            MOV     R5, B
            CALL    GCD
SIMPLE:     RETURN
```

Machine-Independent Programming Languages – Fortran

```
! This example program solves for roots of the quadratic equation,
! ax^2 +bx +c =0,for given values of a, b and c.
!
PROGRAM bisection
IMPLICIT NONE
INTEGER :: iteration
DOUBLE PRECISION :: CC, Er, xl, x0, x0_old, xr

! Set convergence criterion and guess for xl, xr.
CC = 1.d-4
xl = 8.d-1
xr = 11.d-1

! Bisection method.
Er =CC +1
iteration = 0
DO WHILE (Er > CC)
      iteration = iteration + 1

      ! Compute x0 and the error.
      x0_old = x0
      x0 = (xl + xr) / 2.d0
      Er = DABS((x0 - x0_old)/x0)*100.d0
      WRITE (*,10) iteration, x0_old, x0, Er
      10 FORMAT (1X,I4,3(2X,E10.4))
```

this is partial…

Procedures & Functions – Pascal

```
program ValueArg(output);
    {Shows how to arrange for a procedure to have arguments.}

procedure PrintInitials(First, Last : char);
    {Within this procedure, the names First and Last represent the
    argument values. We'll call write to print them.}
    begin
        write('My initials are:   ');
        write(First);
        writeln(Last)
    end;  {PrintInitials}

begin
    PrintInitials ('D', 'C');   {Any two characters can be arguments.}
    PrintInitials ('Q', 'T');   {Like strings, characters are quoted.}
    PrintInitials ('&', '#')
end.   {ValueArg}
```

Objects

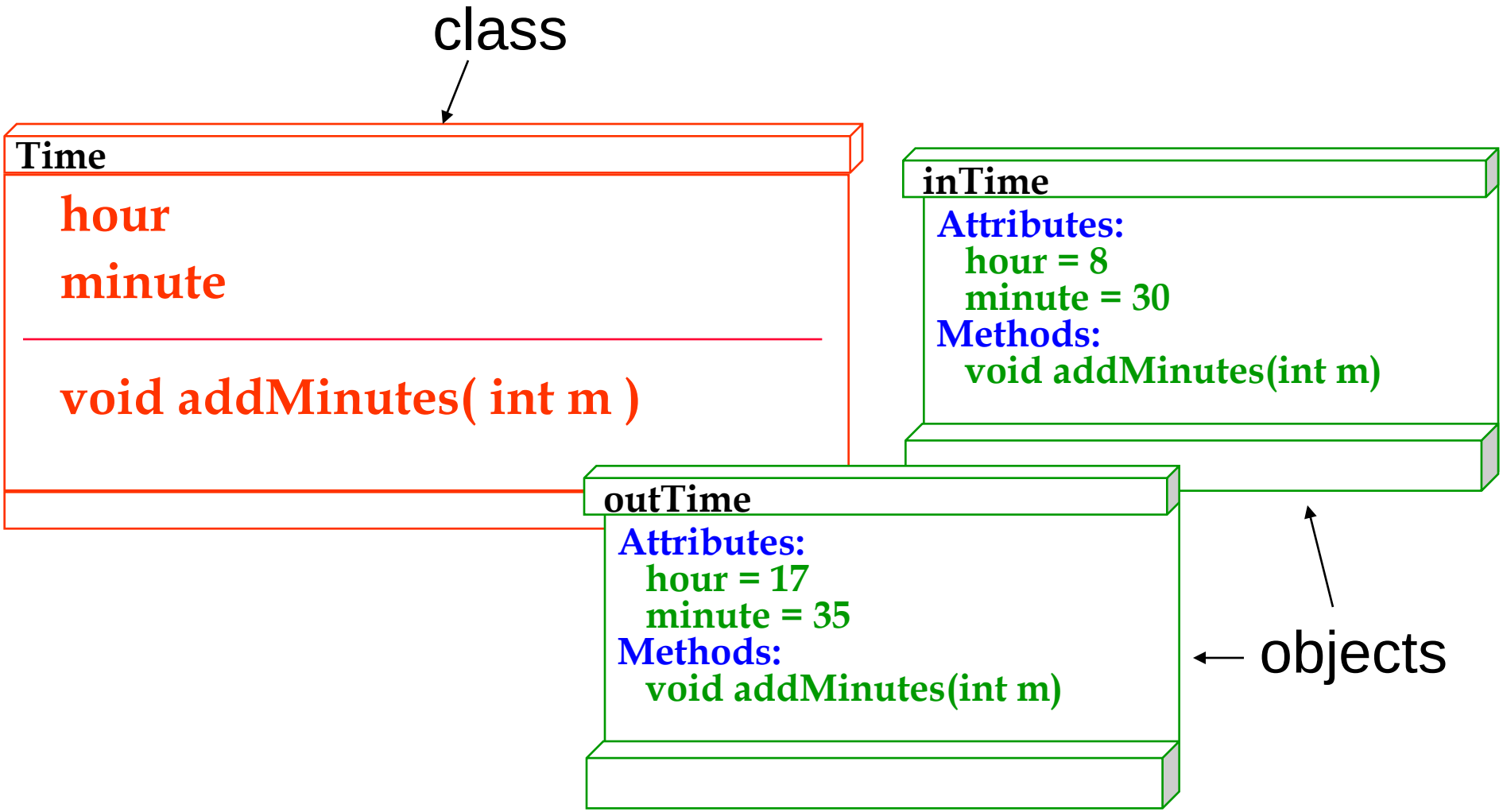
```
class Time {

        private int hour, minute;

        public Time (int h, int m) {
                hour = h;
                minute = m;
        }

        public void addMinutes (int m) {
                int totalMinutes =
                        ((60*hour) + minute + m) % (24*60);
                if (totalMinutes<0)
                        totalMinutes = totalMinutes + (24*60);
                hour = totalMinutes / 60;
                minute = totalMinutes % 60;
        }

}
```

this is partial…

“Intrinsic Power” vs. “Effective Power”

- This progression is *not* a matter of “intrinsic power”
- Anything you can do with a minimally capable computer language, you can theoretically do with any other minimally capable computer language
- But that is like saying a shovel is theoretically as capable as a tractor. In practice, using a shovel might make things very hard…

Objects


class
Time
hour
minute
void addMinutes(int m)
inTime
Attributes:
hour = 8
minute = 30
Methods:
void addMinutes(int m)
outTime
Attributes:
hour = 17
minute = 35
Methods:
void addMinutes(int m)
objects

Classes and Objects

- A *class* is a prototype for creating objects
- When we write a program in an object-oriented language like Java, we define classes, which in turn are used to create objects
- A class has a *constructor* for creating objects

A Simple Class, called "Time" (partial)

```
class Time {

        private int hour, minute;

        public Time (int h, int m) {
                hour = h;
                minute = m;
        }

        public void addMinutes (int m) {
                int totalMinutes =
                        ((60*hour) + minute + m) % (24*60);
                if (totalMinutes<0)
                        totalMinutes = totalMinutes + (24*60);
                hour = totalMinutes / 60;
                minute = totalMinutes % 60;
        }

}
```

constructor for Time

Definition of an "Object"

- An object is a computational entity that:
  1. *Encapsulates* some state
  2. Is able to perform actions, or *methods*, on this state
  3. Communicates with other objects via *message passing*

1) *Encapsulates* some state

- Like a record in Pascal, it has a set of variables (of possibly different types) that describe an object's state
- These variables are sometimes called an object's *attributes* (or *fields*, or *instance variables*, or *datamembers*, or …)

Pascal Example: Represent a Time

```
type  TimeTYPE = record
                    Hour: 1..23;
                    Minute: 0..59;
                end;
var inToWork, outFromWork: TimeTYPE;
```

Java Example: Represent a Time

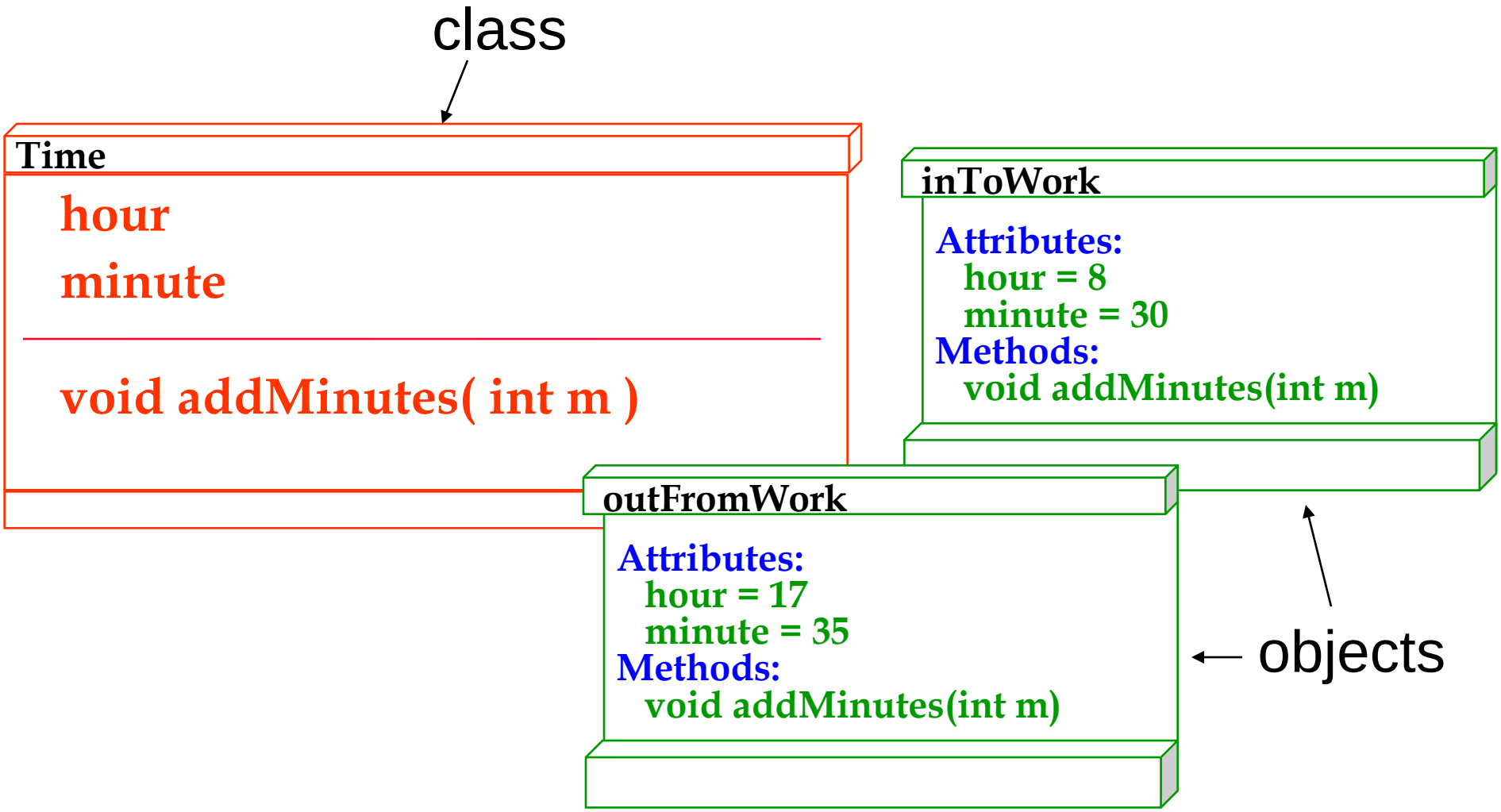
```
class Time {

      private int hour, minute;

      public Time (int h, int m) {
            hour = h;
            minute = m;
      }
…
}
```

attributes of Time

constructor for Time

```
Time inToWork = new Time(8, 30);
Time outFromWork = new Time(17, 35);
```

Objects


class
Time
hour
minute
void addMinutes(int m)
inToWork
Attributes:
hour = 8
minute = 30
Methods:
void addMinutes(int m)
outFromWork
Attributes:
hour = 17
minute = 35
Methods:
void addMinutes(int m)
objects

2) Is able to perform actions, or *methods*, on this state

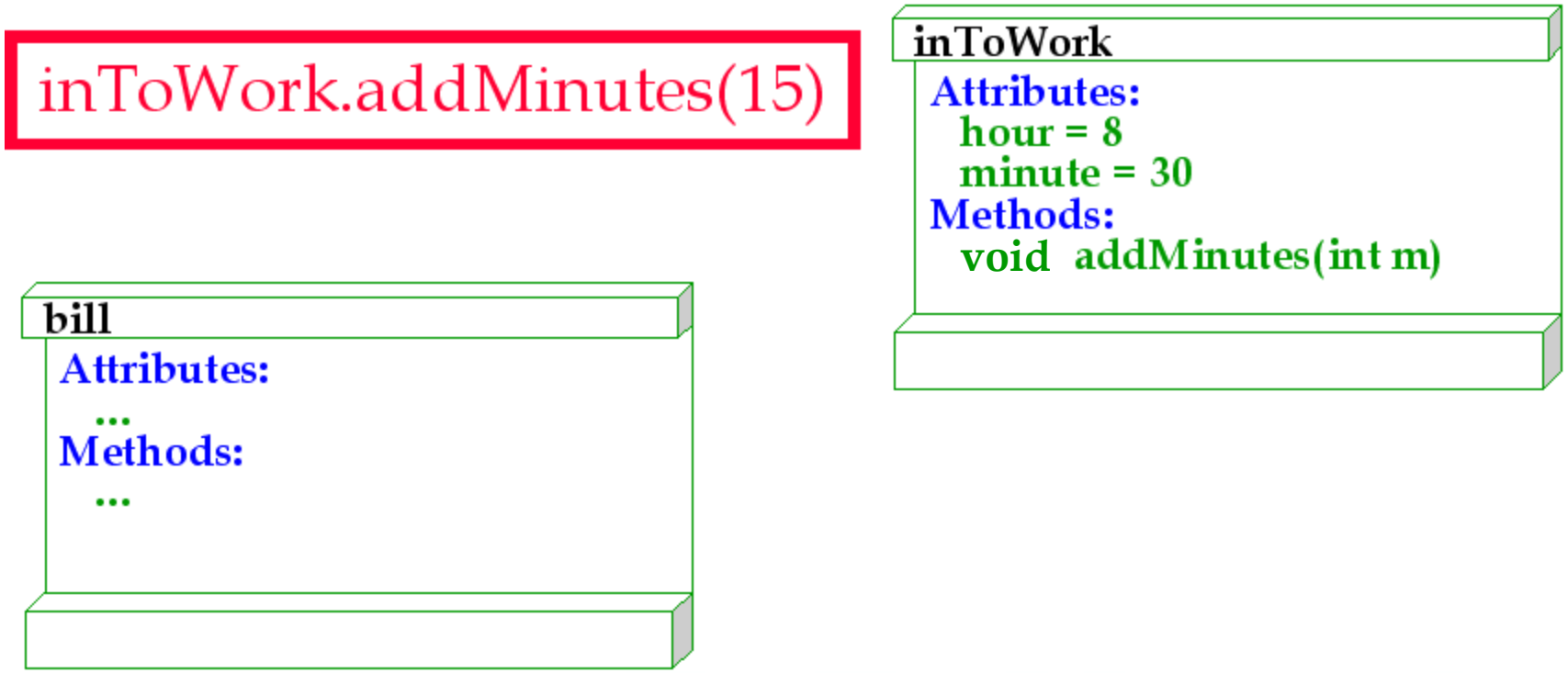
- *More than a Pascal record!*
- An object can also include a group of procedures/functions that carry out actions

```
class Time {

        private int hour, minute;

        public Time (int h, int m) {
                hour = h;
                minute = m;
        }

        public void addMinutes (int m) {
                int totalMinutes =
                        ((60*hour) + minute + m) % (24*60);
                if (totalMinutes<0)
                        totalMinutes = totalMinutes + (24*60);
                hour = totalMinutes / 60;
                minute = totalMinutes % 60;
        }
}
```

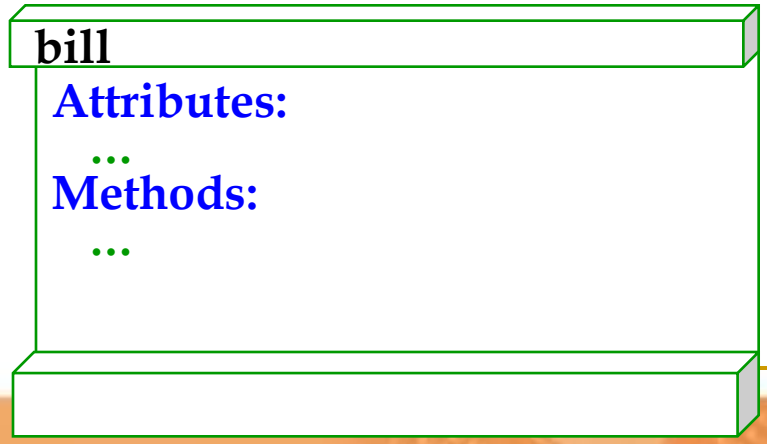
a *method* of Time

3) Communicates with other objects via *message passing*

- Sends messages to objects, triggering methods in those objects



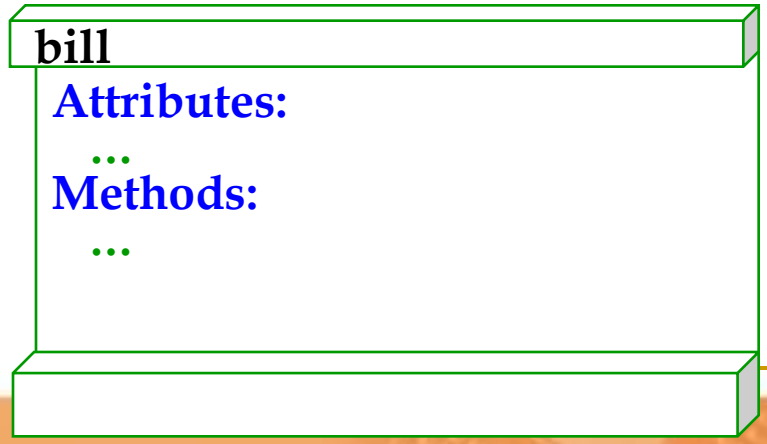
Example of Object Creation and Message Passing



Example of Object Creation and Message Passing

In one of bill's methods, the following code appears:

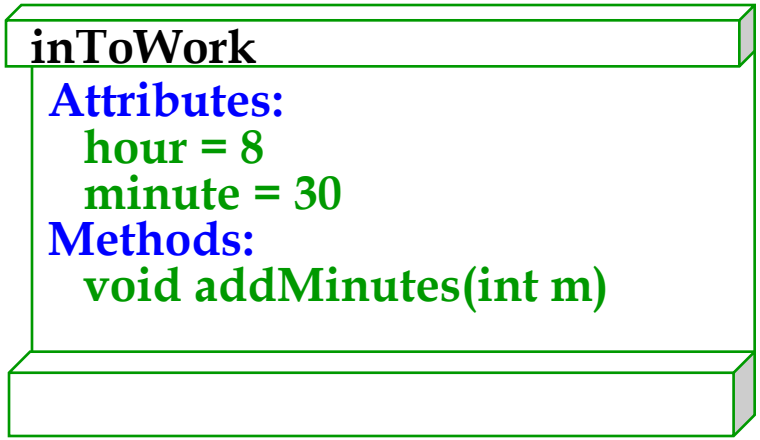
```
Time inToWork = new Time(8, 30);
inToWork.addMinutes(15);
```

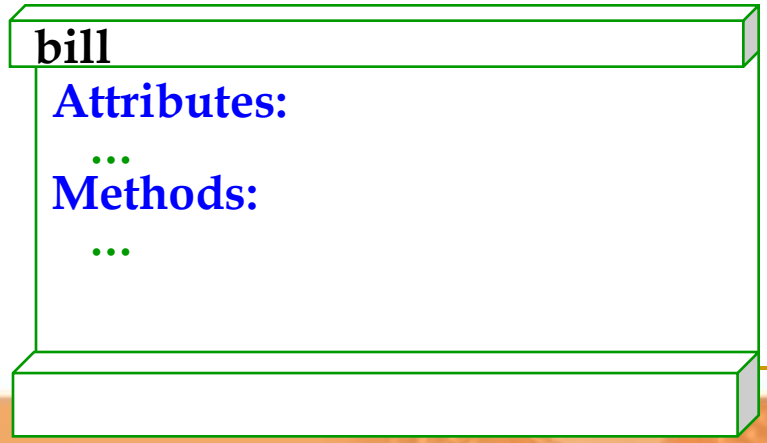


Example of Object Creation and Message Passing

In one of bill's methods, the following code appears:

- Time inToWork = new Time(8, 30);
  inToWork.addMinutes(15);

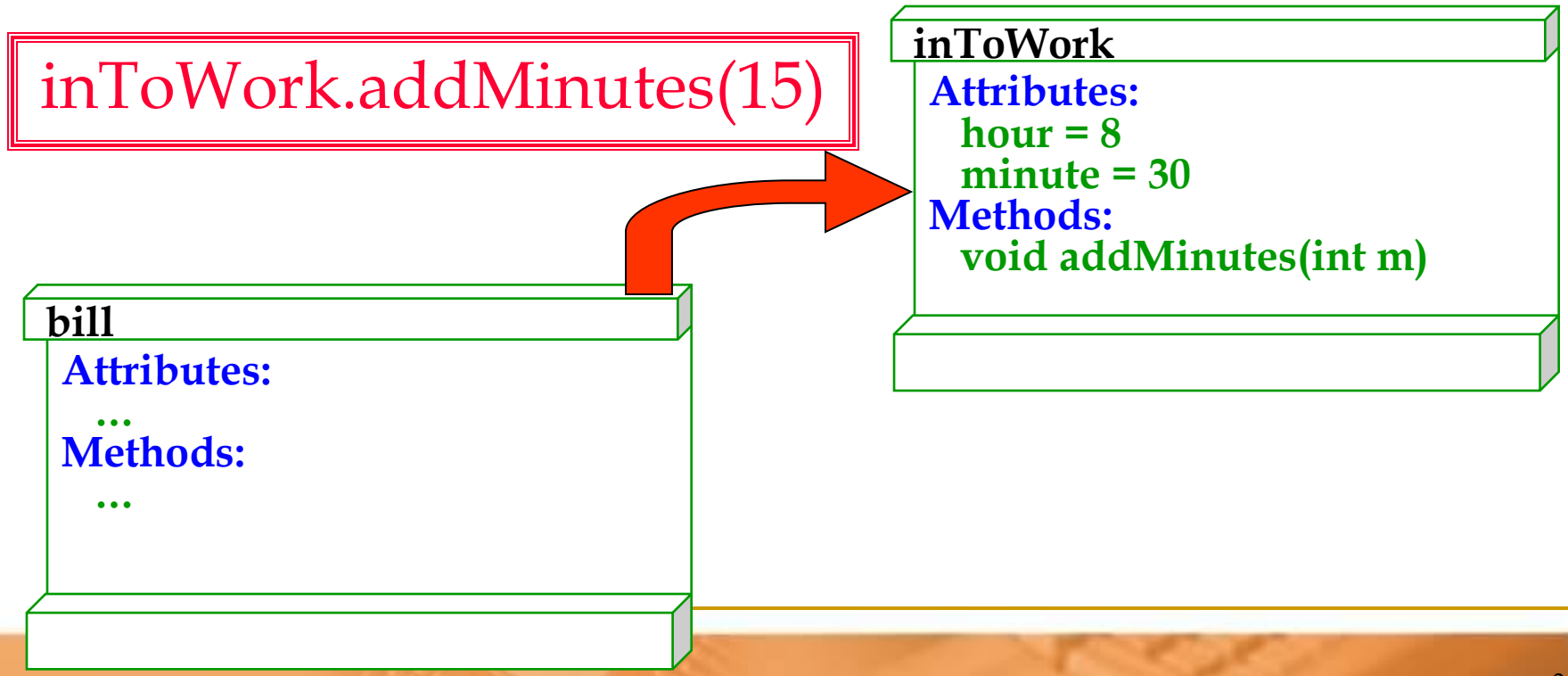




Example of Object Creation and Message Passing

In one of bill's methods, the following code appears:

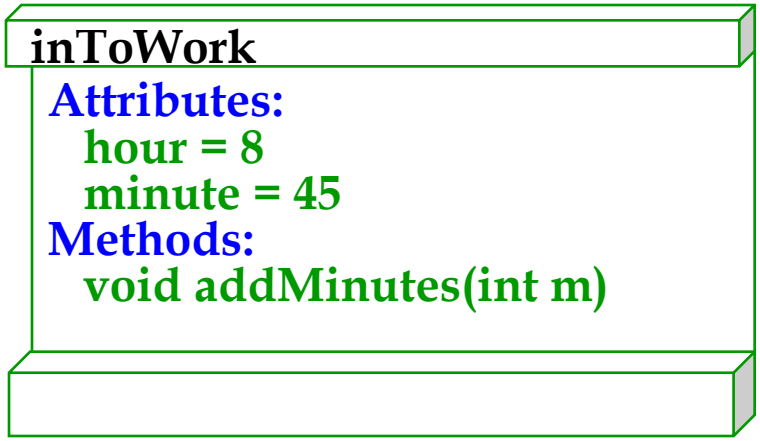
```
Time inToWork = new Time(8, 30);
inToWork.addMinutes(15);
```

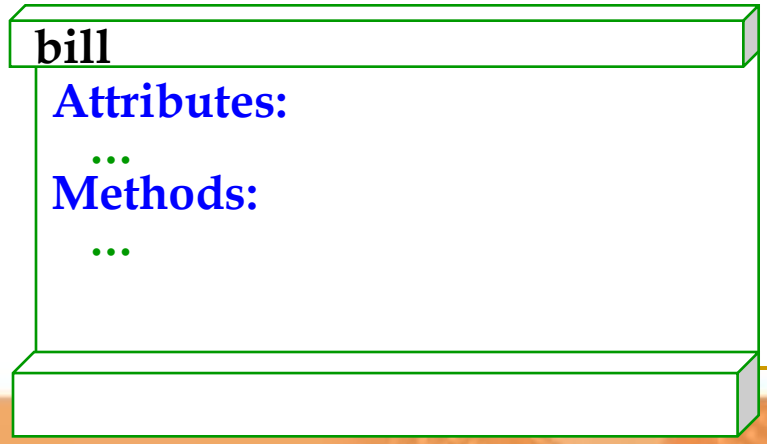


Example of Object Creation and Message Passing

In one of bill's methods, the following code appears:

```
Time inToWork = new Time(8, 30);
inToWork.addMinutes(15);
```





Structure of a Class Definition

```
class name {
        declarations

        constructor definition(s)

        method definitions
}
```

← attributes and symbolic constants

← how to create and initialize objects

← how to manipulate the state of objects

These parts of a class can actually be in any order

History of Object-Oriented Programming

- Started out for simulation of complex man-machine systems, but was soon realized that it was suitable for all complex programming projects
- SIMULA I (1962-65) and Simula 67 (1967) were the first two object-oriented languages
  - Developed at the Norwegian Computing Center, Oslo, Norway by Ole-Johan Dahl and Kristen Nygaard
  - Simula 67 introduced most of the key concepts of object-oriented programming: objects and classes, subclasses (“inheritance”), virtual procedures

The Ideas Spread

- Alan Kay, Adele Goldberg and colleagues at Xerox PARC extend the ideas of Simula in developing Smalltalk (1970's)
  - Kay coins the term "object oriented"
  - Smalltalk is first fully object oriented language
  - Grasps that this is a new programming paradigm
  - Integration of graphical user interfaces and interactive program execution
- Bjarne Stroustrup develops C++ (1980's)
  - Brings object oriented concepts into the C programming language

Other Object Oriented Languages

- Eiffel (B. Meyer)
- CLOS (D. Bobrow, G. Kiczales)
- SELF (D. Ungar et al.)
- Java (J. Gosling et al.)
- BETA (B. Bruun-Kristensen, O. Lehrmann Madsen, B. Møller-Pedersen, K. Nygaard)
- Other languages add object dialects, such as TurboPascal
- …

REVIEW: Definition of an "Object"

- An object is a computational entity that:
  - *Encapsulates* some state
  - Is able to perform actions, or *methods*, on this state
  - Communicates with other objects via *message passing*

Encapsulation

- The main point is that by thinking of the system as composed of independent objects, we keep sub-parts *really* independent
- They communicate only through well-defined message passing
- Different groups of programmers can work on different parts of the project, just making sure they comply with an *interface*
- It is possible to build larger systems with less effort

Advantages

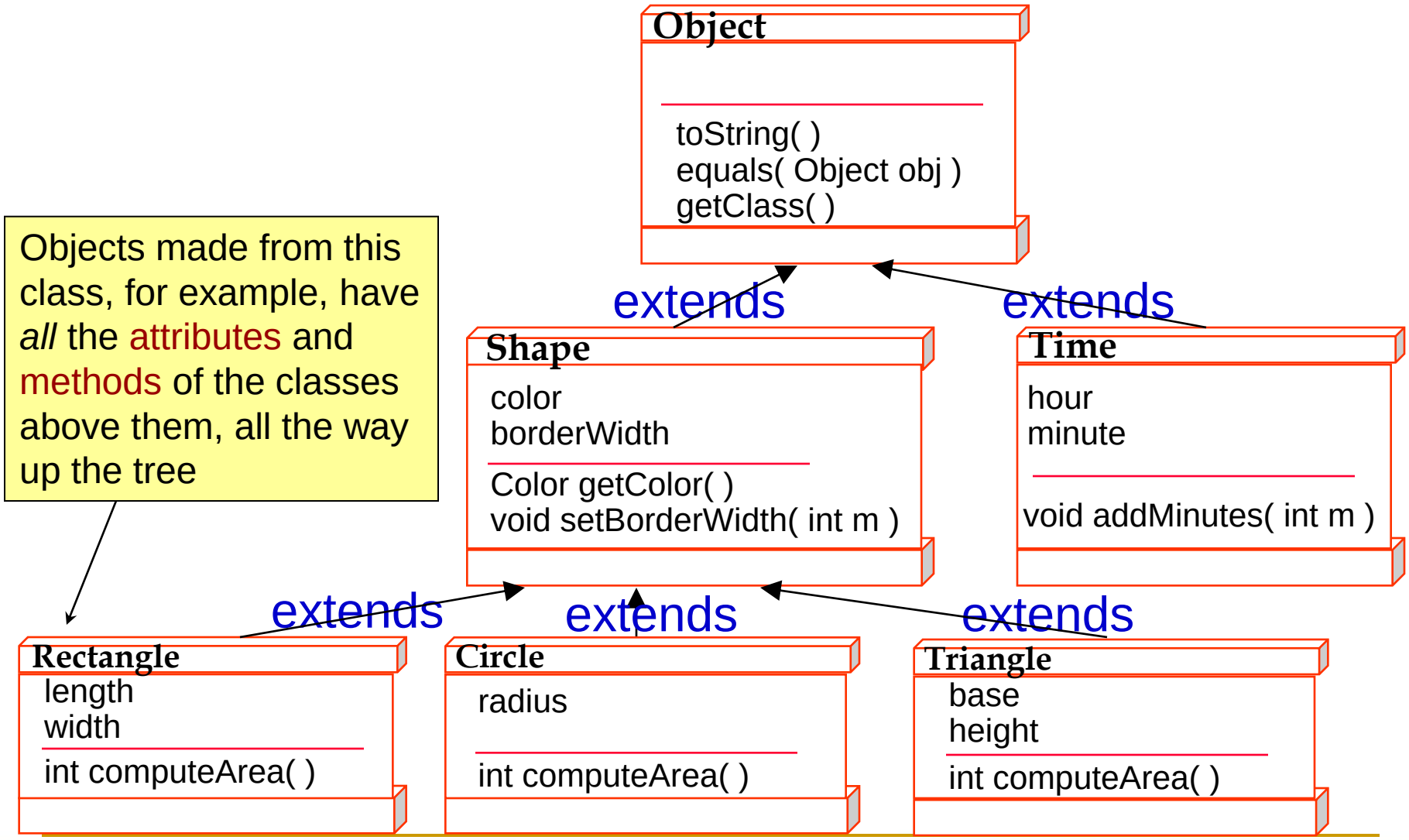
Building the system as a group of interacting objects:

- Allows extreme modularity between pieces of the system
- May better match the way we (humans) think about the problem
- Avoids recoding, increases code-reuse

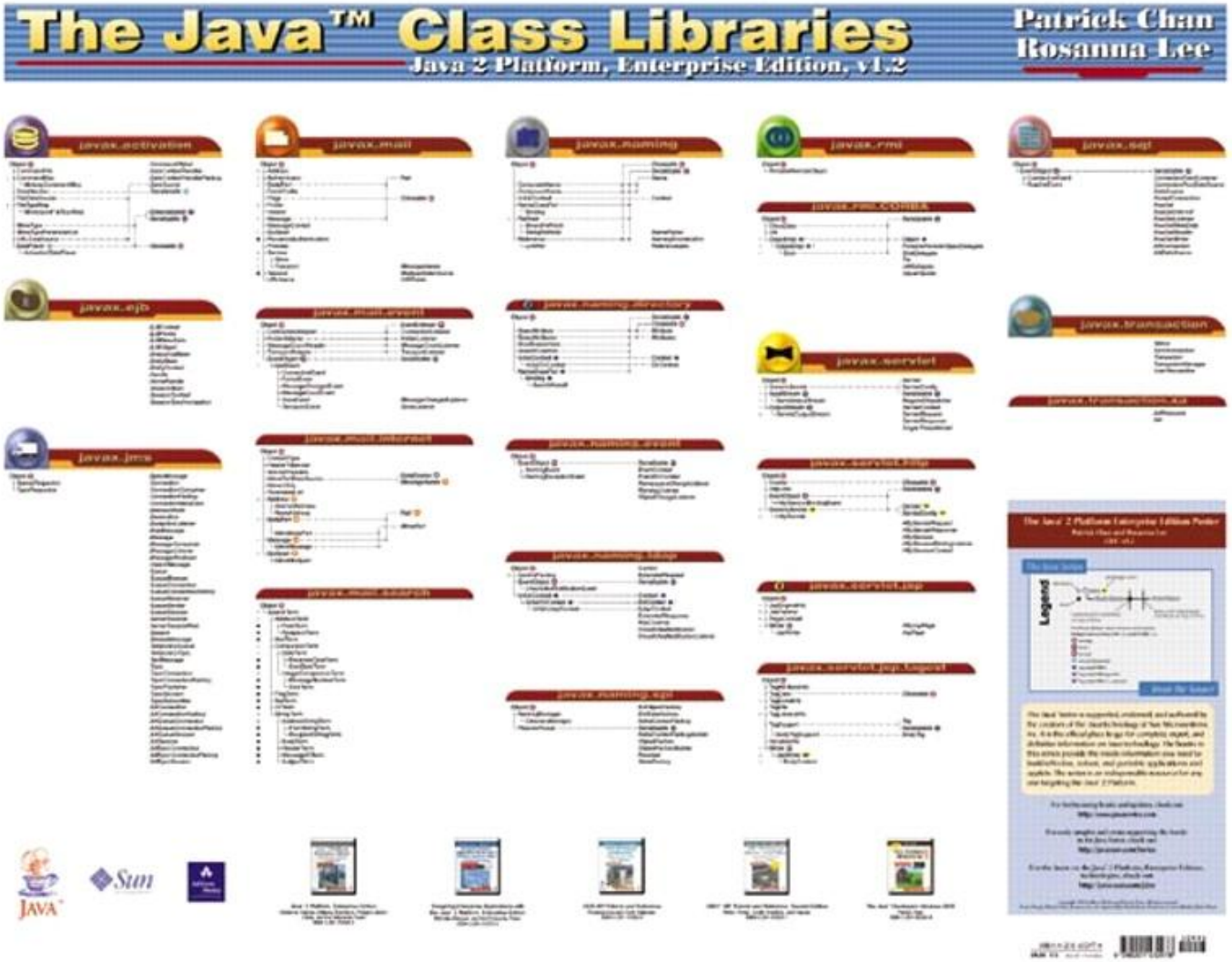
But there's more…

- Classes can be arranged in a hierarchy
- Subclasses *inherit* attributes and methods from their parent classes
- This allows us to organize classes, and to avoid rewriting code – new classes *extend* old classes, with little extra work!
- Allows for large, structured definitions

Example of Class Inheritance



Java Class Hierarchy



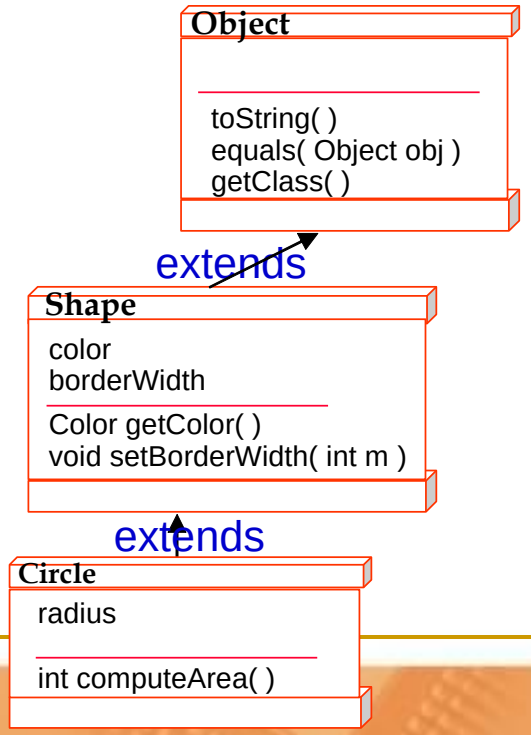
Java Class Hierarchy, another view

Polymorphism

- An object has “multiple identities”, based on its class inheritance tree
- It can be used in different ways

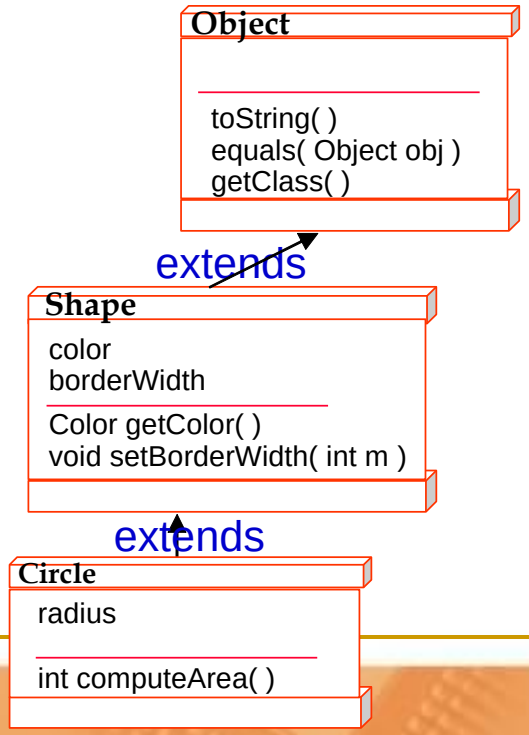
Polymorphism

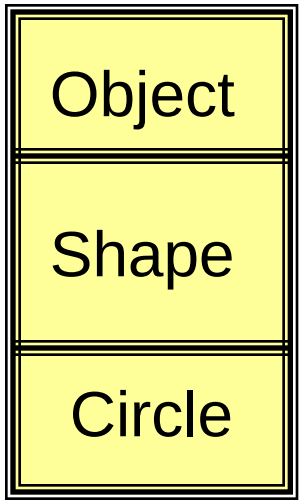
- An object has “multiple identities”, based on its class inheritance tree
- It can be used in different ways
- A Circle is-a Shape is-a Object



Polymorphism

- An object has “multiple identities”, based on its class inheritance tree
- It can be used in different ways
- A Circle is-a Shape is-a Object





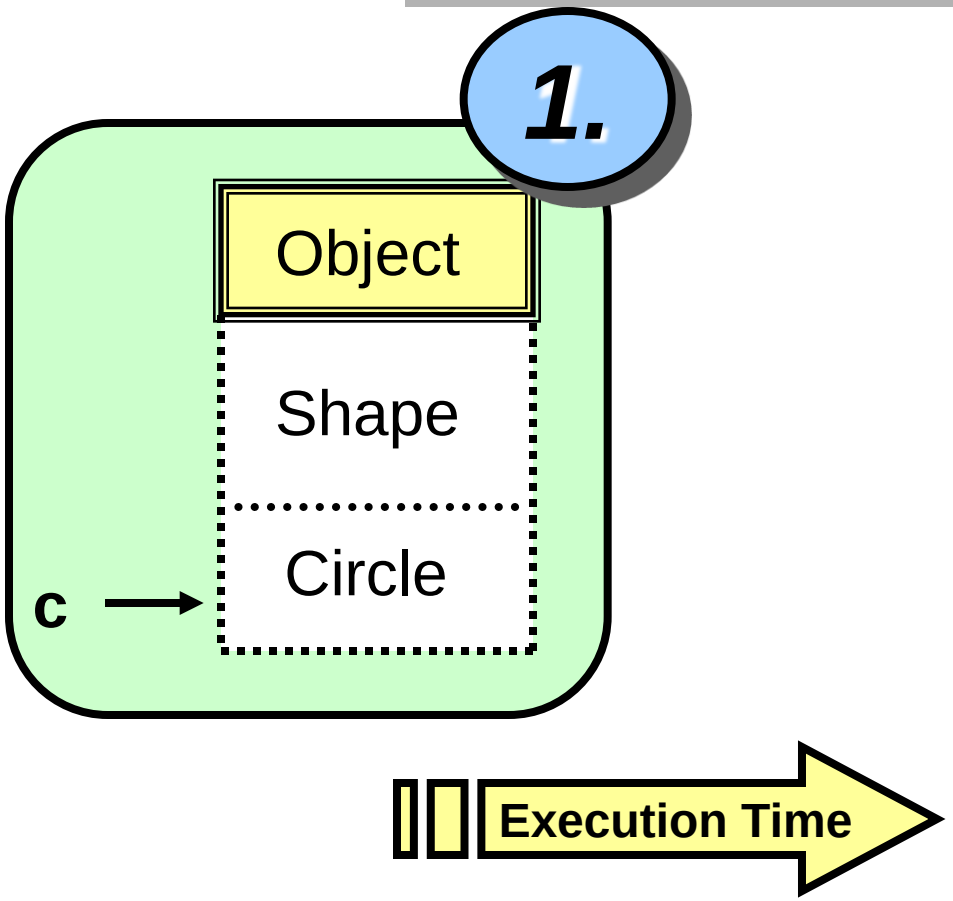
A Circle object really has 3 parts

How Objects are Created

```
Circle c = new Circle( );
```

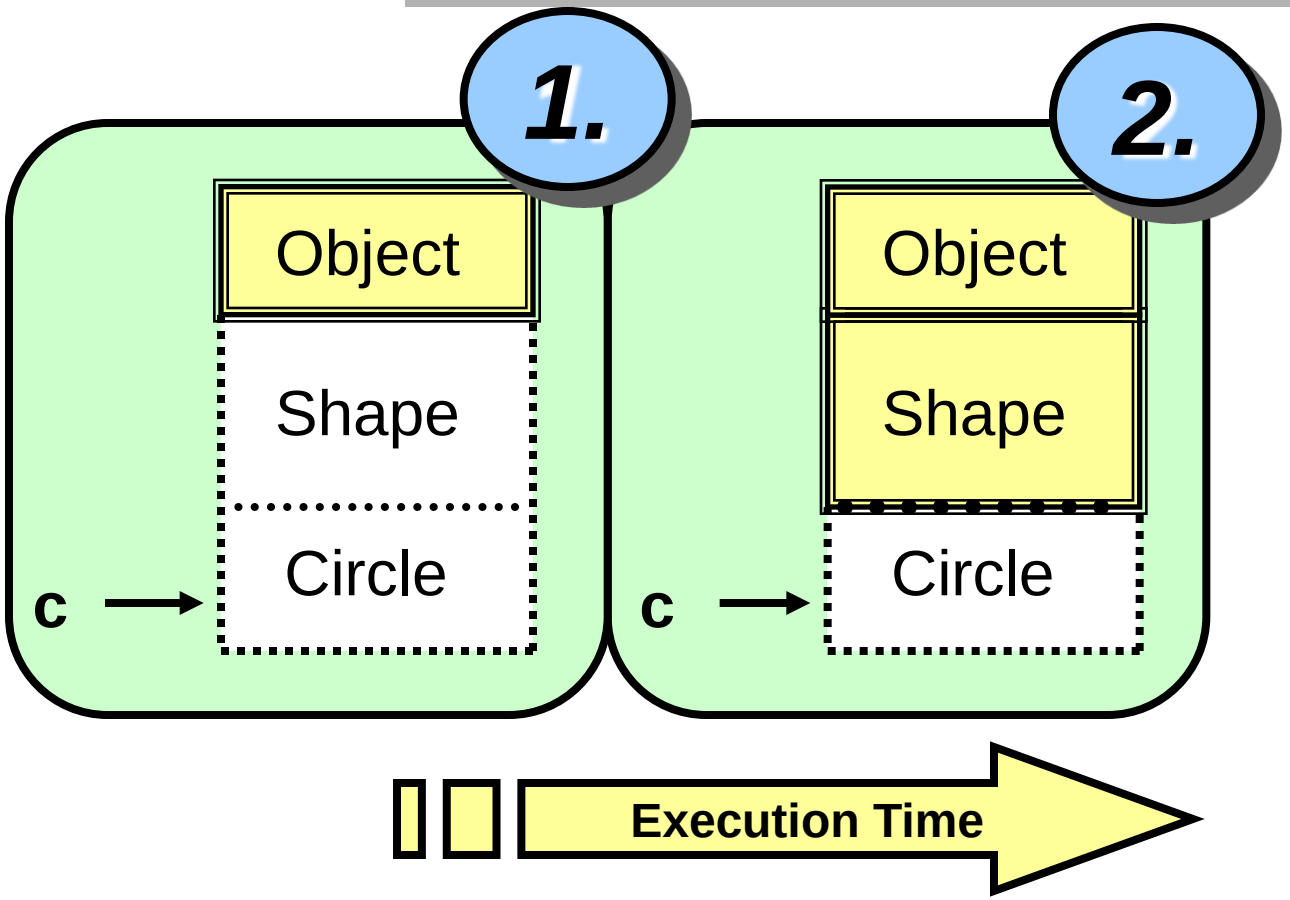
How Objects are Created

```
Circle c = new Circle( );
```



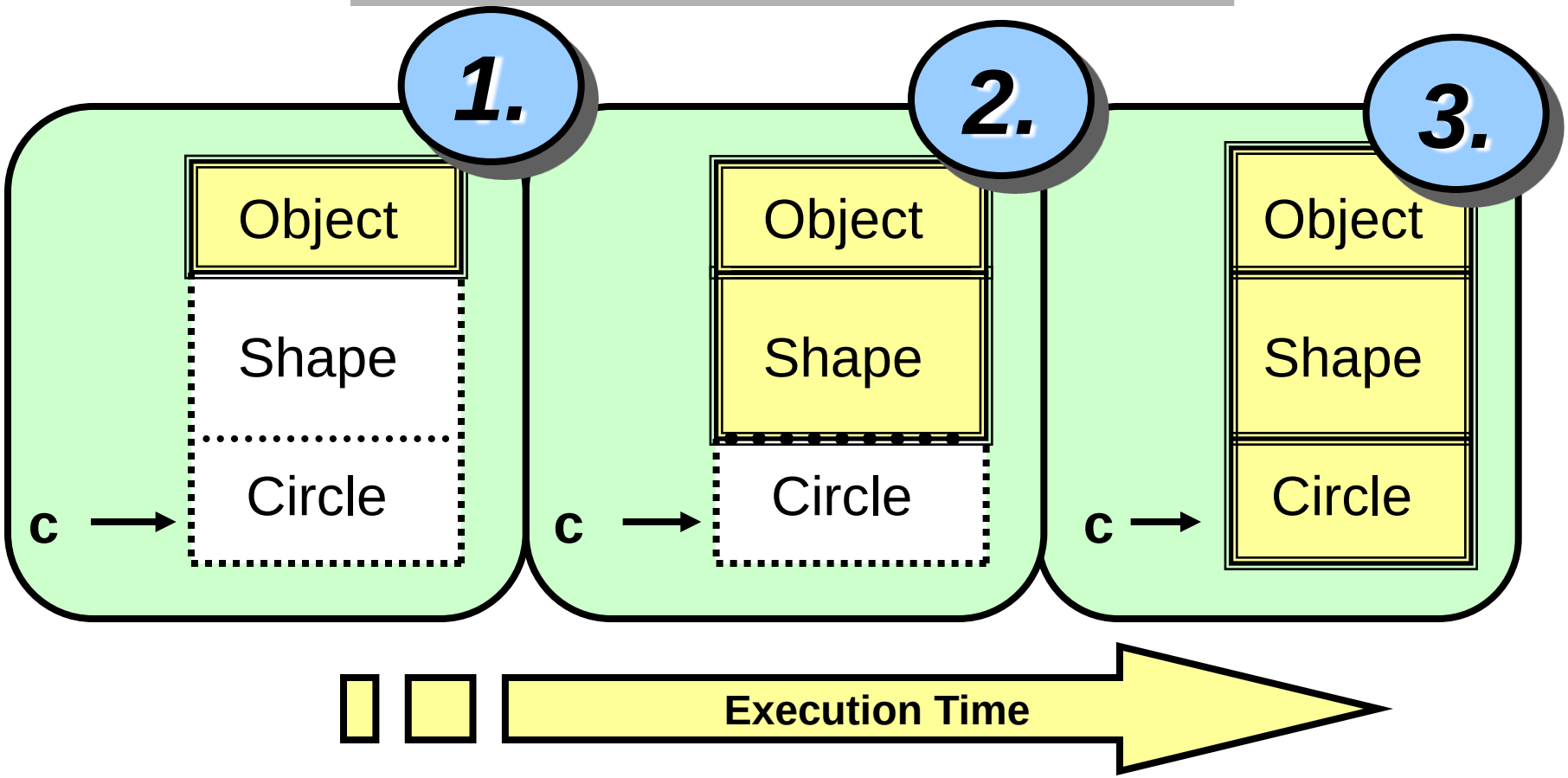
How Objects are Created

```
Circle c = new Circle( );
```



How Objects are Created

```
Circle c = new Circle( );
```

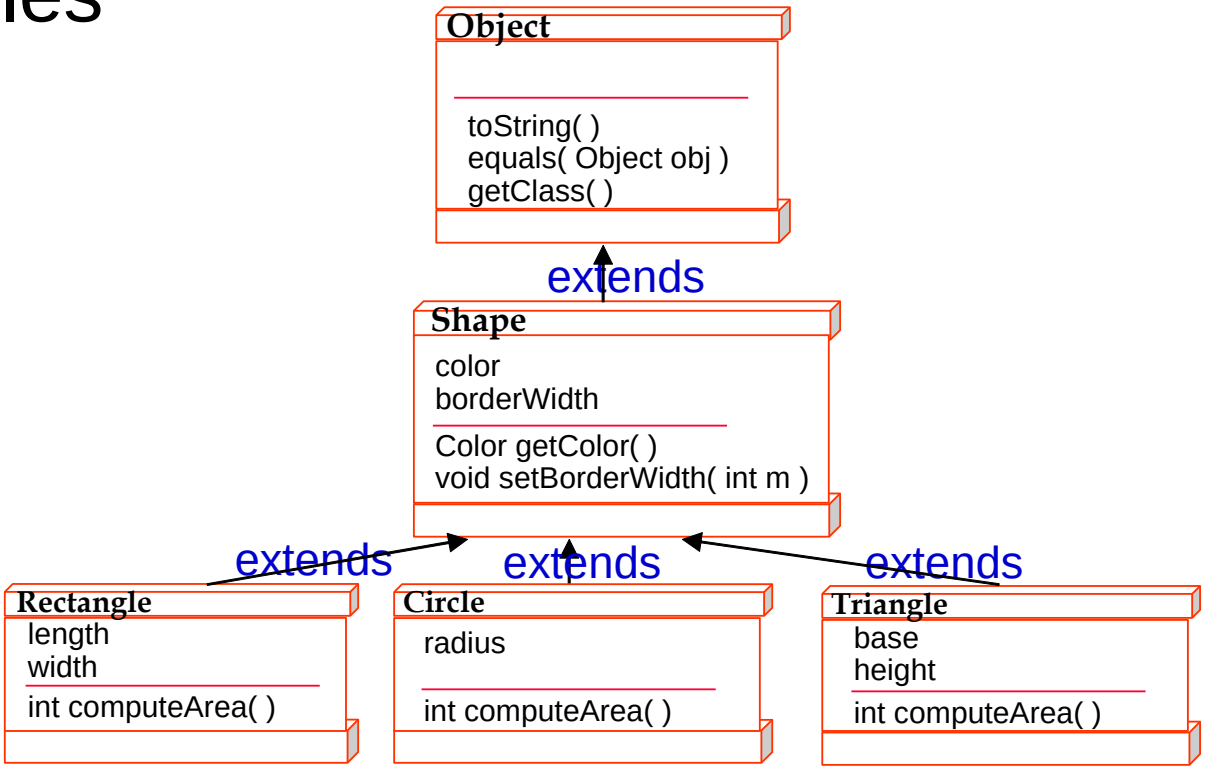


Three Common Uses for Polymorphism

1. Using Polymorphism in Arrays
2. Using Polymorphism for Method Arguments
3. Using Polymorphism for Method Return Type

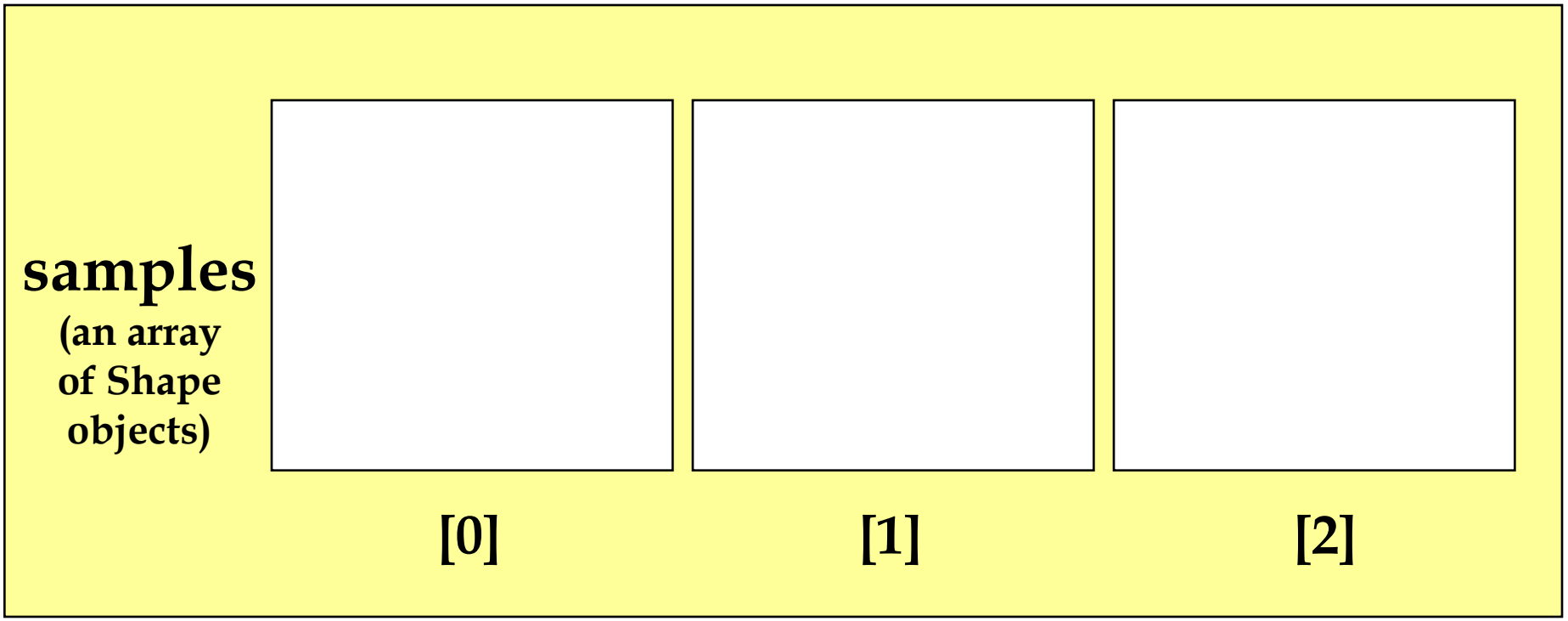
1) Using Polymorphism in Arrays

- We can declare an array to be filled with “Shape” objects, then put in Rectangles, Circles, or Triangles



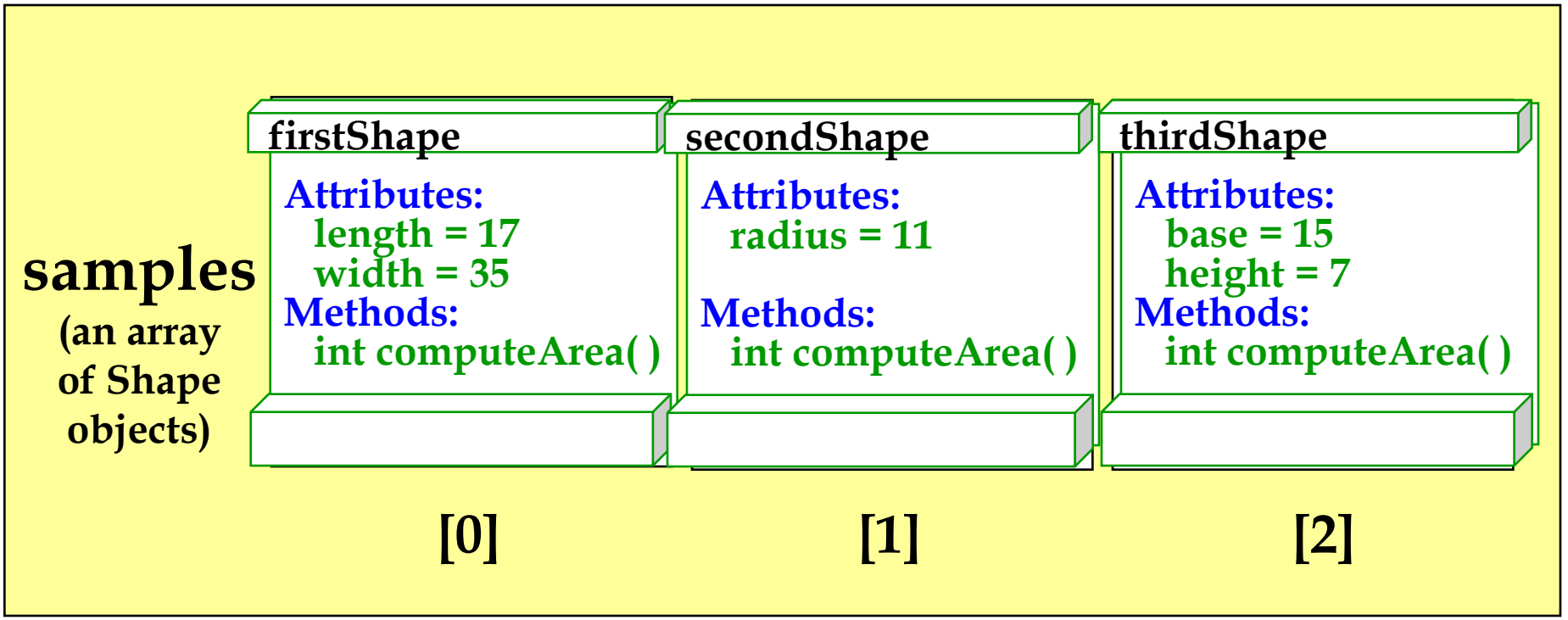
1) Using Polymorphism in Arrays

- We can declare an array to be filled with "Shape" objects, then put in Rectangles, Circles, or Triangles



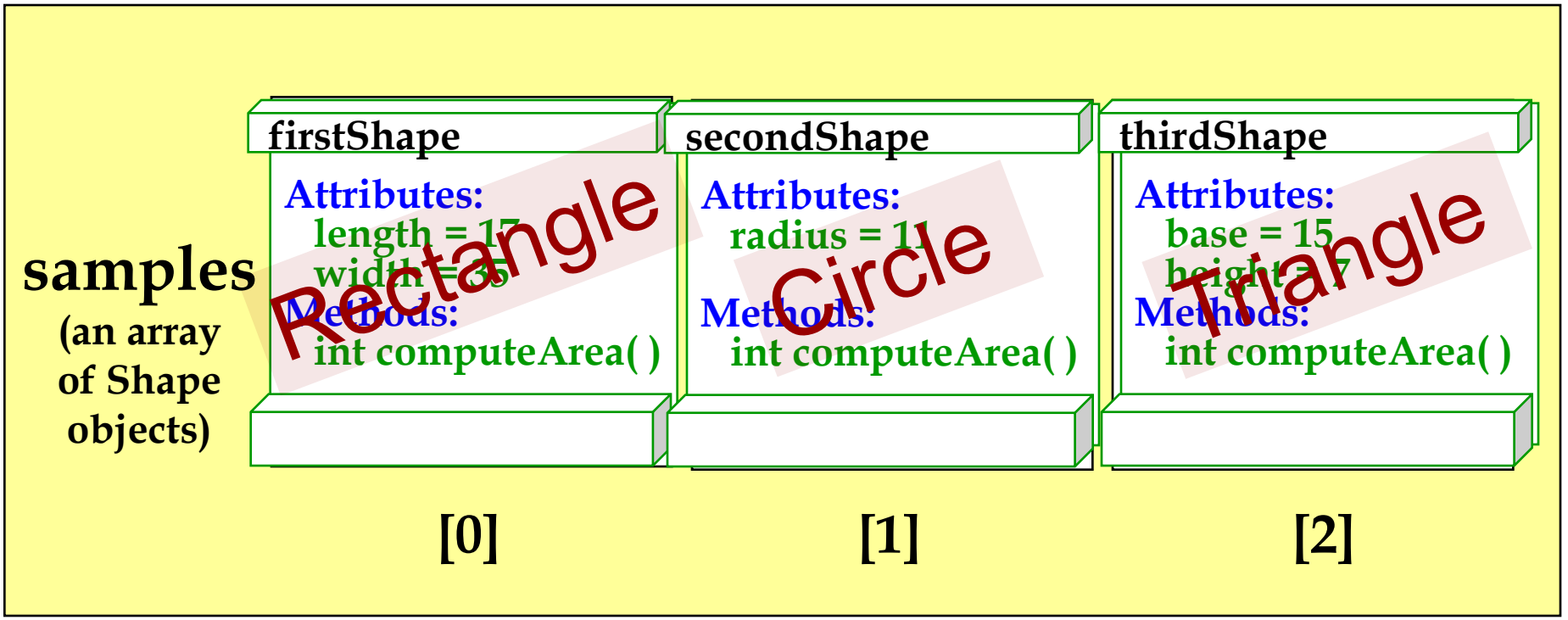
1) Using Polymorphism in Arrays

- We can declare an array to be filled with “Shape” objects, then put in Rectangles, Circles, or Triangles



1) Using Polymorphism in Arrays

- We can declare an array to be filled with “Shape” objects, then put in Rectangles, Circles, or Triangles



2) Using Polymorphism for Method Arguments

- We can create a procedure that has Shape as the type of its argument, then use it for objects of type Rectangle, Circle, and Triangle

```
public int calculatePaint (Shape myFigure) {
    final int PRICE = 5;

    int totalCost = PRICE * myFigure.computeArea( );
    return totalCost;
}
```

The actual definition of computeArea( ) is known only at runtime, not compile time – this is "dynamic binding"

2) Using Polymorphism for Method Arguments

- Polymorphism give us a powerful way of writing code that can handle multiple types of objects, in a unified way

```
public int calculatePaint (Shape myFigure) {
    final int PRICE = 5;

    int totalCost = PRICE * myFigure.computeArea( );
    return totalCost;
}
```

To do this, we need to declare in Shape's class definition that its subclasses will define the method computeArea( )

3) Using Polymorphism for Method Return Type

- We can write general code, leaving the type of object to be decided at runtime

```
public Shape createPicture ( ) {
    /* Read in choice from user */
    System.out.println("1 for rectangle, " +
      "2 for circle, 3 for triangle:");
    SimpleInput sp = new SimpleInput(System.in);
    int i = sp.readInt( );

    if ( i == 1 ) return new Rectangle(17, 35);
    if ( i == 2 ) return new Circle(11);
    if ( i == 3 ) return new Triangle(15, 7);
}
```

Object-Oriented Programming in Industry

- Large projects are routinely programmed using object-oriented languages nowadays
- MS-Windows and applications in MS-Office – all developed using object-oriented languages
- This is the world into which our students are graduating…

Conclusions

- Object-oriented programming provides a superior way of organizing programming projects
  - It encourages a high degree of modularity in programming, making large projects easier to implement
  - It provides powerful techniques like inheritance and polymorphism to help organize and reuse code
- Object-oriented languages like C++ and Java provide a good environment for beginning students to learn programming, and match real-world developments in computer programming