Introduction to Computational Thinking

Polymorphism;
Event-Driven Programming

Qiao Xiang, Qingyu Song

https://sngroup.org.cn/courses/ct-xmuf25/index.shtml

12/24/2025

This deck of slides are heavily based on cs112 at Yale University and cs101 at UCAS, respectively, by courtesy of Dr. Y. Richard Yang and Dr. Zhiwei Xu.

Final Exam

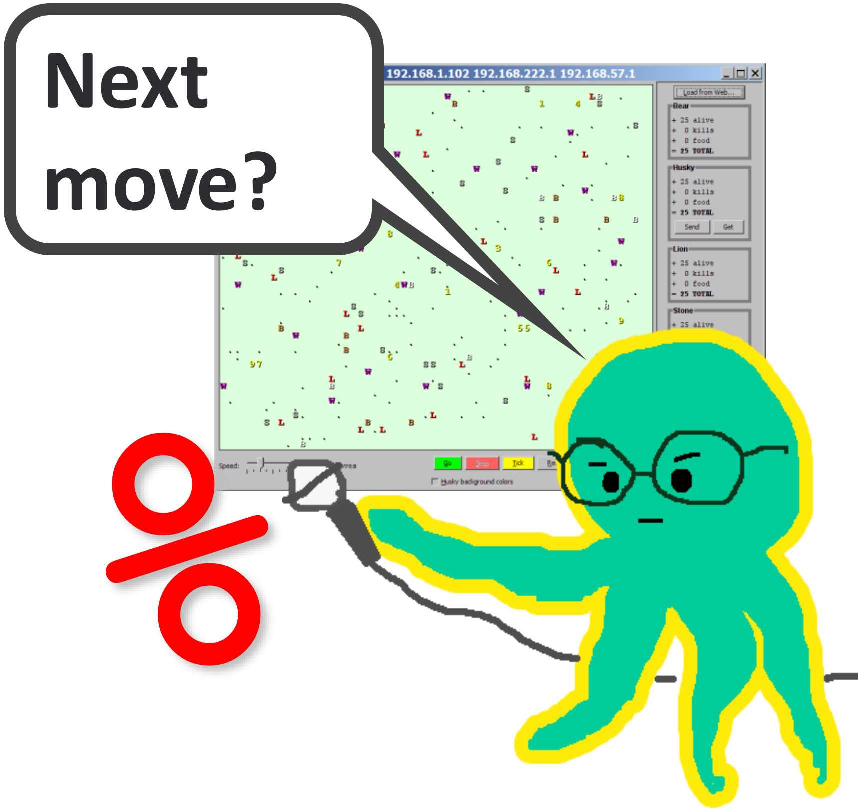
- One double-sided A4 page cheating sheet
- Date, location: 10:30 AM - 12:30 PM, Dec 30, 2025, Xuewu Building 1, A206
- Coverage: https://sngroup.org.cn/courses/ct-xmuf25/exam-coverage.html

Outline

- □ Critters and objects coordination
- □ Polymorphism 多态

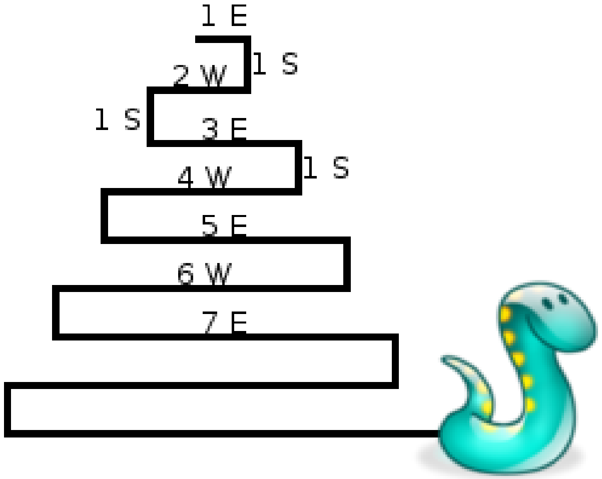
Recap: Critters and Event-Driven Programming

- Key concepts:
  - The simulator is in control, NOT an animal.
    - An animal must keep state (as fields) so that it can make a single move, and know what moves to make later.
    - We say that event-driven programming (EDP) focuses on writing the callback functions of objects

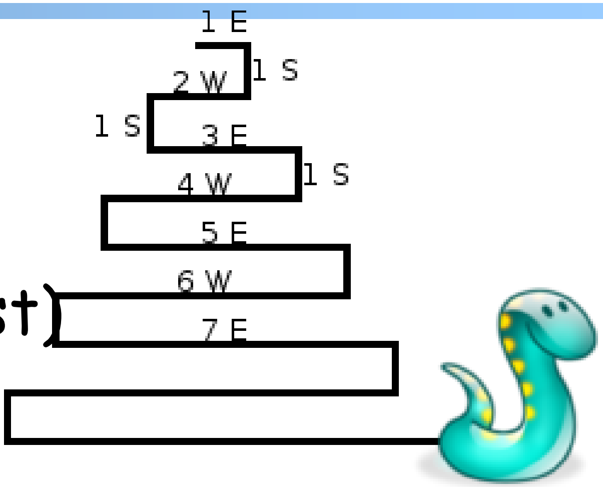


Recap: Critter : `Snake`

Method	Behavior
constructor	`public Snake()`
`eat`	Never eats
`fight`	random pounce(猛扑) or roar
`getColor`	Color(20, 50, 128)
`getMove`	1 E, 1 S; **2** W, 1 S; **3** E, 1 S; **4** W, 1 S; **5** E, ...
`toString`	`"S"`



Recap: EDP for getMove



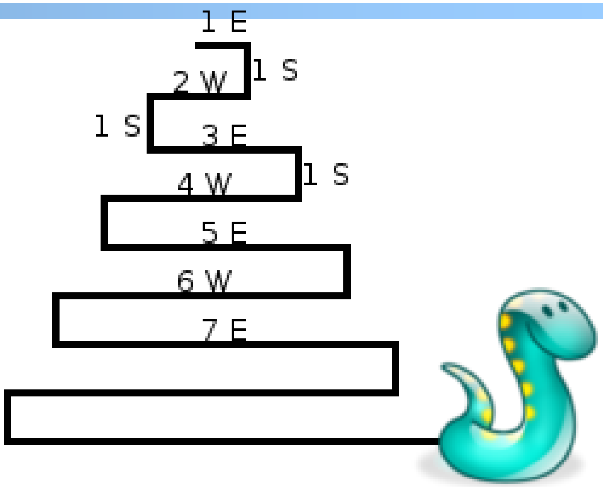
- Variables that determine the state for getMove?
  - Length of current cycle (east-west)
  - Number of moves made in current cycle
- What is the initial state?
  - cycleLength = 1
  - steps = 0

Recap: Non-EDP Version

A non-event driven version

```
cycleLength = 1; steps = 0;
do {
    while (steps < cycleLength)
         if cycleLength % 2 == 1
                  go East
         else
                go West
         steps ++;

    go South
    cycleLength ++; steps = 0;
  } while (true);
```



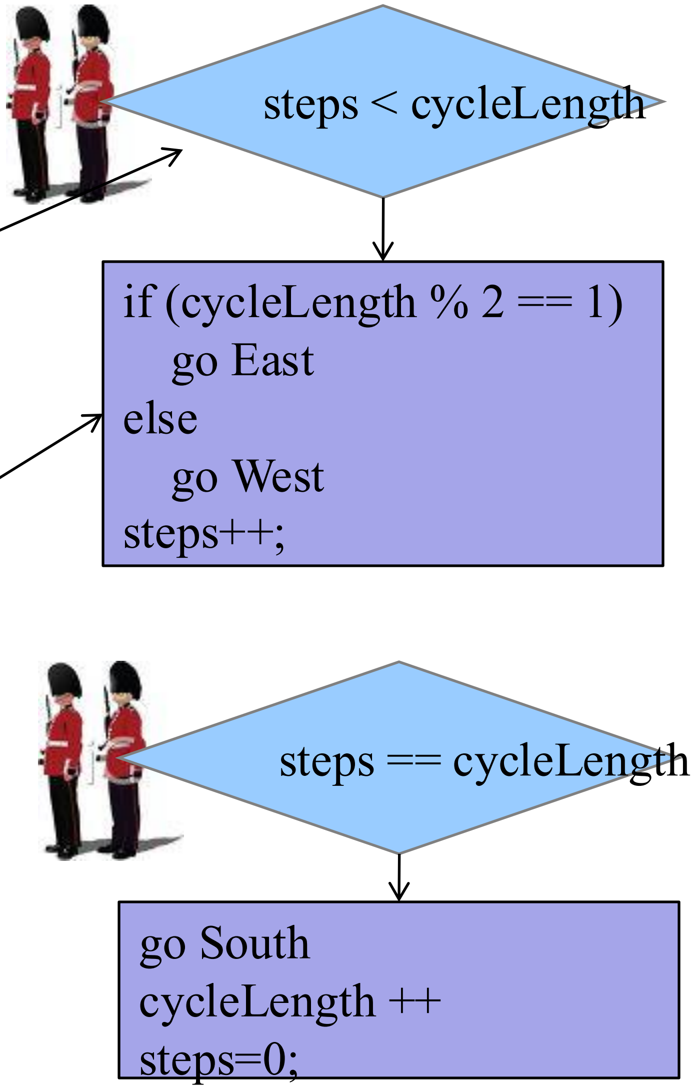
Recap: Non-EDP-> EDP: Guarding Condition

Technique: determine the guarding condition (using state variables) on action statements

```
cycleLength = 1; steps = 0;
do {
   while (steps < cycleLength)
         if cycleLength % 2 == 1
                 go East
         else
                go West
         steps ++;

      go South
      cycleLength ++; steps = 0;

 } while (true);
```



steps < cycleLength

```
if (cycleLength % 2 == 1)
    go East
else
    go West
steps++;
```

steps == cycleLength

```
go South
cycleLength ++
steps=0;
```

Snake solution

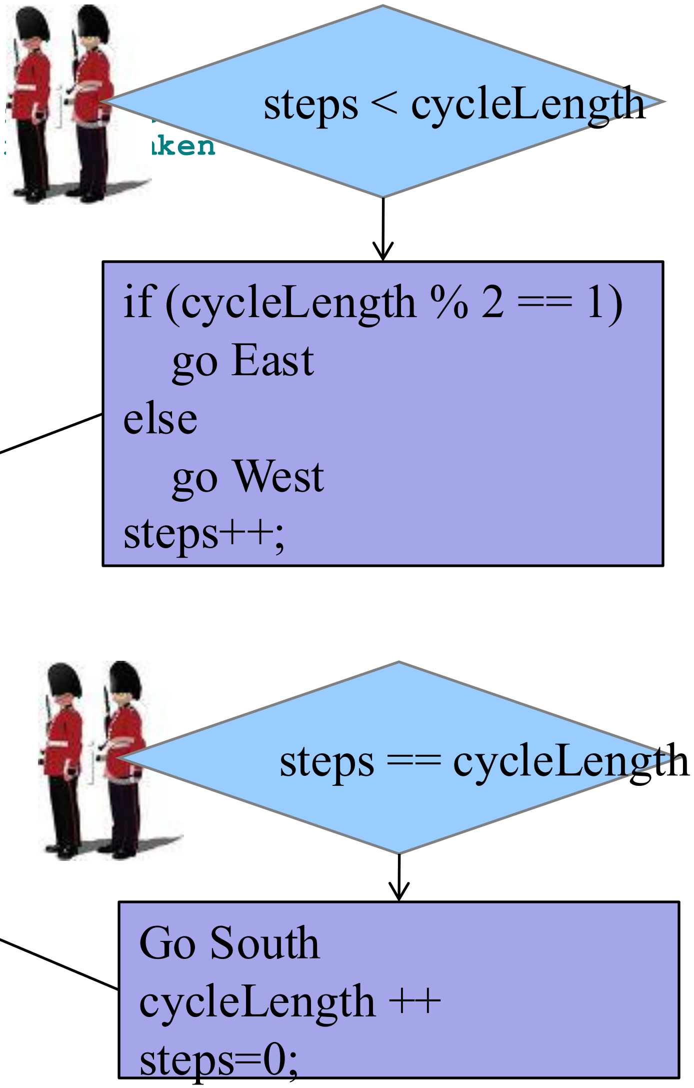
```
import java.awt.*;    // for Color

public class Snake extends Critter {
    private int cycleLength;   // # steps in curr. Hori[illegible]
    private int steps;         // # of cycle's steps al[illegible]ken

    public Snake() {
        cycleLength = 1;
        steps = 0;
    }

    public Direction getMove() {
      if (steps < cycleLength) {
          steps++;
          if (cycleLength % 2 == 1) {
                return Direction.EAST;
            } else {
                return Direction.WEST;
            }
        } else {
           steps = 0;
           cycleLength ++;
           return Direction.SOUTH;
        }
    }

    public String toString() {
        return "S";
    }
}
```



Comment: States

- Counting is helpful:
  - How many total moves has this animal made?
  - How many times has it eaten? Fought?

- Remembering recent actions in fields may be helpful:
  - Which direction did the animal move last?
    - How many times has it moved that way?
  - Did the animal eat the last time it was asked?
  - How many steps has the animal taken since last eating?
  - How many fights has the animal been in since last eating?

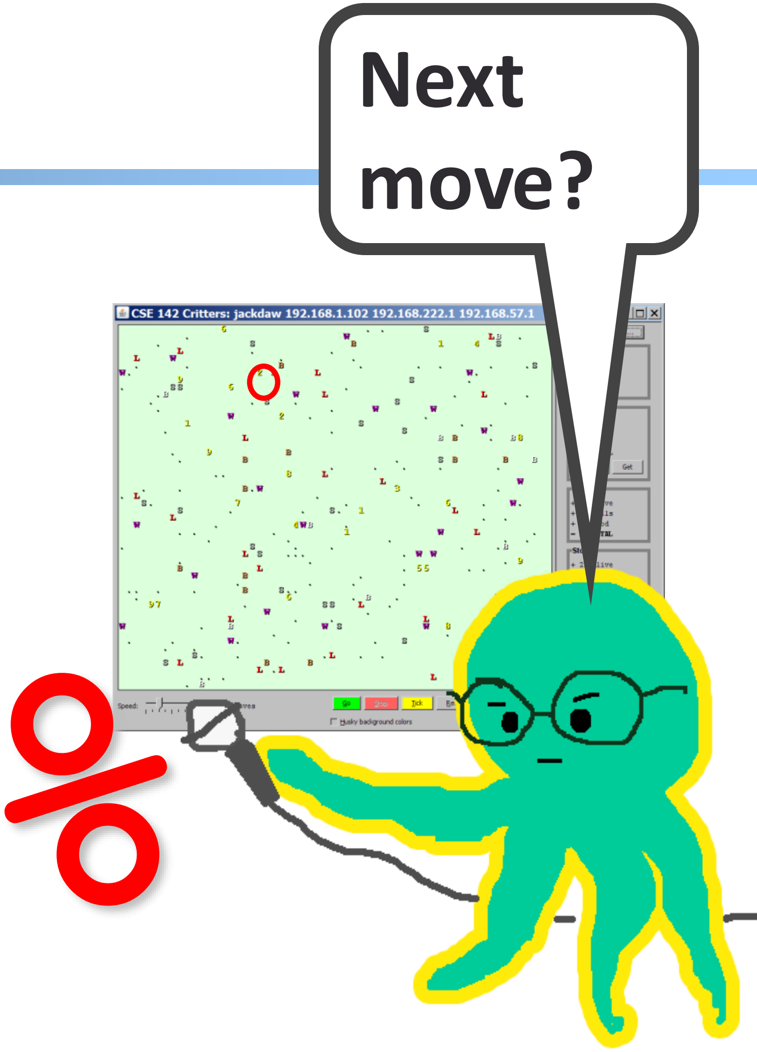
Outline

- ❑ Critters and objects coordination
- ❑ Polymorphism 多态

Motivation

- The controller implemented in CritterMain.java works on all critters objects, even of critter types defined in the future.

- How does one write such a highly reusable, extensible program?


Next move?
CSE 142 Critters: jackdaw 192.168.1.102 192.168.222.1 192.168.57.1
%

What is Polymorphism?

- **polymorphism**: Ability for the same code to be used with different types of objects and behave differently according to the types of objects.

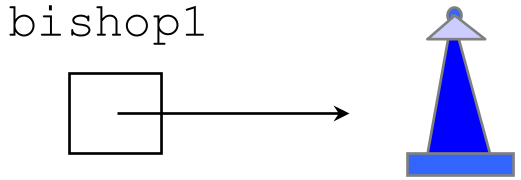
- The foundation of polymorphism is dynamic typing: the method invoked is always determined by the object, not the class.

Recap: Reference Variables

- Interaction with an object occurs through object reference variables

- An object reference variable holds the reference (address, the location) of an object

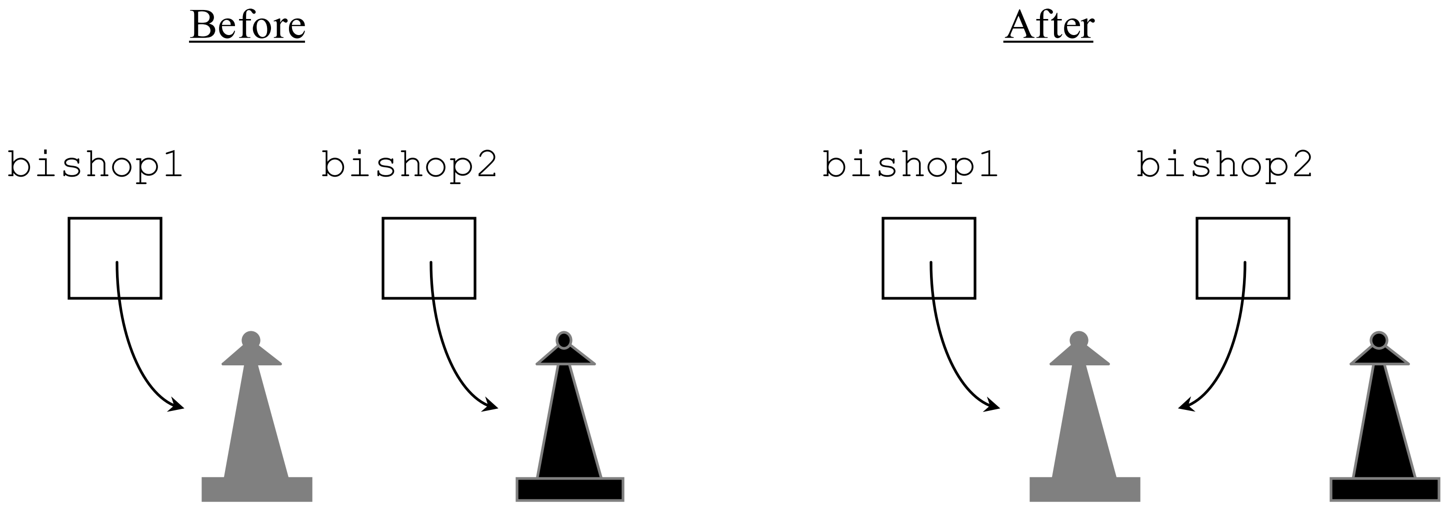
```
ChessPiece bishop1 = new ChessPiece();
```



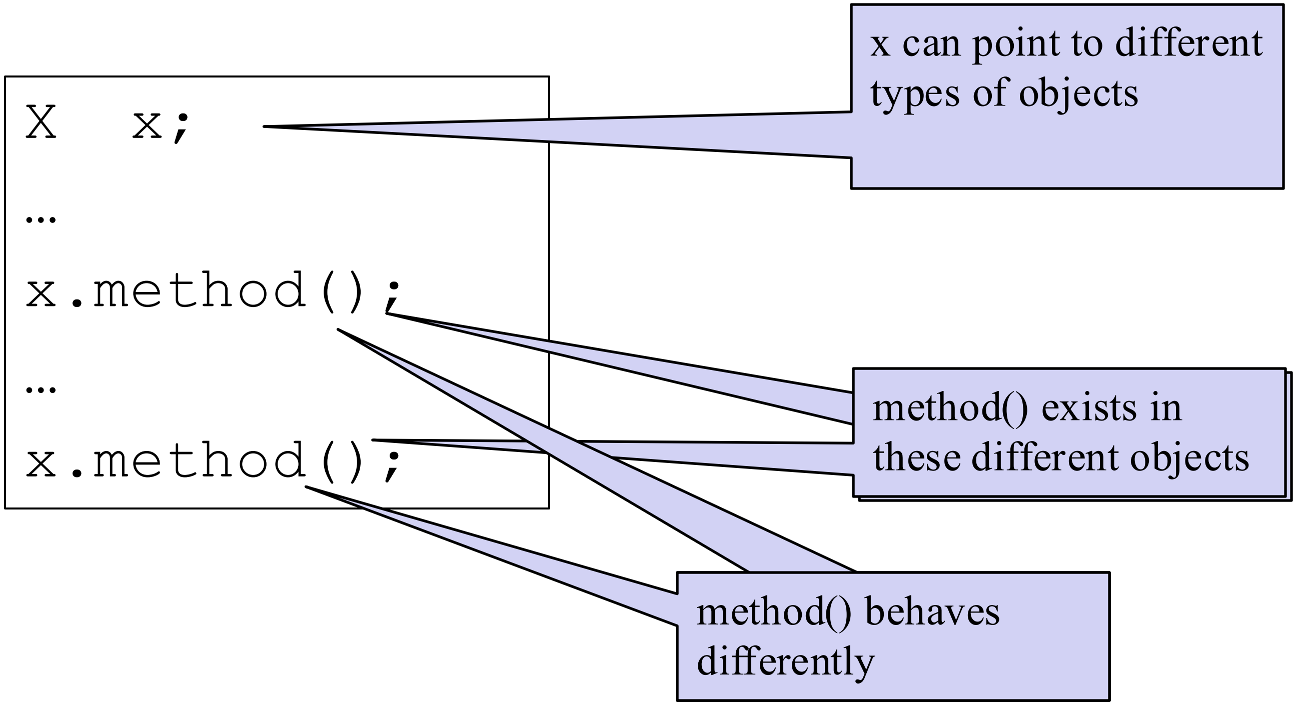
Recap: Object Reference Variable

- Object reference variable assignment copies address, creating aliases

```
bishop2 = bishop1;
```



Requirements on Polymorphic Code



```
X   x;
…
x.method();
…
x.method();
```

x can point to different types of objects

method() exists in these different objects

method() behaves differently

polymorphism: Ability for the same code to be used with different types of objects and behave differently according to the types of objects.

Polymorphism through Inheritance

- Same reference points to different types of objects
  - A variable of type *T* can hold an object of class T or descendent(后代) of *T*, e.g.,

```
Employee emp = new Employee(“Ed”);
emp = new Lawyer(“Larry”);
emp = new LegalSecretary(“Lisa”);
```

- The method used exists in all the objects
  - If the method is defined in the base class
- The method may behave differently
  - The child class can override the method

Polymorphism through Inheritance

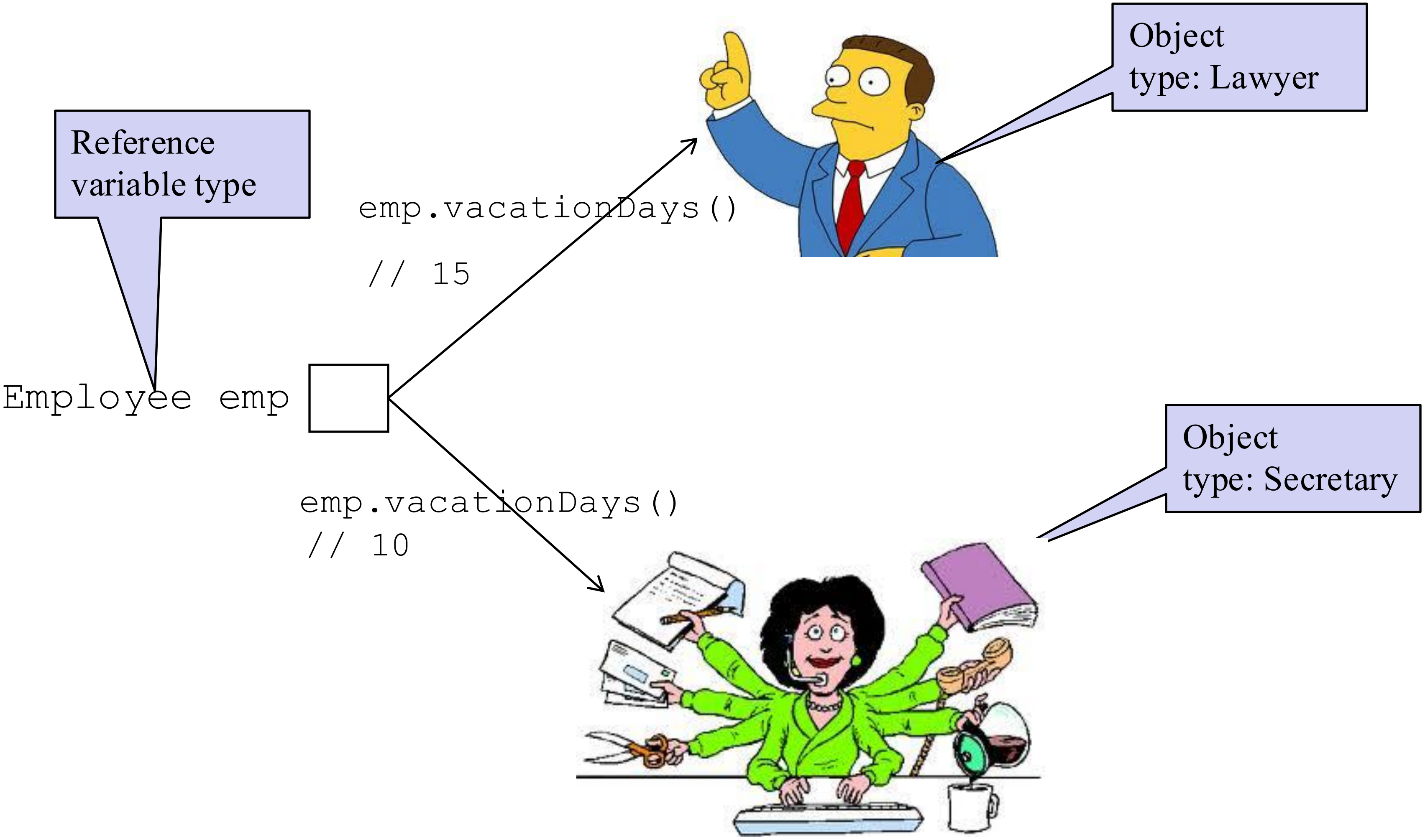
- You can call any methods defined in the based class T (e.g., `Employee`) class on polymorphic reference of type T (e.g., `emp`)
- When you invoke a method through a polymorphic reference variable, **it is the type of the object being referenced, not the reference type, that determines which method is invoked.**
- Careful use of polymorphic references can lead to elegant, robust, highly extensible software designs

Example: Polymorphic Variable

```
Employee emp;  // base type
emp = new Lawyer("Larry");
System.out.println ( emp.vacationDays() );
// OUTPUT: 15
System.out.println ( emp.vacationForm() );
// OUTPUT: pink

emp = new LegalSecretary(“Lisa");
System.out.println ( emp.vacationDays() );
// OUTPUT: 10
System.out.println ( emp.vacationForm() );
// OUTPUT: yellow
```

Example: Polymorphic Variable



Reference variable type

```
Employee emp
```

```
emp.vacationDays()

// 15
```

Object type: Lawyer

```
emp.vacationDays()
// 10
```

Object type: Secretary

Example: Polymorphic Method

- Define a method that can apply to all objects of a base type or its derived types.

- This is how `print` in `PrintStream` is defined:

```
void print(Object obj) {
   // all objects have the toString() method
   // convert to string and then output
}
```

Example: Polymorphic Method

```
public class EmployeeMain {
    public static void main(String[] args) {
        Lawyer lisa = new Lawyer(“Lisa”);
        Secretary steve = new Secretary(“Steve”);
        printInfo(lisa);
        printInfo(steve);
    }

    public static void printInfo(Employee empl) {
        System.out.println("salary: " + empl.pay());
        System.out.println("v.days: " + empl.vacationDays());
        System.out.println("v.form: " + empl.vacationForm());
        System.out.println();
    }
}
```

OUTPUT:

```
salary: 50000.0                 salary: 50000.0
v.days: 15                      v.days: 10
v.form: pink                    v.form: yellow
```

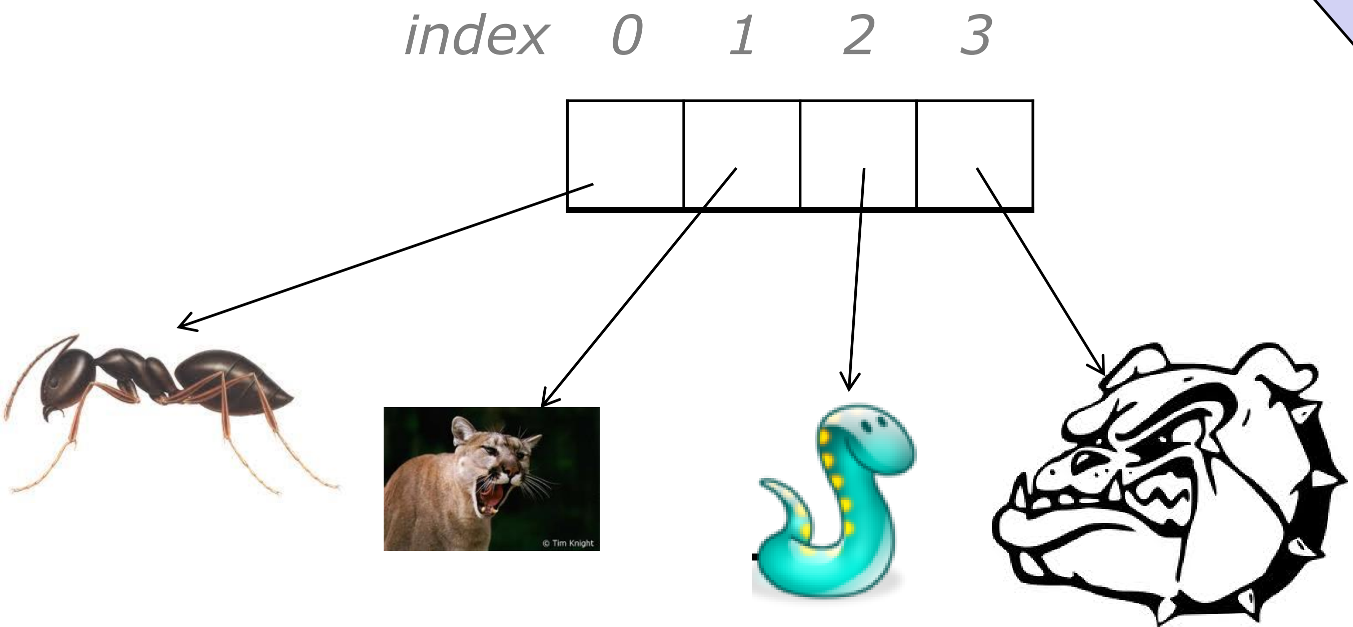
Polymorphic Arrays

- A common usage of polymorphism is to define an array of a base type, but different entries refer to different types of objects
    - To handle a heterogeneous population of objects with uniformity, achieving generic programming

Example: CritterMain Internal

```
Critter[] critters = {
   new Ant(),
    new Cougar(),
    new Snake(),
    new Bulldog()
};
```

```
while (true)
  for (i=0; i<critters.length; i++)
   newPos = critters[i].getMove();
   disp = critters[i].toString();
   … draw disp at pos
```



Not dependent on any specific critters but only the **generic Critter concept**

Example: Polymorphic Array on Firm

```java
public class Staff {
    private Employee[] staffList;
    public Staff() {
        staffList = new Employee[4];
        staffList[0] = new Lawyer("Lisa");
        staffList[1] = new Secretary("Sally");
        staffList[2] = new Marketer("Mike");
        staffList[3] = new LegalSecretary("Lynne");
    }

    public void payday() {
        for (int count = 0; count < staffList.length; count++) {
            System.out.printf("%-10s:", staffList[count].name());
            System.out.printf("$%.2f\n", staffList[count].pay());
            System.out.println("------------------------------------");
        }
    }
}
```

Works on
any mix of
Employee objects

Example: Extending the Program: Hourly

- Include a new type of secretary who works variable number of hours and is paid by the hours.

Extending the Program: Hourly

```
public class Hourly extends Secretary {
    private double payRate;
    private int hours;

    public Hourly(String name, double payRate)
        super(name);
        this.payRate = payRate;
        hours = 0;
    }
    public void addHours(int hours) {
        this.hours += hours;
    }
    public int hours() { return hours; }
    public double pay() {return hours() * payRate;}
}
```

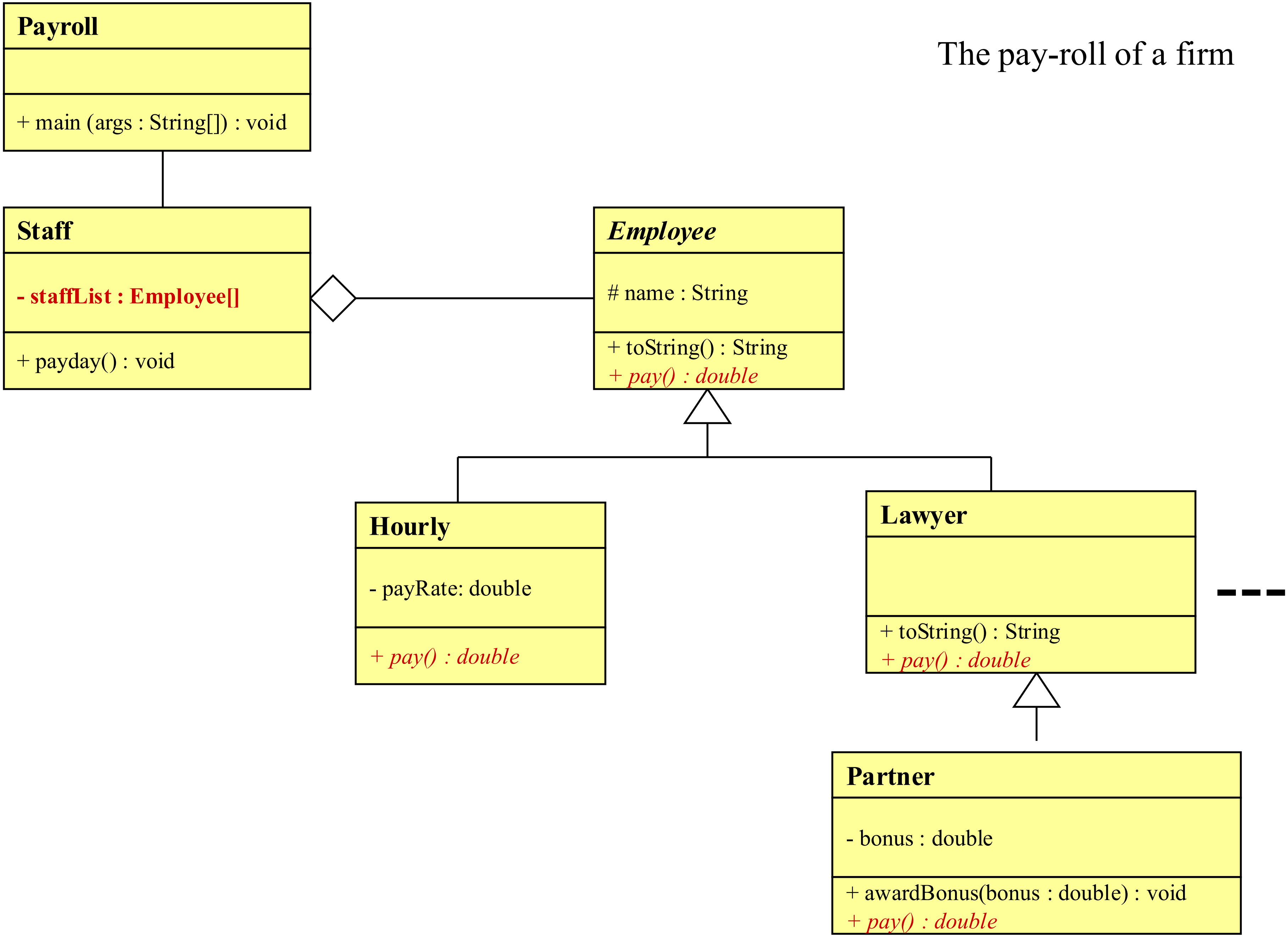
Polymorphic Array Handles Changes

```
public class Staff {
    private Employee[] staffList;
    public Staff() {
        staffList = new Employee[5];
        staffList[0] = new Lawyer("Lisa");
        staffList[1] = new Secretary("Sally");
        staffList[2] = new Marketer("Mike");
        staffList[3] = new LegalSecretary("Lynne");
        Hourly holly = new Hourly(“Holly"); holly.addHours(10);
        staffList[4] = holly;
    }

    public void payday() {
        for (int count = 0; count < staffList.length; count++) {
            System.out.printf("%-10s:", staffList[count].name());
            System.out.printf("$%.2f\n", staffList[count].pay());
            System.out.println("------------------------------------");
        }
    }
}
```

No need to change the payday method at all.

The pay-roll of a firm



Payroll

+ main (args : String[]) : void

Staff

- staffList : Employee[]

+ payday() : void

Employee

# name : String

+ toString() : String
+ pay() : double

Hourly

- payRate: double

+ pay() : double

Lawyer

+ toString() : String
+ pay() : double

Partner

- bonus : double

+ awardBonus(bonus : double) : void
+ pay() : double

Comment: Variable Type and Method

- Through a given type of reference variable, we can invoke only the methods defined in that type

```
class Employee{
    public double pay()
    {…}
}
class Lawyer extends Employee {
    public void sue()
    {…}
}

Employee ed = new Lawyer("Larry");
```

Can we do the following statements:

```
ed.pay();
ed.sue();
```

Comment: Variable Type and Method

- We can “promote” an object back to its original type through an explicit narrowing cast:

```
staffList = new Employee[5];
staffList[0] = new Lawyer("Lisa");
staffList[1] = new Secretary("Sally");
staffList[2] = new Marketer("Mike");
staffList[3] = new LegalSecretary("Lynne");
staffList[4] = new Hourly(“Holly");

Hourly holly = (Hourly)staffList[4];

holly.addHours (5);
```

If the type of object referred to by staff[4] is not Hourly, program error.

Summary: Polymorphism

- **polymorphism**: Ability for the same code to be used with different types of objects and behave differently with each.

  - `CritterMain` can interact with any type of critter.
    - Each one moves, fights, etc. in its own way.
  - `Firm` can use one method to pay for any type of Employee.
    - Each one is paid in its own way.
  - `Print`

Outline

- Class inheritance
  - polymorphism, and polymorphism through inheritance
- Interface as an alternative of inheritance
  - motivation
  - syntax

Interface Syntax

- An *interface* is a collection of constants and abstract methods
  - abstract method: a method header without a method body; we declare an abstract method using the modifier `abstract`
  - since all methods in an interface are abstract, the `abstract` modifier is usually left off

Interface: Example

interface is a reserved word

```
public interface Movable {

    public double getSpeed();
    public void   setSpeed(double speed);
    public void   setDirection(int direction);
    public int    getDirection();
}
```

This interface describes the behaviors common to all movable things.
(Every Movable thing should have these methods.)

A semicolon follows each method header immediately

No method in an interface has a definition (body)

Implementing an interface

- general syntax:

```
public class <name> implements <interface names> {
    ...
}
```

- Example:

```
public class Bicycle implements Movable {
    ...
}
```

(What must be true about the `Bicycle` class for it to compile?)

Interface Implementation

- If we write a class that claims to be an interface (e.g., `Movable`) , but doesn't implement all of the methods defined in the interface, it will not compile.

  - Example:

    ```
    public class Bicycle implements Movable {
    }
    ```

  - The compiler error message:

    ```
    Bicycle.java:1: Bicycle is not abstract
    and does not override abstract method
    getSpeed() in Movable
    ```

Example: Shape interface

- An interface for shapes:

```
public interface Shape {
    public double area();
}
```

  - This interface describes the common features that all shapes should have in your design. (Every shape has an area.)

Example: Circle class

```java
// Represents circles.
public class Circle implements Shape {
    private double radius;

    // Constructs a new circle with the given radius.
    public Circle(double radius) {
        this.radius = radius;
    }

    // Returns the area of this circle.
    public double area() {
        return Math.PI * radius * radius;
    }

}
```

Example: Rectangle class

```
// Represents person.
public class Person implements Shape {
    private double weight;
    private double height;
    …

    public Person(double weight, double height) {
        this.weight = weight;
        this.height = height;
    }

    // Returns the area of a person using Du Bois formula
    public double area() {
        return 0.007184 * Math.power(weight, 0.425)
                        * Math.power(height, 0.725);
    }

    // other methods
}
```

Summary: Interfaces

- **interface**: A list of methods that classes can promise to implement.
  - Analogous to non-programming idea of roles or certifications
    - "I'm certified as a CPA accountant."
- interface vs inheritance
  - inheritance gives an is-a relationship and code-sharing.
    - A Lawyer object can be treated as an Employee, and Lawyer inherits Employee's code.
  - interface gives an is-a relationship without code sharing.

Outline

- ❑ Admin and recap
- ❑ Class inheritance
  - ○ polymorphism, and polymorphism through inheritance
- ❑ Interface as an alternative of inheritance
  - ○ motivation
  - ○ syntax
  - ○ polymorphism through inheritance

Satisfy Polymorphism Requirements using Interface

- Same reference points to different types of objects
  - A variable of interface type *T* can hold an object of any class implementing *T*.

    ```
    Movable mobj = new Bicyle();
    ```
- The method used exists in all the objects
  - Using an interface reference, you can only invoke the methods defined in the interface;
  - A class must implement the methods defined in the interface
- The method may behave differently
  - Different class can implement the method differently

Interface Polymorphism: Example

```
public static void printShapeInfo(Shape s) {
    System.out.println("area : " + s.area());
    System.out.println();
}
```

- Any object that implements the interface may be passed as the parameter to the above method.

```
Circle circ = new Circle(12.0);
Person john = new Person(60, 175);
printShapeInfo(circ);
printShapeInfo(john);
```

Interface Polymorphism: Example

- We can create an array of an interface type, and store any object implementing that interface as an element.

```
Circle circ = new Circle(12.0);
Person john = new John(60, 175);
YaleStudent nicole = new YaleStudent();

Shape[] shapes = {circ, john, nicole};
for (int i = 0; i < shapes.length; i++) {
    printShapeInfo(shapes[i]);
}
```

  - Each element of the array executes the appropriate behavior for its object when it is passed to the `printShapeInfo` method

Interface

- An *interface* provides an abstraction to write reusable, general programs
- Instead of writing a program for **a single class (hierarchy)** of objects, we want to write a program to handle all classes with a given set of behaviors/properties
  - An interface is an abstraction for the common behaviors of these behaviors
- Often interface represents abstract concepts

Summary: Using Interface for General Programming

- When defining a class or method (e.g., sorting), think about the essence (most general) properties/behaviors of the objects you require

- Define those properties in an interface

- Implement the class/method for the interface only so that your design is the most general !