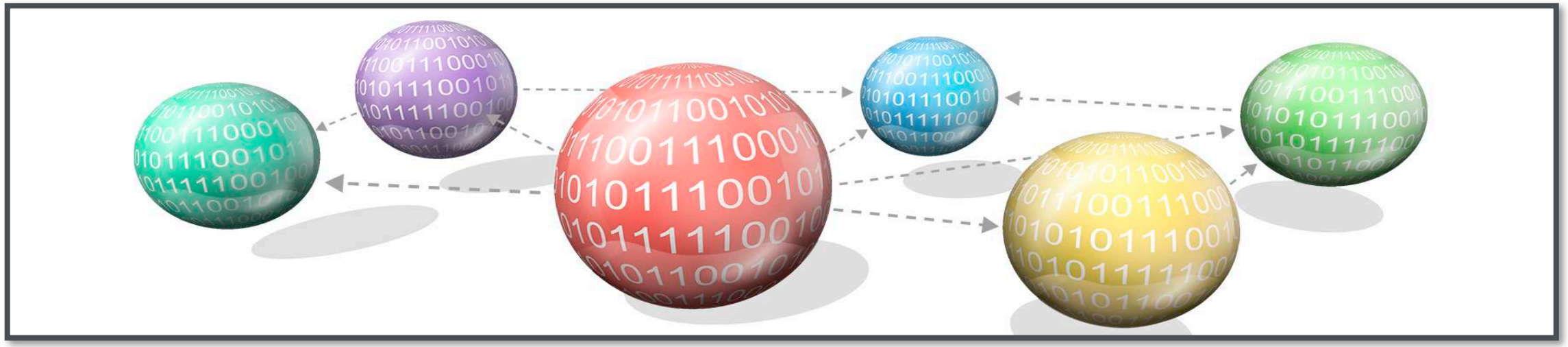
NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Chapter 1: Object-Oriented Modelling

CE2002 Object Oriented Design & Programming

Dr Zhang Jie
Assoc Prof, School of Computer Science and Engineering (SCSE)



Learning Objectives

After the completion of this chapter, you should be able to:

- Differentiate between procedural and object-oriented programming.
- Explain object-oriented concepts.

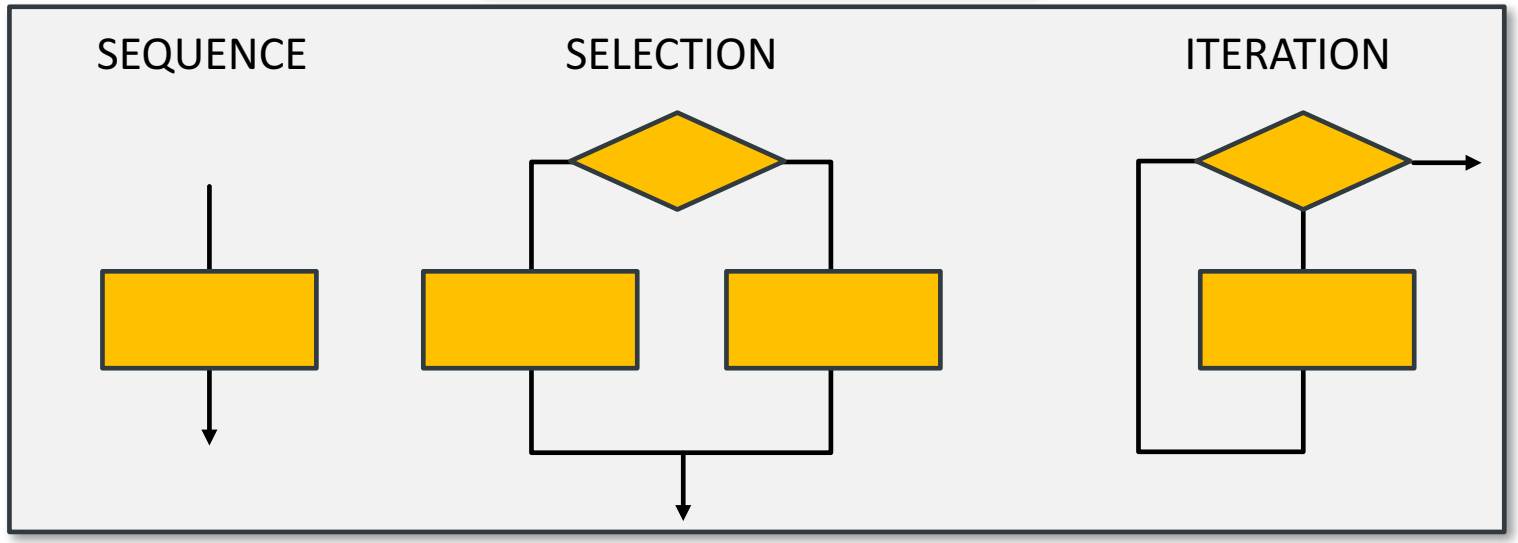
Topic 1: Procedural vs. Object-Oriented Programming

Programming Paradigm

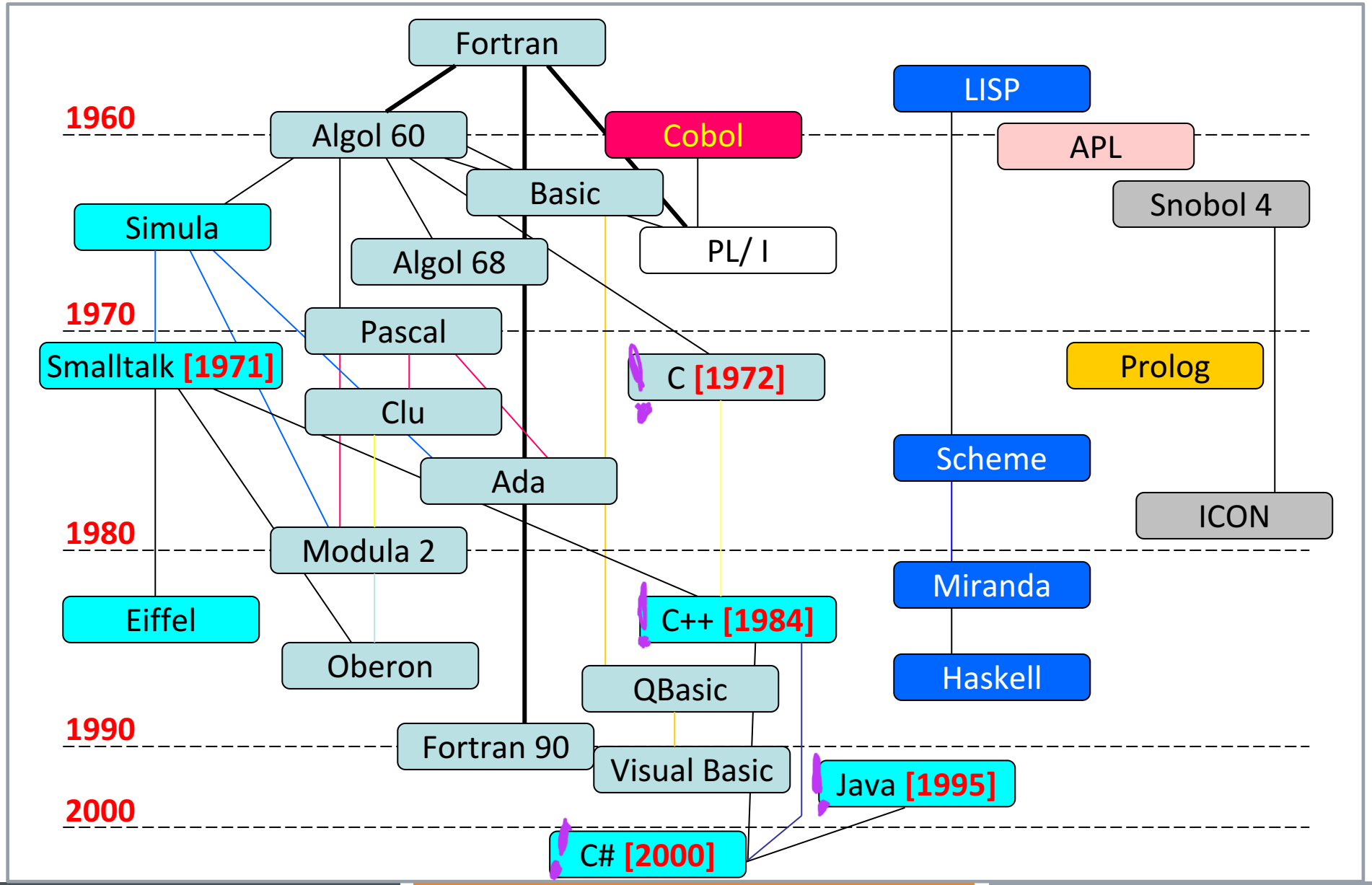
- Unstructured Programming (UP)
  - Example, Assembly language, COBOL, Basic
- Procedural Programming (PP)
  - Example, C, Pascal, ADA, Fortran
- Object-oriented Programming (OOP)
  - Java, C++, C#, Objective C, Smalltalk

```
clrscr proc near
  mov ax,0b800h
  mov es,ax
  mov di,0
  mov al,' '
  mov ah,07d
  loop_clear_12:
      mov word ptr es:[di],ax
      inc di
      inc di
      cmp di,4000
      jle loop_clear_12
      ret
endp
```

```
10  INPUT guess
20  GOTO 40
30  PRINT "This line is skipped"
40  IF guess = answer THEN GOTO 60
50  GOTO 10
60  PRINT "You've won a million"
```



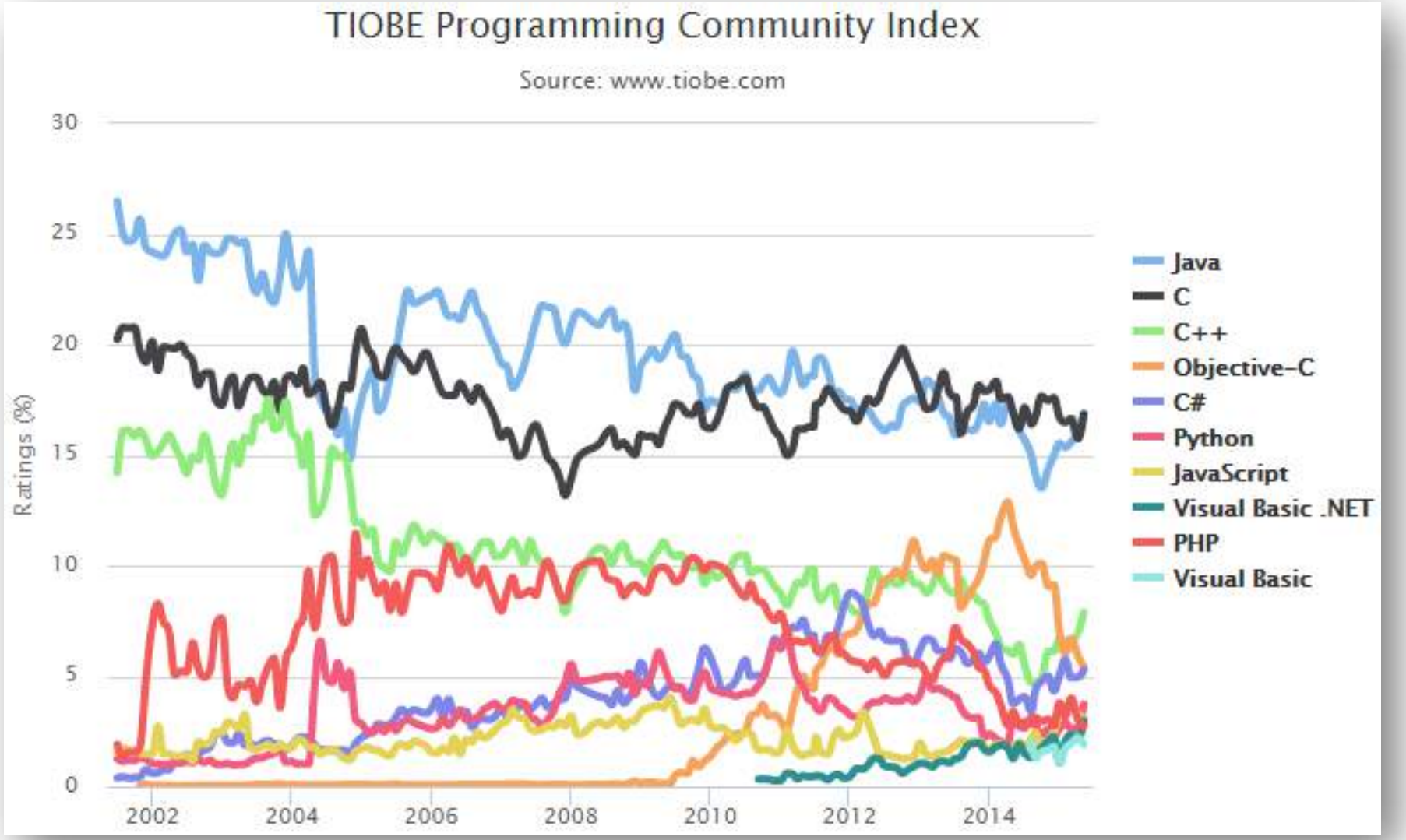
History of Programming Languages



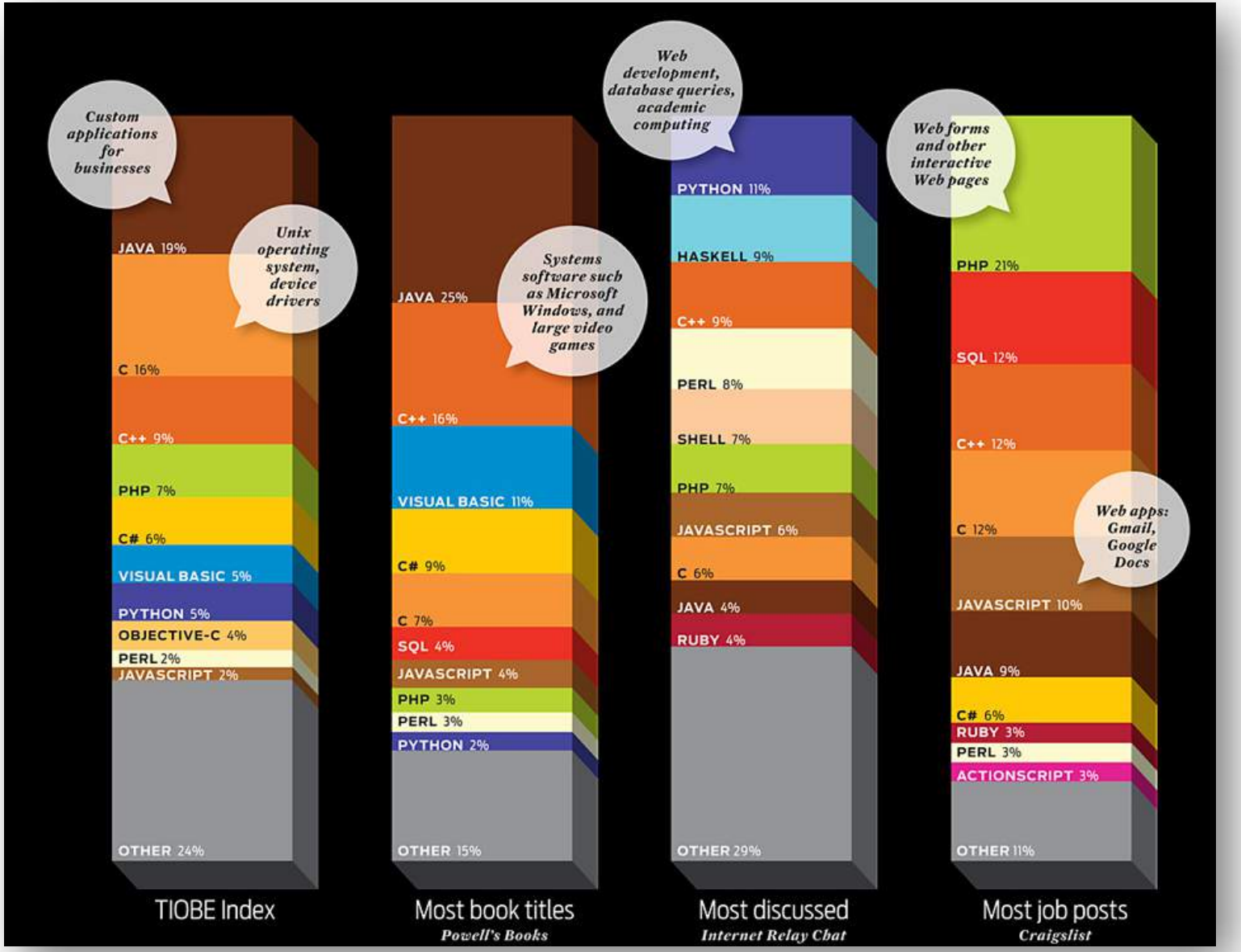
Programming Community Index for August 2011

Position Aug 2011	Position Aug 2010	Delta in Position	Programming Language	Ratings Aug 2011	Delta Aug 2010	Status
1	1	=	Java	19.409%	+1.42%	A
2	2	=	C	17.390%	-0.48%	A
3	3	=	C++	8.433%	-1.23%	A
4	4	=	PHP	6.134%	-3.05%	A
5	6	↑	C#	6.042%	+1.06%	A
6	9	↑↑↑	Objective-C	5.494%	+2.34%	A
7	5	↓↓	(Visual) Basic	5.013%	-0.40%	A
8	7	↓	Python	3.415%	-0.81%	A
9	8	↓	Perl	2.315%	-1.11%	A
10	11	↑	JavaScript	1.557%	-0.84%	A
11	23	↑↑↑↑↑↑↑↑↑↑↑↑	Lua	1.362%	+0.83%	A
12	12	=	Ruby	1.329%	-0.65%	A
13	10	↓↓↓	Delphi/Object Pascal	1.076%	-1.35%	A
14	16	↑↑	Lisp	0.905%	+0.28%	A
15	22	↑↑↑↑↑↑↑	Transact-SQL	0.823%	+0.27%	A-
16	28	↑↑↑↑↑↑↑↑↑↑↑↑	Ada	0.699%	+0.30%	B
17	19	↑↑	RPG (OS/400)	0.660%	+0.05%	B
18	17	↓	Pascal	0.659%	+0.04%	A--
19	46	↑↑↑↑↑↑↑↑↑↑↑↑	F#	0.604%	+0.37%	B
20	-	=	Assembly*	0.599%	-	B

Programming Community Index



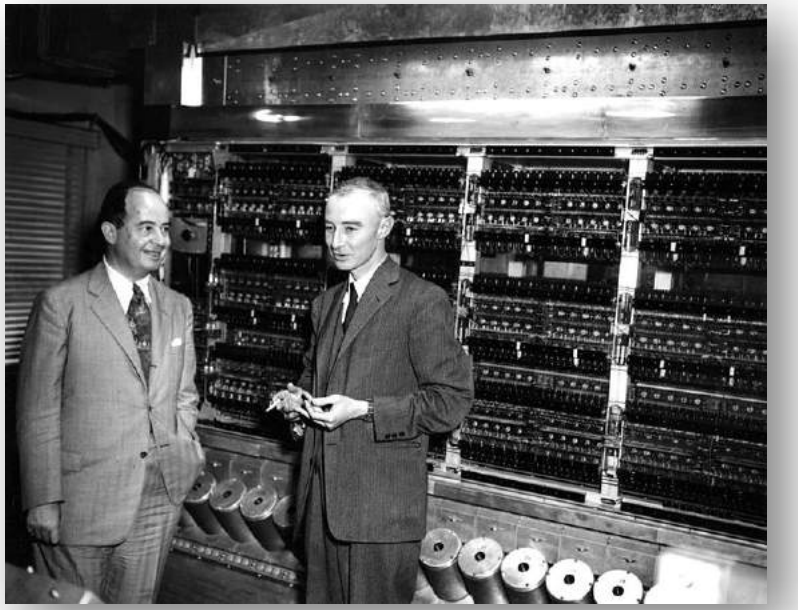
Top-10 Programming Languages



Models of Programming Languages

Procedural Programming:

- Also known as imperative languages.
- Based upon the concept of the procedure call-functions.
- A list or set of instructions telling a computer what to do step-by-step, and how to perform from the first code to the second code.
- Primary feature is a sequence of statements that represent commands.
- Use of *assignment* statements to change values of variables (data) in memory locations.



Edvac-von Neumann.
Retrieved November 11, 2016 from Wikimedia Commons
https://en.m.wikipedia.org/wiki/File:Edvac-vonNeumann.jpg.

C Language

- Developed by Kernihan and Ritchie for implementing the UNIX operating system
- Objective:
  - Provides language that has access to low-level data
  - Generates efficient code
- Programs often expressed with compact code at the expense of readability

```
char (* (*x() ) [] ) ()
```
x: function returning pointer to array[] of pointers to function returning char

1) Brian Kernighan.
Retrieved November 11, 2016 from Wikimedia Commons https://upload.wikimedia.org/wikipedia/commons/a/ae/Brian_Kernighan_in_2012_at_Bell_Labs_1.jpg.

2) Dennis Ritchie.
Retrieved November 11, 2016 from Wikimedia Commons https://upload.wikimedia.org/wikipedia/commons/6/6b/Dennis_ritchie_linux.png.

C Language

- Popularity grew in conjunction with UNIX's acceptance as an operating system (used to port UNIX).

- Excellent for construction of portable systems programs:
  - Portability is enhanced by the ANSI standard for C, the set of rules for C compilers.

Object-Oriented Programming

- Based on the notion of object, which can be described as a collection of memory locations together with all operations that can change values of these memory locations.
- Computation is represented as the interaction among, or communication between objects.

- Fundamental entity is *object* which receives and sends *messages*, performs computations, and has a local state that it can modify.
  - Solve problems by objects sending messages to one another.

Simula, Smalltalk, C++, Java, C#, etc.

C++ Language

- Developed by Bjarne Stroustrup (AT&T)
- Superset of C that includes object-oriented mechanisms without compromising the efficiency and simplicity of C
- Strict type-checking
- Class provides the mechanism for data abstraction and encapsulation

Bjarne Stroustrup

Bjarne Stroustrup
Retrieved November 11, 2016 from Wikimedia Commons https://upload.wikimedia.org/wikipedia/commons/5/5e/Stroustrup_kent_state.jpg.

C++ Language

- Not pure OOP:
  - C++ does not force you to use its OOP features
  - Can create code that uses only C++'s non-OOP features

- C++: A black Firebird, the all macho car comes with optional seatbelt (lint).

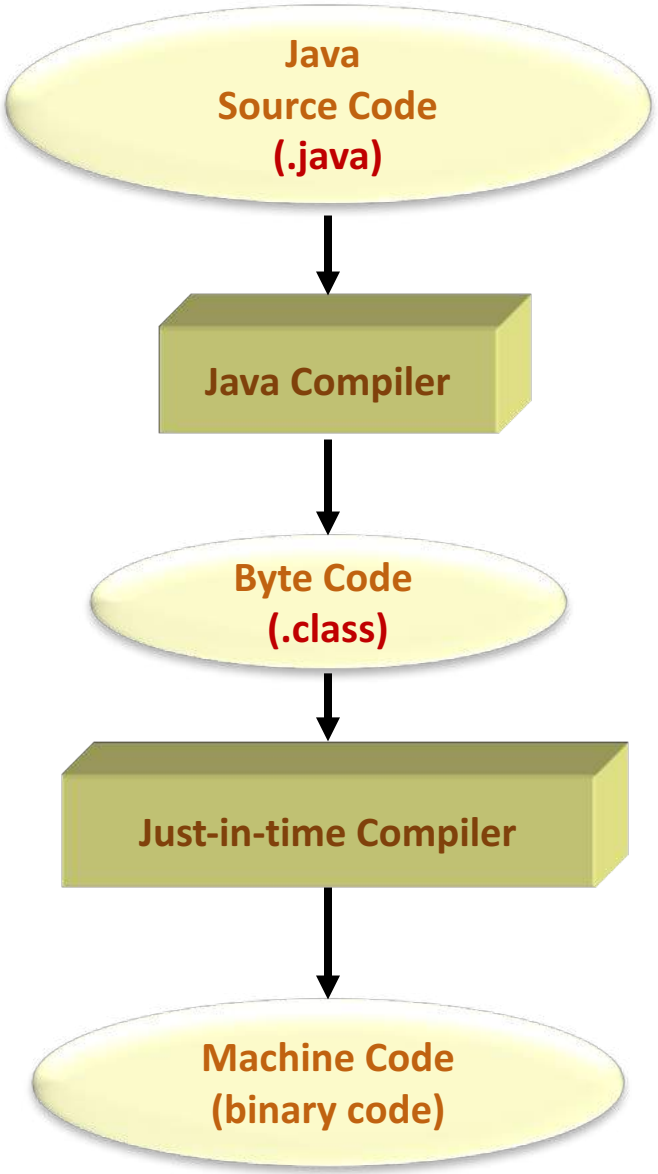
Java

- Developed by SUN for embedded consumer goods; now popular for Web.
- Based on C/C++ but smaller, more reliable, more portable, and has restrictions like:
  - no pointers;
  - no multiple-inheritance; and
  - speed (interpreted byte-code) - JIT compilers.
- Java: All-terrain very slow vehicle.
- “Compile Once, Run Anywhere”.

1) James Gosling
Retrieved November 11, 2016 from Wikimedia Commons
https://commons.wikimedia.org/wiki/File:James_Gosling_2008.jpg

2) Fondo Java
Retrieved November 11, 2016 from Wikimedia Commons
https://commons.wikimedia.org/wiki/File:Fondo_Java.png

Java Translation and Execution



Java Virtual Machine (JVM)
Java Runtime Environment (JRE)

- Developed by Microsoft, led by Anders Hejlsberg (creator of Turbo Pascal, Delphi)
- Based on C++, but many similar features of Java
  - No pointers
  - No multiple-inheritance
  - No globals
  - Strictly OO
  - Automatic garbage collection
  - Multi-threading
  - Cross-platform

Anders Hejlsberg.
Retrieved November 11, 2016 from Wikimedia Commons
https://commons.wikimedia.org/wiki/File:Anders_Hejlsberg.jpg

C

```c
#include <stdio.h>

int  x, y, z ;        // global variable

void function1(…)
{    x = y + 1 ;
….}

int function2(…)
{….  function1(..) ; }

int function3(…)
{….function2(..) ;
    y = z * 2;    }

int main(void)
{
….
    function3(..);
…..
}
```

Procedural Programming

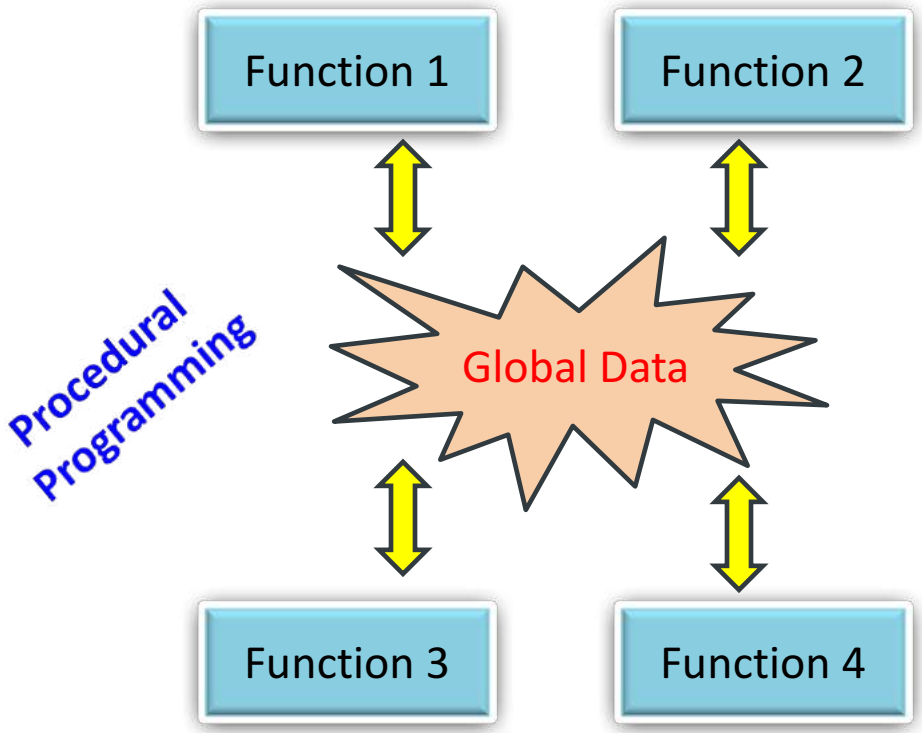
Java

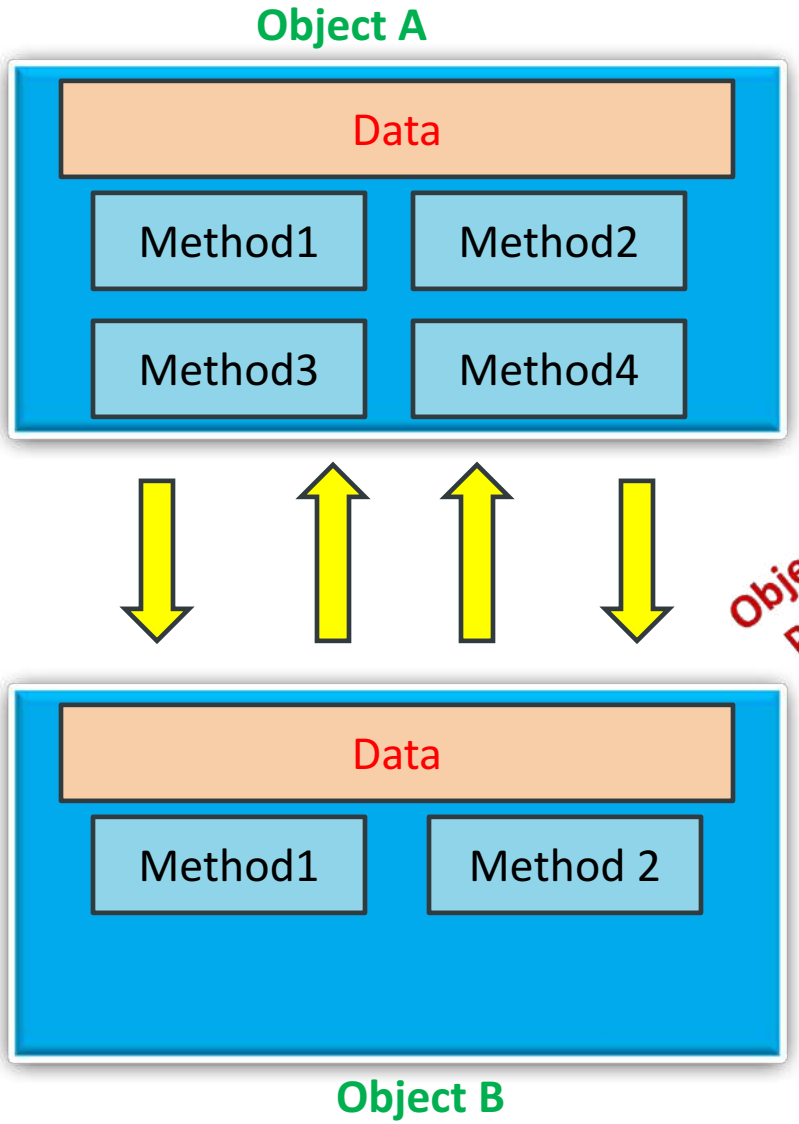
```java
import java.util.*;
class ClassA{          // in ClassA.java
      int x, y = 0 ;    // attributes
     public int methodA(…)
     { ….. }
…..
}
```

```java
…..
class ClassB{          // in ClassB.java
     public int methodB(…)
     { ….. }
…..
}
```

Object-Oriented Programming

```java
class Driver{ // in Driver.java
     public static void main(…)
     {
        ClassA  a = new ClassA(1, 2);
        ClassB  b = new ClassB();
        a.methodA(..);
          b.methodB(..);
     }
}
```





Computation is done by objects talking to each other

more modular

C vs. Java: Simple Comparison

```c
#include <stdio.h>



int getFact(int n) {
  int c, fact = 1;
  if ( n < 0 )
        printf("Number should be non-negative.");
  else
        for (c = 1; c <= n; c++)
            fact = fact * c;
   return fact;
}

int main()
{
  int n = 1;

  printf("Enter a number to calculate it's factorial\n");
  scanf("%d", &n);

  printf("Factorial of %d = %d\n", n, getFact(n));

  return 0;
}
// save as <anyname>.c
```

```java
import java.util.Scanner;

class Factorial   // class name
{
  public static int getFact(int n) {
     int c, fact = 1;
     if ( n < 0 )
       System.out.println("Number should be non-negative.");
     else
          for ( c = 1 ; c <= n ; c++ )
          fact = fact*c;
     return fact ;
  }

  public static void main(String args[])
  {
     int n = 1;

     System.out.println("Enter an integer to calculate it's factorial");
     Scanner in = new Scanner(System.in);
     n = in.nextInt();
     System.out.println("Factorial of "+n+" is = " + getFact(n));


  }
}
//  must save as <class name>.java => Factorial.java
```

Additional Reading

- Procedural vs. Object-Oriented
  - Text book: Page 7-12
- Java Programming
  - Lecture Slides: Blackboard/ NTU Learn -> Content
  - Recorded Lectures: Blackboard/ NTU Learn
- C vs. Java
- Java Quick Guide
  - Blackboard/ NTU Learn -> Content -> Course -> Additional Resources

Topic 2: Classes and Objects

What is an Object?

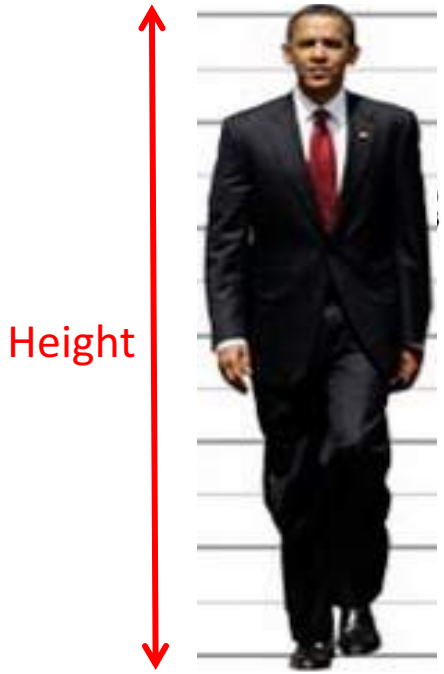
- Objects are everywhere.

- Characteristics of objects:
  - Identity
  - State
  - Behaviour

One Example

Characteristics?

Person



Weight

Identity: Barack Obama

Attributes/ States

height: 187cm
weight: 81.6 kg
gender: M
age: 49
wealth:
.....

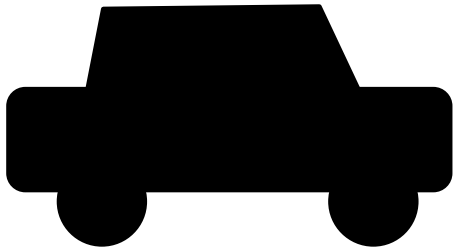
Behaviours

Eat

Study/ Sleep

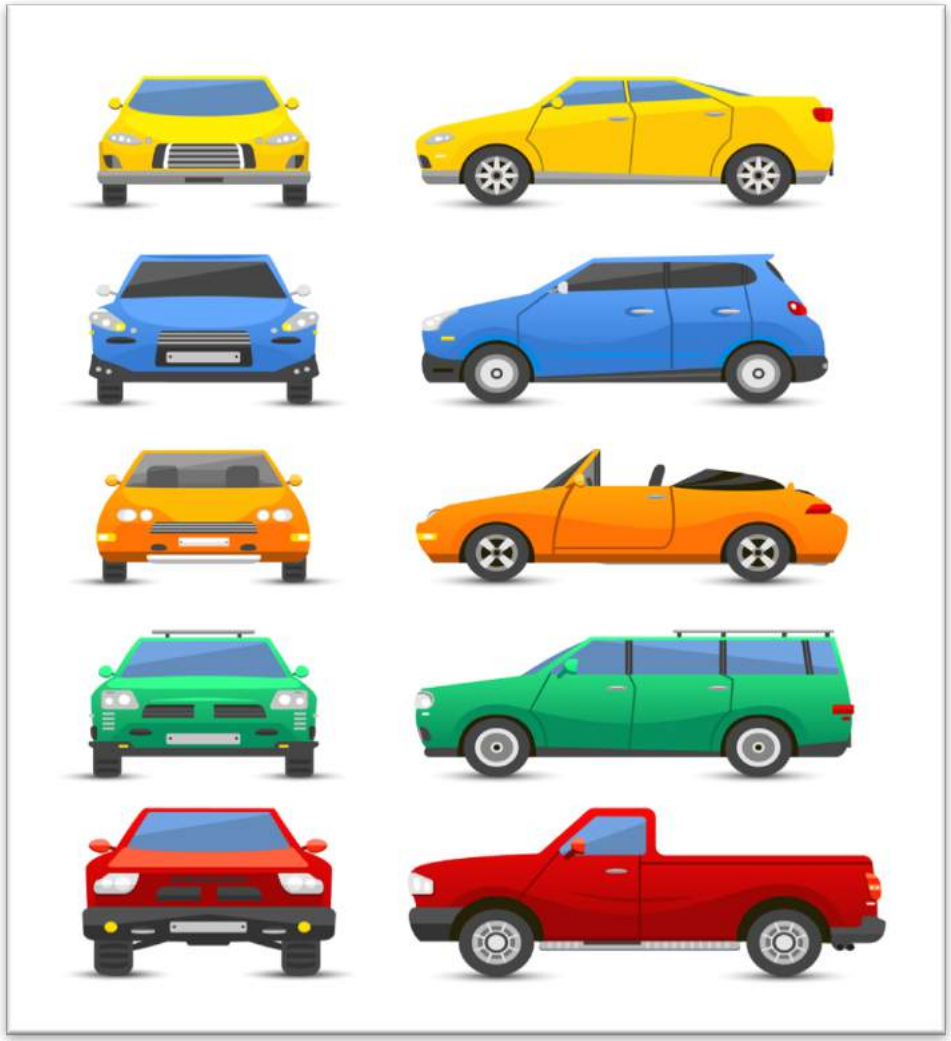
Person is a *Class*
Obama is an Object (instance) of Person Class

Another Example



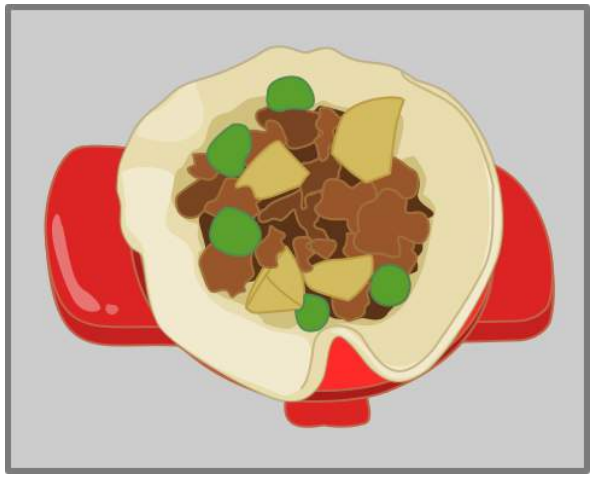
Attributes

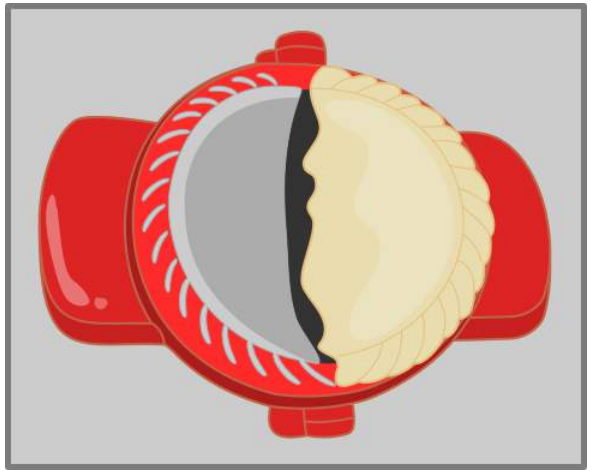
- Identity: Machine serial number
- State/ Properties: Color, maximum speed, etc.
- Behavior/ Methods: Turn, brake and accelerate

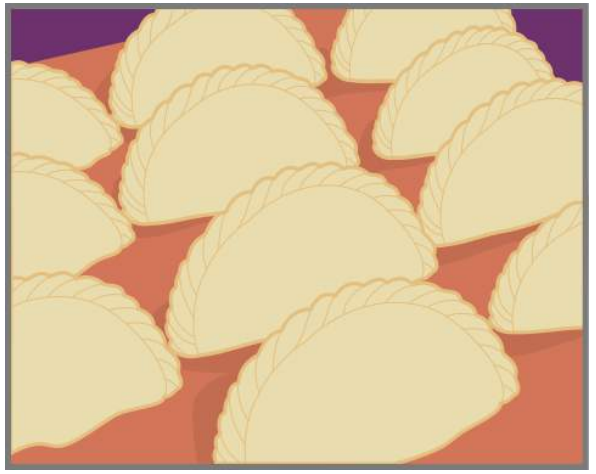


Four Basic Components of Object-Oriented Model

- **Objects:** An entity that contains both the *attributes* that describe the state of a real-world object and the *actions* that are associated with the real-world objects.
- **Messages:** Requests from one object to another for the receiving object to produce some desired result.
- **Methods:** Descriptions of operations/ actions that an object performs when it receives a message.
- **Classes:** A template for objects which consists of methods and state descriptions that objects belong to.







Four Main Concepts/ Features of Object-Oriented Model

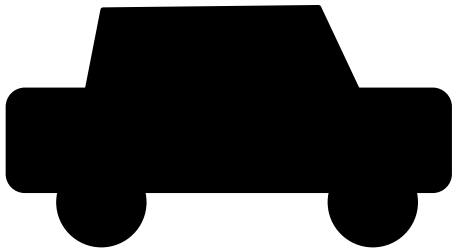
- Abstraction
- Encapsulation/ Information Hiding
- Inheritance
- Polymorphism

Abstraction

Definition:

"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."

Grady Booch, *Object-Oriented Analysis and Design with Applications*, 1994



hide unwanted data from user

Abstraction

- **Object:** An entity that contains both the *attribute* that describes the state of a real-world object and the *actions* that are associated with the real-world object.

- **State** of an object: *"encompasses all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties"* {Grady Booch, *Object-Oriented Analysis and Design with Applications*, 1994}.

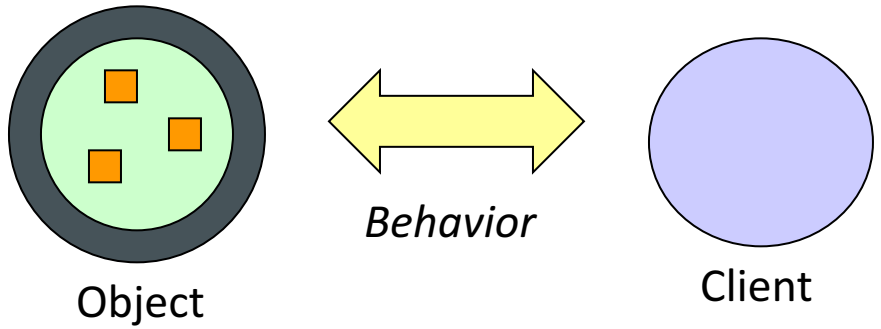
- **Attributes:** The data or variables that characterise the state of an object.

Abstraction

- **Behaviour** of an object:
  - *Methods* describe the behaviour associated with an object. They represent the:
    - Services that the object provides to other objects.
    - Operation that the object may perform upon other objects.
    - Interfaces provided by the object.

Abstraction Barrier

Object	**Aston Martin Car**	
Attributes	**State**	**Behaviour**
Color	Black	Steer
NoOfDoors	2	ChangeGear
….	….	….



Encapsulation and Information Hiding

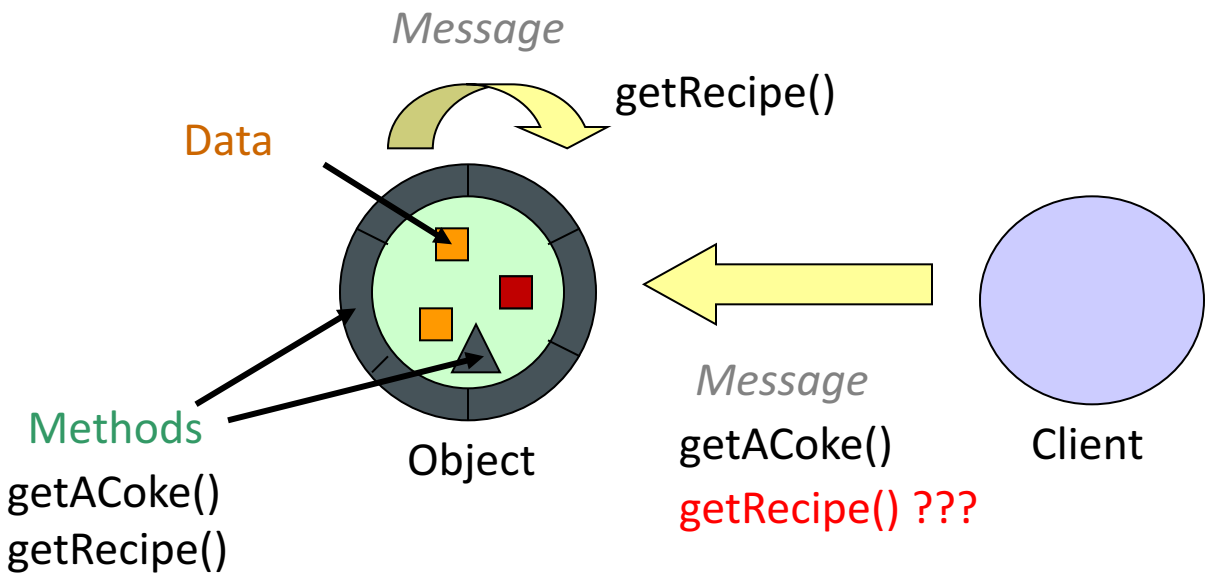
- **Encapsulation** builds a barrier to protect an object's private data. Access to private data can only be done through the public methods of the object's class (e.g. get and set methods).
- **Information hiding** hides the details or implementation of the class from users.
- With encapsulation, the users of a class only need to know **what** a class does and **how** to call the methods to perform the tasks.
- The users **do not** need to know the **implementation** details for the methods.

Encapsulation and Information Hiding

My name is Shadow, I am 48 years old and weigh 70 kg.

Encapsulation and Information Hiding

Actual value of internal state is determined by methods that the object performs in response to ***messages*** received.



Encapsulation and Information Hiding

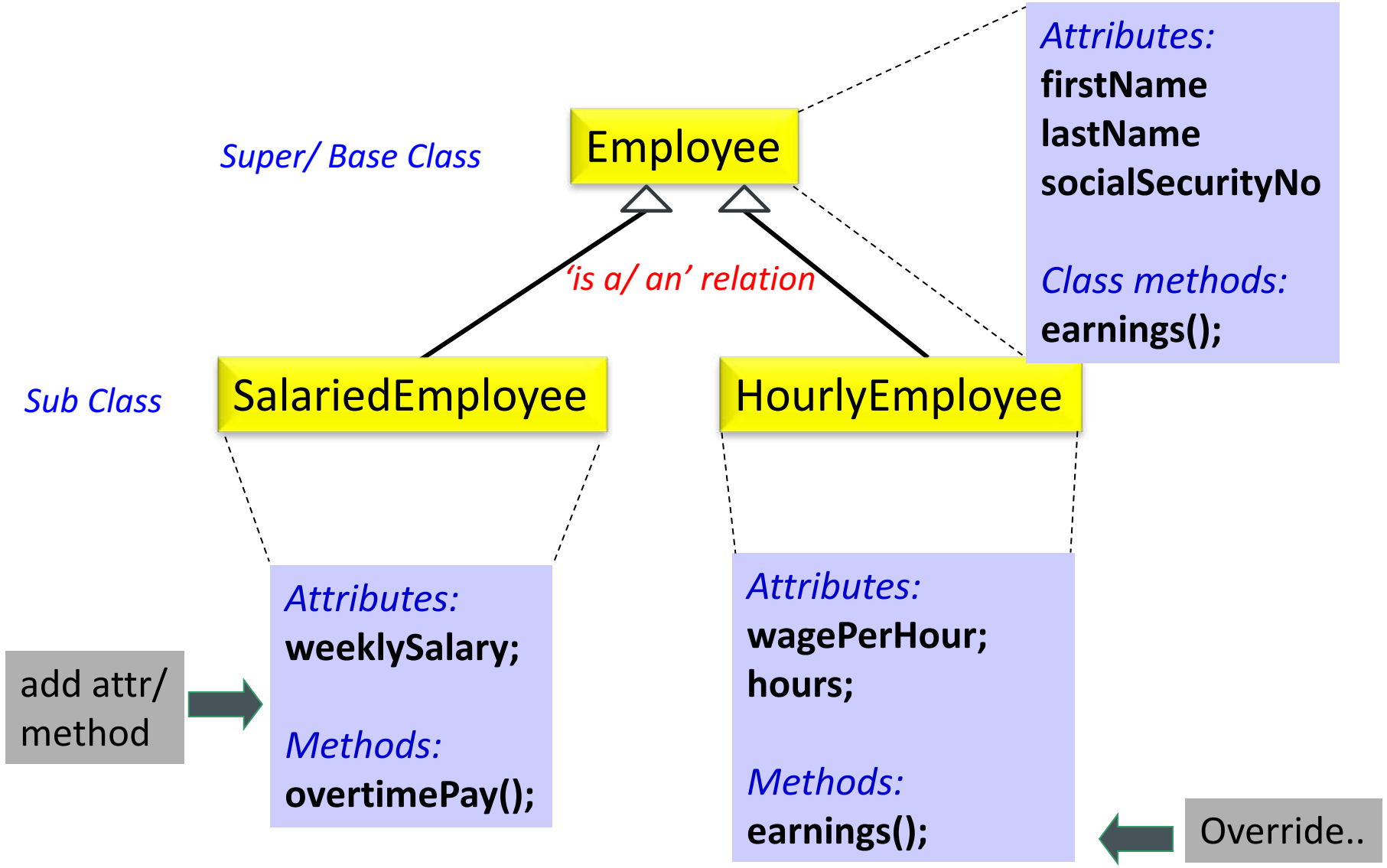
They help to:

- Protect what you have
- Protect your data

Inheritance

- A mechanism that defines a new class which inherits the properties and behaviours (methods) of a parent class.
- **Superclass** or **base class (parent)**:
  - The class from which another class inherits structures and/or behaviours
- **Subclass** or **derived class (child)**:
  - A class that inherits from one or more other classes
  - Any inherited behaviour may be redefined in subclass, *superseding* inherited definition
- Can create new classes without extensive duplication of code
- Reuse parent's code

Inheritance Example

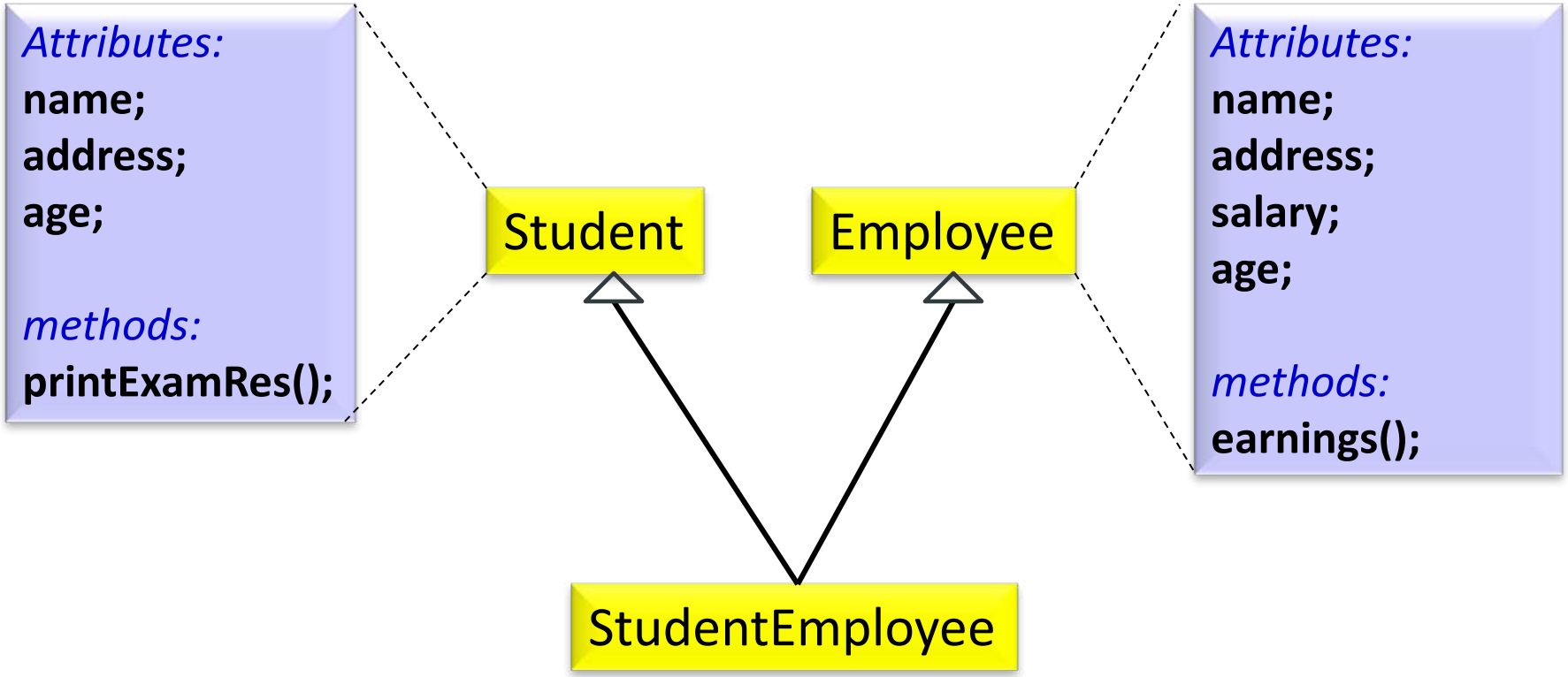


Multiple Inheritance

- A class inherits from more than one superclass
- Problem that arises is which property/ method to inherit if property/ method is present in more than one superclass
- User specified precedence (Smalltalk)
- Rename one property/ method
- Accept both if signatures are different

```
print(String s)
print(Char c)
```

Multiple Inheritance



Polymorphism

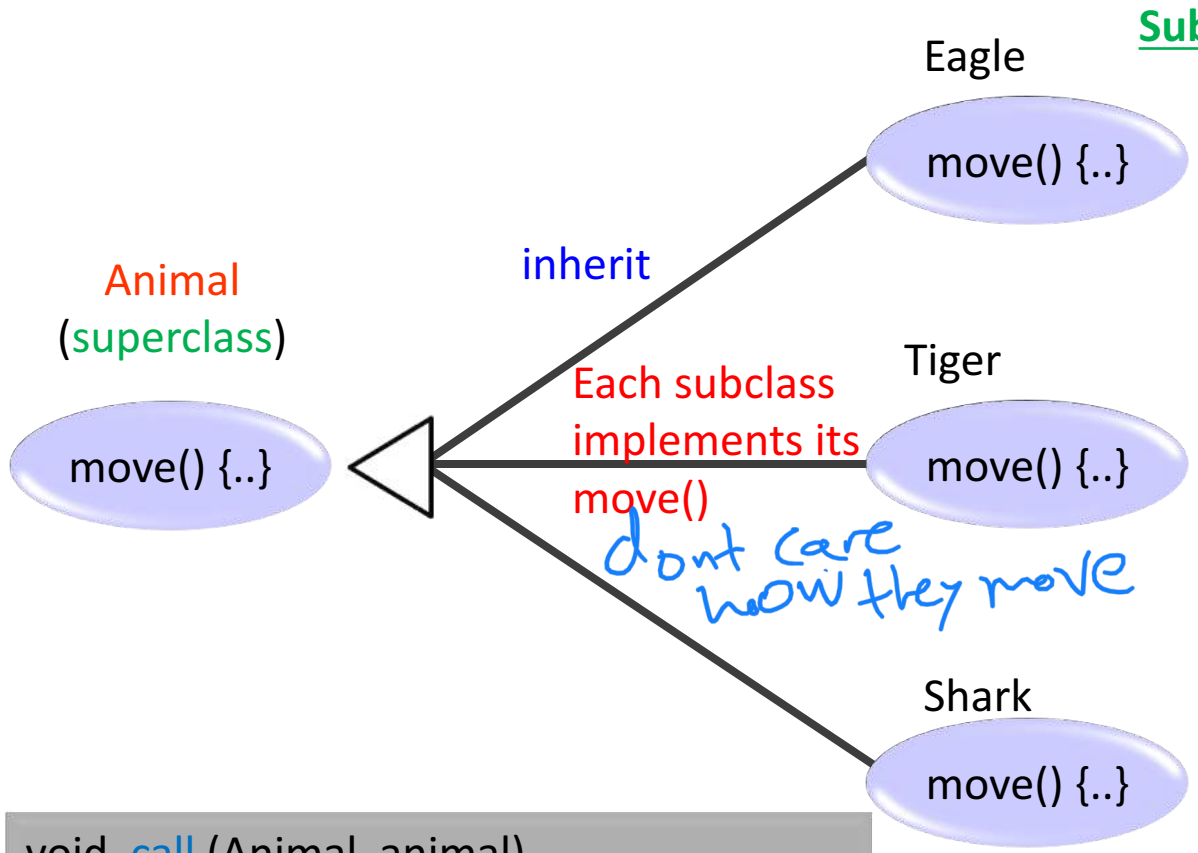
- Greek word that literally mean *many shapes* or *multiple forms*

- Same message can be sent to different objects**
  - Each object performs operation appropriate to its class
  - Do similar things differently

- Sending object does not need to know the class of receiving object or how the object will respond

- Most powerful concept of OO

Polymorphism

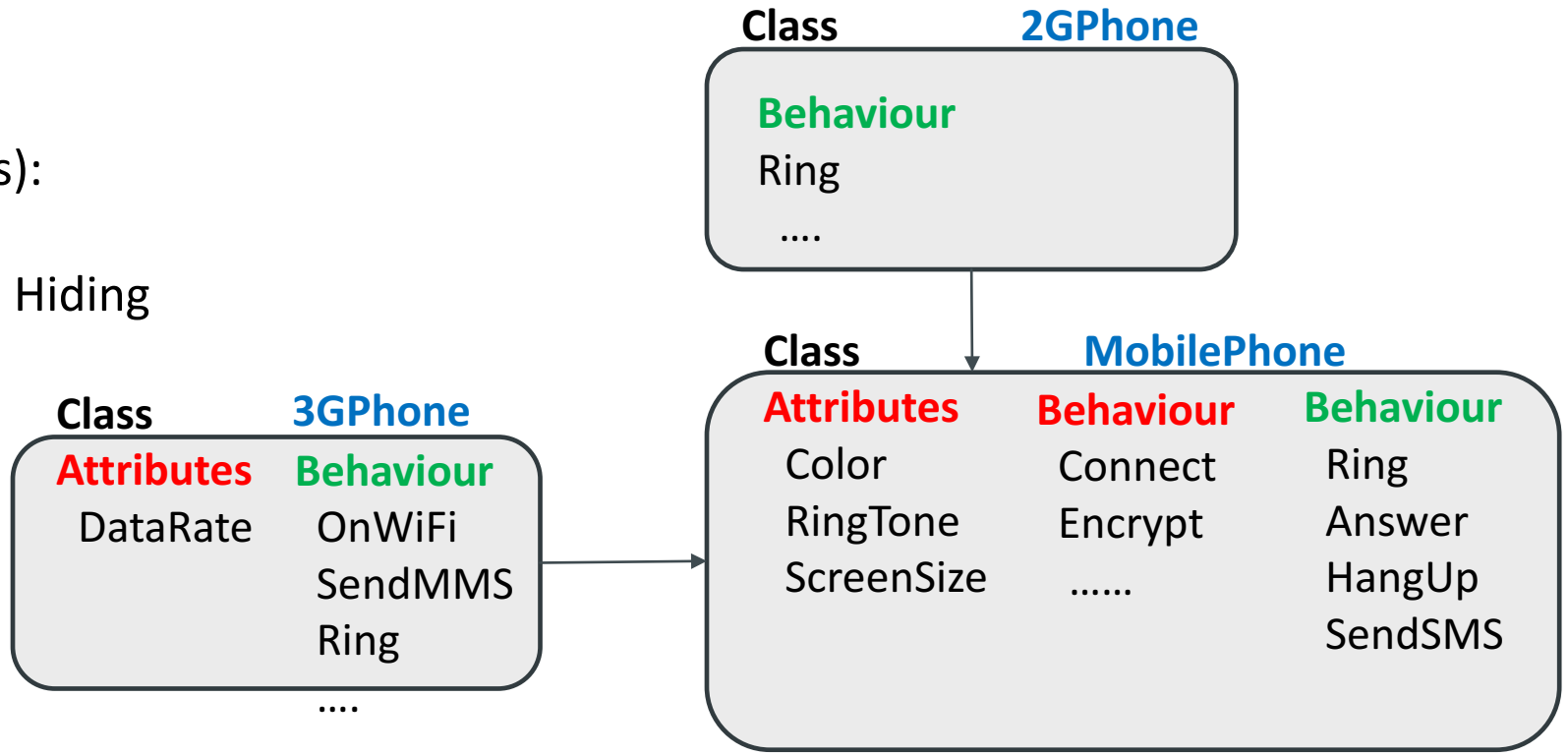



```
void  call (Animal  animal)
{
     animal.move();
}
```

Summary

Key points from this chapter:

- Four basic components:
  - Objects
  - Messages
  - Methods
  - Classes
- Four main properties (concepts):
  - Abstraction
  - Encapsulation/ Information Hiding
  - Inheritance
  - Polymorphism



OO Conceptual Terms

- Object
- Class
- Attribute
- Operation
- Interface
- Implementation
- Association
- Aggregation
- Composition
- Generalisation

- Super-Class
- Sub-Class
- Abstract Class
- Concrete Class
- Polymorphism

Additional Reading

- Class and Object
  - Text Book: Page 12-20

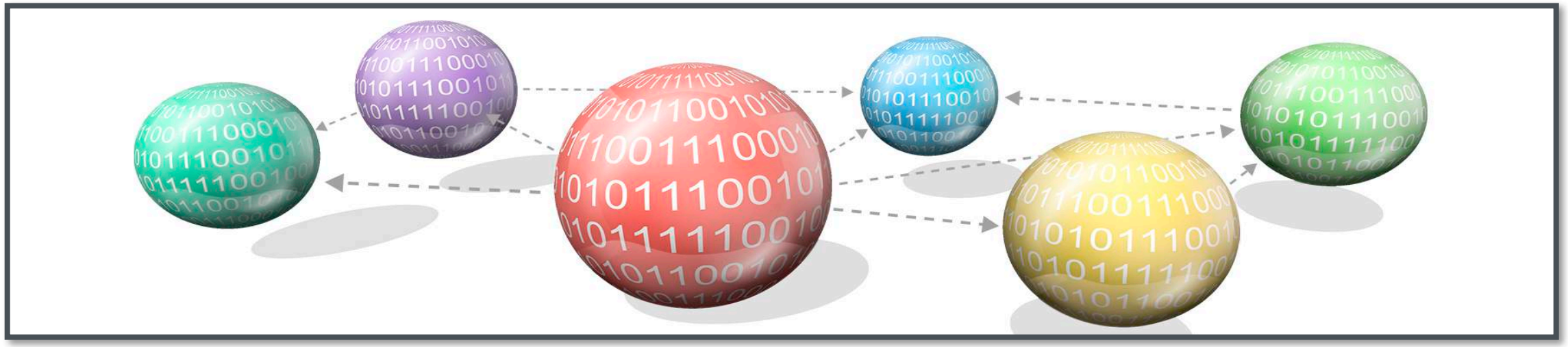
- Four Main Concepts of OO
  - Text Book: Page 20-33

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Chapter 2: Class and Object

CE2002 Object Oriented Design & Programming

Dr Zhang Jie
Assoc Prof, School of Computer Science and Engineering (SCSE)



Learning Objectives

After the completion of this chapter, you should be able to:

- Describe classes and objects in Object-Oriented Programming (OOP).
- Define class in OOP.
- Explain the concept of message sending.
- Explain copying objects.
- Explain '*this*' keyword.
- Describe Accessors and Mutators.
- Describe static keyword.
- Differentiate between static and instance method.
- Differentiate between encapsulation and information hiding.
- Describe object composition.
- Explain '*string*' class.

Additional Reading

- Class Definition:
  - Textbook: Chapter 4, Page 75-86
- Constructors:
  - Textbook: Chapter 3, Page 53-60
- Java Programming:
  - Lecture Slides: Blackboard/ NTU Learn -> Content
  - Recorded Lectures: Blackboard/ NTU Learn
- C vs. Java
- Java Quick Guide
  - Blackboard/ NTU Learn-> Content -> Course -> Additional Resources

Topic 1: Classes and Objects

What is an Object?

Objects are everywhere!

Characteristics:

- Identity
- State
- Behaviour

Object Identity

Each object has its own **unique** identity (or Object Identifier (OID)).

Bill Gates

John Hui

Bill Gates. Retrieved November 11, 2016 from Wikimedia Commons https://commons.wikimedia.org/wiki/File:Bill_Gates_July_2014.jpg.
John Hui. Retrieved November 11, 2016 from Wikimedia Commons https://en.wikipedia.org/wiki/Lap_Shun_Hui

Object State (Data Members)

- Made up of **attribute(s)** (or property)
- Example: A person has properties

- Gender
- Age
- Monthly Salary

Bill Gates

- Male
- 57 years old

>>> • $72,000,000

John Hui

- Male
- 47 years old

≈ • $7,000

Object Behaviour

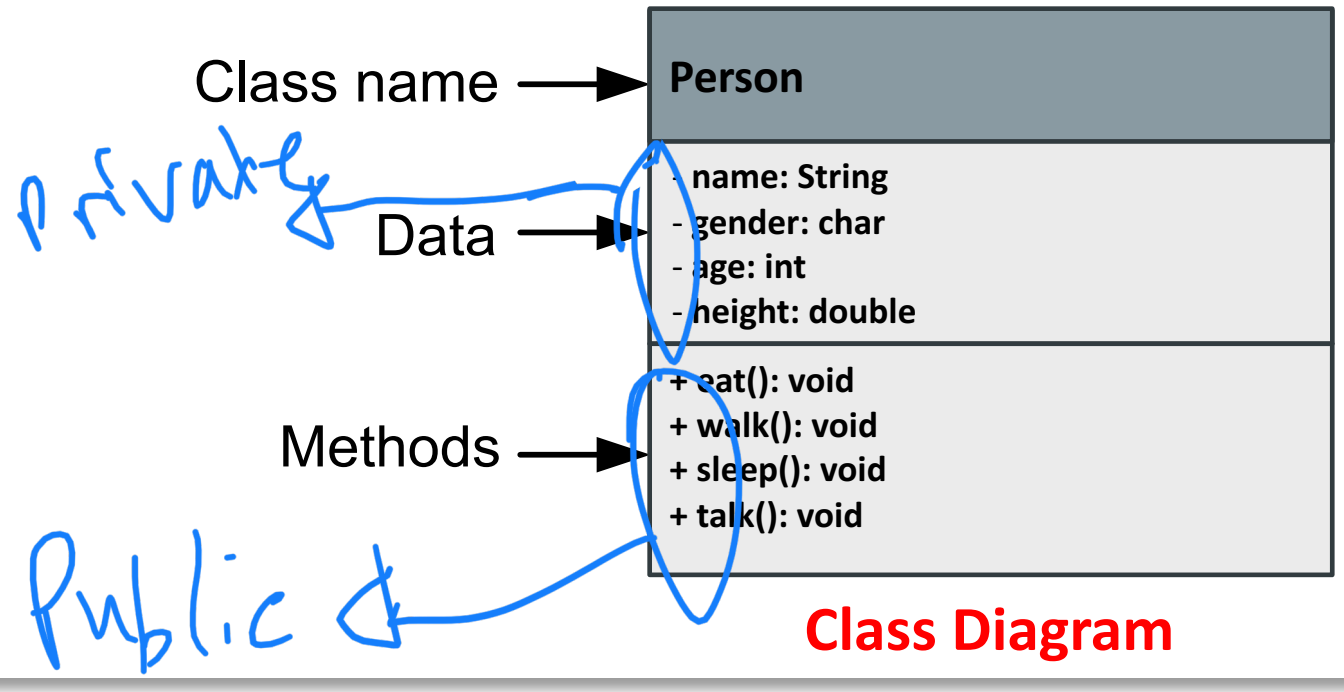
Behaviour refers to the **things that an object can do**.

Example: What can a person do?

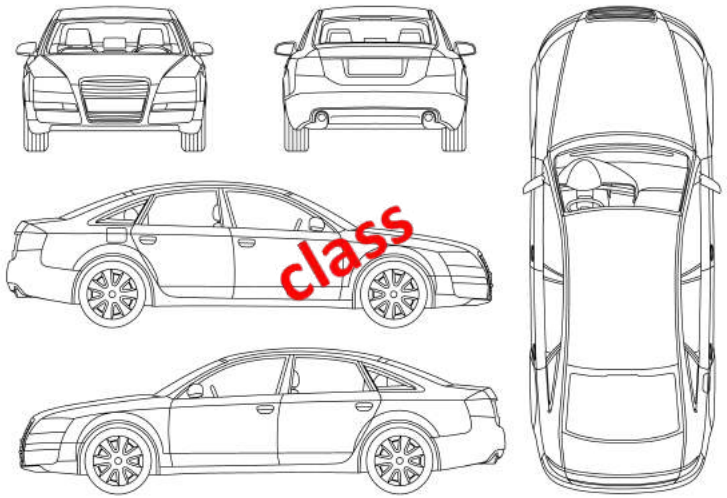
These operations that can be acted upon (performed) by an object are referred to as methods.

What is a Class?

- A class defines the **blueprint/ structure** for creating objects. In Java, a class contains:
  - **Data properties** (or attributes)
  - **Methods**
- To create an object, we need to define a class for it.
- UML (Unified Modelling Language) diagram:

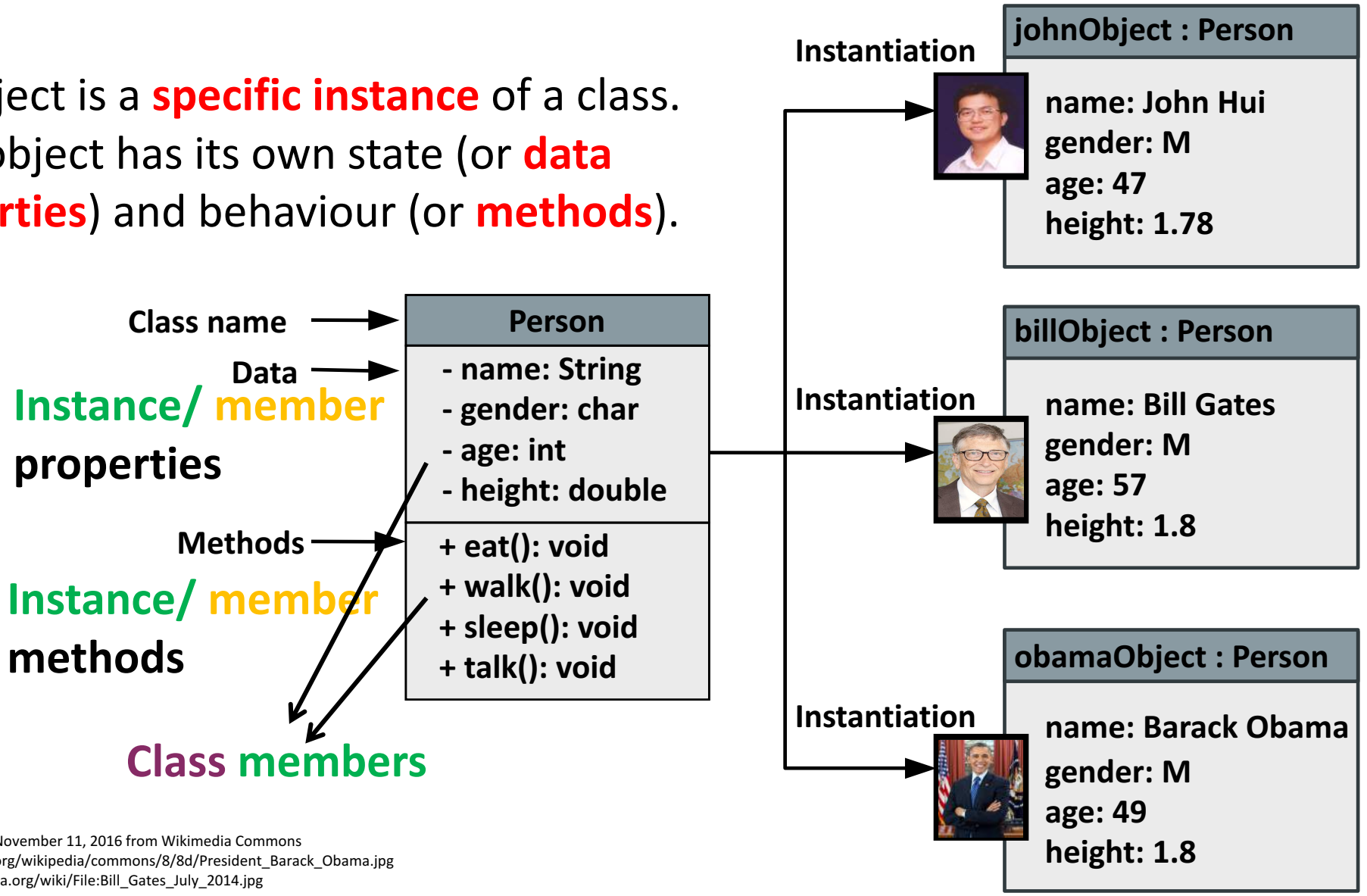


Class Diagram



Classes and Objects

- An object is a **specific instance** of a class.
- Each object has its own state (or **data properties**) and behaviour (or **methods**).

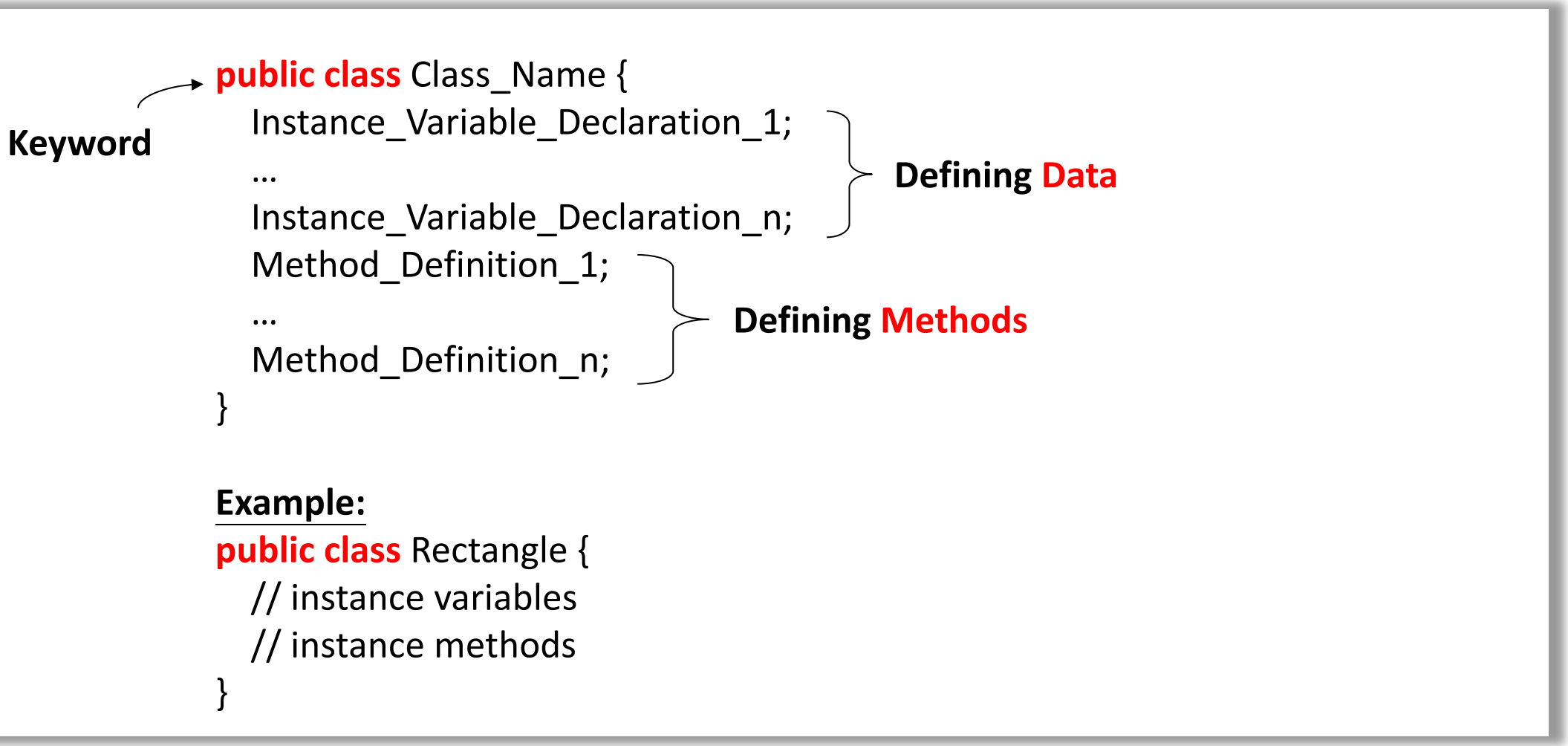


Barack Obama. Retrieved November 11, 2016 from Wikimedia Commons
https://upload.wikimedia.org/wikipedia/commons/8/8d/President_Barack_Obama.jpg
https://commons.wikimedia.org/wiki/File:Bill_Gates_July_2014.jpg

Topic 2: Class Definition

Class Definition (in Java)

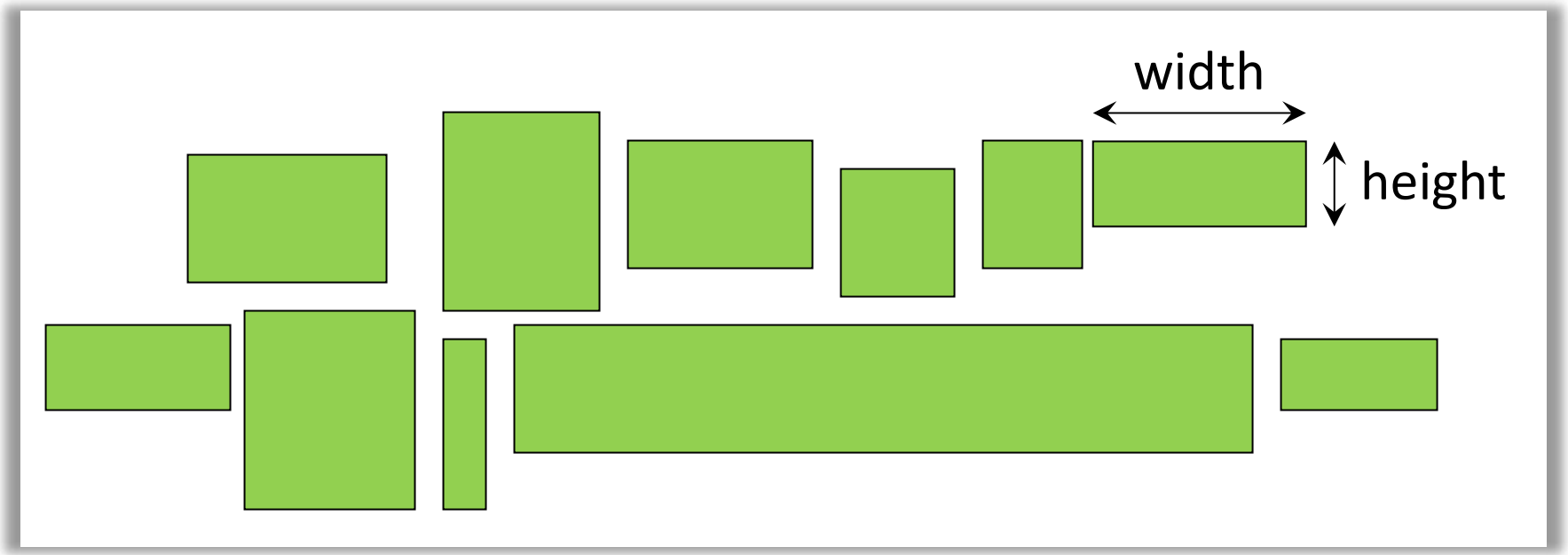
The syntax for defining a class:



Class Definition Example: The Rectangle

- There are an infinite number of rectangles.
- What do all have in common? Any attribute?
  - Width
  - Height
- What common operations we would like to perform?
  - Find area
  - Find Perimeter

OO: abstraction



Class Definition (Data Abstraction)

- Defining Data Attribute – syntax:

  private Type Instance_Variable_Name;

- Defining Method – syntax:

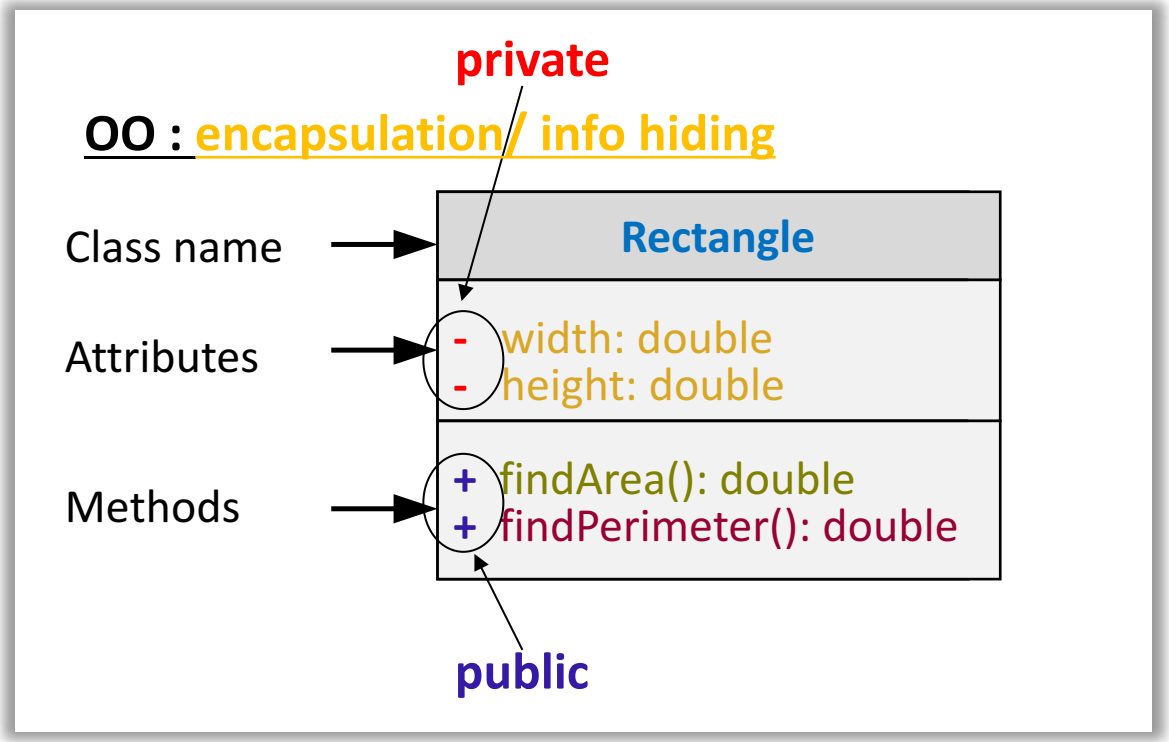
  public Return_Type Method_Name(Parameter_List) {

  Method_body

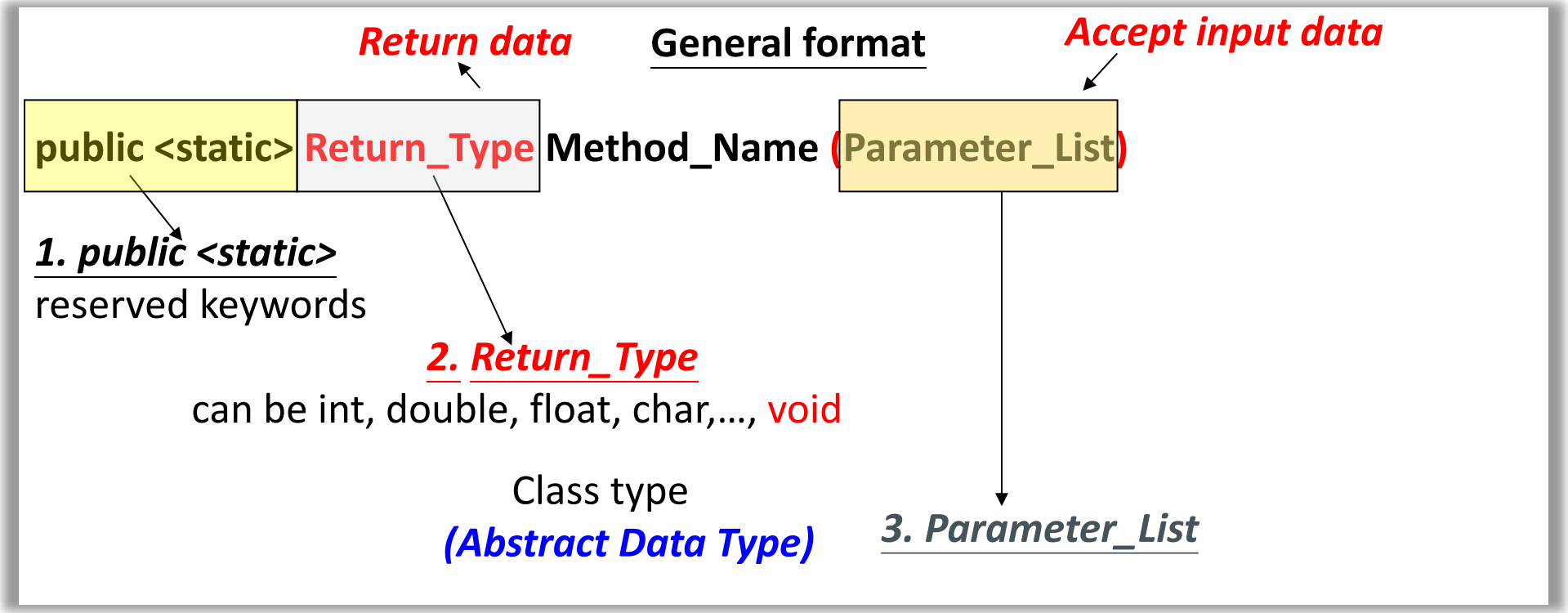
  }

Example:

```
public class Rectangle {
  // instance variables
  private double width  ;
  private double height ;
  // instance methods
  public double findArea() {
    return width * height ;
  }
  public double findPerimeter() {
    return ( width + height ) * 2;
  }
}
```



Method Header



Example

public static **int** successor(*double num*) { …… }

public **boolean** *savetoDB(String name, int age, char gender*)
{……}

Creating Objects

2 steps:

Object Identifier

1. Create a variable for the object:

declaration Class_Name Object_Variable_Name ;

2. Allocate memory:

creation Object_Variable_Name = Class_Name() ;

keyword

Example:

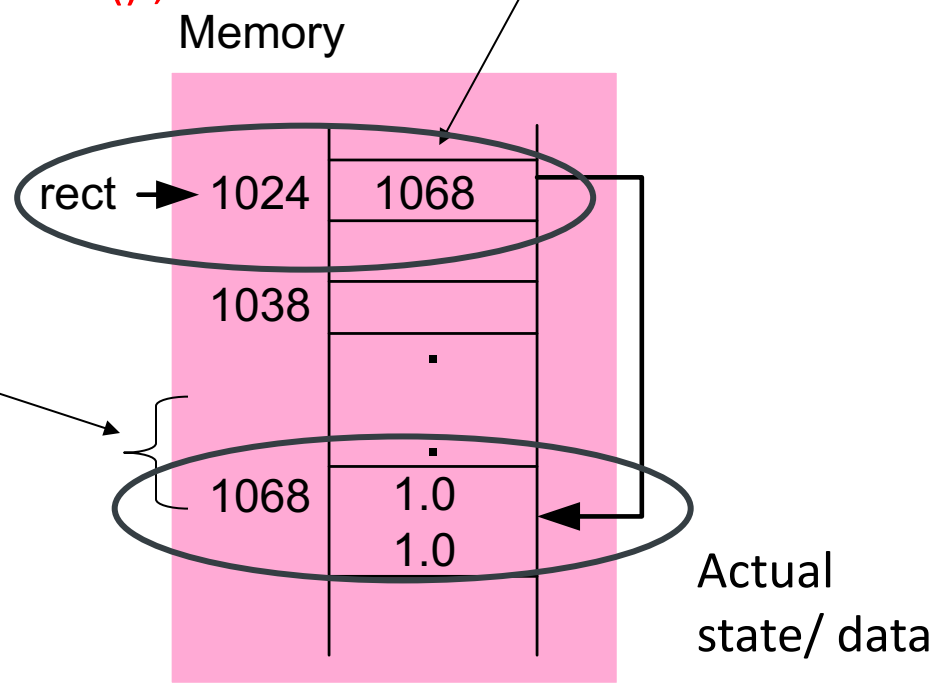
1. **Rectangle rect** ;
2. **rect** = new **Rectangle**() ;

Memory space is allocated to store the object

Or

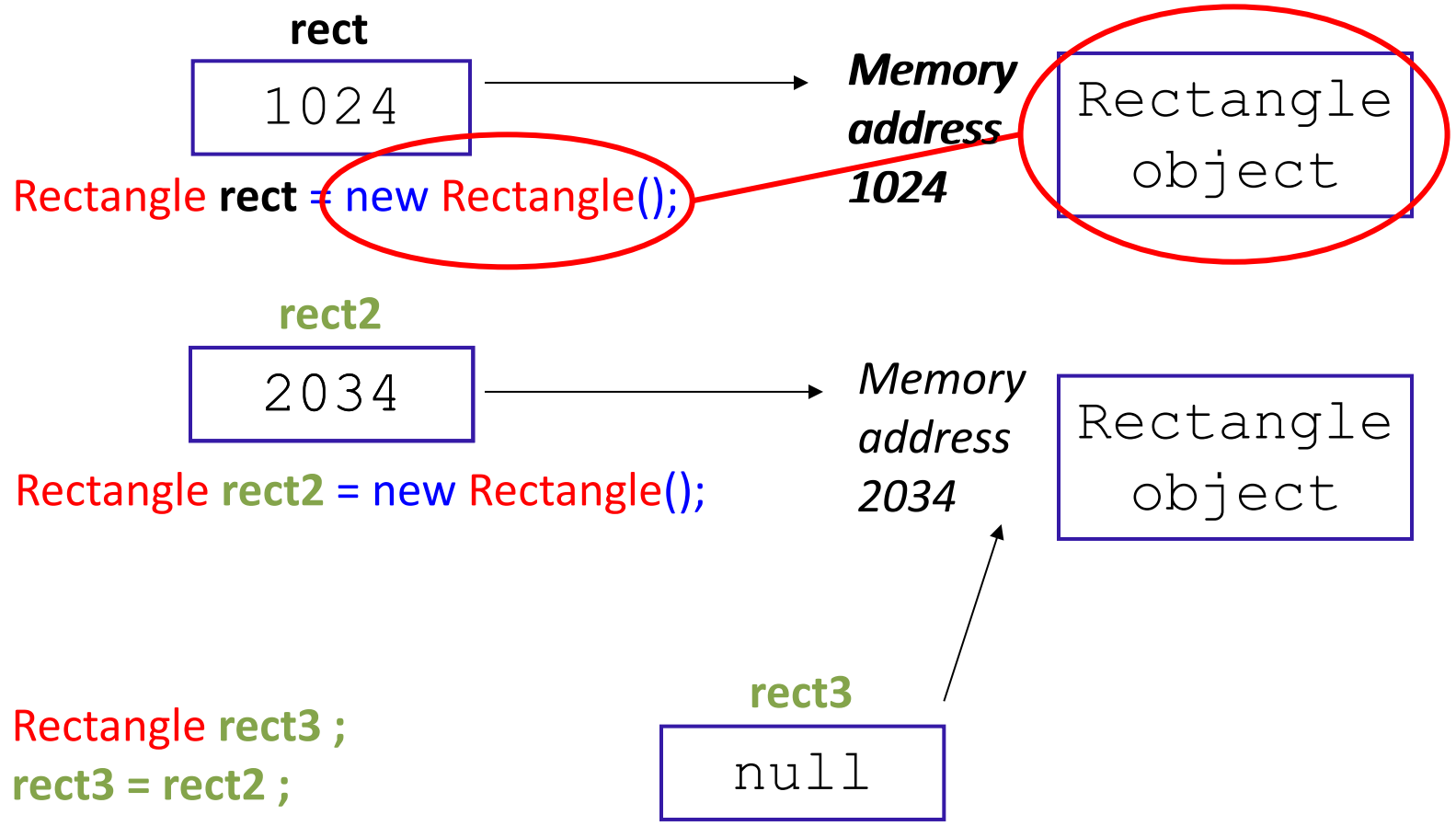
```
Rectangle rect = new Rectangle() ;
```

```
int [] array = new int[10] ;
```



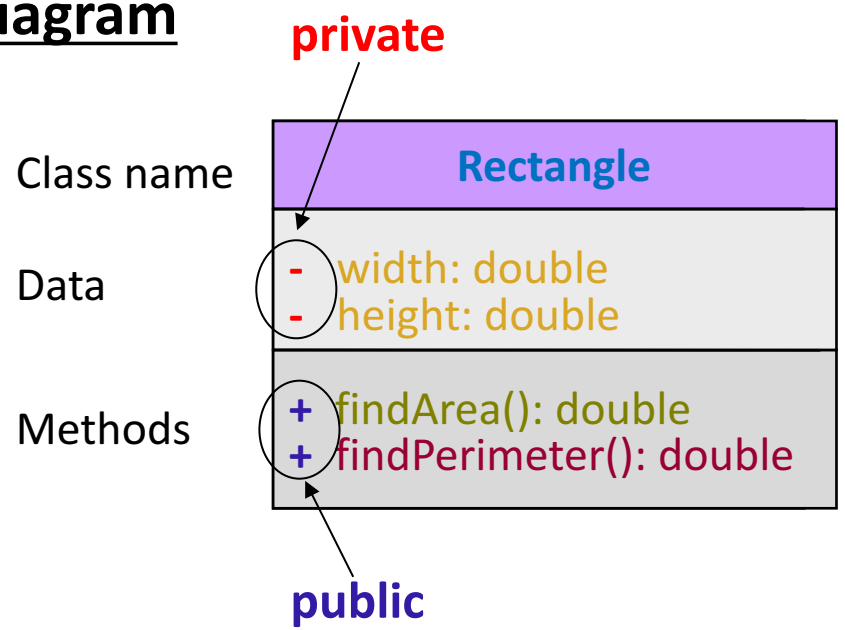
Object Reference and Object

The **object reference**, **rect**, does not store the 'Rectangle' object; instead it stores the memory address of the 'Rectangle' object.



Defining Class

Class Diagram




```
public class Rectangle {
   // instance variables
   private double width  ;
   private double height ;
   // instance methods
   public double findArea() {
      return width * height ;
   }
   public double findPerimeter() {
      return ( width + height ) * 2;
   }
}
```

Creating Objects (Instantiating Object)

```
Rectangle rect ;
rect = new Rectangle() ;
```

OR `Rectangle rect = new Rectangle();`

↳ Object created at this step and allocated to memo space

Constructors

- Constructors are used for initialising object data (Constructing obj).
- If no constructors are written by programmers, Java has a default constructor without argument; the default constructor does nothing.

Example: Rectangle rect = new Rectangle();
Constructor

Default Constructor
public Rectangle( ) { }

- To create objects with different initial values, we can create constructors using the following syntax (basically, **overloading**):

```
public                ( Parameter_List ) {
    // initialise instance variables
}
```

No parameter

Examples:

```
public Rectangle( ){
    width = 1.0 ; height = 1.0 ;
}
```

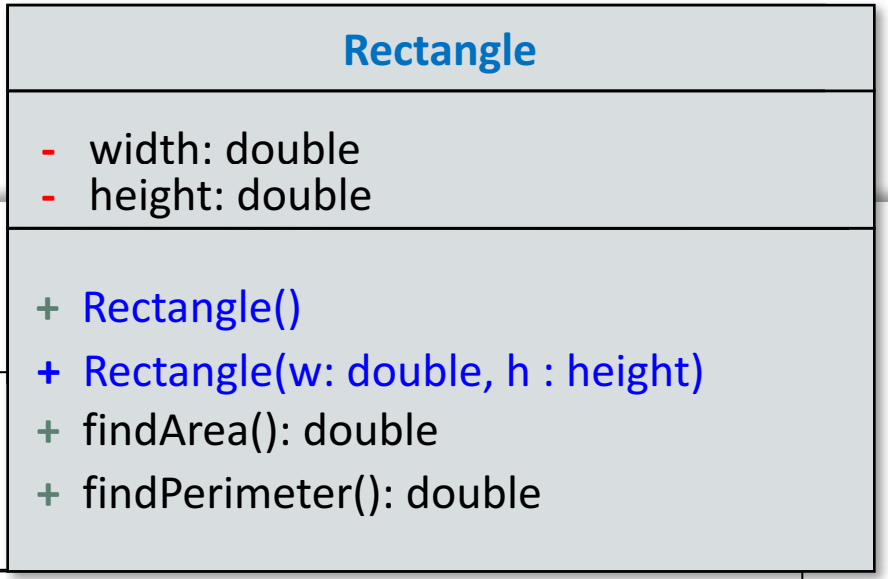
Initialise data properties/ attributes

```
public Rectangle( double w , double h ) {
    width = w ;
    height = h ;
}
```

With parameters

Initialise data properties using parameter values

Constructors




```
public class Rectangle {
  private double width ;
  private double height ;
  public Rectangle() {
     width = 1.0 ;
     height = 1.0 ;
  }
  public Rectangle( double w , double h ) {
     width = w ;
     height = h ;
  }
  public double findArea()     // methods
        { return width * height; }
  public double findPerimeter() {
      return( width + height ) * 2 ;
  }
}
```

Creating Rectangle Objects

Calling the two constructors:

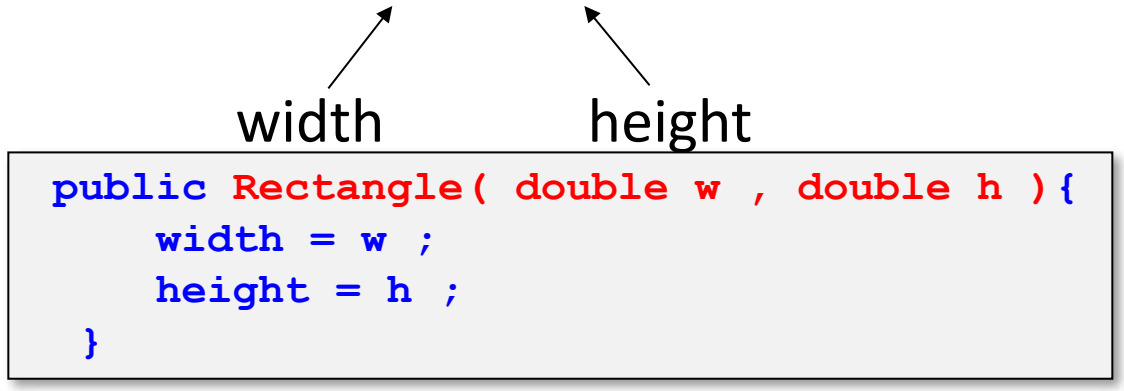
1. Using default values for width and height:

Rectangle rect1 = new Rectangle();

```
public Rectangle() {
     width = 1.0 ;
    height = 1.0 ;
 }
```

2. Using argument values 10.0 and 20.0 for width and height:

Rectangle rect2 = new Rectangle(10.0 , 20.0);



Finalizer

Now you know about a **Constructor!**

Is there also a **Destructor ?**

*In **Java**, it is called* **finalizer**

- **finalize( ) method**
  - **release resources**

Garbage Collector

Topic 3: Message Sending

Message Sending

When I want to ask the student **Bill** to come down here, I will send a voice message to him by saying.

"Bill, please come down!"

Ask an object to do something.

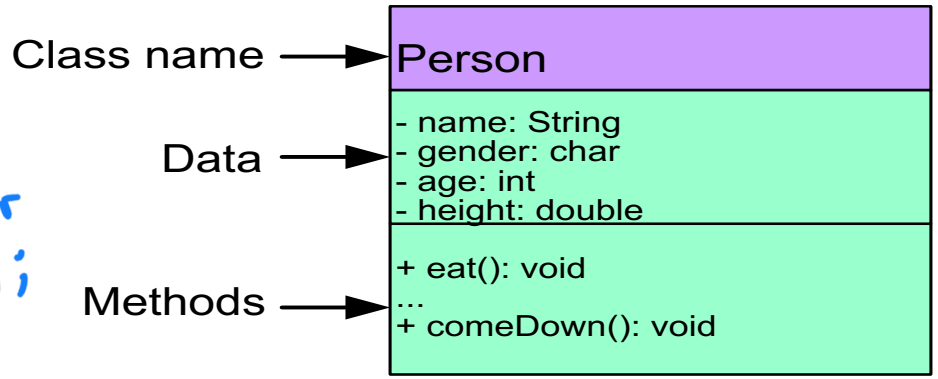
Walking down

Message Sending

Syntax:

Object_Variable_Name.Instance_Variable
Object_Variable_Name.Instance_Method(Argument_List);

Example:



in OOP, we send a message to an object for Operation:
Person bill = new Person();
bill.ComeDown();

```
Person bill ;

bill.comeDown( );
   // make sure bill reference an object that exists.
```

NullPointerException Error!!

- Object reference
- dot operator
- Method name
- Argument(s) to the method

Message Sending

Example:

```
(1) public class Rectangle {
        private double width = 5.0 ;
        private double height = 10.0 ;

        public double findArea() {
          return width * height ;
        }
        public double findPerimeter() {
          return ( width + height ) * 2;
        }
    }
```

Data properties

Methods

Testing with Application Class

```
(2) public class RectangleApp {
      public static void main( String[] args ){
        Rectangle rect = new Rectangle();  // instantiation
        System.out.println(
          "The area of rectangle is "
          + rect.findArea() );
        System.out.println(
          "The perimeter of rectangle is "
          + rect.findPerimeter() );
      }
    }
```

Object reference

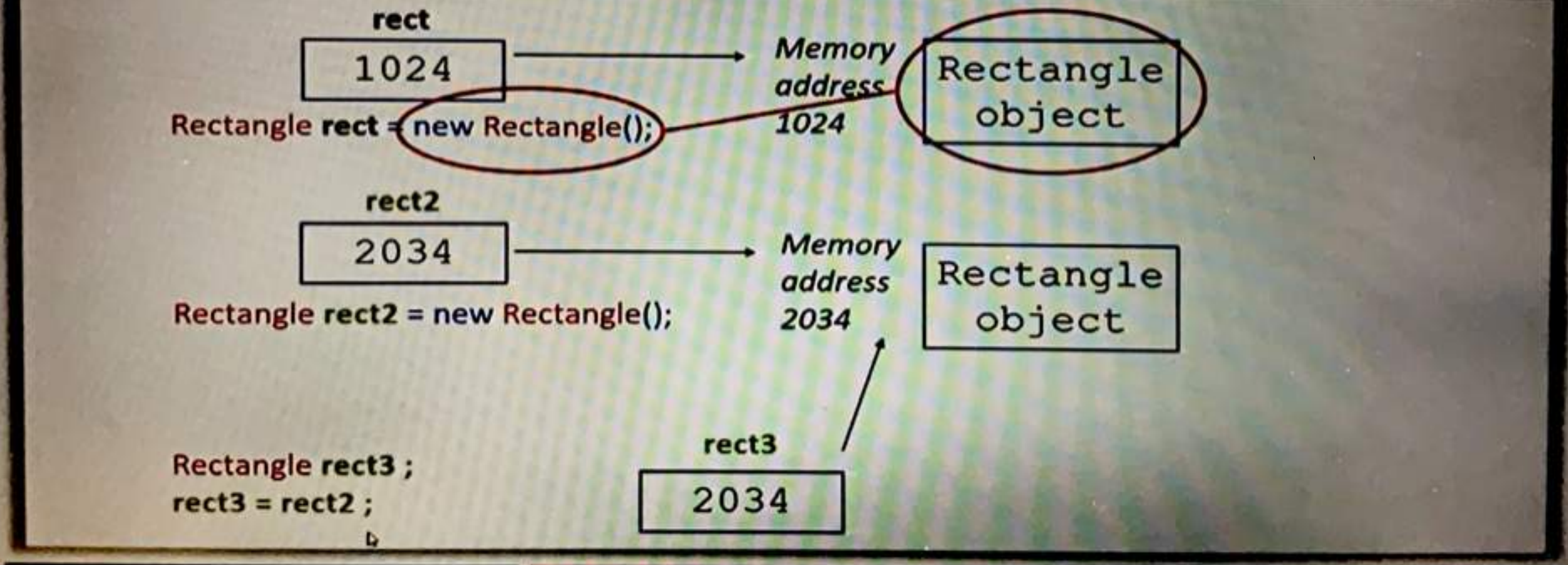
Method name

```
Program Input and Output
The area of rectangle is 50.0
The perimeter of rectangle is 30.0
```

Topic 4: Copying Objects

Object Reference and Object

The object reference, rect, does not store the 'Rectangle' object; instead it stores the memory address of the 'Rectangle' object.



Copying Objects

- Consider the following code:

```
public class CopyObjectsApp {
    public static void main(String[] args){
        Rectangle rect1, rect2;
        rect1 = new Rectangle( 10.0 , 20.0 );
        rect2 = rect1;
    }
}
```

➡ This code will create only one object with two object references rect1 and rect2 both pointing to it.

- copyObjects() method:

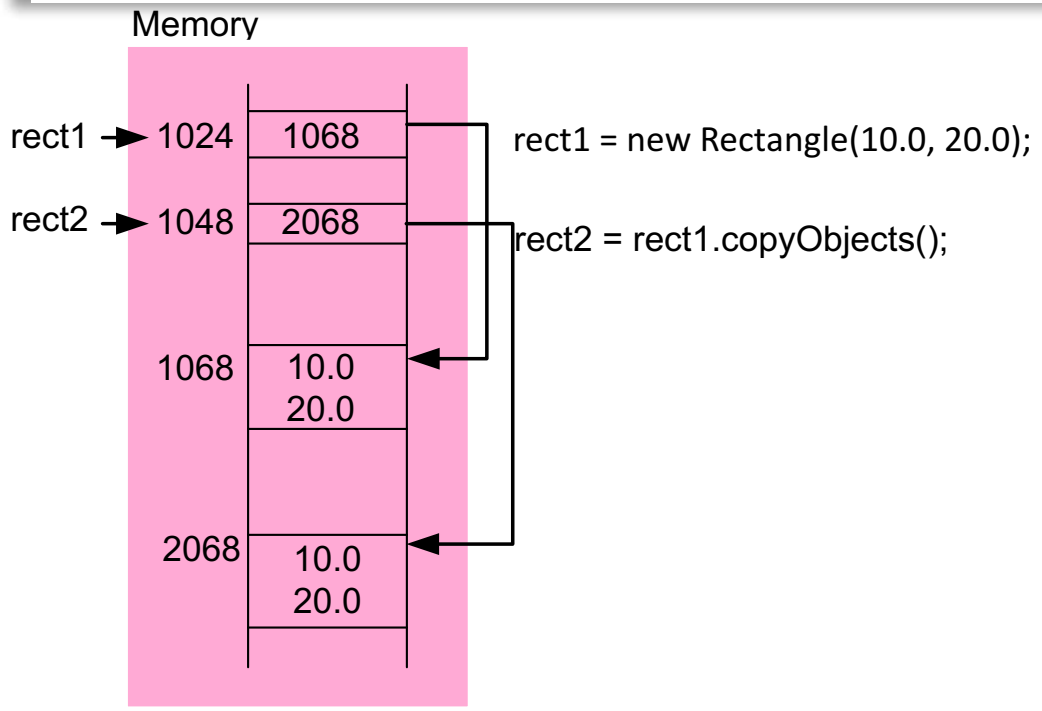
```
public class Rectangle {
    // instance variables and methods
    ...
    public Rectangle copyObjects() {
        Rectangle rect = new Rectangle( width , height );
        return rect;
    }
}
```

➡ This method will create an object using the data properties and pointed at by reference rect.

Application Class

```
public class CopyObjectsApp {
   public static void main(String[] args){
      Rectangle rect1, rect2;
        rect1 = new Rectangle( 10.0 , 20.0 );
        rect2 = rect1.copyObjects();
   }
}
```

This method will create two separate objects with the same data properties and pointed at by references rect1 and rect2 respectively.



Topic 5: The Keyword ‘this’

The Keyword ‘this’

- The keyword **this** refers to the **receiver object** with which you **call the method (or access the object’s variable)**.
- Actually, **this** is the **object reference** that stores the receiver object.
- If the receiver object is not specified when a method is invoked, Java will use **this** automatically.
- The syntax is:

```
this.Method_Name;
```

```
public class Rectangle {
……
 public void print() {
    System.out.println("The area of rectangle is "
                       + this.findArea() );
    System.out.println("The perimeter of rectangle is "
                       + this.findPerimeter() );
 }
}
```

Current instance of this class

The Keyword ‘this’

```
public class Rectangle {
  private double width ;   // data properties
  private double height ;
  public Rectangle () {    // constructors
     width = 1.0 ;
     height = 1.0 ;
  }
  public Rectangle ( double w , double h ) {
     width = w ;
     height = h ;
  }
  public double findArea()     // methods
        { return width * height; }
  public double findPerimeter() {
      return( width + height ) * 2 ;
  } . . .
  public void print() {
    System.out.println("The area of rectangle is "
                       + this.findArea() );
    System.out.println("The perimeter of rectangle is "
                       + this.findPerimeter() );
 }
}
```

The Keyword 'this'

- The '**this**' keyword can also be used for **instance variables**.
- The syntax:

```
this.Instance_Variable;
```

```
E.g.
  public class Rectangle {
  . . .
    public double findArea() {
      return this.width * this.height ;
    }
  }
```

In the statement:

```
  double area1 = rect1.findArea();
  double area2 = rect2.findArea();
```

- The keyword '**this**' above contains the value (reference) of rect1 when calling rect1.findArea() in the first statement.

The Keyword ‘this’

- **This** is commonly used in **constructors**.
- Note that the names of **parameters** or variables declared within constructors or methods must not be the same as the names for the instance variables.

```
public Rectangle(double width, double height){
    width  = width  ;
    height = height ;
}
```

To solve this problem, we can use **this**:

```
public Rectangle(double width, double height)
{
    this.width  = width  ;
    this.height = height ;
}
```

- The syntax form: **this(…);** refers to constructor. E.g. this(10.0, 20.0); activates the Rectangle (double width, double height) constructor.

The Keyword 'this'

- **This** is co
- Note tha
  construc
  instance

```
public class Rectangle {
   private double width  ;      // data properties
   private double height ;
   public Rectangle() {
      width  = 1.0 ;
      height = 1.0 ;
   }
   public Rectangle( double width , double height ) {
      this.width  = width ;
      this.height = height ;
   }
……..
}
```

```
public R
    width
    heigh
}
```

To solve th

```
public Rectangle(double width, double height)
{
    this.width  = width  ;
    this.height = height ;
}
```

- The syntax form: **this(…);** refers to constructor. E.g. this(10.0, 20.0); activates the Rectangle (double width, double height) constructor.

The Keyword ‘this’

- **This** is co
- Note tha
  construc
  instance

```
public R
    widt
    heig
}
```

To solve th

```
public Rectan
{
    this.widt
    this.heig
}
```

- The syntax fo
  activates the

```
public class Rectangle {
   private double width  ;      // data properties
   private double height ;
   public Rectangle() {
      width  = 1.0 ;
      height = 1.0 ;
   }
   public Rectangle( double width , double height ) {
   }
   }
.....
}
```

```
public class Rectangle {
   private double width  ; // data properties
   private double height ;
   public Rectangle () {        // constructors
      this(1.0, 1.0);
   }
   public Rectangle( double w , double h ) {
      width  = w ;
      height = h ;
   }
.....
}
```

Topic 6: Accessors and Mutators

Accessors and Mutators

- Question: Can we change the values of the **data properties** (state) of an object by another object or function **directly**?
  - NO!!!

```
public   double height ;
```

  - Well, OK, yes, but it is considered a very bad OO design.

- Two common types of methods:
  - **Accessors** (or **get** methods): Responsible for returning the value of a data property.
  - **Mutators** (or **set** methods): Responsible for changing the values of a data property (Help to maintain data consistency).

Accessors and Mutators

```
public class Rectangle {
    private double width  ;
    private double height ;

    public Rectangle() {
       this(1.0, 1.0);
    }
    public Rectangle(double width, double height) {
       this.width  = width ;
       this.height = height ;
    }
    public void setWidth(  double w ) { width  = w ; }
    public void setHeight( double h ) { height = h ; }
    public double getWidth()  { return width  ; }
    public double getHeight() { return height ; }

    public double findArea() { return width * height ; }
    public double findPerimeter() {
       return ( width + height ) * 2; }
}
```

1

2

3

4

Set methods

Get methods

Accessors and Mutators

```
public class GetAndSetApp {
   public static void main( String[] args ) {
      Rectangle rect= new Rectangle(5.0,10.0); print(rect);
      rect.setWidth(  20.0 ); print( rect );
      rect.setHeight( 40.0 ); print( rect );
   }
   public static void print(Rectangle r) {
     System.out.println( "Width = "    + r.getWidth()
                         + "; Height = " + r.getHeight() );
     System.out.println( "The area of rectangle is "
                         + r.findArea() );
     System.out.println( "The perimeter of rectangle is "
                         + r.findPerimeter() );
   }
}
```

Program Output

```
Width = 5.0; Height = 10.0
The area of rectangle is 50.0
The perimeter of rectangle is 30.0
Width = 20.0; Height = 10.0
The area of rectangle is 200.0
The perimeter of rectangle is 60.0
Width = 20.0; Height = 40.0
The area of rectangle is 800.0
The perimeter of rectangle is 120.0
```

Topic 7: The Keyword ‘static’

The Keyword 'static'

- The **static** keyword can be used in the declaration of an instance variable or method.
- Applied to the **whole class** instead of individual objects.
- Static variables - class variables
- Static methods - class methods

1. Static Variables (per class)

- Count number of created **Rectangle** objects:

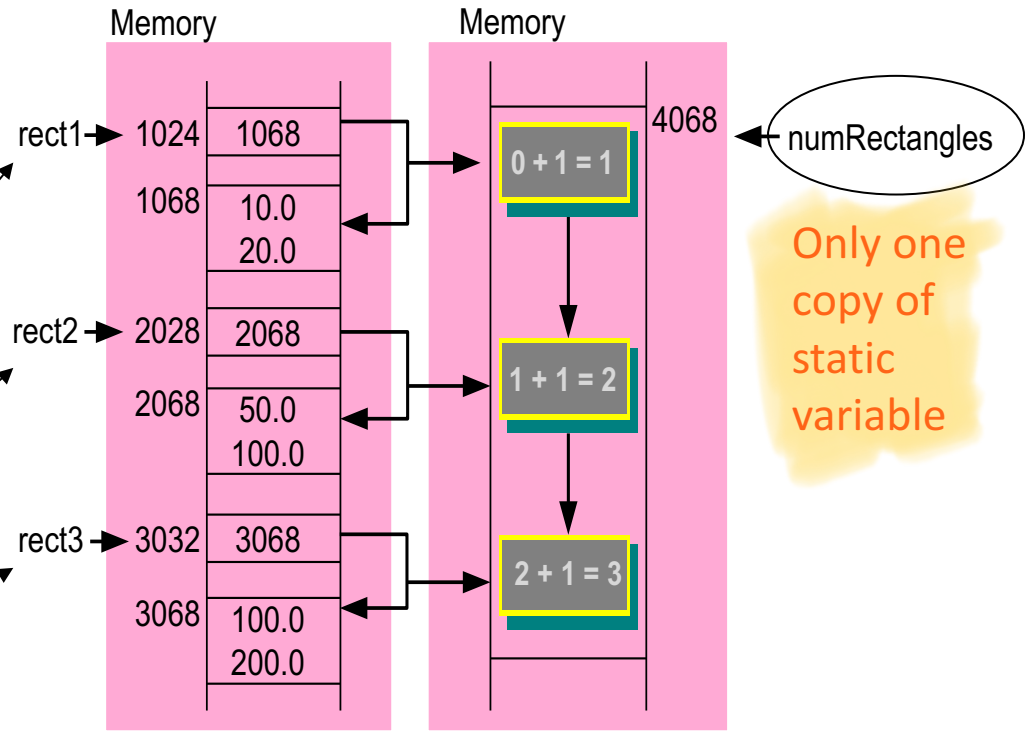
```
private static int numRectangles ;
```

- A **common** separate memory location is allocated to store the value for the static instance variable, and only a **single copy** of the static instance variable is allocated for all objects. i.e. the static variable numRectangles is **shared** by **all objects of the class Rectangle**.

The Keyword 'static'

To count the number of **Rectangle** objects that are created, we can modify the **constructors**:

```
public Rectangle()
   {
     width  = 1.0 ;
     height = 1.0 ;
     numRectangles++;
   }
   public Rectangle
   (double w , double h)
   {
     width  = w ;
     height = h ;
     numRectangles++;
   }
```




```
Rectangle rect1 = new Rectangle(  10.0 ,  20.0 );
Rectangle rect2 = new Rectangle(  50.0 , 100.0 );
Rectangle rect3 = new Rectangle( 100.0 , 200.0 );
```

Static Methods (Methods belong to the Class)

- The **static** keyword can also be applied to any **methods** in the class Rectangle.

Cannot have instance method

```
public class Rectangle {
  // instance variables and methods
  private static int numRectangles = 0;
     ...
  // class methods
  public static void setNumRectangles(int number){
     numRectangles = number;
  }
  public static int getNumRectangles() {
     return numRectangles;
  }
}
```

Initialise static variable (class data)

Set method

Get method

- Since these two methods **do not** manipulate any of the **Rectangle** non-static object data, we define them as **class methods**.

We can use class name instead of object name

Static Methods (Methods belong to the Class)

- To access a class method outside its class, we use:

```
Class_Name.Method_Name();
```

```
Application Class:
   public class UsingStaticApp {
        public static void main(String[] args){
          Rectangle rect1 = new Rectangle( 5.0, 10.0);
          Rectangle rect2 = new Rectangle(50.0,100.0);
          System.out.println("Number of rectangles = "
            + Rectangle.getNumRectangles());
       }
   }
```

The Math class is an excellent example of calling “class methods”.

Static Methods (Methods belong to the Class)

- Also notice that class methods can reference class variables and methods, but not instance variables and methods in the class.

```
public class Rectangle {
  // instance variables and methods
  ...
  public double findArea() {return width * height; }
  public static void countArea() {
     findArea();    // an error
  }
}
```

- This will give an error as the receiver object for the method findArea() is the object reference this. This violates the use of class method as it does not apply to individual object's methods.

➡ Can we just add **static** in front of findArea() and make it a class method???

Class Constants (Class Variables that are Constant)

- Use keywords **static final**.

```
public static final double MAX_WIDTH  = 100;
public static final double MAX_HEIGHT = 100;

public void setWidth( double w ) {
   if ( w     >= MAX_WIDTH )
        width  = MAX_WIDTH ;     // data consistency
   else
        width  = w ;
}
public void setHeight( double h ) {
   if (  h     >= MAX_HEIGHT )
         height = MAX_HEIGHT ;
   else
         height = h ;
}
```

- **Advantages:** (1) Easy to change to new value.

  (2) Symbolic names are easier to understand.

Complete Class Example

```
public class Rectangle {
   public  static final double MAX_WIDTH  = 100;
   public  static final double MAX_HEIGHT = 100;
   private double width  ;
   private double height ;
   // class variable
   private static int numRectangles=0;
   public Rectangle() {
      width  = 1.0;
      height = 1.0;
      numRectangles++;
   }
   public Rectangle( double w , double h ) {
      width  = w ;
      heigth = h ;
      numRectangles++;
   }
   public void setWidth( double w ) {
      if ( w     >= MAX_WIDTH )
           width  = MAX_WIDTH ;
      else
           width  = w ;
   }
```

Class constant (per class)

Instance variables (per object)

Static variable (per class)

Class constant

Complete Class Example

```
    public void setHeight( double h ) {
       if ( h      >= MAX_HEIGHT )
            height  = MAX_HEIGHT ;
       else
            height = h ;
    }

    // instance methods
    public double getWidth()      { return width  ; }
    public double getHeight()     { return height ; }
    public double findArea()      { return  width * height ; }
    public double findPerimeter(){ return (width + height) * 2; }

    // class methods
    public static void setNumRectangles(int number) {
       numRectangles = number ;
    }
    public static int getNumRectangles() {
       return numRectangles ;
    }
}
```

Class constant

Instance methods

Static methods

Application Class

```
public class UsingStaticApp {
   public static void main( String[] args ) {
      Rectangle rect1 = new Rectangle( 5.0 , 10.0 );
      print(rect1);
      rect1.setWidth(  40.0 );
      rect1.setHeight( 10.0 );
      print(rect1);
      Rectangle rect2 = new Rectangle( 50.0 , 100.0 );
    System.out.println( "Number of rectangles = " +
      Rectangle.getNumRectangles());
   }

   public static void print( Rectangle r ){
      System.out.println( "Width = "  + r.getWidth()
                          + "Height = " + r.getHeight() );
      System.out.println( "The area of rectangle is "
                          + r.findArea() );
      System.out.println( "The perimeter of rectangle is "
                          + r.findPerimeter() );
   }
}
```

static/class methods are called by class name

Calling instance methods

Program Output

Program Output:

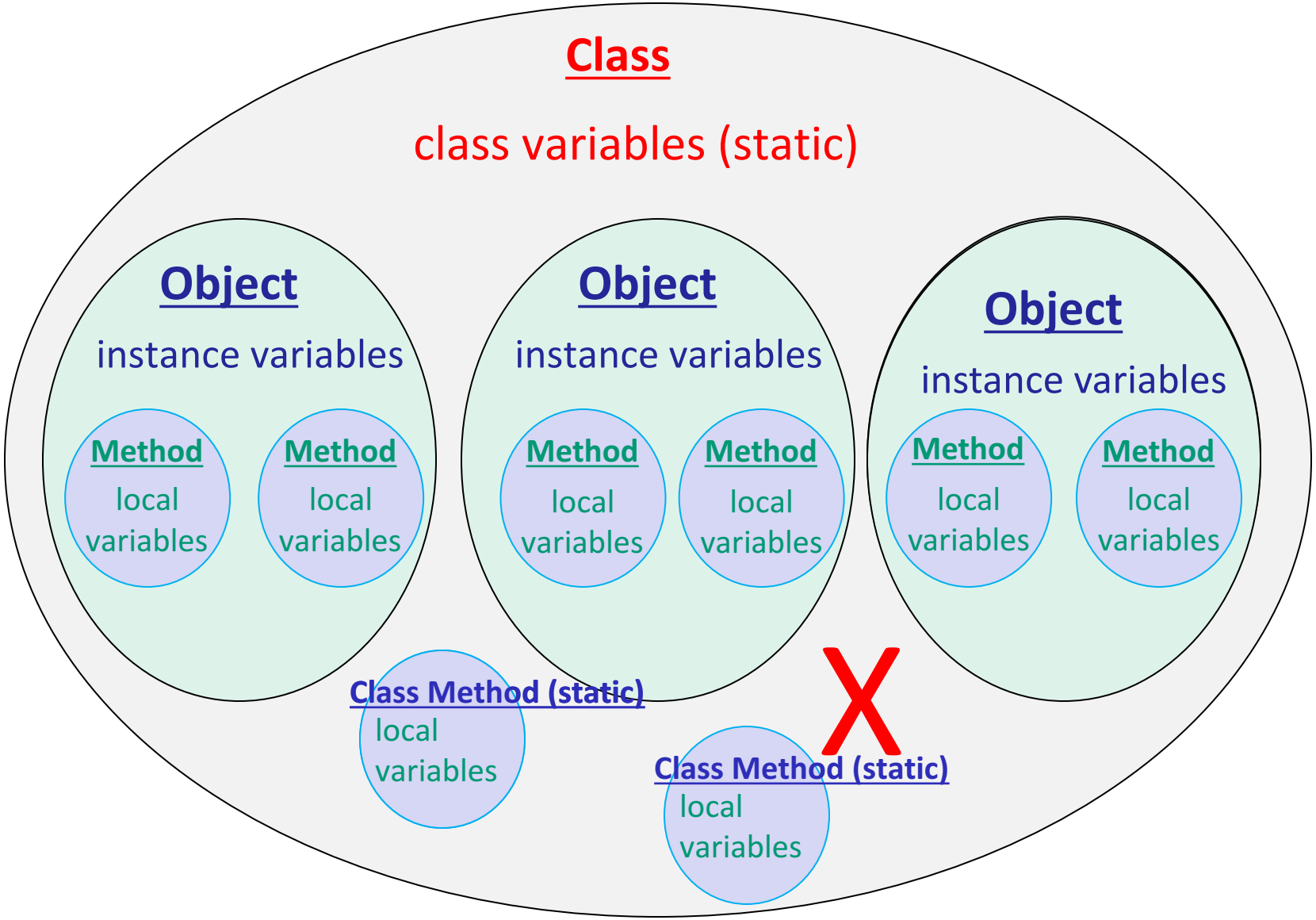
```
Width = 5.0; Height = 10.0
The area of rectangle is 50.0
The perimeter of rectangle is 30.0
Width = 40.0; Height = 10.0
The area of the rectangle is 400.0
The perimeter of rectangle is 100.0
Number of rectangles = 2
```

Class Variable
Class method

Topic 8: Static vs. Instance methods

Can use Static

Class


Class
class variables (static)
Object
instance variables
Method
local variables
Method
local variables
Object
instance variables
Method
local variables
Method
local variables
Object
instance variables
Method
local variables
Method
local variables
Class Method (static)
local variables
X
Class Method (static)
local variables

Class

```
public class Person{
  double height  ;
  double weight ;

  public Person(double h, double w)
    height = h ;
    weight = w ;
  }
  public double getHeight() {
    return height ;
  }
  public double getWeight() {
    return weight ;
  }
  public double calculateBMI() {
     calculateBMI( height,  weight) ;
  }
  public static double calculateBMI(double height,
                                    double weight) {
    return (weight/(height * height));
  }
}
```

Category	BMI
Very severely underweight	< 15
Severely underweight	15 – 16
Underweight	16 – 18.5
Normal (healthy weight)	18.5 – 25
Overweight	25 – 30
Obese Class I	30 – 35
Obese Class II	35 – 40
Obese Class III	> 40

Class

```
public class PersonApp{
  public static void main(String[] args) {
    Person bill = new Person(1.80 , 70);
    double bmi = bill .calculateBMI(); //instance method

    System.out.println("BMI is " + bmi);

    bmi =  bill .calculateBMI(bill.getHeight(), bill.getWeight());

    System.out.println("BMI is " + bmi);

    bmi = Person.calculateBMI (1.90, 90);

    System.out.println("BMI is " + bmi);
  }
}
```

```
BMI is 21.604938271604937
BMI is 21.604938271604937
BMI is 24.930747922437675
```

Topic 9: Encapsulation and Information Hiding

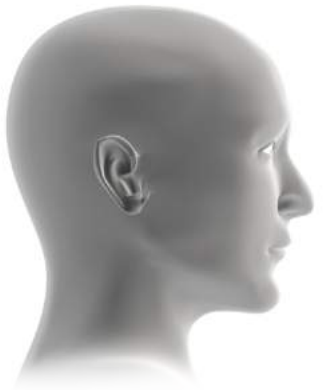
Encapsulation and Information Hiding

- **Encapsulation** – Builds a barrier to protect an object's private data. Access to private data can be done through the public methods of the object's class (e.g. get and set methods).
- **Information hiding** – Hides the details or implementation of the class from users.

  - With encapsulation, the users of a class only need to know **what** a class does and **how** to call the methods to perform the tasks.
  - The users **do not** need to know the **implementation** details for the methods.

Encapsulation and Information Hiding

My name is Shadow, I am 48 years old and weigh 70 kg.

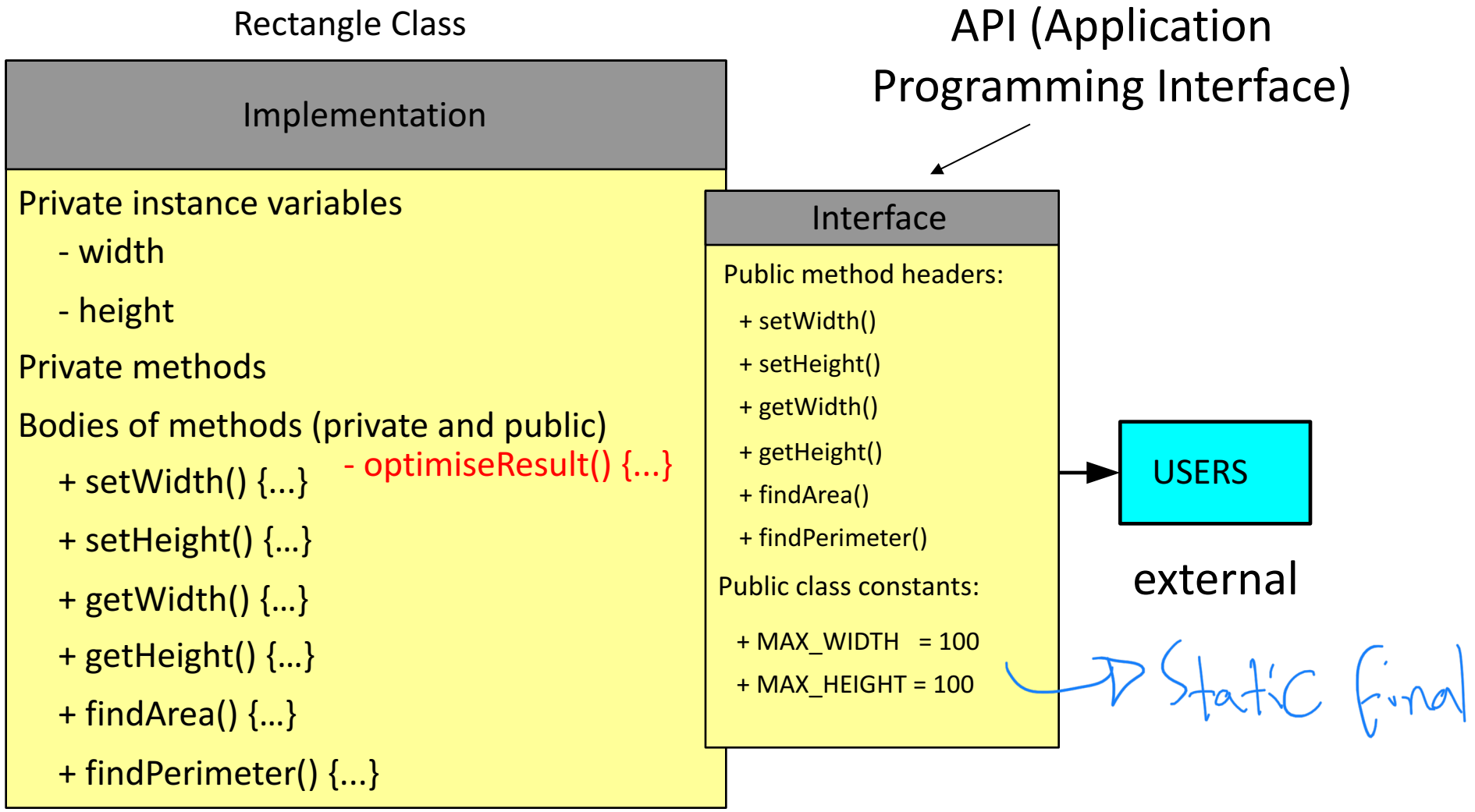


Encapsulation and Information Hiding

Encapsulation and Information Hiding help to achieve the following:

- Protect what you have
- Protect your data

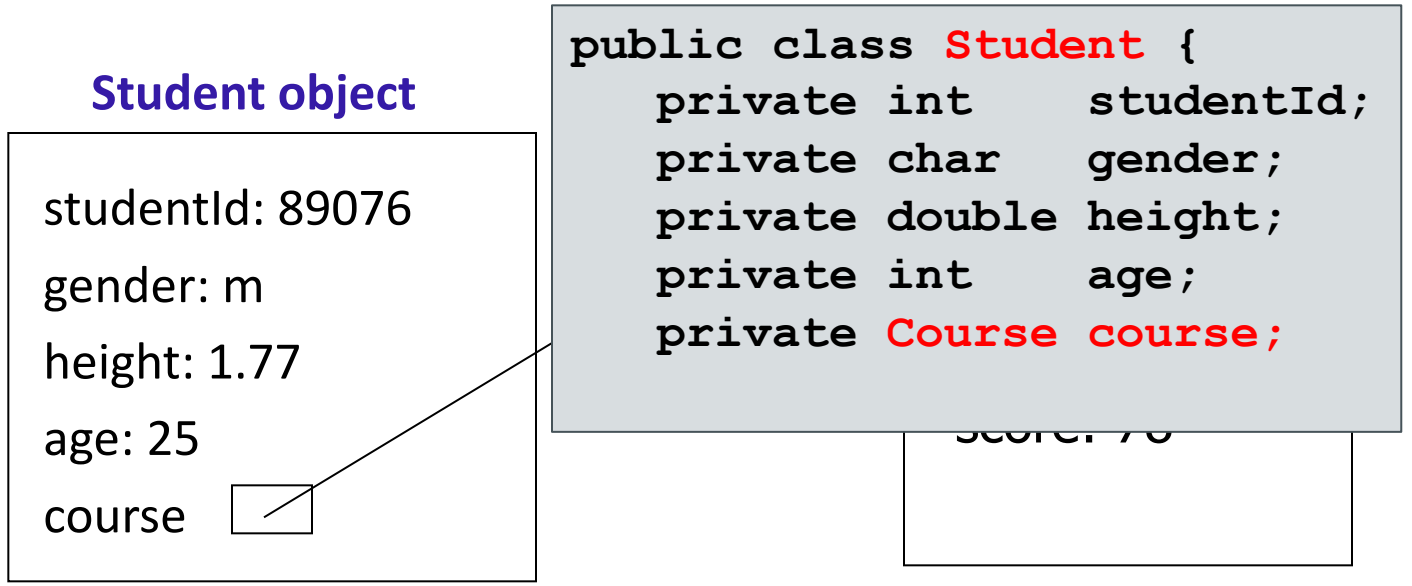
Example of Information Hiding



Topic 10: Object Composition

Object Composition

- An object can include other objects as its **data** member. This is called **object composition**.
- A class contains object references from other classes as its instance variables.
- Object references are of reference data types.
- In OO, this is called the "has-a" relationship.
- Example: a student *has a* course.



Object Composition

```
public class Course {
    private int courseId;
    private int year;
    private int semester;
    private int score;
    public Course( int courseId , int year   ,
                   int semester , int score ) {
        this.courseId = courseId ;
        this.year     = year     ;
        this.semester = semester ;
        this.score    = score    ;
    }
    public void setCourseId(int courseId) {
        this.courseId = courseId; }
    public void setYear(int year) {
        this.year     = year;     }
    public void setSemester(int semester) {
        this.semester = semester; }
    public void setScore(int score) {
        this.score    = score;    }
```

=> Define the Course class

=> Constructor

=> Set methods

Object Composition

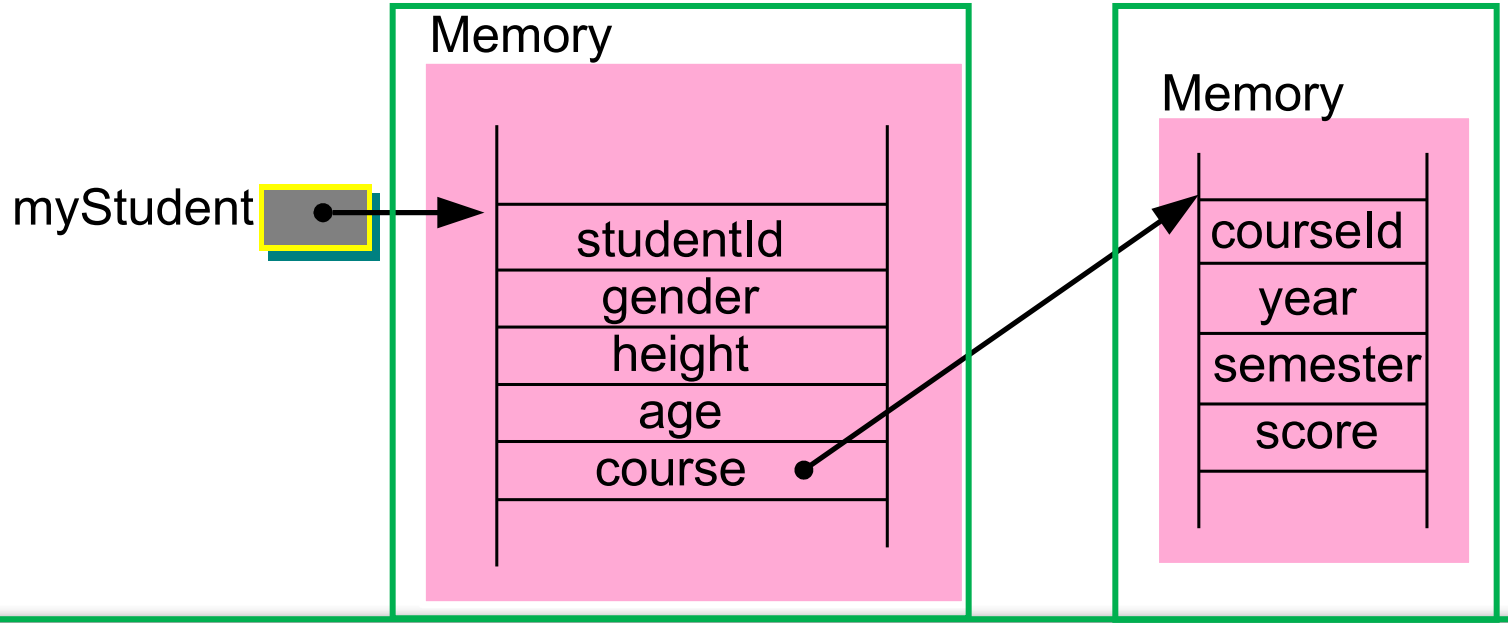
```
    public int getCourseId(){
         return courseId;
    }
    public int getYear(){
         return year;
    }
    public int getSemester(){
         return semester;
    }
    public int getScore(){
         return score;
    }
}
```

=> Get methods

```
public class Student {
    private int    studentId;
    private char   gender;
    private double height;
    private int    age;
    private Course course;
```

=> Define an object as an instance variable (data member)

Object Composition




```
public Student(int studId, char sex, double
   studHeight, int studAge, Course courseIn)
{
     studentId = studId;
     gender    = sex;
     height    = studHeight;
     age       = studAge;
     course    = courseIn;
}
public int    getStudentId() { return studentId ; }
public double getHeight()    { return height    ; }
```

=> In the constructor

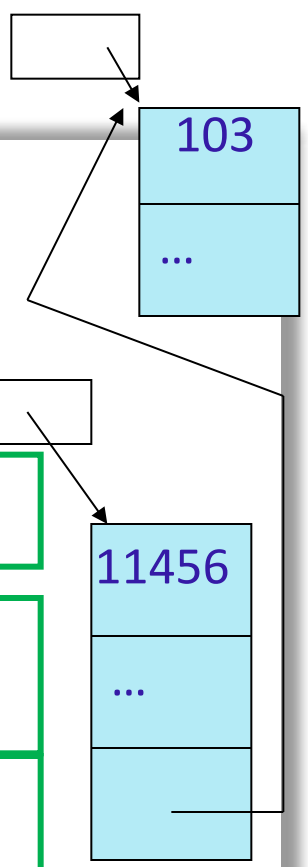
Object Composition

```
    public int getAge() { return age ; }
    public void setStudentId( int studId ) {
         studentId = studId ;
    }
    public void setHeight( double studHeight ) {
         height = studHeight ;
    }
    public void setAge( int studAge ) { age = studAge ; }
    public void printInfo() {
         System.out.println( "=== Student Info ==="     );
         System.out.println( "Student ID: " + studentId );
         System.out.println( "Gender: "     + gender    );
         System.out.println( "Height: "     + height    );
         System.out.println( "Age: "        + age       );
         System.out.println( "Course: "
                           + course.getCourseId());
         System.out.println( "Year: " + course.getYear());
         System.out.println( "Semester: "
                           + course.getSemester());
         System.out.println( "Score: " + course.getScore());
    }
}
```

Object Composition

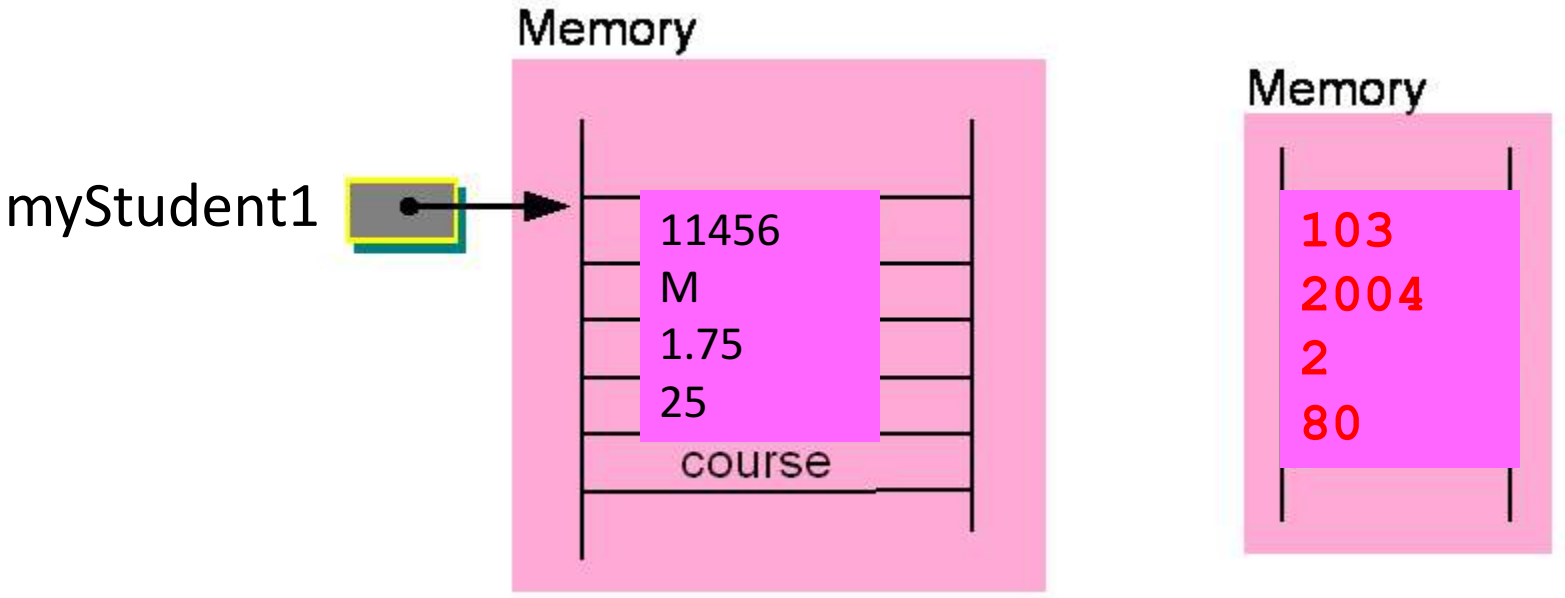
=> Define the application class

```
public class StudentApp {
  public static void main(String[] args){
   Course course1 = new Course(103,2004,2,80);

   StudentInfo myStudent1 =
        new Student(11456,'M', 1.75,25,course1);

    Course course2 = new Course(104,2004,1,60);
   StudentInfo myStudent2 =
        new Student(11457,'F', 1.55,21,course2);
   Course course3 = new Course(105,2004,1,90);
   StudentInfo myStudent3 =
        new Student(11458,'F', 1.85,28,course3);
   myStudent1.printInfo();
   myStudent2.printInfo();
   myStudent3.printInfo();
 }   }
```



Object Composition

```
Course course1 = new Course(103,2004,2,80);
StudentInfo myStudent1 =
    new Student(11456,'M', 1.75,25,course1);
```

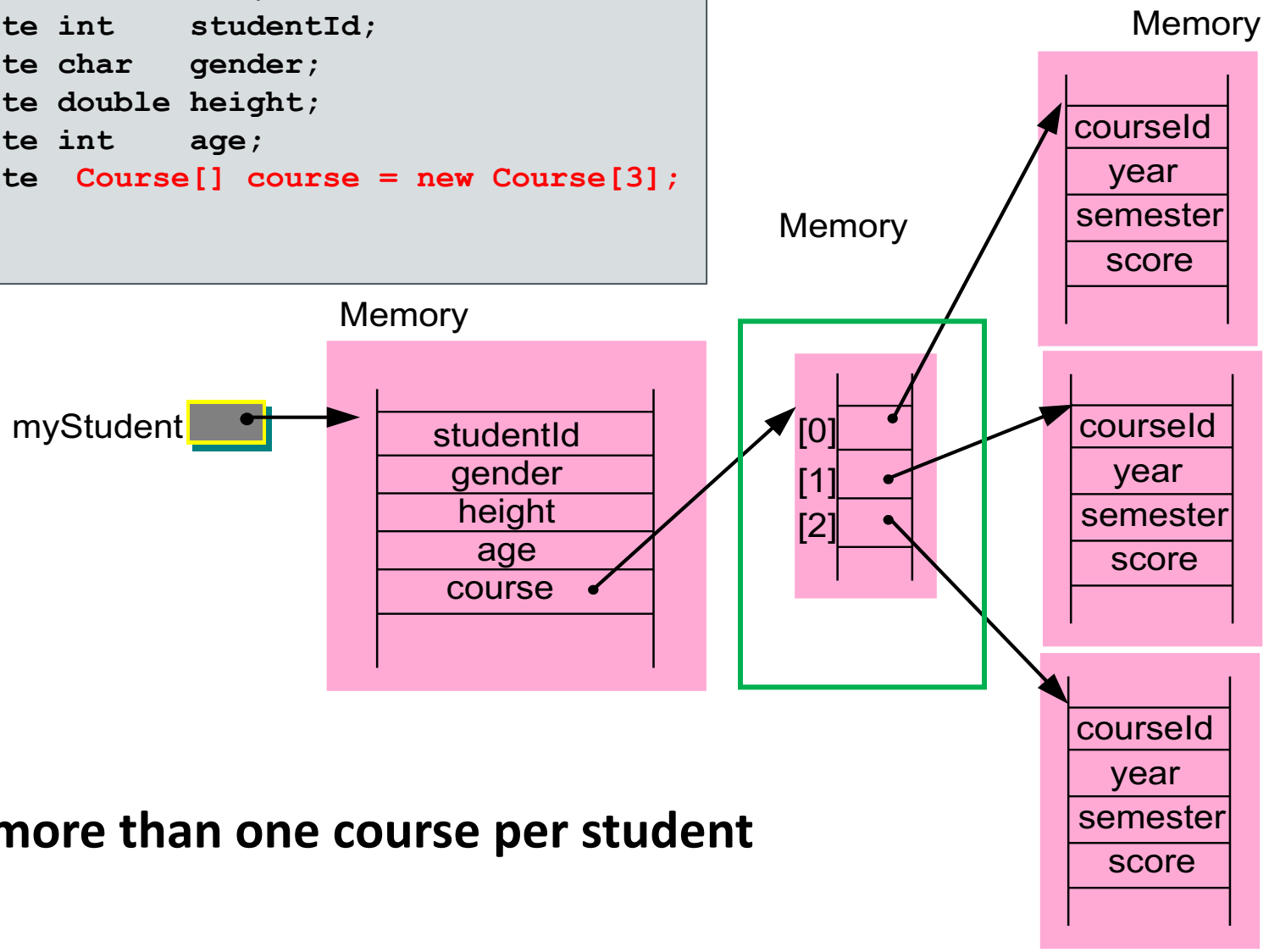


Program Output

```
=== Student Info ===
Student ID: 11456
Gender: M
Height: 1.75
Age: 25
Course: 103
Year: 2004
Semester: 2
Score: 80
=== Student Info ===
Student ID: 11457
Gender: F
Height: 1.55
Age: 21
Course: 104
Year: 2004
Semester: 1
Score: 60
=== Student Info ===
Student ID: 11458
Gender: F
Height: 1.85
Age: 28
Course: 105
Year: 2004
Semester: 1
Score: 90
```

Object Composition with Array of Objects

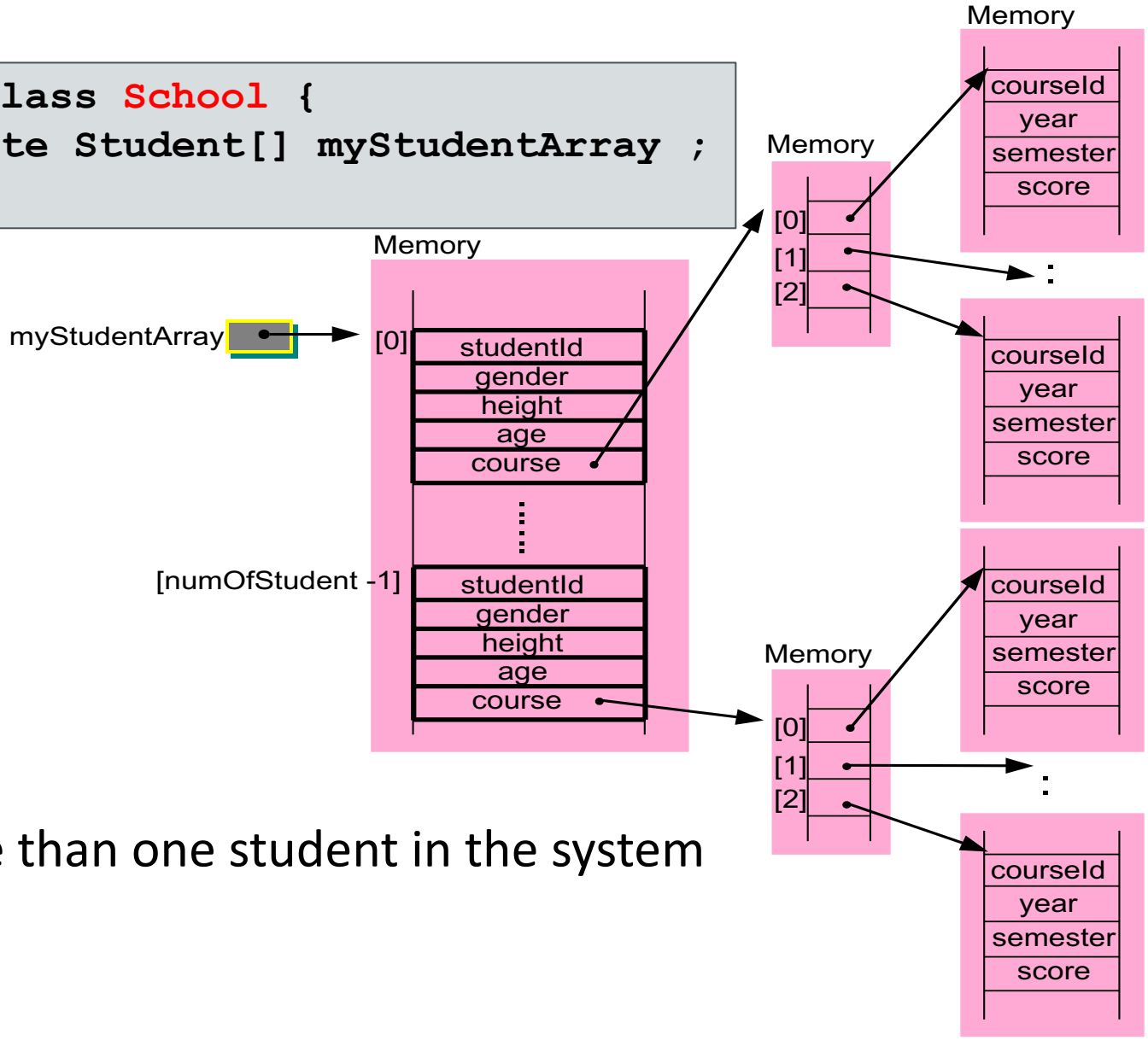
```
public class Student {
   private int    studentId;
   private char   gender;
   private double height;
   private int    age;
   private  Course[] course = new Course[3];
…
}
```



Having more than one course per student

A Database of Composite Objects

```
public class School {
    private Student[] myStudentArray ;
    .....
```

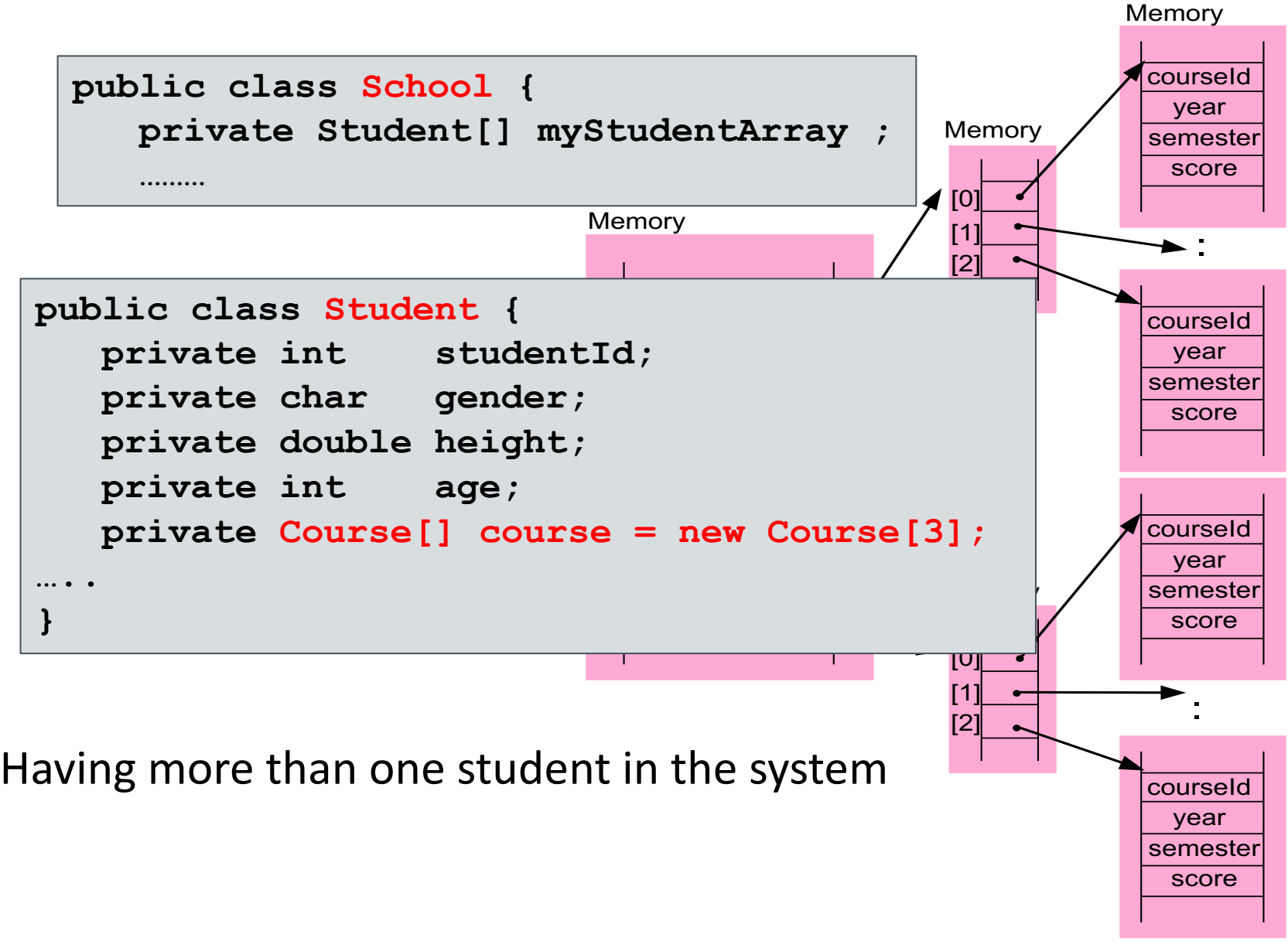


Having more than one student in the system

A Database of Composite Objects

```
public class School {
    private Student[] myStudentArray ;
    .........
```

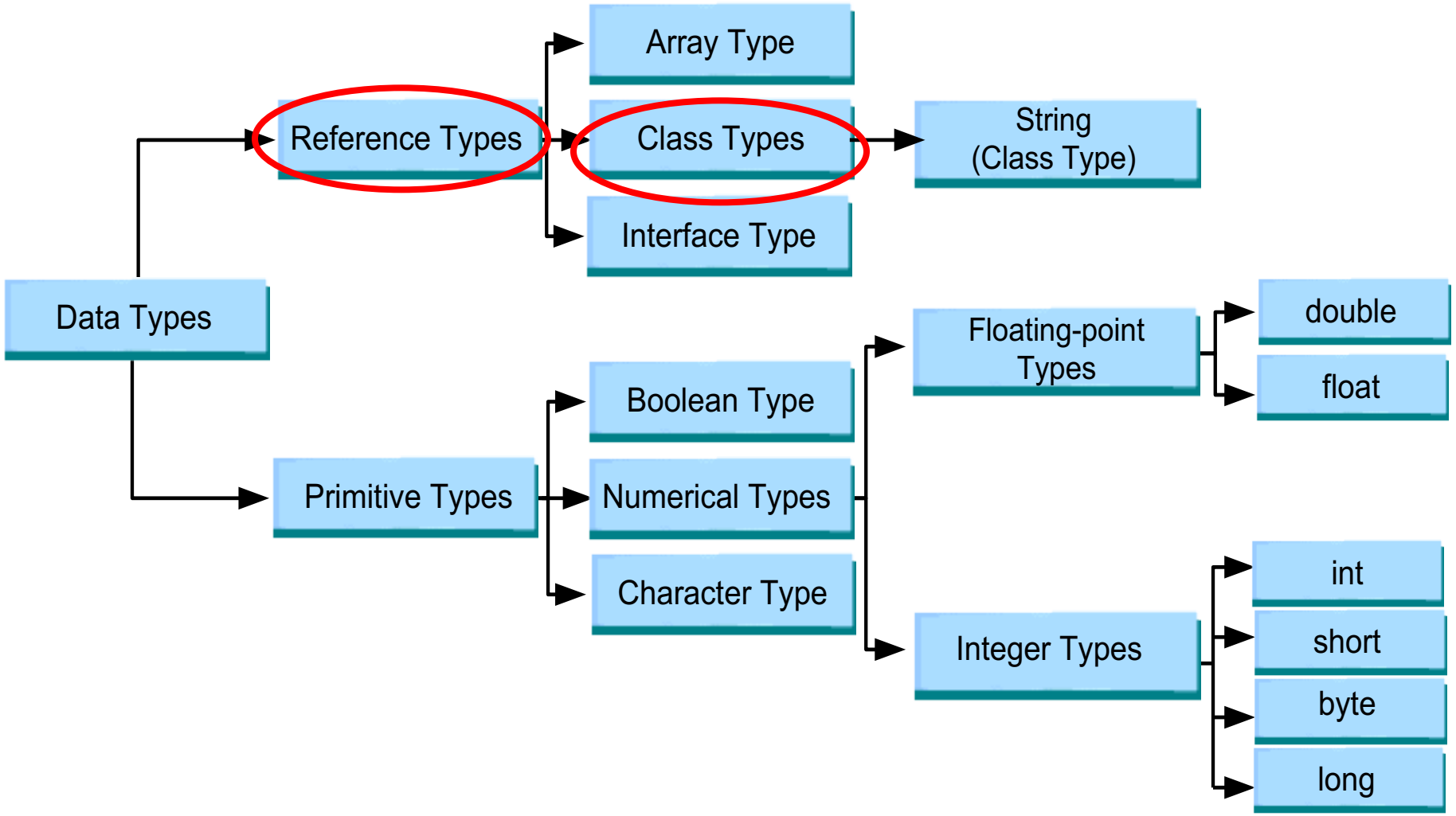
```
public class Student {
    private int    studentId;
    private char   gender;
    private double height;
    private int    age;
    private Course[] course = new Course[3];
…..
}
```



Having more than one student in the system

Topic 11: String Class

Data Types


Data Types
Reference Types
Array Type
Class Types
Interface Type
String
(Class Type)
Primitive Types
Boolean Type
Numerical Types
Character Type
Floating-point
Types
double
float
Integer Types
int
short
byte
long

String Class

- java.lang.String is a class representing strings.
- A string or string constant is a series of characters in double quotes, e.g. “Java Programming”.
- In Java, strings are always created as objects.
- Syntax to declare a string:
  String Variable_Name;

- Since it is an object, memory will only be allocated when:
  Variable_Name = new String(String_Value);

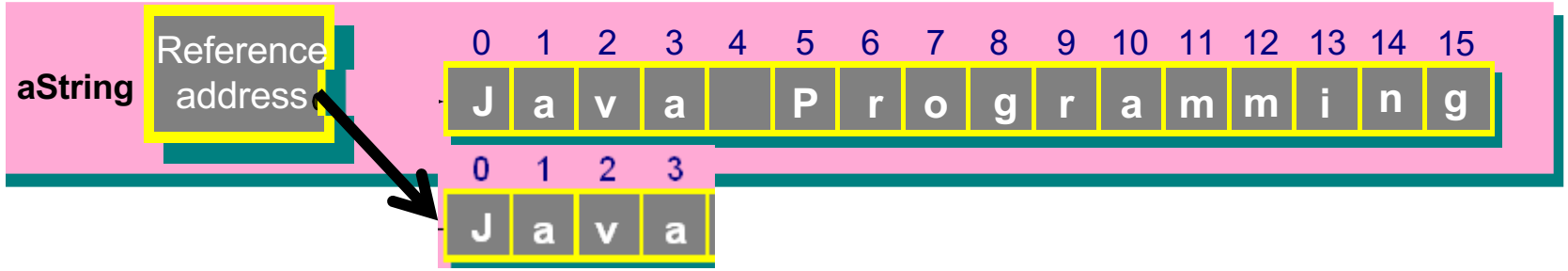
  Combined into one:
  String Variable_Name = new String(String_Value) ;
  or
  String Variable_Name = String_Value ;

Examples

String is Immutable

String aString = new String(“Java Programming”); OR
String aString = “Java Programming” ;

aString = “Java” ;



The variable aString contains reference address of the string object
The length of the string is also stored in the string object’s storage.

String can be used as arguments for System.out.println()

```
System.out.println( “Java Programming” ); OR
System.out.println( aString );      aString.charAt(0)
```

Java Naming Convention

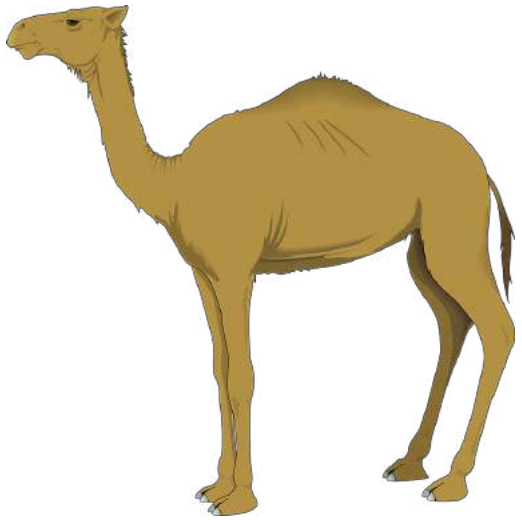
Using the right letter case is the key to following a naming convention:

1. **Lowercase** is where all the letters in a word are written without any capitalisation (e.g., packagename). **For package name.**

2. **Uppercase** is where all the letters in a word are written in capitals. When there are more than two words in the name use underscores to separate them (e.g., MAX_HOURS, FIRST_DAY_OF_WEEK). **For constants, enums.**

3. **CamelCase** (also known as *Upper* CamelCase) is where each new word begins with a capital letter (e.g., CamelCase, CustomerAccount, PlayingCard). **For Classes and Interfaces.**

4. **Mixed case** (also known as *Lower* CamelCase) is the same as CamelCase except the first letter of the name is in lowercase (e.g., **h**as**C**hildren, **c**ustomerFirstName, For **methods, variables**.



Summary

Key points from this chapter:

- Objects have states and behaviors. For example, a dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.

- A Class can be defined as a template/ blueprint that describes the behavior/ state that the object of its type support.

Additional Reading

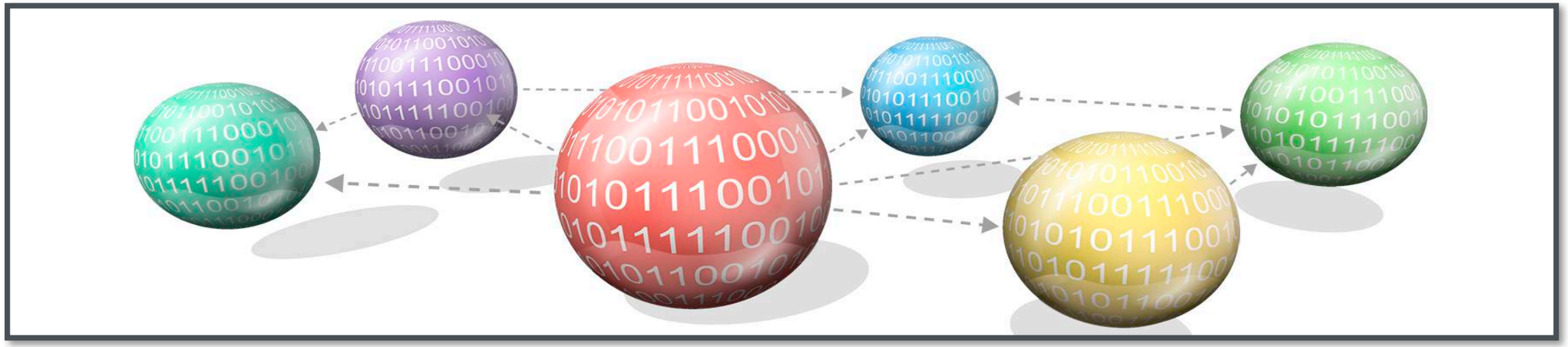
- Class Definition:
  - Text Book: Chapter 4, Page 75-86
- Constructors:
  - Text Book: Chapter 3, Page 53-60
- Java Programming:
  - Lecture Slides: Blackboard/ NTU Learn -> Content
  - Recorded Lectures: Blackboard/ NTU Learn
- C vs. Java
- Java Quick Guide
  - Blackboard/ NTU Learn-> Content -> Course -> Additional Resources

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Chapter 3: Inheritance

CE2002 Object Oriented Design & Programming

Dr Zhang Jie
Assoc Prof, School of Computer Science and Engineering (SCSE)



Learning Objectives

After the completion of this chapter, you should be able to:

- Explain inheritance in Object Oriented Programming (OOP).
- Describe method overloading and method overriding.
- Describe visibility modifiers.
- Write final classes and methods.
- Describe abstract classes and methods.
- Describe multiple inheritance and interfaces.

Topic 1: Inheritance

Inheritance

- Inheritance: An important OO feature that allows us to derive new classes from existing classes by:
  - Absorbing their attributes and behaviours (inherits).
  - Adding new capabilities in the new classes.
    - This enables code reuse and can greatly reduce the programming effort in implementing new classes
    - "is-a" relationship
- Examples:
  - Person (superclass) and Student (subclass).
  - Student is a person.

- In Java, the *root/ ancestor* ('great^{n} – grandfather') of all classes is java.lang.Object (http://docs.oracle.com/javase/7/docs/api/).

Inheritance

- The generalisation relationship:
  - The superclass is a generalisation of the subclasses.
  - The subclasses are specialisations of the superclass.

Inheritance Hierarchy

Superclass
(parent class or base class)

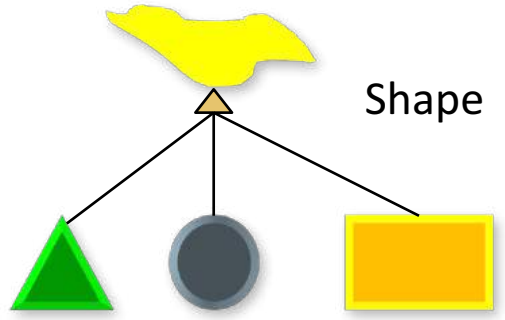
Super, base, parent, generalised

Shape

- xPos: double
- yPos: double

+ Shape()
+ Shape(xCoor: double, yCoor: double)
+ getXPos(): double
+ getYPos(): double
+ print(): void

Common attributes and methods



is-a

is-a

Rectangle

- width: double
- height: double

+ Rectangle()
+ Rectangle(xCoor: double,
yCoor: double,
w: double, h: double)
+ getWidth(): double
+ getHeight(): double
+ findArea(): double
+ findPerimeter(): double
+ print(): void

Additional attributes and methods

Circle

- radius: double

+ Circle()
+ Circle(xCoor: double, yCoor: double
rad: double)
+ getRadius(): double
+ findArea(): double
+ findPerimeter(): double
+ print(): void

Sub, derived, child, specialised

Subclasses: Inherits all the instance variables and methods of the superclass.

(EXCEPT those declared as private - accessibility)

Superclass: Shape Class

```
public class Shape {
   private double xPos ;
   private double yPos ;

   public Shape() {
      xPos = 0 ;
      yPos = 0 ;
   }
   public Shape( double xCoor , double yCoor ) {
      xPos = xCoor ;
      yPos = yCoor ;
   }
   public double getXPos() { return xPos ; }
   public double getYPos() { return yPos ; }
   public void   print()   {
     System.out.println( "[x, y] = [" + xPos
                                    + ", " + yPos + "]");
   }
}
```

Constructors

Subclass: Rectangle Class

```
public class Rectangle extends Shape {
  private double width  ;
  private double height ;
  public Rectangle()
    { super() ; width = 0 ; height = 0 ; }
  public Rectangle( double xCoor , double yCoor ,
                    double w     , double h     )
    { super( xCoor , yCoor ); width = w ; height = h ; }
  public double getWidth()       { return width  ; }
  public double getHeight()      { return height ; }
  public double findPerimeter() { return 2*(width+height) ; }
  public double findArea()       { return width * height   ; }
  public void   print() {
      System.out.println( "Rectangle print() method: " );
      System.out.print(   "Center " );
    super.print();
      System.out.println( "Width = "     + width
                        + ";Height = "   + height );
      System.out.println( "Perimeter = " + findPerimeter() );
      System.out.println( "Area = "      + findArea()      );
  }
}
```

Inherits superclass's attributes and methods

Constructors

From superclass

Subclass Definition

- Subclass is defined using the extends keyword.

- super:
  - Can be used in subclass constructor to call the superclass's constructor. The call to **super()** must be the **first statement** in the constructor for the class.
  - Can also be used in method definition to call the superclass's method, i.e., **super.print();.**

```java

public class FiveGPhone extends MobilePhone{

    private double dataRate;
    private double credit;

    public FiveGPhone(String owner, String color, double screenSize, double dataRate, doubl
        super(owner, color, screenSize);
        this.dataRate = dataRate;
        this.credit = credit;
    }

    public double getDataRate() {
        return dataRate;
    }

    public double getCredit() {
        return credit;
    }

    public void setDataRate(double dataRate) {
        this.dataRate = dataRate;
    }

    public void setCredit(double credit) {
        this.credit = credit;
    }
```

Subclass: Circle Class

```
public class Circle extends Shape {
  private double radius ;

  public Circle()
    { super() ; radius = 0 ; }
  public Circle( double xCoor , double yCoor , double rad )
    { super( xCoor , yCoor ); radius = rad ; }

  public double getRadius()      { return radius;                 }
  public double findPerimeter()  { return 2*Math.PI*radius;       }
  public double findArea()       { return Math.PI*radius*radius; }

   public void print() {
      System.out.println( "Circle print() method: " );
      System.out.print(   "Center "                  );
     super.print();
      System.out.println( "Radius = "    + radius           );
      System.out.println( "Perimeter = " + findPerimeter() );
      System.out.println( "Area = "      + findArea()       );
   }
}
```

Constructors

The Application Class

```
public class ShapesApp {
 public static void main(String[] args) {
  Rectangle rect = new Rectangle( 5 , 5 , 10 , 10 ) ;
  System.out.print( "Rectangle: " );
  System.out.println( "Center Coordinates [x, y] = ["
                    + rect.getXPos() + ", "
                    + rect.getYPos() + "]" );
  System.out.println( "Width = "      + rect.getWidth()
                    + "; Height = "  + rect.getHeight()     );
  System.out.println( "Perimeter = " + rect.findPerimeter() );
  System.out.println( "Area = "      + rect.findArea()      );
  rect.print();
  Circle circ = new Circle( 5 , 10 , 5 );
  System.out.print( "Circle: " );
  System.out.println( "Center Coordinates [x, y] = ["
                    + circ.getXPos() + ", "
                    + circ.getYPos() + "]" );
  System.out.println( "Radius = "    + circ.getRadius()     );
  System.out.println( "Perimeter = " + circ.findPerimeter() );
  System.out.println( "Area = "      + circ.findArea()      );
  circ.print();
 }
}
```

Methods in superclass

Program Input and Output

```
Program Input and Output
Rectangle: Center Coordinates [x, y] = [5.0, 5.0]
Width = 10; Height = 10.0
Perimeter = 40.0
Area = 100.0
Rectangle print() method:
Center [x, y] = [5.0, 5.0]
Width = 10.0; Height = 10.0
Perimeter = 40.0
Area = 100.0
Circle: Center Coordinates [x, y] = [5.0, 10.0]
Radius = 5.0
Perimeter = 31.41592653589793
Area = 78.53981633974483
Circle print() method:
Center [x, y] = [5.0, 10.0]
Radius = 5.0
Perimeter = 31.41592653589793
Area = 78.53981633974483
```

Topic 2: Method Overloading

Method Overloading

(Mentioned here for comparison with Method Overriding, ***NOT*** *a behaviour due to Inheritance.*)

When a method is overloaded, it is designed to perform differently when it is supplied with different ***signatures***, i.e.

Same Method Name but:

1) **different number of parameters.**
2) **different parameter *types*.**

OR

Advantage:

- Overloading allows **methods** to have **similar tasks**, but **differ** only in the number of parameters required by the method **or** the data types of the parameters.
- Without overloading, we need to come up with **different method names** for each similar task instead of just one.

```
printString(…), printInt(…), printBool(….)??
print(String), print(int), print(boolean)
```

Overloading of Method

```
public class Minimum {

    public int findMin(int num1, int num2){
        if (num1 < num2)
            return num1;
        else
            return num2;
    }
    public double findMin(double num1, double num2){
        if (num1 < num2)
            return num1;
        else
            return num2;
    }
    public int findMin(int num1, int num2, int num3){
       return findMin(findMin(num1,num2),findMin(num2,num3));
    }
    public int findMin(int a, int b, int c){
            ..................
        }
```

Overloading of Method

```
    public static void main(String[] args){
        int x=10, y=20, z=5;
        double i=4.5, j=5.5;
         Minimum m = new Minimum();

        System.out.println("findMin(x,y) with int
            args = "    + m.findMin(x,y));

        System.out.println("findMin(i,j) with double
            args ="     + m.findMin(i,j));

        System.out.println("findMin(x,y,z) with int
            args = "    + m.findMin(x,y,z));
    }
}
```

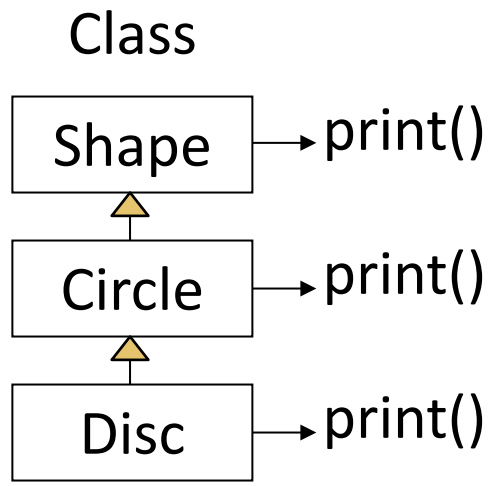
Program Output

```
findMin(x,y) with int args = 10
findMin(i,j) with double args = 4.5
findMin(x,y,z) with int args = 5
```

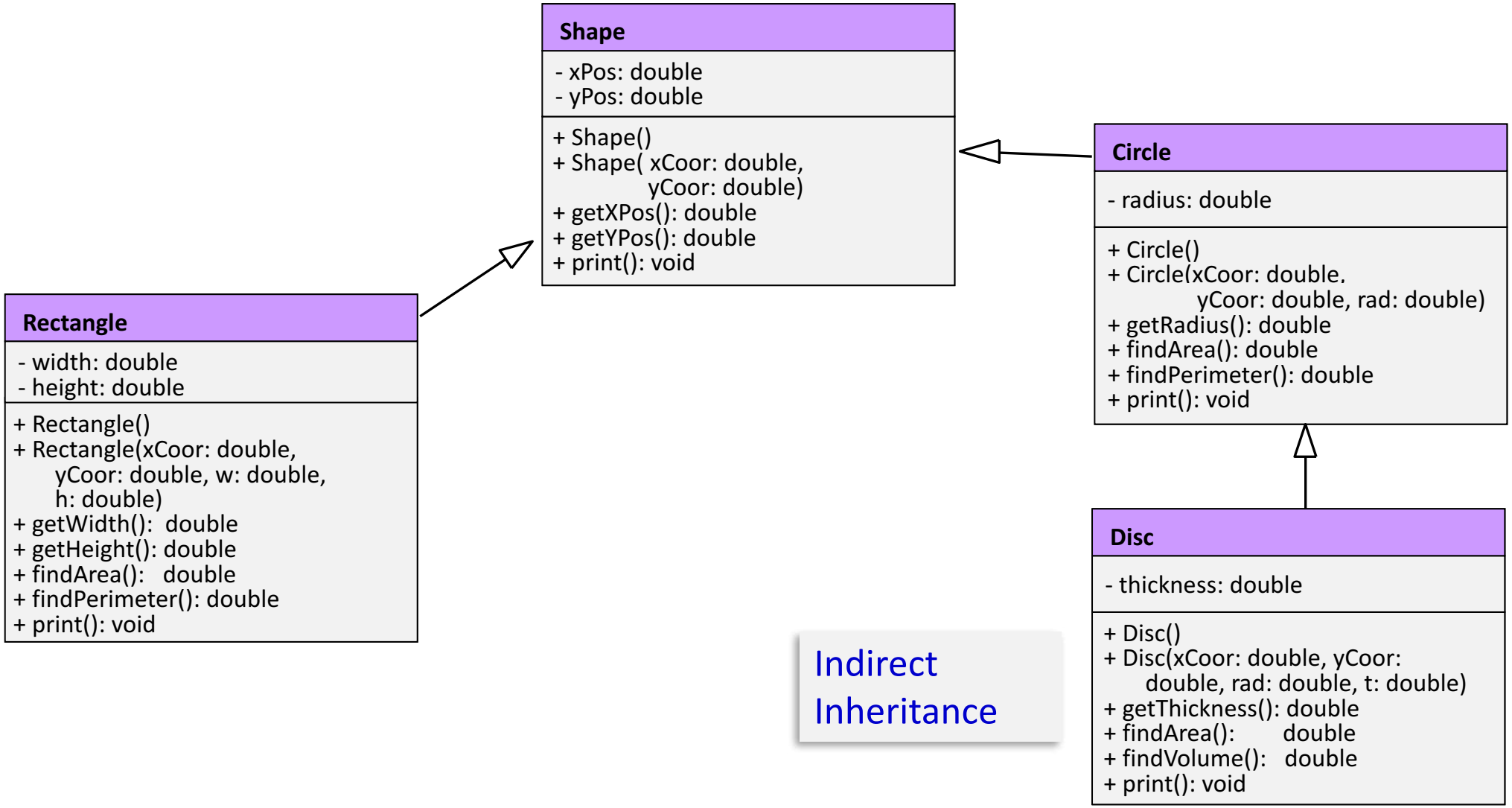
Topic 3: Method Overriding

Method Overriding

- A subclass inherits properties and methods from the superclass. Sometimes, it is necessary to alter the methods of the superclass.
- When a subclass alters a method inherited from a superclass, it overrides that method.
- To override superclass method, we define a method with *exactly* the same signature (argument(s)) and return type as the method in the superclass.
- Two ways:
  1) Refinement: Reuse the implementation of superclass method with some refinement – using the super keyword.
  2) Replacement: Replace the method completely.



Subclass: Disc Class



Subclass: Disc Class

```
public class Disc extends Circle {
  private double thickness ;
  public Disc() { super() ; thickness = 0 ; }
  public Disc( double xCoor , double yCoor ,
               double rad , double t      )
    { super( xCoor , yCoor , rad ) ; thickness = t ; }
  public double getThickness() { return thickness ; }
  public double findArea()
    { return 2 * super.findArea()
           + 2 * Math.PI * getRadius() * thickness ; }
  public double findVolume()
    { return super.findArea() * thickness ; }
  public void print() {
      System.out.println( "Disc print() method: " );
      super.print();
      System.out.println( "Radius = "    + getRadius()   );
      System.out.println( "Thickness = " + thickness     );
      System.out.println( "Area = "      + findArea()    );
      System.out.println( "Volume = "    + findVolume() );
  }
}
```

method overriding

Testing the Disc Class

```
public class DiscApp {
  public static void main( String[] args )
  {
    Circle circ = new Circle( 1   ,  5 , 10       );
    Disc   disc = new Disc(   10 , 20 , 5   , 10 );

    circ.print() ;
    disc.print() ;

    System.out.println( "Volume = " + disc.findVolume() );
    System.out.println( "XPos = "   + circ.getXPos()
                        + " YPos = "  + circ.getYPos() );
    System.out.println( "XPos = "   + disc.getXPos()
                        + " YPos = "  + disc.getYPos() );
  }
}
```

Resolving Method Calls

- When a message is sent (method call) to an object, the search for a matching method begins at the class of the object.

- If not found, the search continues to the **immediate superclass**.

- This proceeds through each immediate superclass until:
  1) A matched method is found; or
  2) No further superclass remain, where an error is reported.

Topic 4: Visibility Modifiers

Visibility Modifiers

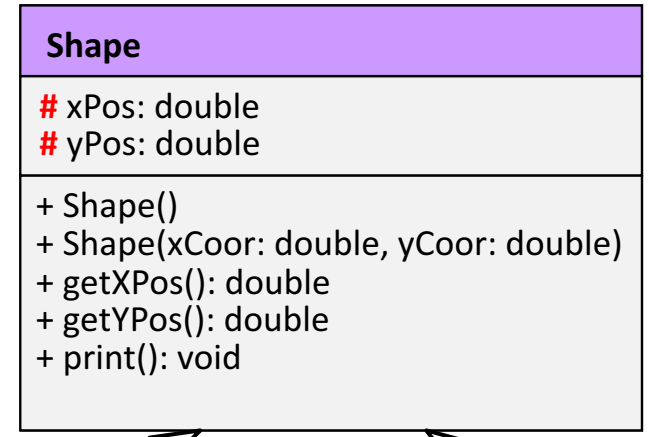
- Visibility (accessibility/ access control):
  - public
    - Visible (accessible) to anywhere in an application.
  - private
    - Visible (accessible) only within that class's implementation.
  - protected *Stay in the family*
    - Visible (accessible) to methods of the class, the methods of subclasses, or any classes in the ***same package***. (class, subclasses, Object class)

  Note: Accessible means read and modify.

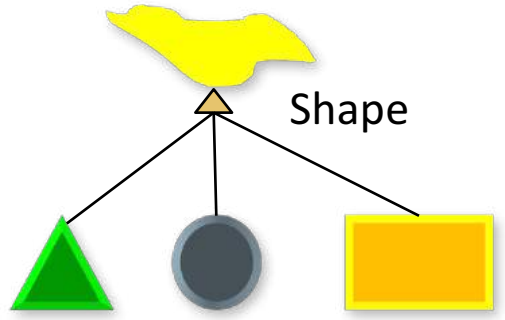
- Protected variables and methods are denoted as '#' in an UML (Unified Modeling Language) diagram.

Inheritance Hierarchy

Superclass
(parent class or
base class)

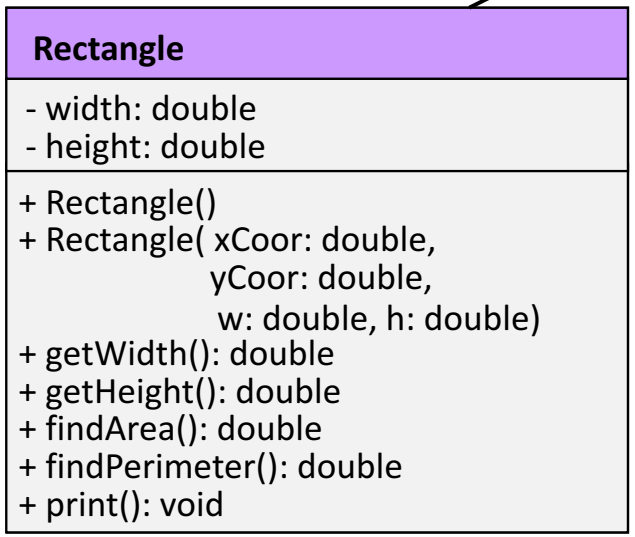


Common attributes
and methods

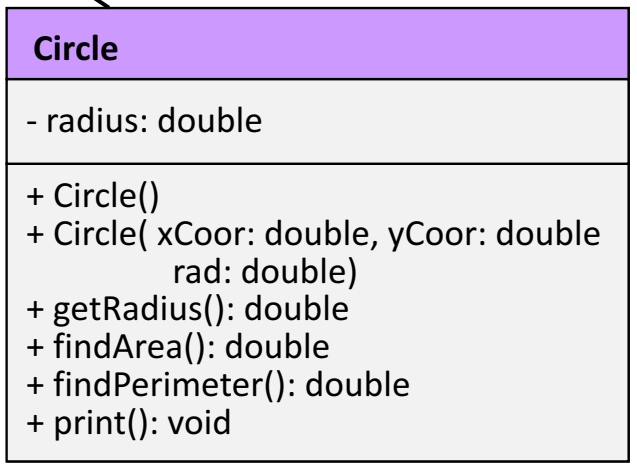


is-a

is-a



Additional
attributes
and methods



Subclasses: Inherits all the instance variables and methods of the superclass. (EXCEPT those declared as private - accessibility)

Defining Package

- A package contains a set of classes that are **grouped** together in the same directory.
- Non-private data can be accessed by any object in the same package.

Example: Two classes named Class1 and Class2 in packageOne:

- The source file **Class1** contains:

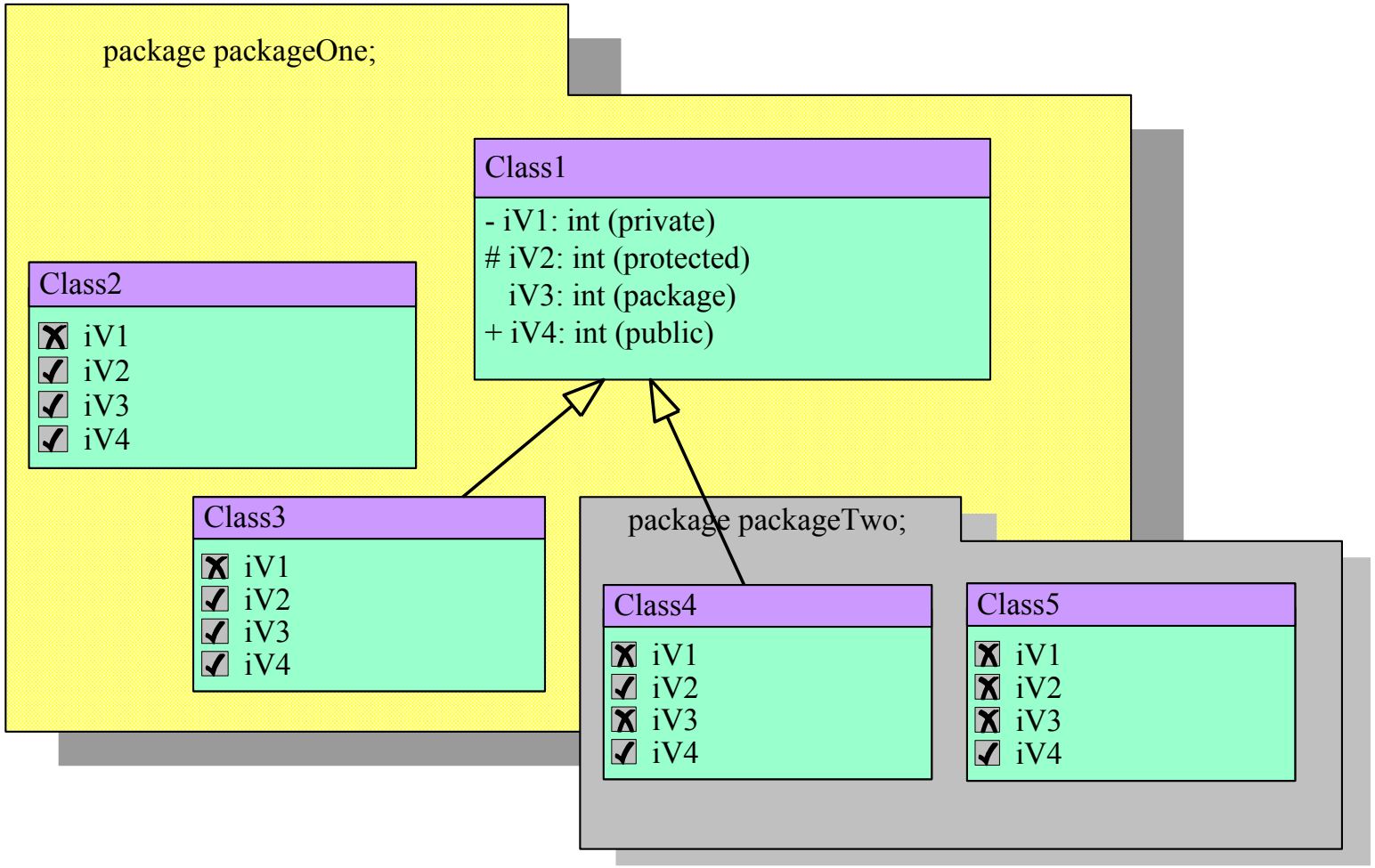
  package packageOne**;**
  public class Class1 { ... }

- The source file **Class2** contains:

  package packageOne**;**
  public class Class2 { ... }

```
package packageOne;

import java.util.Scanner;

public class Class1 {
// the usual……


}
```

Package Access



- In **package access**, instance variables and methods (except private) can be accessed by name inside the definition of any class in the same package.

Visibility for Java

If Visibility is *(Access Level)*	Visible			
	Within Class?	Within Package?	to a Subclass?	to the World? (outside Classes)
public	Y	Y	Y	Y
protected	Y	Y	Y	N
Not defined	Y	Y	N	N
private	Y	N	N	N

Topic 5: Final Classes and Methods

Final Classes and Methods

Final method: When a method is declared as final, the method **cannot be overridden in subclasses**. This aims to prevent the method to be overridden.

Final class: When a class is declared as final, the class **cannot be a superclass**. This means the final class will not have subclasses. The methods in this class is implicitly final.

This helps to:

1) Improve security: Ensures the behaviour of the method will not be changed by subclasses.
2) Improve efficiency: Compile-time type checking and binding can be made instead of waiting till runtime (refer 'Chapter 5: Polymorphism and Dynamic Binding').

Topic 6: Abstract Classes and Methods

Abstract Classes and Methods

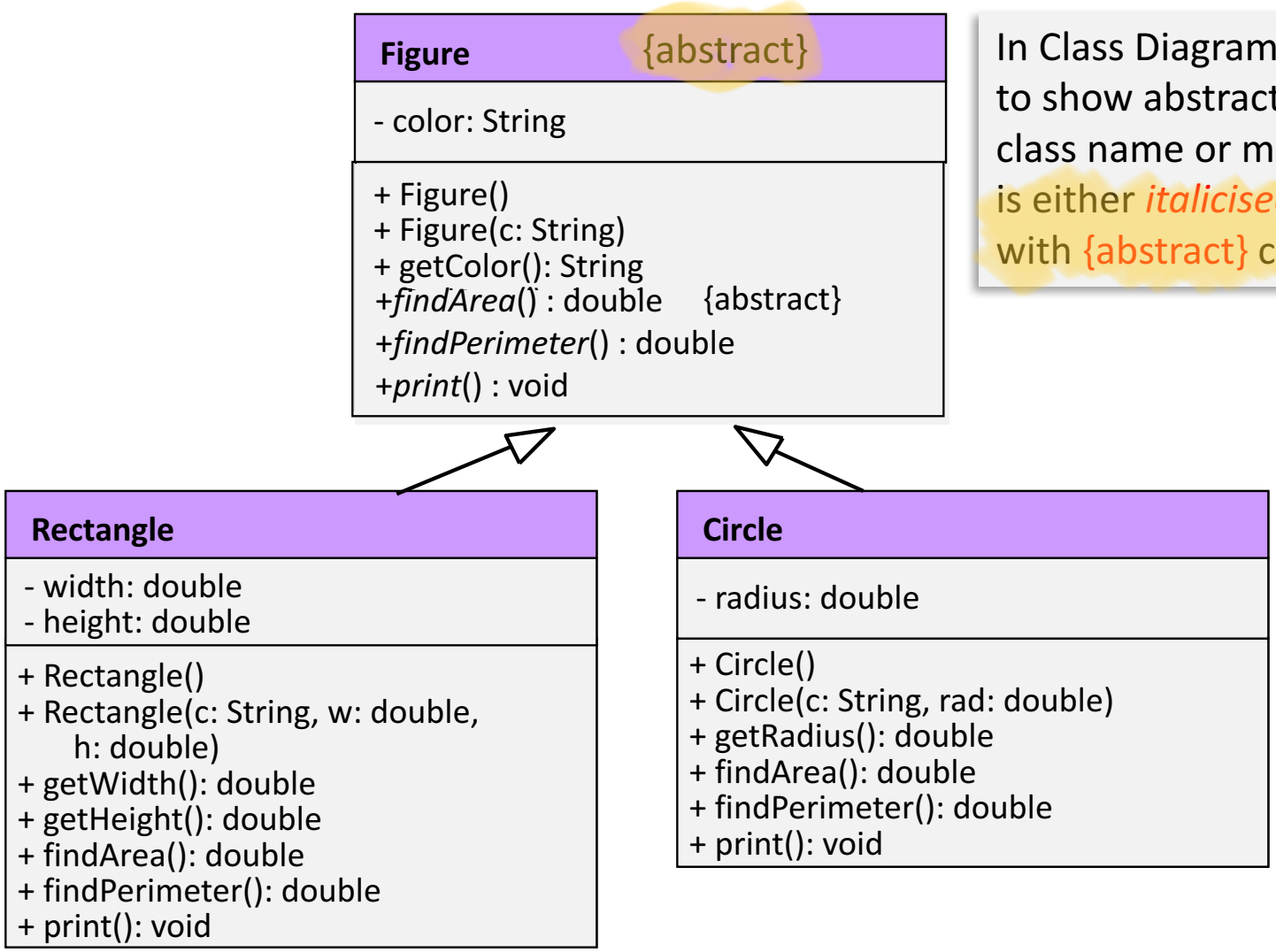
- In a class inheritance hierarchy, a superclass is more **general** than its subclasses. As we move down in the hierarchy, subclasses become more **specific (specialise)** and concrete.

- Therefore, a superclass should contain general features that can be shared by subclasses. ***abstraction***

- But if a superclass is too general, then no meaningful object can be created (instantiated) from it, such classes are called abstract classes (Abstract Base Class (ABC)).

- Abstract method (with keyword abstract) does not have the implementation (no code) in the abstract class. The implementation must be provided by the subclasses.

Inheritance Hierarchy with Abstract Classes



In Class Diagram,
to show abstract,
class name or method name
is either *italicised* OR
with {abstract} constraint

The Abstract Class Figure

Cannot used to create object

```
public abstract class Figure
{
   private String color;
   public  Figure()            { color      = "black"; }
   public  Figure( String c ) { this.color = c;       }

   public String getColor() { return color; }

   // abstract methods - no method body
   public abstract double findArea()      ;
   public abstract double findPerimeter() ;
   public abstract void   print()         ;
}
```

no method implementation

Subclass: Rectangle Class

```
public class Rectangle extends Figure {
   private double width  ;
   private double height ;

   public Rectangle()
      { super()     ; this.width = 0 ; this.height = 0 ; }
   public Rectangle( String c , double w , double h )
       { super( c ) ; this.width = w ; this.height = h ; }
   public double getWidth()  { return width  ; }
   public double getHeight() { return height ; }

   // implementation of the abstract methods
   public double findArea()      { return width * height   ; }
   public double findPerimeter() { return 2*(width+height) ; }
   public void print() {
      System.out.println( "Rectangle print() method: " );
      System.out.println( "Width = "    + width
                        + "; Height = " + height           );
      System.out.println( "Perimeter = " + findPerimeter() );
      System.out.println( "Area = "      + findArea()      );
   }
}
```

Subclass: Circle Class

```
public class Circle extends Figure
{
  private double radius;
  public Circle() { super()   ; this.radius = 0    ; }
  public Circle( String c , double rad )
                  { super(c) ; this.radius = rad ; }

  public double getRadius() { return radius; }

  // implementation of the abstract methods
  public double findArea()
    { return Math.PI * radius * radius ; }
  public double findPerimeter()
    { return 2 * Math.PI * radius ; }
  public void print() {
    System.out.println( "Circle print() method: "         );
    System.out.println( "Radius = "     + radius          );
    System.out.println( "Perimeter = " + findPerimeter() );
    System.out.println( "Area = "       + findArea()      );
  }
}
```

Testing the Figure Abstract Class

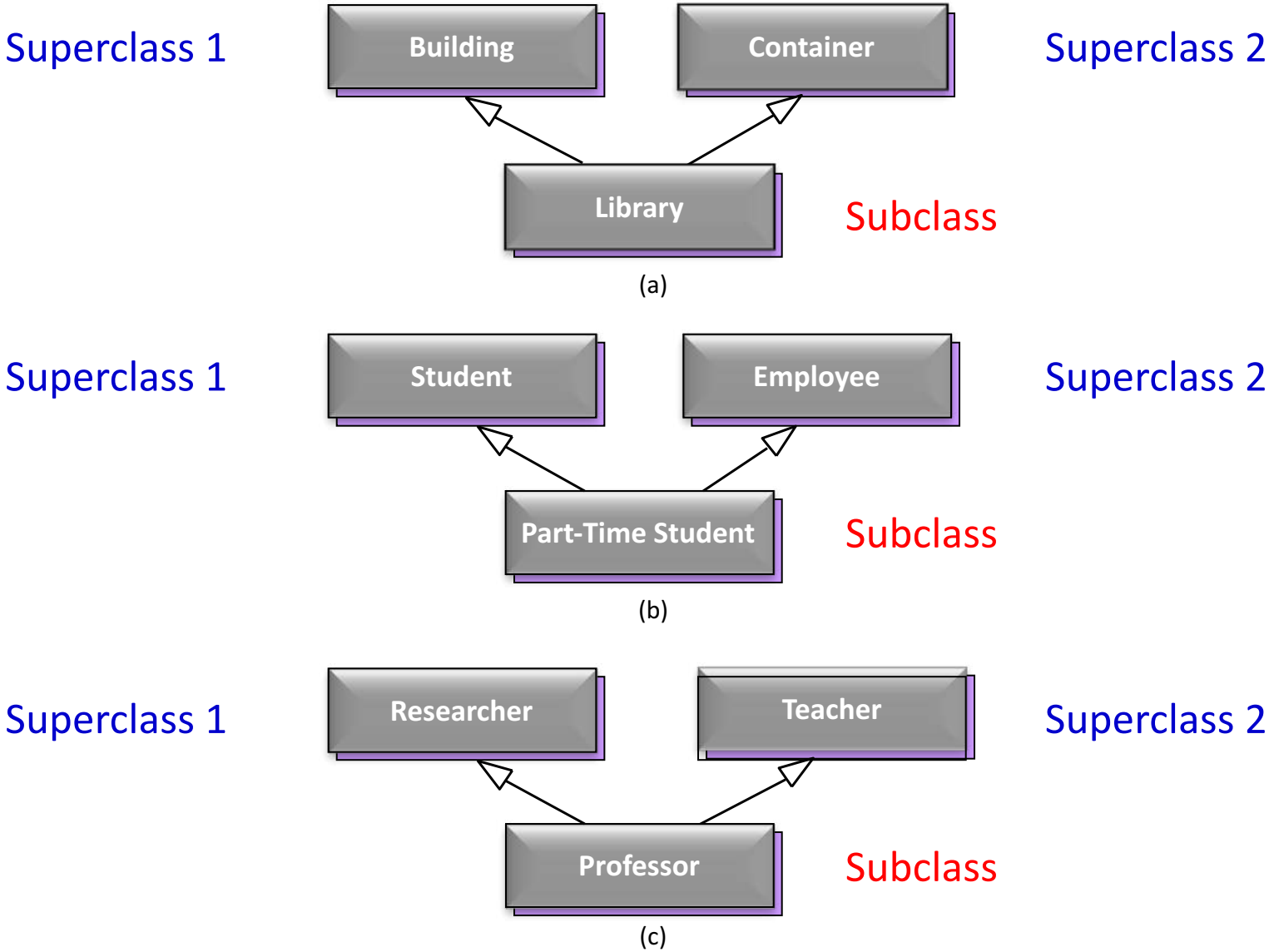
```
public class FigureApp {
 public static void main( String[] args )
 {
  Rectangle rect = new Rectangle( "Red" , 10 , 10 );
  System.out.println( "Rectangle: " );
  System.out.println( "Color : "     + rect.getColor()       );
  System.out.println( "Width = "     + rect.getWidth()       );
  System.out.println( "Height = "    + rect.getHeight ()     );
  System.out.println( "Perimeter = " + rect.findPerimeter() );
  System.out.println( "Area = "      + rect.findArea()       );
  rect.print();
  System.out.println();

  Circle circ = new Circle( "Orange" , 5 );
  System.out.println( "Circle: " );
  System.out.println( "Color : "     + circ.getColor()       );
  System.out.println( "Radius = "    + circ.getRadius()      );
  System.out.println( "Perimeter = " + circ.findPerimeter() );
  System.out.println( "Area = "      + circ.findArea()       );
  circ.print();
 }
}
```

Program Input and Output

```
Program Input and Output
Rectangle:
Color : Red
Width = 10.0
Height = 10.0
Perimeter = 40.0
Area = 100.0
Rectangle print() method:
Width = 10.0; Height = 10.0
Perimeter = 40.0
Area = 100.0
Circle:
Color : Orange
Radius = 5.0
Perimeter = 31.41592653589793
Area = 78.53981633974483
Circle print() method:
Radius = 5.0
Perimeter = 31.41592653589793
Area = 78.53981633974483
```

Topic 7: Multiple Inheritance and Interfaces

Multiple Inheritance (Multiple is-a)



Interfaces

- Java does not support multiple inheritance.
- Java supports **multiple interface**. A class can implement more than one interface and allow it to reflect the behaviour of two (or more) 'parents'.
- An interface is like an abstract class, except it contains **only abstract methods** and **constants** (**with static final**).
- A class implementing/ realising an interface promises that it will provide an implementation to **all the abstract methods** in the interface. Otherwise, the new class will also be abstract.

Example

```
public interface IntFigure
// interface is similar to abstract class
{
  // static final constants for instance variables
  // abstract methods
  public abstract double findPerimeter();
  public abstract double findArea();
  public          void   print();
}

public class Circle implements IntFigure // circle example
{
  private double radius;
  public Circle()                { this.radius = 0    ; }
  public Circle( double rad )    { this.radius = rad ; }
  public double getRadius()      { return radius ; }
  public double findArea()       { return Math.PI*radius*radius; }
  public double findPerimeter() { return 2*Math.PI*radius;      }
  public void   print()          { ... }
}
```

Derived Interfaces

- Like classes, an interface may be derived from a base interface.
  - This is called extending the interface.
  - The derived interface must include the phrase extends <BaseInterfaceName>.

- A concrete class that implements a derived interface must have definitions for any methods in the derived interface as well as any methods in the base interface.

Derived Interfaces

```
public interface IOrdered
{
    public boolean precedes(Object other);

    public boolean follows(Object other);
}
```

```
public interface IShowablyOrdered extends IOrdered
{
    public void showOneWhoPrecedes();
}
```

```
public class Order implements IShowablyOrdered
{
    public boolean precedes(Object other) {….}
    public boolean follows(Object other) {….}
    public void showOneWhoPrecedes() {….}
}
```

Abstract vs. Interfaces

- An abstract class is a real parent (base class); an interface is not a real parent (~~base class~~).

  type

- An abstract class can have object attributes (data members) and non-abstract methods. An interface can only have constants and abstract methods.

  Constructor?

- A class can implement more than one interface, but can extend (inherit) from only one abstract class.

Note: A Concrete class has implementation for all methods, i.e. NO abstract methods.

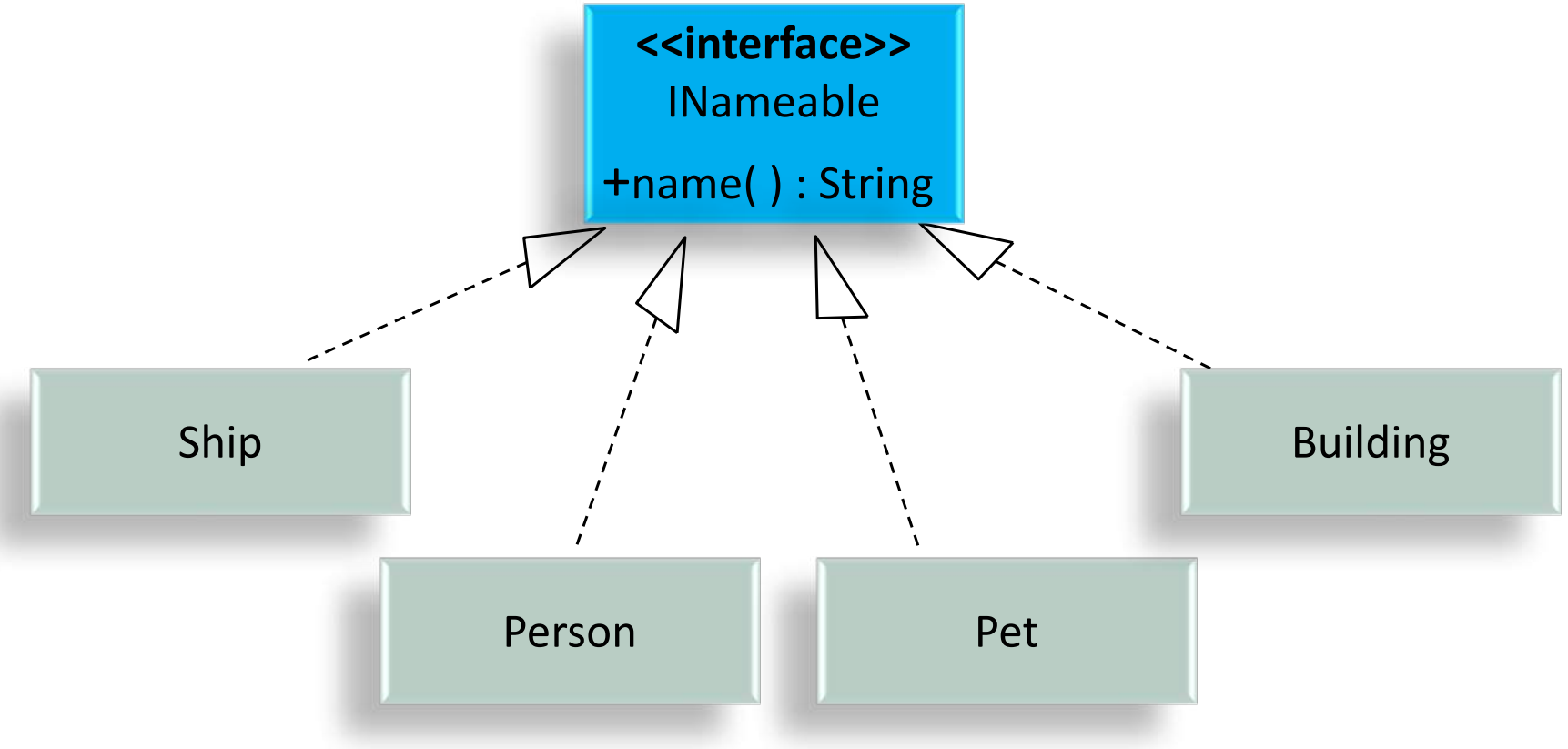
Example

```
public class Shape  {      // CONCRETE
        String color ;
        public Shape(String color )  {  this.color = color ; }
        public String getColor()  { return color ; }
        public double findArea()  { return 0 ; }
        public double findPerimeter() { return 0 ; }
}
```

```
public abstract class Shape  {
        String color ;
        public Shape(String color )  {  this.color = color ; }
        public String getColor()  { return color ; }
        public abstract double findArea()  ;
        public abstract double findPerimeter() ;
}
```

```
public interface IShape  {

        public double findArea()  ;
        public double findPerimeter() ;
}
```

Interface Use



Interfaces are particularly useful for assigning common functionality to possibly ***unrelated classes.***

Abstract vs. Interfaces

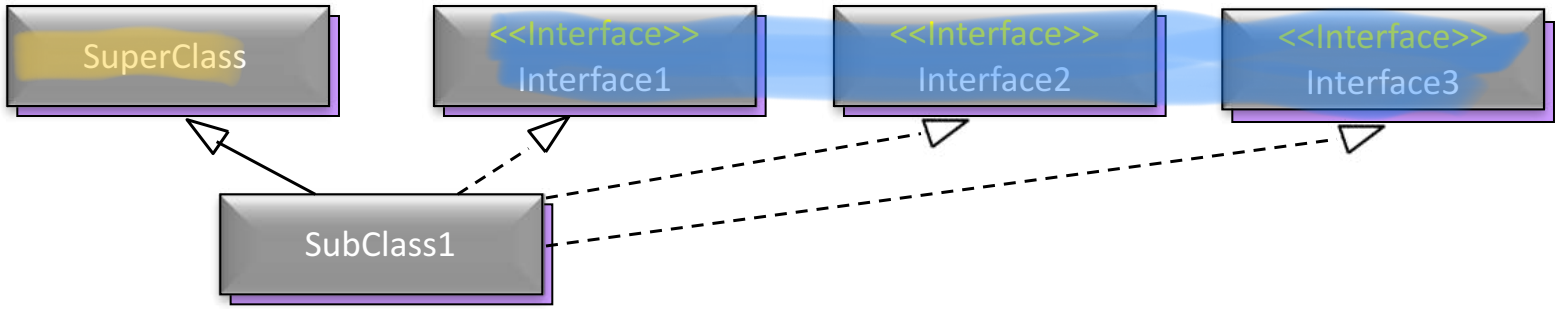
Abstract	Interface
May have some methods declared as `abstract`.	Can only have abstract methods.
May have `protected` properties and `static` methods.	Can only have `public` methods with no implementation.
May have `final` and non-final data attributes.	Limited to only constants (`static final`).

Both Abstract class and Interface CANNOT be instantiated with *new*, i.e.,

..... = new <AbstractClass>( ) ;

..... = new <Interface>( ) ;

Multiple Inheritance in Codes




```
public class SubClass1 extends SuperClass implements Interface1,
  Interface2, Interface3
{
  ..........
}
```

diff from C++ that allow multiple inheritance

Topic 8: Java Packages

Java Packages

- Packages: Organise/ group java classes (files) into different directories according to their functionality, usability as well as category they should belong to.
- Derives from Internet domain space (namespace), i.e., com.foo.users, sg.edu.ntu.projects.
- Package namespace is global. Same class name can be unique via using namespace.
- If no name is given, classes are in an anonymous "default package".

Java Packages

- Using 2 Classes with same class name
- Example:
  - **`Deck`** class for deck of cards
  - **`Deck`** class for ship deck (level/ floor of a ship)

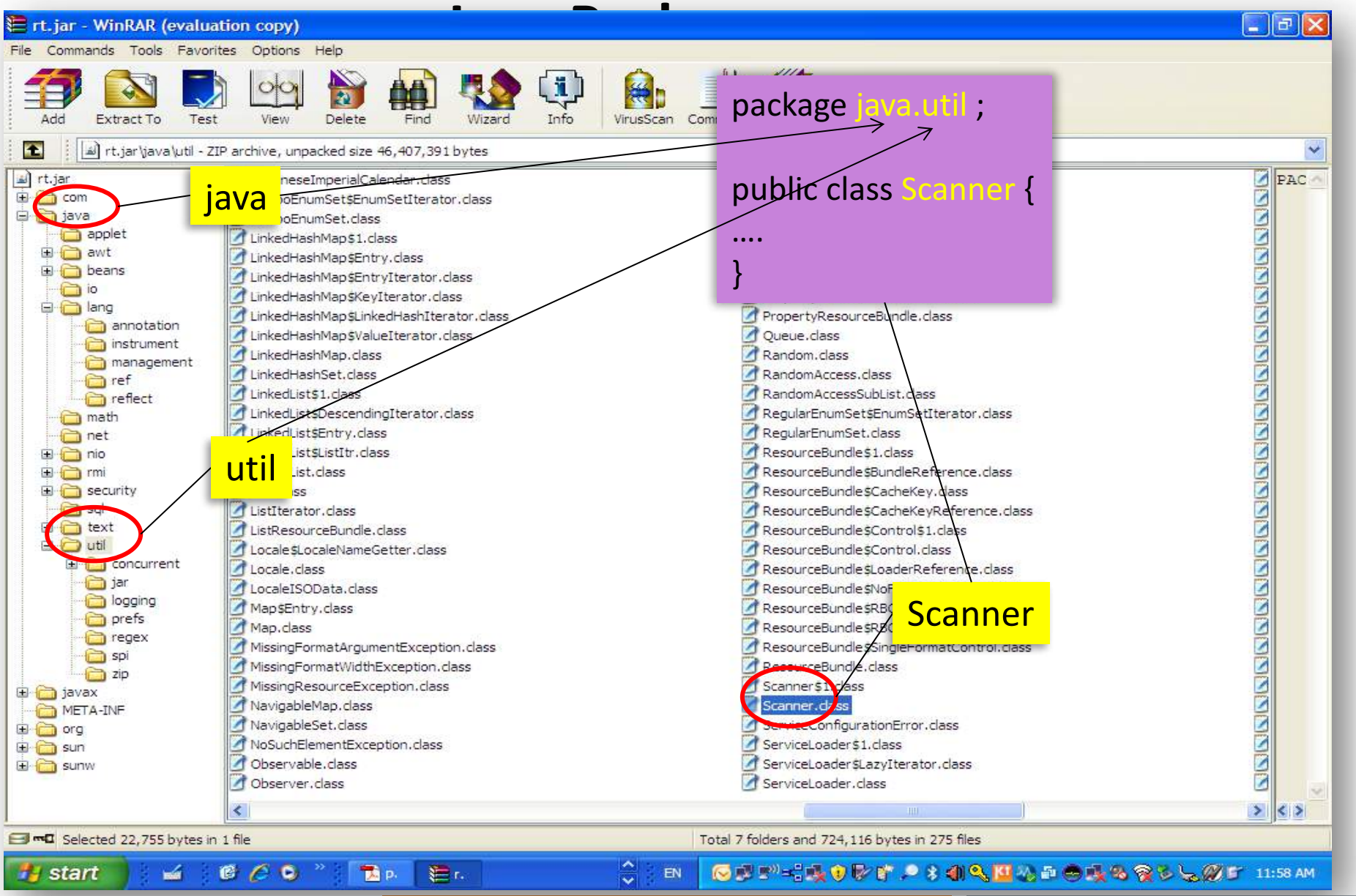
Casino System in a Cruise Ship

`casino.game.Deck` for Deck of cards

`star.cruise.Deck` for Ship Deck

```
public class CruiseTour {
  casino.game.Deck cards = new casino.game.Deck()
  star.cruise.Deck deck = new star.cruise.Deck();
……

}
```

Java Packages



Notes

```
Scanner sc = new Scanner(System.in) ;
char choice = sc.nextLine().charAt(0) ;
                    String
```

```
System.out.println("blah blah…");
      PrintStream
```

```
casino.game.Deck.getTotalCards()  ;
```

java.lang.System.out.println("");

Method Header

General format

public <static> Return_Type Method_Name (Parameter_List)

Method Signature

1. Public <static>

reserved keywords

2. Return_Type

can be int, double, float, char,…, void

Class type

(Abstract Data Type)

3. Parameter_List

Example

public static **int** successor(*double num*) { …… }

public **boolean** *savetoDB(String name, int age, char gender)*

{……}

Summary

Key points from this chapter:

- Inheritance is an important OO feature that allows us to derive new classes from existing classes.
- Methods overloading: When a method is overloaded, it is designed to perform differently, when it is supplied with different signatures.
- Method overriding: When a subclass alters a method inherited from a superclass, it overrides that method.
- Visibility modifiers: Visibility (accessibility/ access control); public, private, and protected.

Summary

Key points from this chapter:

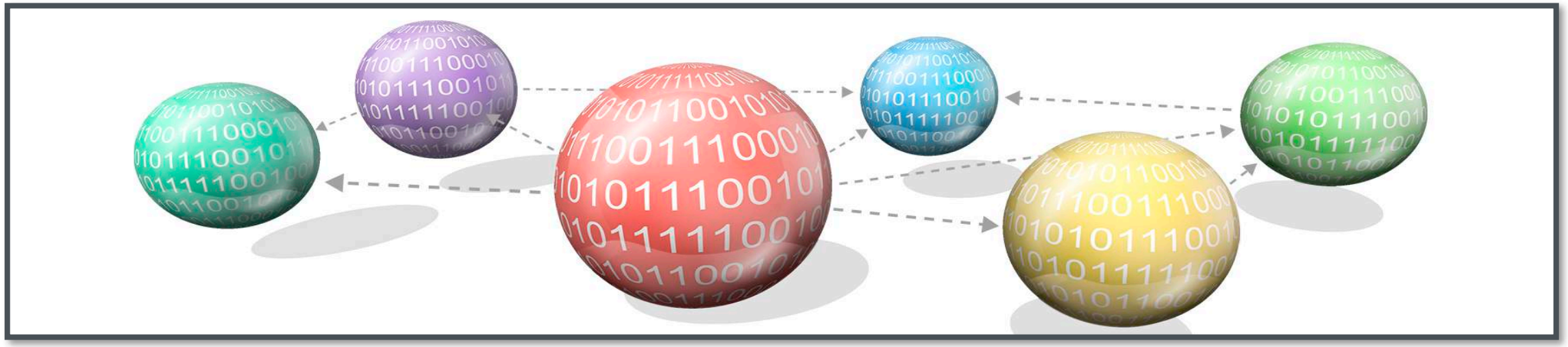
- Final method: When a method is declared as final, the method cannot be overridden in subclasses.
- Final class: When a class is declared as final, the class cannot be a superclass.
- Abstract method (with keyword abstract) does not have the implementation (no code) in the abstract class.
- Java does not support multiple inheritance, but supports multiple interface.
- Java Packages: Organise/ group java classes (files) into different directories according to their functionality, usability as well as category they should belong to.

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Chapter 4: Exception Handling

CE2002 Object Oriented Design & Programming

Dr Zhang Jie
Assoc Prof, School of Computer Science and Engineering (SCSE)



Learning Objectives

After the completion of this chapter, you should be able to:

- Handle error in Java.
- Handle Java's exception.
- Summarise Java's exception hierarchy.
- Use exception classes.
- Create your own exception classes.

Topic 1: Error Handling

Error Handling

- Exception: Run-time anomalies that a program may detect.
- Examples: Memory exhaustion, an input file cannot be opened, division by zero, etc.
  - May cause serious problems and disrupt normal execution of the program.
  - Well-designed programs should have error-handling code inside the program code to handle these run-time anomalies.
- In Java, exception handling is provided to catch and handle run-time exceptions.
- Exception handlers (or recovery procedure) may catch an exception in order to recover from the problem.

Computing Average Marks Version 1

```
import java.util.Scanner ;
public class AverageMarksV1 {
   public static void main( String[] args ) {
      int     i , numOfStudents ;
      double  totalMarks = 0 , avgMarks = 0 ;
      Scanner sc         = new Scanner( System.in );
      System.out.print( "Enter number of students: " );
      numOfStudents = sc.nextInt();
      System.out.print("Enter student marks: ");
      for ( i = 0 ; i < numOfStudents ; i++ )
         totalMarks += sc.nextDouble() ;
      avgMarks = totalMarks / (double)numOfStudents ;
                 // error if numOfStudents = 0
      System.out.println( "Average marks = " + avgMarks );
   }
}
```

```
Program Input and Output
Enter number of students: 5
Enter student marks: 70 80 90 60
50
Average marks = 70.0
```

Computing Average Marks Version 2

```
import java.util.Scanner ;
public class AverageMarksV2 {
  public static void main( String[] args ) {
    int     i , numOfStudents ;
    double  totalMarks = 0 , avgMarks = 0 ;
    Scanner sc = new Scanner( System.in );
    System.out.print( "Enter number of students: " );
    numOfStudents = sc.nextInt() ;
    if ( numOfStudents <= 0 ) {
      System.out.print(   "Error: no of students " );
      System.out.println( "must not equal to 0!"   );
      System.out.println( "Program Terminating!"   );
      System.exit( 0 ) ;
    }
    System.out.print( "Enter student marks: " );
    for ( i = 0 ; i < numOfStudents ; i++ )
      totalMarks += sc.nextDouble() ;
    avgMarks = totalMarks / (double)numOfStudents ;
    System.out.println( "Average marks = " + avgMarks );
  }
}
```

Program Input and Output

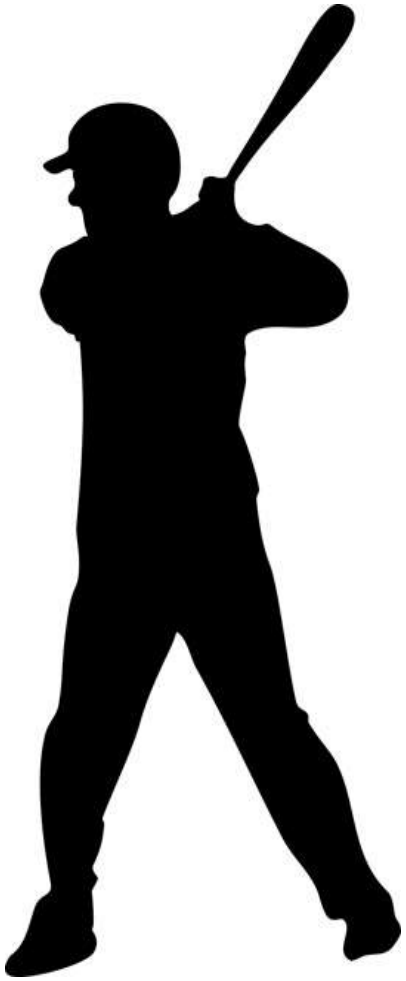
```
Enter number of students: 0
Error: no of students must not equal to 0!
Program Terminating!
```

Topic 2: Java’s Exception Handling

Java's Exception Handling

- **Trying an Exception**:
  - **Try** to see if there is any exception.
  - Every method must first state the types of exceptions it can handle.

- **Throwing an Exception:**
  - When the method detects an error or exception on a statement contained in it, it creates (raises) an exception object that contains information on the type of the exception, and the state of the program when the error occurred, to signal the abnormal condition.
- **Catching an Exception**:
  - When an exception is thrown, the **JVM** looks for an **exception handler** that can **catch** and handle the exception. The exception handler must match the type of the exception thrown.



‘try/throw/catch’ Mechanism

```
try {
    // statements for normal flow of program execution
    // at least one statement should be capable of
    // throwing an exception based on some conditions
    if ( /* some conditions happen */ )
        throw new Exception_Name
        ( Optional_String_Arguments );
}
catch ( Exception_Class_Name1 Parameter_Name_1 ) {
    // statements to handle the exception
}
...
catch ( Exception_Class_NameN Parameter_Name_n ) {
    // statements to handle the exception
}
finally {
    // (optional) statements to be executed regardless
    // of whether an exception is thrown or not
}
```

Computing Average Marks Version 3

```
import java.util.Scanner ;
public class AverageMarksV3 {
   public static void main( String[] args ) {
      int     i, numOfStudents;
     double  totalMarks = 0, avgMarks = 0 ;
      Scanner sc          = new Scanner( System.in );
      try {
         System.out.print( "Enter number of students: " );
         numOfStudents = sc.nextInt();
         if ( numOfStudents <= 0 )
            throw new Exception(
               "Error: no of students must not equal to 0!" );
         System.out.print( "Enter student marks: " );
         for ( i = 0 ;  i < numOfStudents ; i++ )
           totalMarks += sc.nextDouble() ;
         avgMarks = totalMarks / (double) numOfStudents ;
         System.out.println( "Average marks = " + avgMarks );
      }
      catch ( Exception e ) {
         System.out.println( e.getMessage() );
      }
      System.out.println( "End of program execution!" );
} }
```

#1: Flow of Control: When No Exception

```
import java.util.Scanner ;
public class AverageMarksV3 {
   public static void main( String[] args ) {
      int     i, numOfStudents;
     double  totalMarks = 0, avgMarks = 0 ;
      Scanner sc          = new Scanner( System.in );
      try {
         System.out.print( "Enter number of students: " );
         numOfStudents = sc.nextInt();
         if ( numOfStudents <= 0 )
            throw new Exception(
               "Error: no of students must not equal to 0!" );
         System.out.print( "Enter student marks: " );
         for ( i = 0 ;  i < numOfStudents ; i++ )
           totalMarks += sc.nextDouble() ;
         avgMarks = totalMarks / (double) numOfStudents ;
         System.out.println( "Average marks = " + avgMarks );
      }
      catch ( Exception e ) {
         System.out.println( e.getMessage() );
      }
      System.out.println( "End of program execution!" );
} }
```

skip

skip

E.g., numOfStudents = 5

#2: Flow of Control: When Exception is Thrown

```
import java.util.Scanner ;
public class AverageMarksV3 {
   public static void main( String[] args ) {
      int     i, numOfStudents;
     double  totalMarks = 0, avgMarks = 0 ;
      Scanner sc          = new Scanner( System.in );
      try {
         System.out.print( "Enter number of students: " );
         numOfStudents = sc.nextInt();
         if ( numOfStudents <= 0 )
           throw new Exception(
              "Error: no of students must not equal to 0!" );
         System.out.print( "Enter student marks: " );
         for ( i = 0 ;  i < numOfStudents ; i++ )
           totalMarks += sc.nextDouble() ;
         avgMarks = totalMarks / (double) numOfStudents ;
         System.out.println( "Average marks = " + avgMarks );
      }
      catch ( Exception e ) {
         System.out.println( e.getMessage() );
      }
      System.out.println( "End of program execution!" );
} }
```

skip

E.g., numOfStudents = 0

Computing Average Marks Version 3

Case # 1

Case # 2

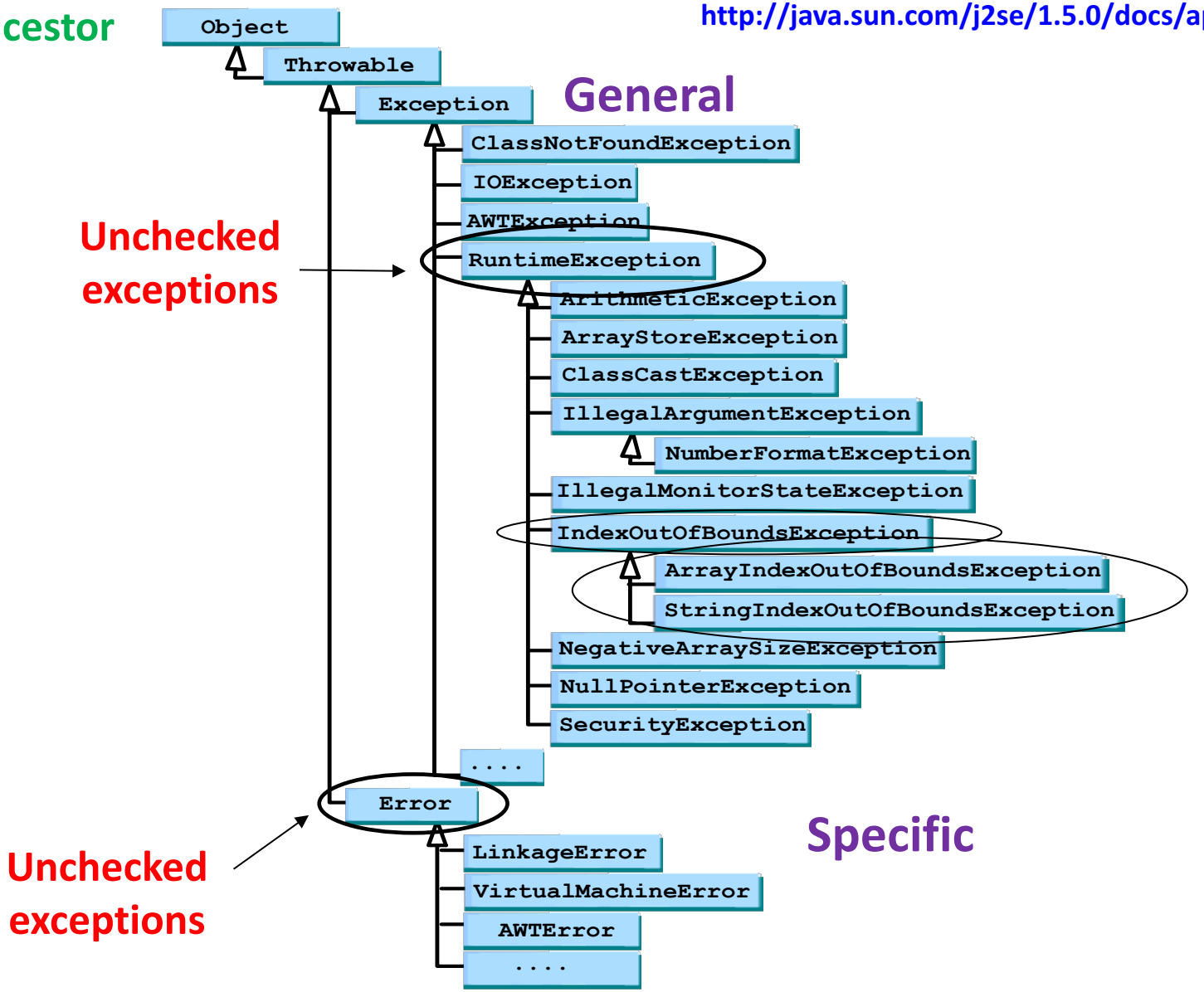
```
Program Input and Output
Enter number of students: 5
Enter student marks: 70 80 90 60 50
Average marks = 70.0
End of program execution!

Enter number of students: 0
Error: no of students must not equal to 0!
End of program execution!
```

Topic 3: Java’s Exception Hierarchy

Java's Exception Hierarchy

root/ancestor

http://java.sun.com/j2se/1.5.0/docs/api/java/lang/package-tree.html



Some Useful Exceptions (System Generate)

Exception (Predefined)	Description
ArithmeticException	This indicates division by zero or some kinds of arithmetic exceptions.
IndexOutOfBoundsException	This indicates that an array or string index is out of bound.
ArrayIndexOutOfBoundsException	This indicates that an **array** index is less than zero or greater than or equal to the array's length.
StringIndexOutOfBoundsException	This indicates that a **string** index is less than zero or greater than or equal to the string's length.
FileNotFoundException	This indicates that the reference to a file cannot be found.
IllegalArgumentException	This indicates that an improper argument is used when calling a method.
NullPointerException	This indicates that an object reference has not been initialised yet.
NumberFormatException	This indicates that illegal num format is used.

Some Useful Exceptions (System Generate)

Exception (Predefined)	Description
ArithmeticException	This indicates division by zero or some kinds of arithmetic exceptions.
IndexOutOfBoundsException	This indicates that an array or string index is out of bound.
ArrayIndexOutOfBoundsException	This indi…o or greater than or equal to the array's length.
StringIndexOutOfBoundsException	This indi… or greater t…
FileNotFoundException	This indicates that the reference to a file cannot be found.
IllegalArgumentException	This indicates that an improper argument is used when calling a method.
NullPointerException	This indi…een initialise…
NumberFormatException	This indicates that illegal num format is used.

```
int [] array = new int[5];
int  j = array[5];
```

```
String str = "Hello" ;
 char c =  str.charAt(str.length()) ;
```

```
String str = null;
int j = st.length();
```

```
int  i  =  Integer.parseInt("abc");
```

Exception Class

- **Exception** is the root class of all exceptions.
- The **Exception class** has two constructors:

```
public Exception()
public Exception( String message )
```

- The Exception class contains some useful **instance methods** to get information related to the exception:
  - **String getMessage()**: Returns the message of the exception object.
  - **String toString()**: Returns a short description of the exception object.
  - **void printStackTrace()**: Print on screen a **trace** of all the methods that were called, leading up to the method that threw the exception.

The Exception Class

- **Example:**

```
try
{
   // statements that may throw ArithmeticException
}
catch (ArithmeticException e)
{
  // An example exception handling code:
  System.out.println( e.getMessage() );
  e.printStackTrace() ;
  System.exit(0);    // terminate the program
}
```

The Exception Class

- **Example:**

```
try
{
   // stater
}
catch (Arit
{
   // An exa
   System.ou
   e.printSt
   System.ex
}
```

```
public class  CreateException
{
     public static void main(String[] args) {
method1(3, 0)  ;
    }
     public static void method1(int i, int j) {
    method2(i,j);
     }
     public static void method2(int i, int j) {
    method3(i,j);
     }
     public static int method3(int i, int j) {
    int k = 0 ;
           try {
          k = i / j ;
           }catch (ArithmeticException e)  {
    System.out.println( e.getMessage() );
    e.printStackTrace() ;
    System.exit(0);   // terminate the program
          }
           return k ;
      }
}
```

The Exception Class

- **Example:**

```
try
{
   // stater
}
catch
{
  // 
  Sys
  e.p
  Sys
}
```

```
public class  CreateException
{
     public static void main(String[] args) {
method1(3, 0)  ;
    }
     public static void method1(int i, int j) {
    method2(i,j);
```

```
/ by zero
java.lang.ArithmeticException: / by zero
        at CreateException.method3(CreateException.java:36)
        at CreateException.method2(CreateException.java:31)
        at CreateException.method1(CreateException.java:28)
        at CreateException.main(CreateException.java:16)
Press any key to continue . . .
```

```
    System.out.println( e.getMessage() );
    e.printStackTrace() ;
    System.exit(0);   // terminate the program
          }
           return k ;
      }
}
```

Topic 4: Using Exception Classes

Using Exception Class

- There are basically **two types** of exceptions in Java's exception hierarchy:
  1) Checked Exceptions
  2) Unchecked Exceptions

- **Checked Exception**
  - It refers to those exceptions that **can be analysed** by the **compiler**.
  - For example, statements that might cause possible **IOException** when reading a file input data from users.
  - For checked exceptions:
    1. A checked exception **can be caught** within the method that **threw** the exception using the **try/catch** blocks.
    2. However, it is also possible to **delay** the handling of an exception when it is not clear how to handle the exception in the method (using the **throws** clause).

Unchecked Exceptions

- Refers to the exceptions that are **not checked** by the **compiler**.
  - Also refers to exceptions that belong to:
    1) any of the subclasses of class **RuntimeException** or
    2) class **Error**

  - These exceptions can be caused by actions, e.g., pressing a return key without entering any input or entering incorrect data.
    -> These actions might lead to exceptions such as **ArithmeticException** or **IndexOutOfBoundsException**.

  - These exceptions are usually **not easy to be checked explicitly** and are always avoided by programming.

Unchecked Exceptions (Cont`d)

- If **unchecked exceptions** are **not handled** inside the program (left uncaught), they will be handled by **Java's default exception handlers**. The program will terminate with an error message (with the name of the exception class).

- **Guidelines for checked and unchecked exceptions**:
  - If the exception comes from a pre-defined class **RuntimeException** or the class **Error**, it does not necessary need to be caught within the programs (UNCHECKED EXCEPTIONS).
  - Otherwise, the exception must be either caught within the method in which it is thrown using a **catch block**, or declared that the exception might be thrown in the method using a **throws** clause (CHECKED EXCEPTIONS).

Computing Average Marks Version 4

```
import java.util.Scanner ;
public class AverageMarksV4 {
  public static void main( String[] args ){
     double average ;
     try {
       average = computeAvgMarks();
       System.out.println( "Average marks = " + average );
     }
     catch ( ArithmeticException e ) {
       System.out.println( e.getMessage() );
     }
     finally {
       System.out.println( "End of program execution!" );
     }
   }
   // to continue in next page
```

Calling this method must be prepared to handle any exceptions that may be thrown

Throwing Exceptions in Methods

```
public static double computeAvgMarks()
   throws ArithmeticException
{
   int      i , numOfStudents ;
   double   totalMarks = 0 ;
   double   avgMarks   = 0 ;
   Scanner sc          = new Scanner( System.in );

   System.out.print( "Enter number of students: " );
   numOfStudents = sc.nextInt();
   if ( numOfStudents <= 0 )
      throw new ArithmeticException(
            "Error: no of students must not equal to 0!" );
   System.out.print( "Enter student marks: " );
   for ( i = 0 ; i < numOfStudents ; i++ )
      totalMarks += sc.nextDouble() ;
   avgMarks = totalMarks / (double) numOfStudents ;
   return avgMarks ;
}
}
```

Need to declare any exceptions that might be thrown from this method **but is not caught in the method**

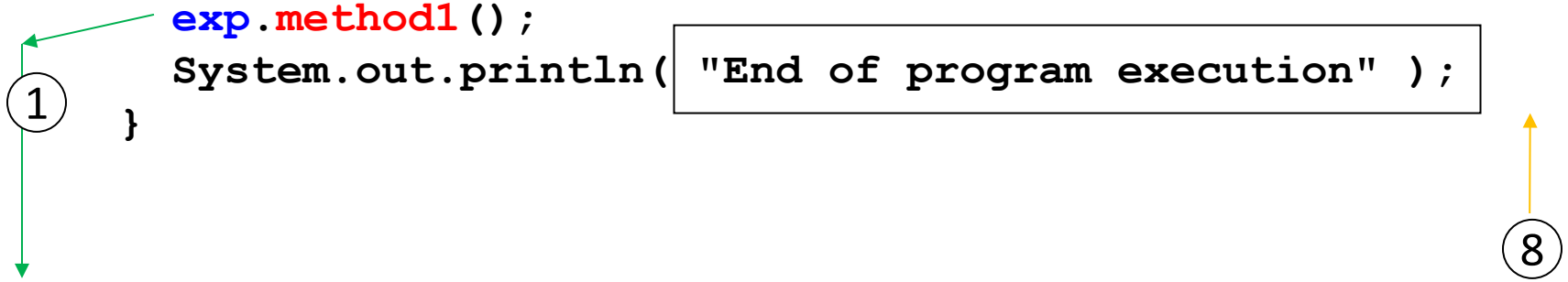
Throwing Exceptions in Methods

```
Program Input and Output
Enter number of students: 5
Enter student marks: 70 80 90 60 50
Average marks = 70
End of program execution!

Enter the number of students: 0
Error: no of students must not equal to 0!
End of program execution!
```

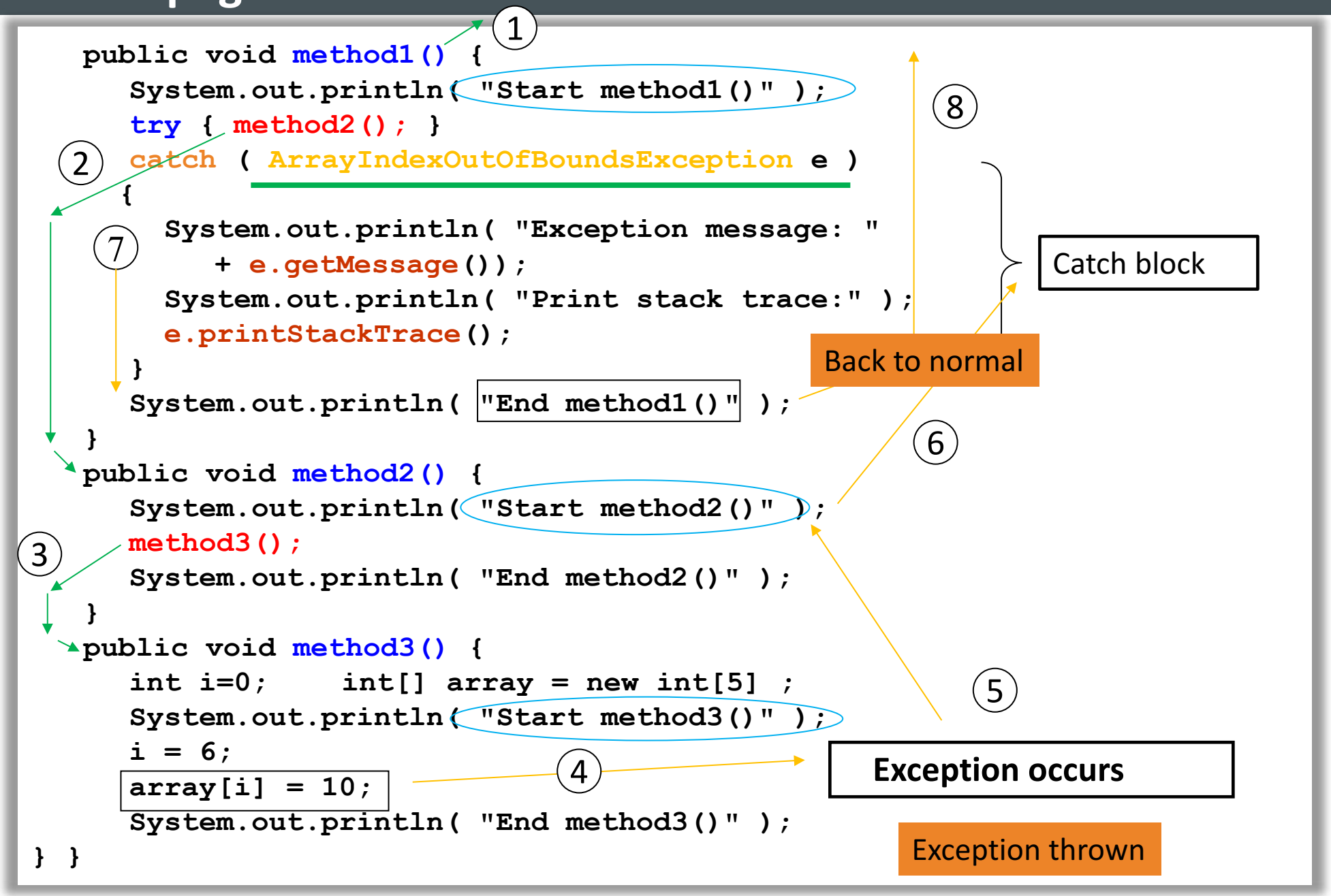
Exception Propagation

- **Exception propagation** - If an exception is thrown and not caught by the handlers after the try block where it occurs, control is then **transferred to the method** that invoked the method that threw the exception.

```
public class ExPropagation {
   public static void main( String[] args ) {
      System.out.println( "Start program execution" );
      ExPropagation exp = new ExPropagation();
      exp.method1();
      System.out.println( "End of program execution" );
   }
```



Exception Propagation



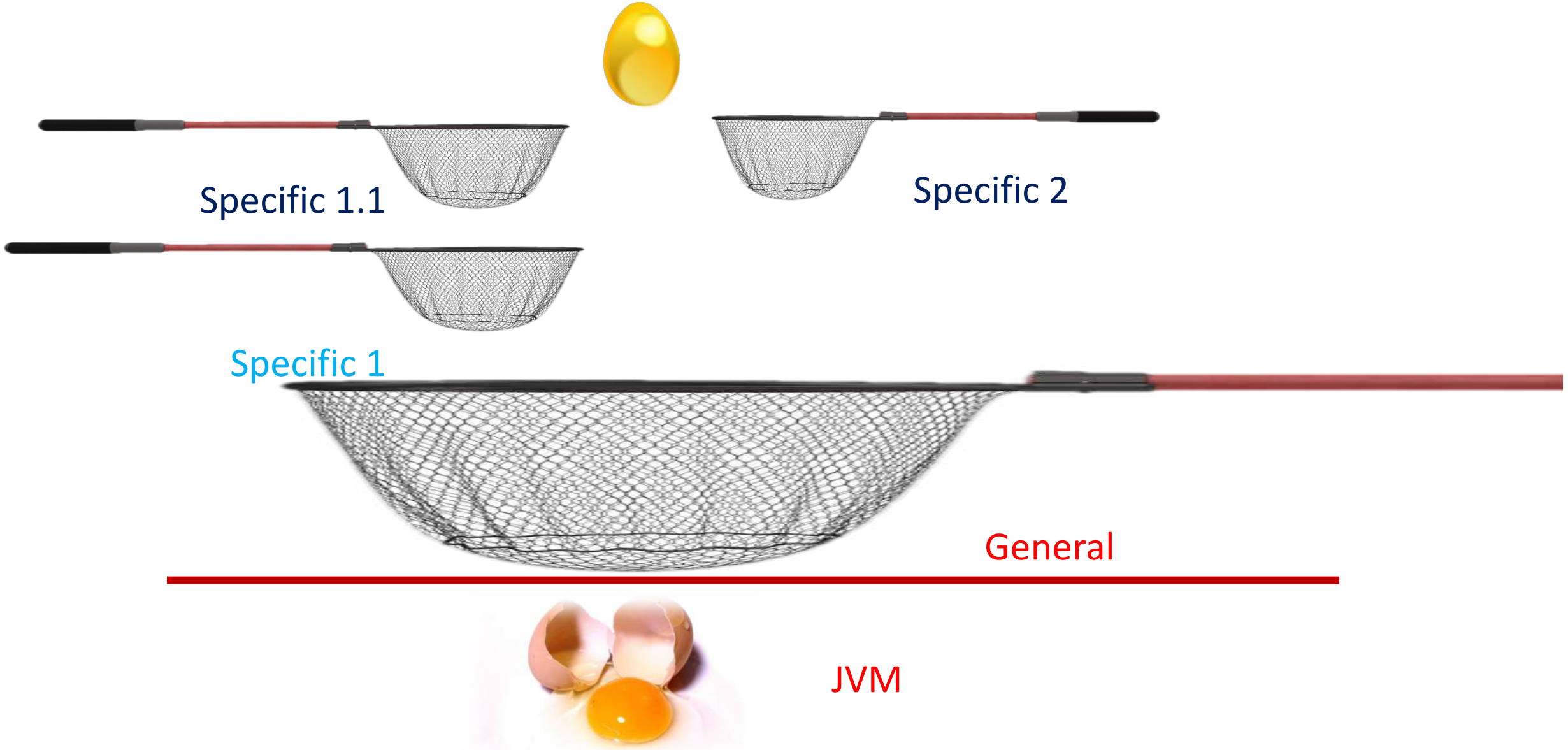
Exception Propagation

```
Program Output
Start program execution
Start method1()
Start method2()
Start method3()
Exception message: 6
Print stack trace:
Java.lang.ArrayIndexOutOfBoundsException: 6
at ExPropagation.method3(ExPropagation.java:31)
at ExPropagation.method2(ExPropagation.java:22)
at ExPropagation.method1(ExPropagation.java:11)
at ExPropagation.main(ExPropagation.java:5)
End method1()
End of program execution
```

Multiple Throws and Catches

- Methods can throw **more than one exception**.
- More than one **catch** block (handler) can be made available for catching exceptions.
- The **catch** blocks immediately following the try block are searched **in sequence** for matching the exception type being thrown previously.
  - Only the **first catch** block that matches the exception type will be executed.
- **Specific** exceptions are derived from classes of more general types (see **class inheritance hierarchy**).
  - Thus, specific exceptions can be caught by both general and specific exception types (IS-A relationship).
- **Recommendation**: First put the catch blocks for the more **specific**, then the derived exceptions, and then the more **general** ones near the end.

Multiple Throws and Catches



Multiple Throws and Catches

```
import java.util.Scanner ;
public class AverageMarksV5 {
   public static void main( String[] args ) {
      double average ;
      try {
         average = computeAvgMarks() ;
         System.out.println( "Average marks = " + average );
      }
      catch ( ArithmeticException e ) {
         System.out.println( e.getMessage() );
         System.exit(0);
      }
      catch ( Exception e ) {
         System.out.println( e.getMessage() );
      }
      finally {  // optional
         System.out.println( "End of program execution!" );
      }
   }
```

More specific one

More general

Multiple Throws and Catches

```
  public static double computeAvgMarks()
     throws ArithmeticException
  {
     int      i , numOfStudents ;
     double   totalMarks = 0 ;
     double   avgMarks   = 0 ;
     Scanner sc          = new Scanner( System.in );
     System.out.print( "Enter number of students: " );
     numOfStudents = sc.nextInt();
     if ( numOfStudents <= 0 )
      throw new ArithmeticException(
         "Error: no of students must not equal to 0!" );
     System.out.print( "Enter student marks: " );
     for ( i = 0 ; i < numOfStudents ; i++ )
       totalMarks += sc.nextDouble() ;
     avgMarks = totalMarks / (double) numOfStudents ;
     return avgMarks ;
  }
}
```

Multiple Throws and Catches

```
Program Output
Enter number of students: 5
Enter student marks: 70 80 90 60 50
Average marks = 70
End of program execution!

Enter number of students: 0
Error: no of students must not equal to 0!
```

Topic 5: Creating Your Own Exception Classes

Creating Your Own Exception Classes

```
public class IntNonNegativeException extends Exception
{
  // constructors
  public IntNonNegativeException() {
    super( "Integer input is a negative number!! " );
  }

  public IntNonNegativeException( String message ) {
    super( message );
  }
}
```

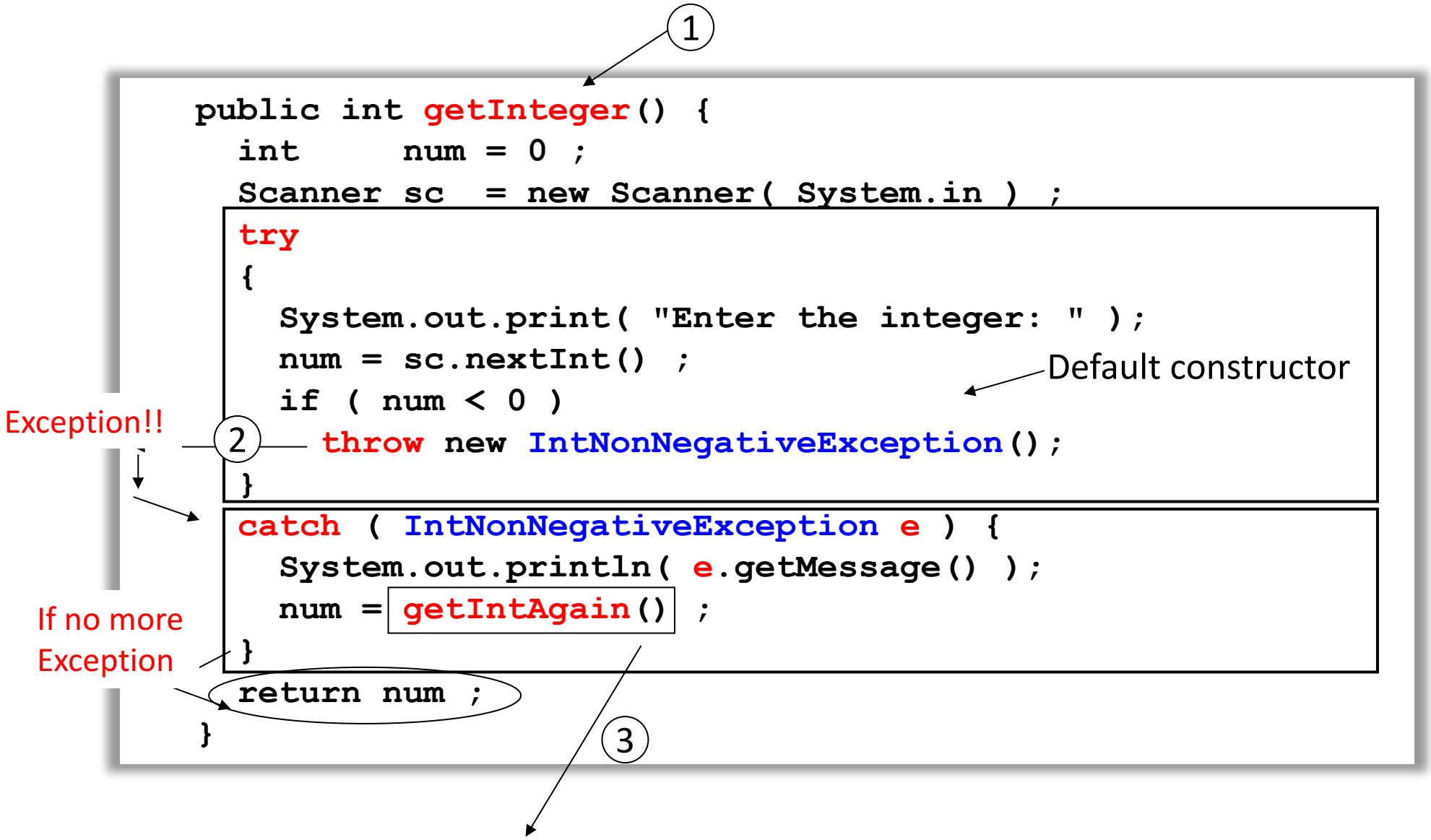
with parameter

- You can define **your own exception class** and use it in your throw statement (in your own code).
- Must be derived (inherited) from an existing **exception** class.
- Usually, we define **only** the **constructors** and call the default constructor using **super**.

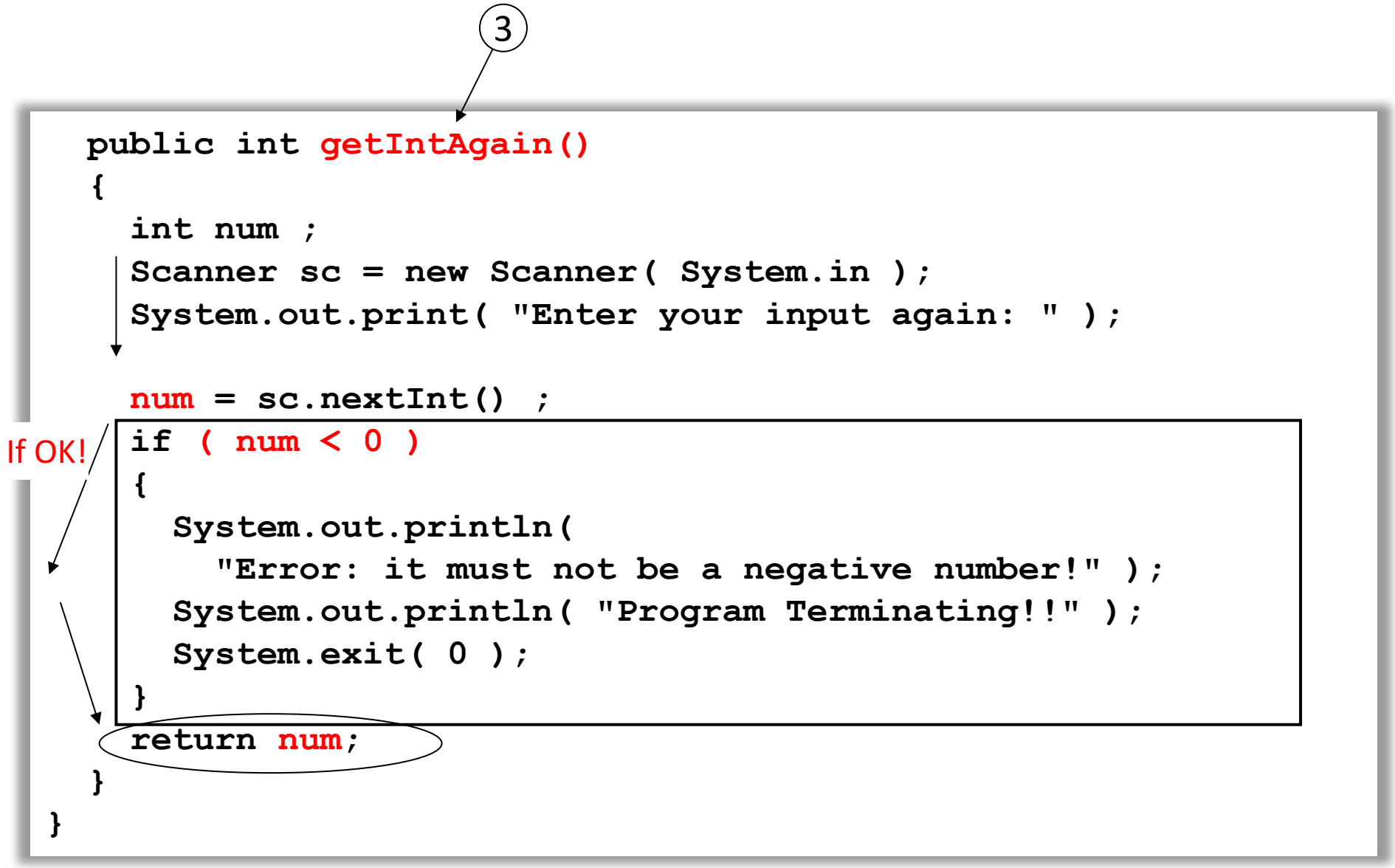
Example

```
import java.util.Scanner ;
public class IntNonNegativeExceptionApp
{
  public static void main( String[] args ) {
    IntNonNegativeExceptionApp sumEx
                = new IntNonNegativeExceptionApp() ;
    int      inputNum ;
    int      sum = 0 ;
    Scanner sc   = new Scanner( System.in );

    System.out.print( "Enter total no. of integers: " );
    int total = sc.nextInt();

    for ( int i = 0 ; i < total ; i++ ) {
      inputNum = sumEx.getInteger();
      sum += inputNum ;
    }
    System.out.println( "The sum of integers: " + sum );
  }
```

Example


1
public int getInteger() {
int num = 0 ;
Scanner sc = new Scanner(System.in) ;
try
{
System.out.print("Enter the integer: ");
num = sc.nextInt() ;
if (num < 0)
throw new IntNonNegativeException();
}
catch (IntNonNegativeException e) {
System.out.println(e.getMessage());
num = getIntAgain() ;
}
return num ;
}
Default constructor
Exception!!
2
If no more Exception
3

Example


3
public int getIntAgain()
{
int num ;
Scanner sc = new Scanner(System.in);
System.out.print("Enter your input again: ");
num = sc.nextInt() ;
If OK!
if (num < 0)
{
System.out.println(
"Error: it must not be a negative number!");
System.out.println("Program Terminating!!");
System.exit(0);
}
return num;
}
}

Example

```
Program Output
Enter total no. of input integers: 5
Enter the integer: 1
Enter the integer: 2
Enter the integer: 3
Enter the integer: 4
Enter the integer: 5
The sum of integers: 15

Enter total no. of input integers: 5
Enter the integer: 1
Enter the integer: 2
Enter the integer: 3
Enter the integer: -2
Integer input is a negative number!!
Enter your input again: 4
Enter the integer: 5
The sum of integers: 15
```

Exception occurs

Example

```
Program Output

Enter total no. of input integers: 4
Enter the integer: 1
Enter the integer: -2
Integer input is a negative number!!
Enter your input again: -4
Error: it must not be a negative number!
Program Terminating!!
```

Summary

Key points from this chapter:

- Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.

- The core advantage of exception handling is to maintain the normal flow of the application. An exception normally disrupts the normal flow of the application and that is why we use exception handling.

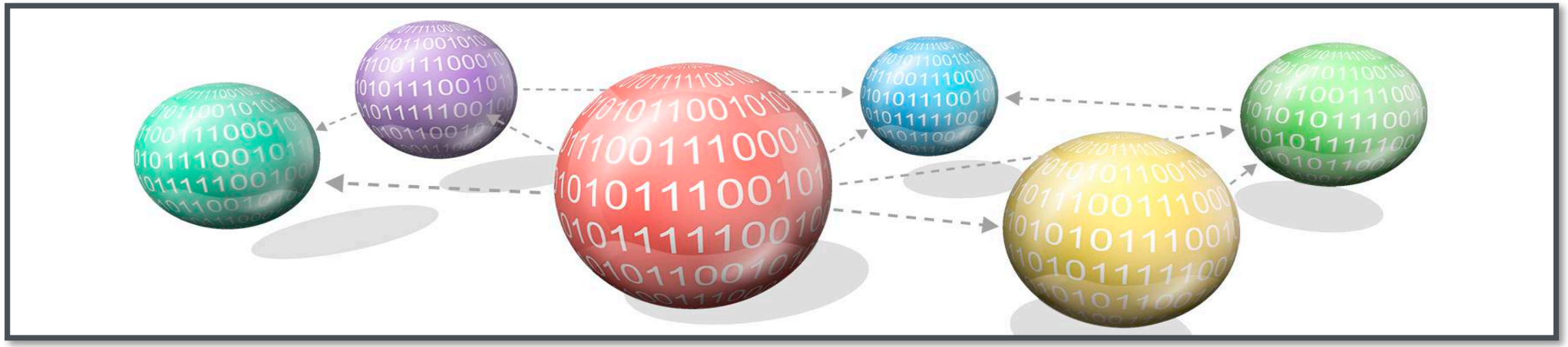
- Suppose there are 10 statements in your program and an exception occurs at statement 5, rest of the code will not be executed, i.e. statement 6 to 10 will not run. However, if we perform exception handling, rest of the statement will be executed.

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Chapter 5: Polymorphism

CE2002 Object Oriented Design & Programming

Dr Zhang Jie
Assoc Prof, School of Computer Science and Engineering (SCSE)



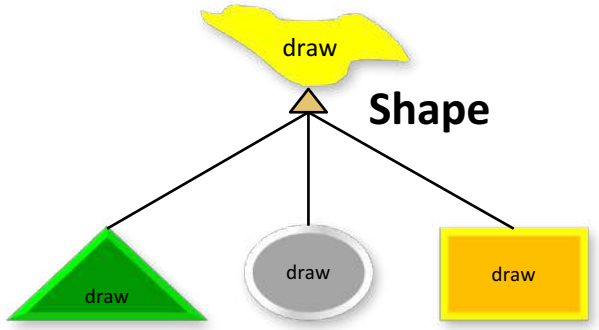
Learning Objectives

After the completion of this chapter, you should be able to:

- Explain polymorphism.
- Explain the concept of binding.
- Describe object typecasting.
- List the benefits of polymorphism.
- Explain the three ways of method overriding.

Topic 1: Polymorphism

Polymorphism



- Polymorphism means “many forms” (in Greek).
- In OOP, it means the ability of an object reference being referred to different types; knowing which method to apply depends on where it is in the inheritance hierarchy.
- Overriding - a necessary tool for polymorphism.
- A subclass can override a method in the parent class by defining a method with exactly the same signature and return type.
- The subclass method can replace or refine the method in the parent class.

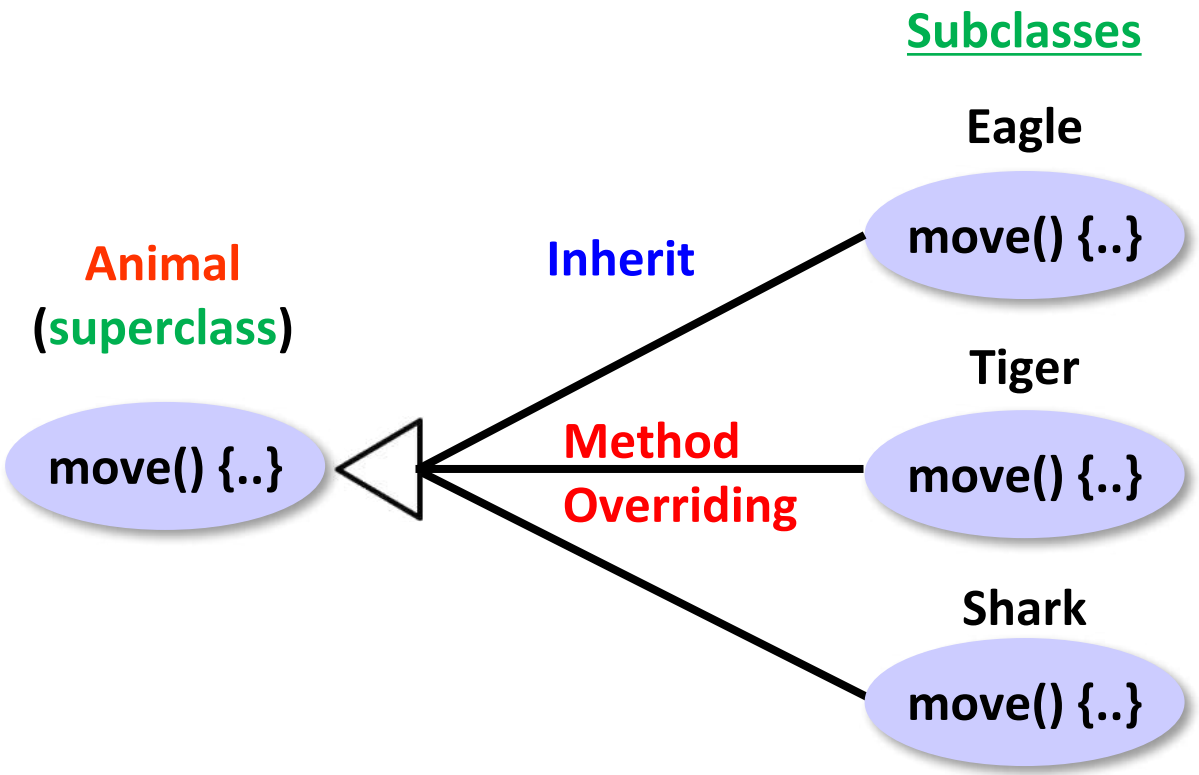
Example

```
public class Person{
   int age ;
   String name;
// usual constructor, accessor mutator methods
…..
   public void printInfo() {
       System.out.println("Name is " + name);
       System.out.println("Age is " + age);
   }
   ……..
}
public class Employee extends Person{
   double salary ;
// usual constructor, accessor mutator methods
…….
   public void printInfo() {  // refine printInfo method
      System.out.println("Employee name is :" + getName()+
                         " and age is " + age );
      System.out.println("Salary is " + salary) ;
   }
}                  // replace printInfo method
```

method overriding

we can use super.printInfo(); (also overriding)

Polymorphism




```
Animal animal = new Eagle();
animal.move();

animal = new Tiger();
animal.move();

animal = new Shark();
animal.move();
```

Examples of Polymorphism

- Polymorphism:
  - When a program invokes a method through a superclass variable, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable.
    object
  - The same method name and signature can cause different actions to occur, depending on the type of object on which the method is invoked.
  - Facilitates adding new classes to a system with minimal modifications to the system's code.

```
void  call (Animal animal)
{
        animal.move();
}
```

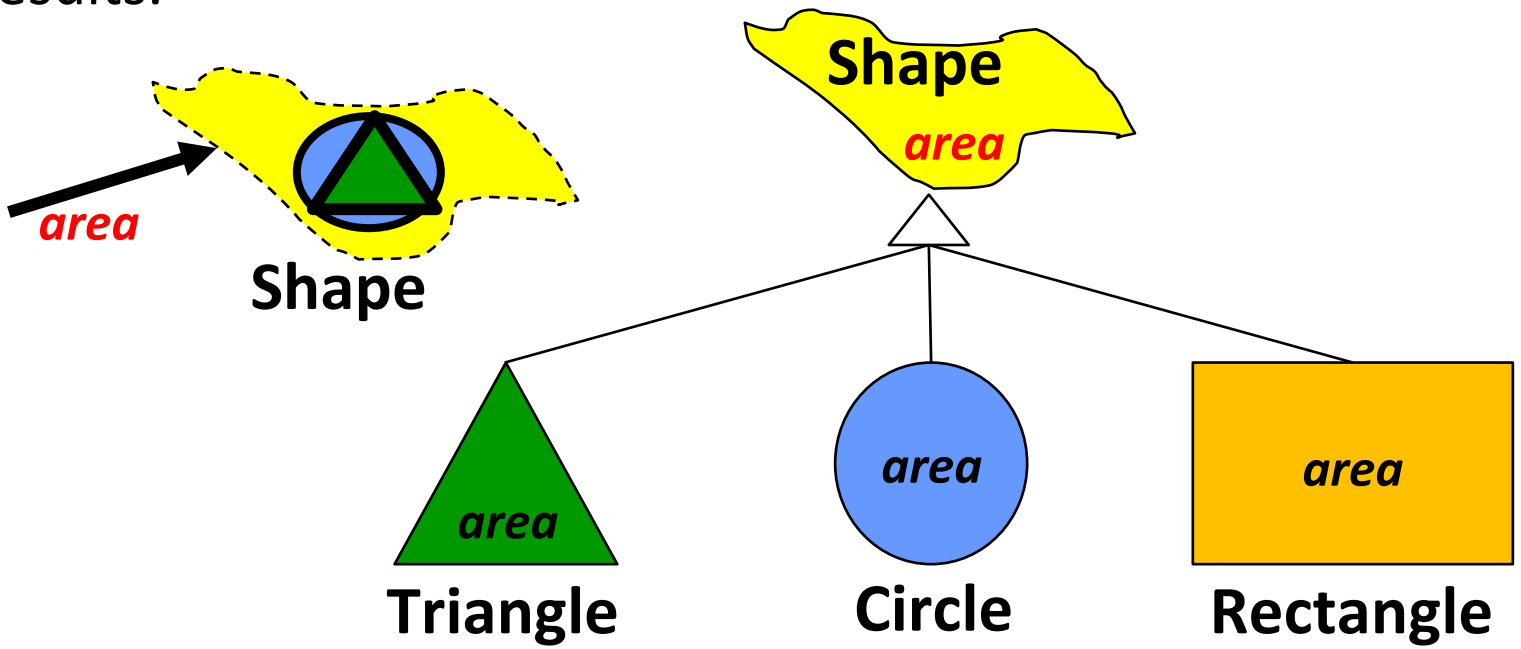
```
call(new Eagle());
animal = new Eagle();

call(new Tiger());
animal = new Tiger();

call(new Shark());
animal = new Shark();
```

↳ it doesn't effect subclasses only superclasses

Examples of Polymorphism

- For example, given a base class *shape,* polymorphism enables the programmer to define different *area* methods for any number of derived classes, such as *circles, rectangles* and *triangles.*

- No matter what shape an object is, applying the area method to it will return the correct results.



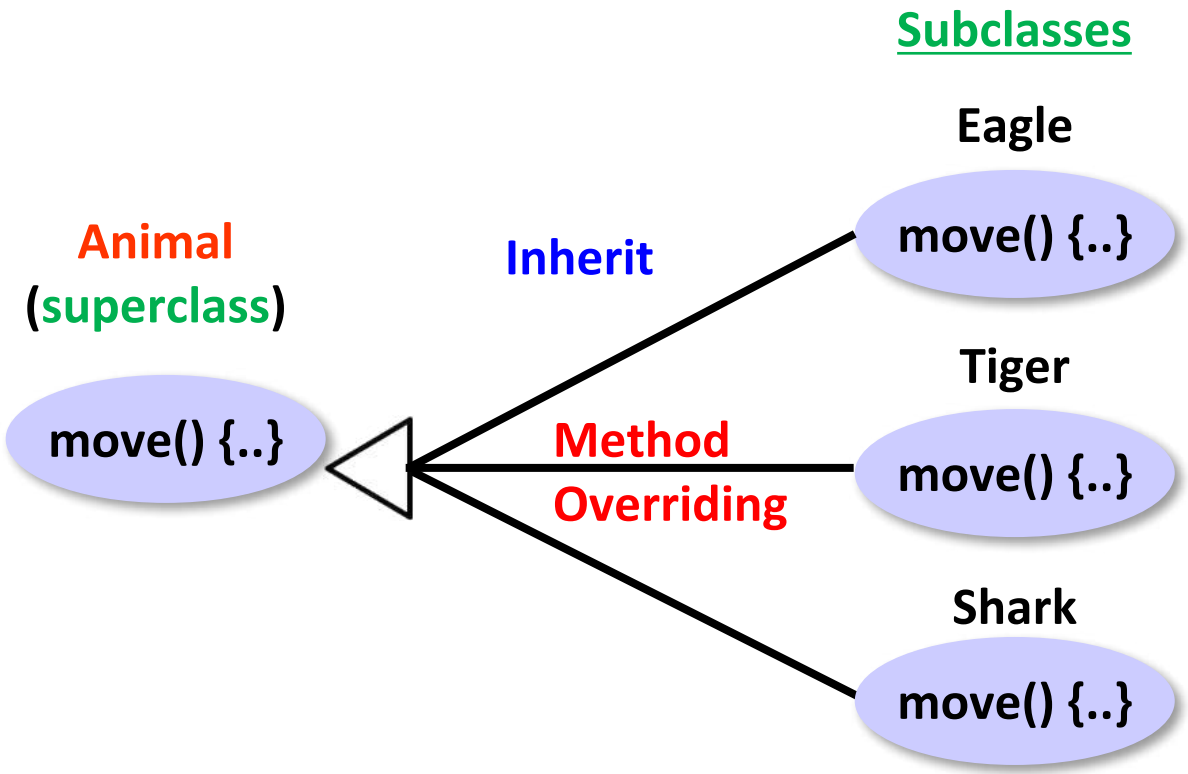
The Liskov Substitution Principle

- Barbara Liskov first wrote in 1988:

  "If for each object o_1 of type **S** there is an object o_2 of type **T** such that for all programs **P** defined in terms of **T**, the behavior of **P** is unchanged when o_1 is substituted for o_2 then **S** is a subtype of **T**."

- Paraphrased
  - Subtypes must be substitutable for their base types.

Polymorphism




```
void  call (Animal animal)
{
        animal.move();
}
```

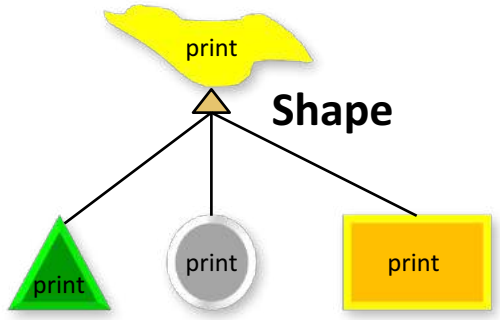
```
call(new Eagle());
animal = new Eagle();

call(new Tiger());
animal = new Tiger();

call(new Shark());
animal = new Shark();
```

Topic 2: Binding

Binding

- Binding - Refers which method to be called at a given time [i.e. connecting a **method call** to a ***method body***],
  circle.print();

  print() { …//implementation……}



- Static binding (early binding)
  - It occurs when the method call is "bound" at compile time

Shape circle = new Circle();

- Dynamic binding (late binding)
  - The selection of method body to be executed is delayed until execution time (based on the **actual object** referred by the reference) - (*execution time = runtime*)

Polymorphism uses this

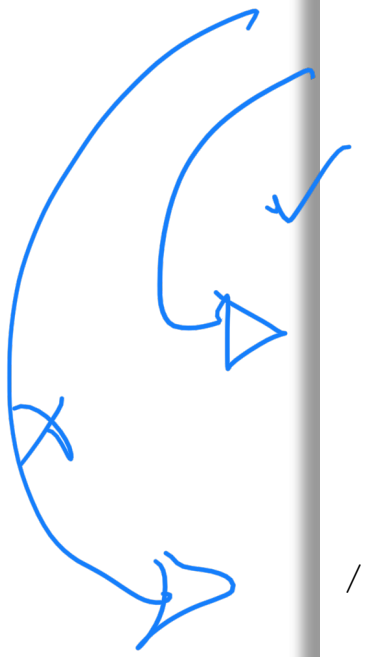
Binding

- Java uses dynamic binding (by default) which enables the actual type of the object (object type), not the declared type (reference type), to govern which method to use.

Class_Name Object_Variable_Name = new *Class_Name*()
Interface_Name Object_Variable_Name = new *Class_Name*()
<reference type> <reference name> = new <Object Type> ()
<declared type> <object reference>
<variable type>

Example: Object anObj = new Person()

Example of Static Binding



```
public class StaticBindingTest {

    public static void main(String args[])  {
        Collection c = new HashSet();
        StaticBindingTest et = new StaticBindingTest();
        et.sort(c);

    }

    //overloaded method takes Collection argument
    public Collection sort(Collection c){
        System.out.println("Inside Collection sort method");
        return c;
    }


//another overloaded method which takes HashSet argument which is sub class
    public Collection sort(HashSet hs){
        System.out.println("Inside HashSet sort method");
        return hs;
    }

}
```

Example of Dynamic Binding

```
public class DynamicBindingTest {

    public static void main(String args[]) {
        Vehicle vehicle = new Car(); // here Type is vehicle but object will
                    // be Car
        vehicle.start();  //Car's start called because start() is overridden
             // method
    }
}
```

```
class Vehicle {
    public void start() {
        System.out.println("Inside start method of Vehicle");
    }
}
```

```
class Car extends Vehicle {
    // Override
    public void start() {
        System.out.println("Inside start method of Car");
    }
}
```

```
class Bus extends Vehicle {
    // Override
    public void start() {
        System.out.println("Inside start method of Bus");
    }
}
```

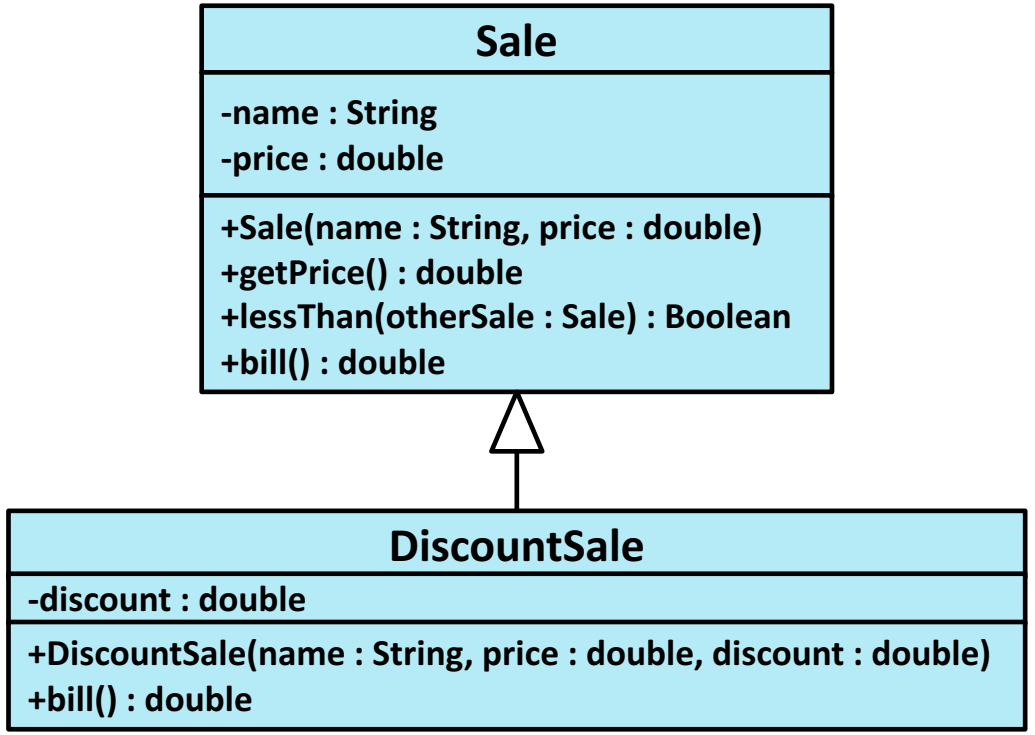
```
Output: Inside start method of Car
```

Dynamic Binding

- Java uses dynamic binding for all methods (except private, final, and static methods).
- Because of dynamic binding, a method can be written in a base class to perform a task, even if portions of that task aren't yet defined.

*'Program in the **general**' rather than 'program in **specific.**'*

- For an example, the relationship between a base class called *Sale* and its subclass *DiscountSale* will be examined.



Dynamic Binding

```
public class Sale {
……
public double bill( ){
   return price;
}
public boolean lessThan (Sale otherSale){
  if (otherSale == null)
  {
    System.out.println("Error: null
                        object");
    System.exit(0);
  }
  return (bill( ) < otherSale.bill( ));
}
}
```

```
public class DiscountSale extends Sale {
    …….
    public double bill( ) {
        ……???
    }
}
```

```
Sale simple = new sale("floor mat", 10.00);
DiscountSale discount = new DiscountSale("floor mat", 11.00, 10);
  . . .
if (!simple.lessThan(discount))
  System.out.println("code can still be compiled!");
```

Topic 3: Object Typecasting

Object Typecasting

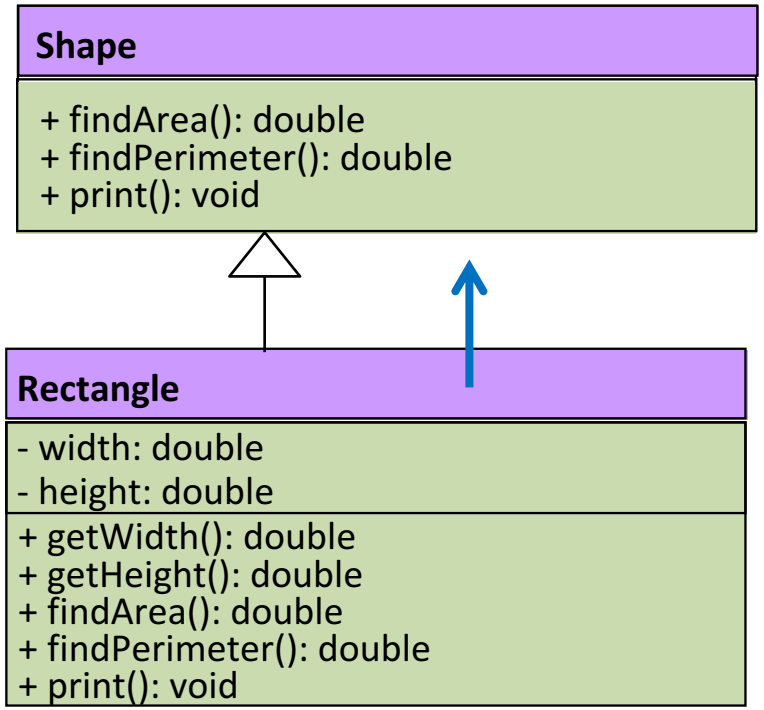
- **Upcasting** is when an object of a derived class is assigned to a variable of a base class (or any ancestor class).

```
Shape shape; //Base class
Rectangle rect= new Rectangle(15,10); //Derived class
shape = rect ; //Upcasting
```

```
System.out.println(shape.findArea());
```

- Because of dynamic binding, findArea() above uses the definition given in the Rectangle class.

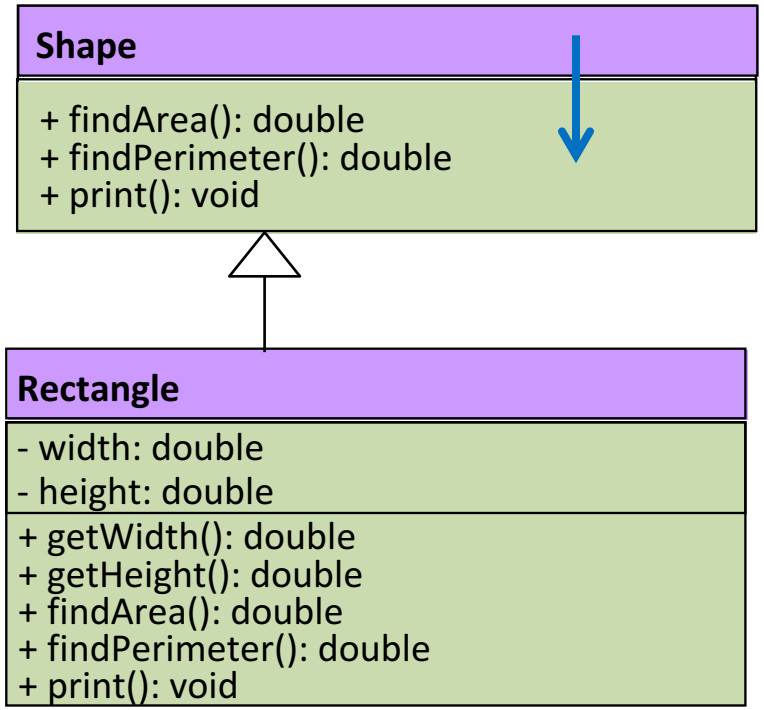
```
Shape shape = new Rectangle(15,10));
```



Upcasting and Downcasting

- **Downcasting** is when a type cast is performed from a base class to a derived class (or from any ancestor class to any descendent class).
  - Downcasting has to be done very carefully
  - In many cases it doesn't make sense, or is illegal:

```
Shape shape = new Shape() ;

Rectangle rect = shape ;
// will have compilation error

Rectangle  rect = (Rectangle)shape ;
// explicit cast, accept by compiler
//will produce runtime error !!!!

int height = rect.getHeight();
// compiler OK
```



Upcasting and Downcasting

- It is the responsibility of the programmer to use downcasting only in situations where it makes sense.
- The compiler does not check to see if downcasting is a reasonable thing to do.
- Using downcasting in a situation that does not make sense usually results in a run-time error.

Checking to See if Downcasting is Legitimate

- Downcasting to a specific type is only sensible if the object being cast is an instance of that type

```
Shape shape = new Rectangle();
Rectangle rect = (Rectangle) shape;
```

- This is exactly what the instanceof operator tests for:

```
object instanceof ClassName
```

- It will return true if *object* is of type *ClassName*

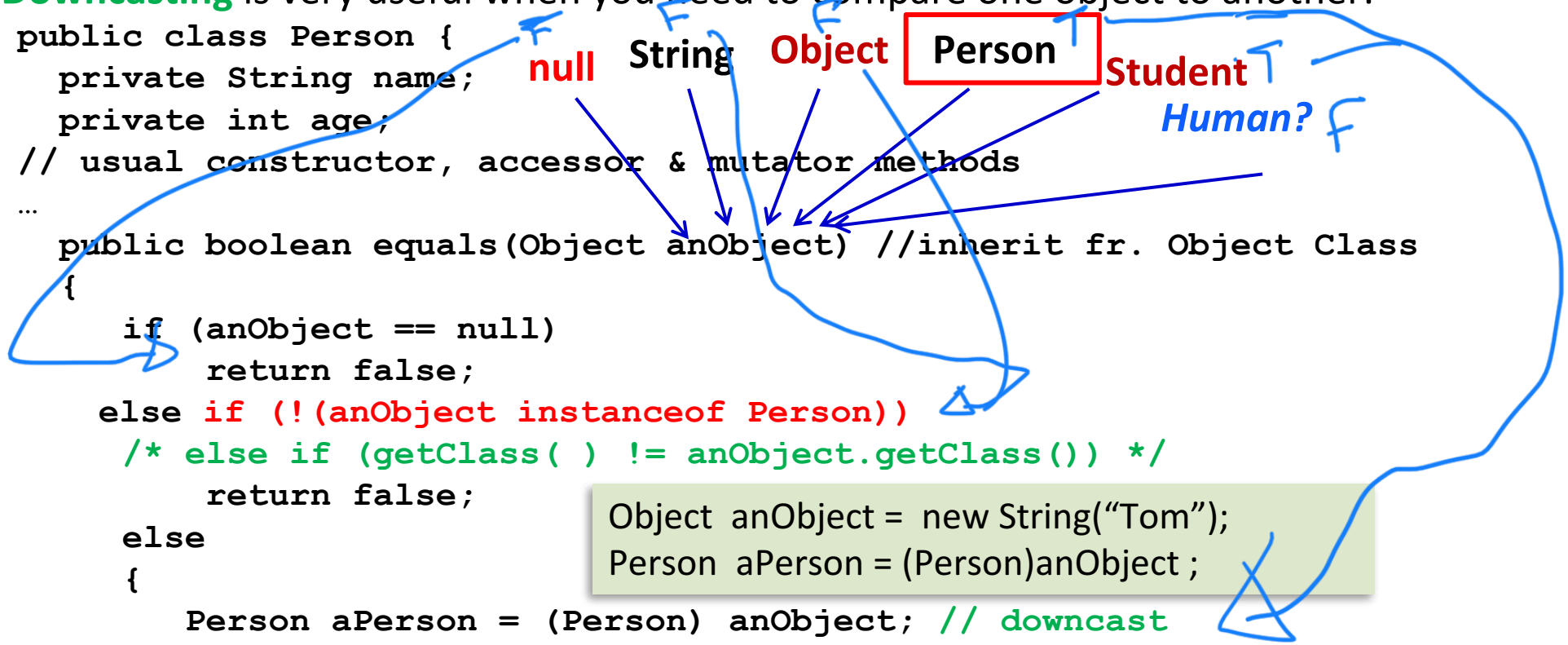
- In particular, it will return true if *object* is an instance of any **descendent** class of *ClassName*

```
personObj instanceof Object
```

Use of Downcasting

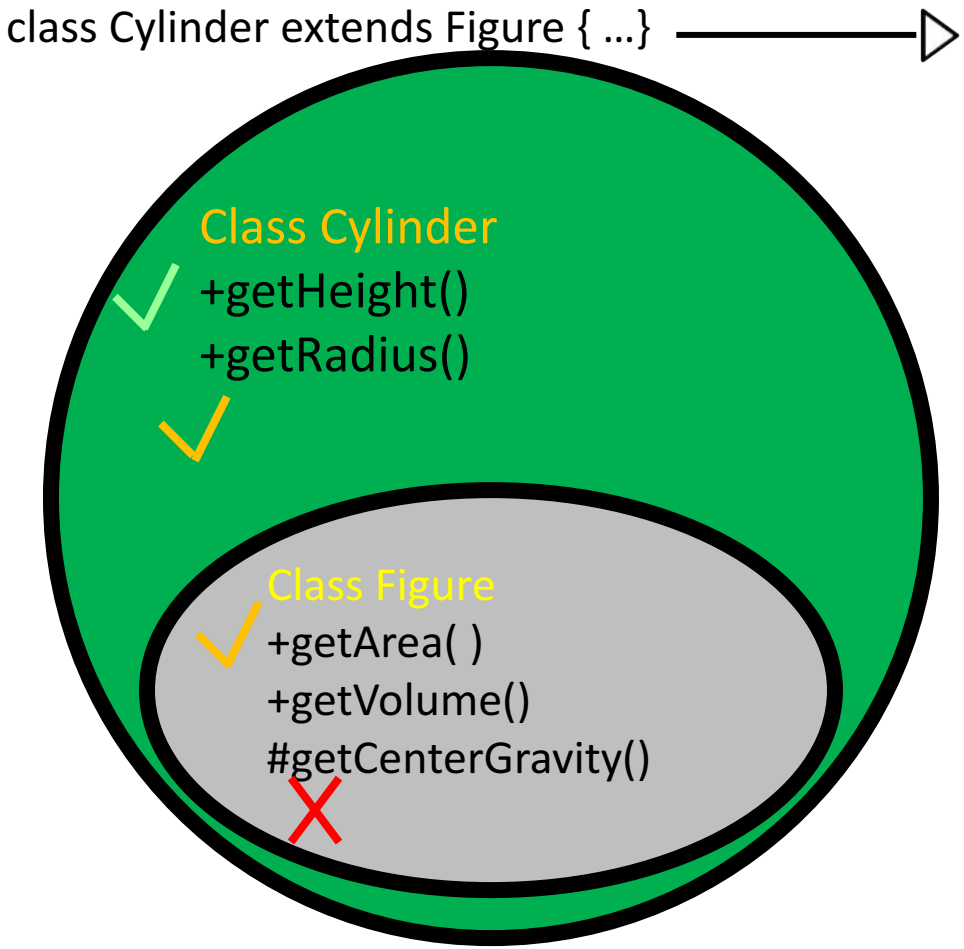
- **Downcasting** is very useful when you need to compare one object to another:

```
public class Person {
   private String name;
   private int age;
// usual constructor, accessor & mutator methods
…
   public boolean equals(Object anObject) //inherit fr. Object Class
   {
      if (anObject == null)
          return false;
     else if (!(anObject instanceof Person))
      /* else if (getClass( ) != anObject.getClass()) */
          return false;
      else
      {
         Person aPerson = (Person) anObject; // downcast
         return ( name.equals(aPerson.getName())
                  && (age == aPerson.getAge()));
      }
   }
}
```




```
Object  anObject =  new String("Tom");
Person  aPerson = (Person)anObject ;
```

Use of Downcasting



Upcast

```
Figure fig = new Cylinder() ;
fig.getArea() ;
// compile & runtime both OK
```

Downcast

```
Figure fig = new Figure()
Cylinder cy = fig;  // compile NOK

Cylinder cy = (Cylinder) fig;
// explicit cast, compile OK,
// runtime NOK
cy.getHeight() ; // compile OK
```

<reference type> <reference name> = new <Object Type> ()

use at compile time to check

use at runtime to check

Allowed Assignments Between Superclass and Subclass Variables

- Superclass and subclass assignment rules
  - Assigning a superclass reference to a superclass variable is straightforward **Object ob = new Object()**
  - Assigning a subclass reference to a subclass variable is straightforward **String st = new String()**
  - Assigning a subclass reference to a superclass variable is safe because of the *is-a* relationship **Object st = new String()**
    - Referring to subclass-only members through superclass variables is a compilation error **st.charAt(0)**
  - Assigning a superclass reference to a subclass variable is a compilation error **String ob = new Object()**
    - Explicit downcasting can get around this error **String ob = (String)new Object()**

Topic 4: Benefits of Polymorphism

Benefits of Polymorphism

- Simplicity
  - If you need to write code that deals with a family of types, the code can ignore type-specific details and just interact with the base type of the family.
  - Even though the code thinks it is using an object of the base class, the object's class could actually be the base class or any one of its subclasses.
  - This makes your code easier for you to write and easier for others to understand.

Benefits of Polymorphism

- Extensibility
  - With polymorphism, programs become easily extensible.
  - We can add new functionality by creating new classes (or data types) inherited from an off-the-shelf base class without modifying the base class and the other classes derived from the base class.

Benefits of Polymorphism

```
public abstract class Shape {
        public abstract double area() ;
}
```

```
public class Rectangle extends Shape {
        private double width, height;
        public Rectangle(double w, double h)  { width = w ; height = h ; }
        public double area()  { return width * height ; }
}
```

```
public class Circle extends Shape {
        private double radius;
        public Rectangle(double r)  { radius = r ;  }
        public double area()  { return  Math.PI * radius * radius ; }
}
```

```
public class ShapeApp {
       public static void main(String[ ] args) {
shapes[1] = new Circle(8) ;
              for (Shape s : shapes) { System.out.println("Area  = " + s.area() ; }
      ArrayList<Circle> shapes = new ArrayList<Circle>() ;
      shapes.add(new Circle(0.1));
      shapes.add(new Circle(0.8));
```

Topic 5: Three Ways of Method Overriding

Three Ways of Method Overriding

- Method overriding: Methods of a subclass override the methods of a superclass.
- Method overriding (implementation) of the abstract methods: Methods of a subclass implement the abstract methods of an abstract class.
- Method overriding (implementation) through the Java interface: Methods of a concrete class implement the methods of the interface.

Topic 6: More Examples

Example

```
interface IAnimal {
        void move();
        void speak();
}
class Cat implements IAnimal{
    public void move() {
        System.out.println("Cat
        moves....");
  }
    public void speak() {
        System.out.println("Meow !"); }
}
class Lion implements IAnimal {
    public void move() {
        System.out.println("Lion
        moves...");}
    public void speak() {
        System.out.println("ROAR !"); }
}
```

```
class CareTaker
{
  public void takeAWalk(IAnimal pet)
    {       pet.move();
            pet.speak();
     }

}

class  AnyClass {
…….
    CareTaker ct = new CareTaker();
        Cat cat = new Cat();
        Lion lion = new Lion();

        ct.takeAWalk(cat);
        ct.takeAWalk(lion);
……..
}
```

Example

```
interface IAnimal {
        void move();
        void spea
}
class Cat implem
    public void m
        System.o
        moves....'
  }
    public void sp
        System.out.println("Meow !"); }
}
class Lion implements IAnimal {
```

Method Overloading.
Which method to be called is known during compile time
=>
Static Binding

Dog implements IAnimal
Elephant implements IAnimal
Tiger implements IAnimal
.....

```
class CareTaker
{
public void takeAWalk (Cat cat)  {....}
public void takeAWalk (Lion lion) {.....}
public void takeAWalk (.....) {.....}

}

class  AnyClass {
..............
    CareTaker ct = new
        Cat cat = new
        Lion lion = new

        ct.takeAWalk(
        ct.takeAWalk(
...............
}
```

Only during runtime, will the object implementing IAnimal be known
=>
Dynamic Binding

Example

```
abstract class LivingThings {
        public  void grow( ) {
                System.out.println("LivingThings
                grow!"); }
       public abstract void speak( );
  }
  abstract class Mammal extends LivingThings {
      public void move( ) {
                System.out.println("Mammal move!"); }
        static
      public void grow()  {
                System.out.println("Mammal  grow!"); }

      public void eat()  {
                System.out.println("Mammal eat!"); }
      }
   class Dog extends Mammal {  // concrete class
      public void speak( ) {
                System.out.println("Woof!"); }
      public  void move( ) {
      static System.out.println("Dog move!"); }
      public void grow( ) {
                System.out.println("Dog grow!"); }
   }
```

```
class House {
    public static void main(
            String[] args) {
        Dog mDog = new Dog( );
        mDog.speak( );
        mDog.move( );
        mDog.eat();
        mDog.grow();

        Mammal m = new Dog( );
        m.speak( );
        m.move( );
        m.eat();
        m.grow();


    }
}

}
```

Example

```
abstract class LivingThings {
        public  void grow( ) {
                System.out.println("LivingThings
                grow!"); }
       public abstract void speak( );
  }
  abstract class Mammal extends LivingThings {
      public void move( ) {
                System.out.println("Mammal move!"); }
        static
      public void grow()  {
                System.out.println("Mamm
      public void eat()  {
                System.out.println("Mammal eat!"); }
      }
   class Dog extends Mammal {  // concrete class
      public void speak( ) {
                System.out.println("Woof!"); }
      public  void move( ) {
      static    System.out.println("Dog move!"); }
      public void grow( ) {
                System.out.println("Dog grow!"); }
   }
```

```
class House {
    public static void main(
          String[] args) {
       Dog mDog = new Dog( );
       mDog.speak( );
                                  = new Dog( );
       m.eat();
       m.grow();
```

abstract, subclass must 'implement'

Mammal did not implement abstract method speak so has to be abstract

Dog inherit eat method
And overrides move & grow methods
And implements abstract method speak

Program Output

```
Woof!
Dog moves!
Mammal eats!
Dog grows!

Woof!
Dog moves!
Mammal eats!
Dog grows!
```

Summary

Key points from this chapter:

- Polymorphism means “many forms” (in Greek). In OOP it means the ability of an object reference being referred to different types; knowing which method to apply depends on where it is in the inheritance hierarchy.
- The Liskov Substitution Principle states that subtypes must be substitutable for their base types.
- The benefits of polymorphism is simplicity and extensibility.
- It is the responsibility of the programmer to use downcasting only in situations where it makes sense; otherwise results in a run-time error.

Example

```
01. package relationsdemo;
02. public class Bike
03. {
04.     private String color;
05.     private int maxSpeed;
06.     public void bikeInfo()
07.     {
08.         System.out.println("Bike Color= "+color + " Max Speed= " + maxSpeed);
09.     }
10.     public void setColor(String color)
11.     {
12.         this.color = color;
13.     }
14.     public void setMaxSpeed(int maxSpeed)
15.     {
16.         this.maxSpeed = maxSpeed;
17.     }
18. }
```

In the code above the Bike class has a few instance variables and methods.

```
01. package relationsdemo;
02. public class Pulsar extends Bike
03. {
04.     public void PulsarStartDemo()
05.     {
06.         Engine PulsarEngine = new Engine();
07.         PulsarEngine.stop();
08.     }
09. }
```

Pulsar is a type of bike that extends the Bike class that shows that Pulsar is a Bike. Pulsar also uses an Engine's method, stop, using composition. So it shows that a Pulsar has an Engine.

```
01. package relationsdemo;
02. public class Engine
03. {
04.     public void start()
05.     {
06.         System.out.println("Started:");
07.     }
08.     public void stop()
09.     {
10.         System.out.println("Stopped:");
11.     }
12. }
```

The Engine class has the two methods start( ) and stop( ) that are used by the Pulsar class.

```
01. package relationsdemo;
02. public class Demo
03. {
04.     public static void main(String[] args)
05.     {
06.         Pulsar myPulsar = new Pulsar();
07.         myPulsar.setColor("BLACK");
08.         myPulsar.setMaxSpeed(136);
09.         myPulsar.bikeInfo();
10.         myPulsar.PulsarStartDemo();
11.     }
```