Introduction to Computational Thinking

Inheritance and object construction;

Method Overriding; Object Hierarchy;
Event-Driven Programming

Qiao Xiang, Qingyu Song

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

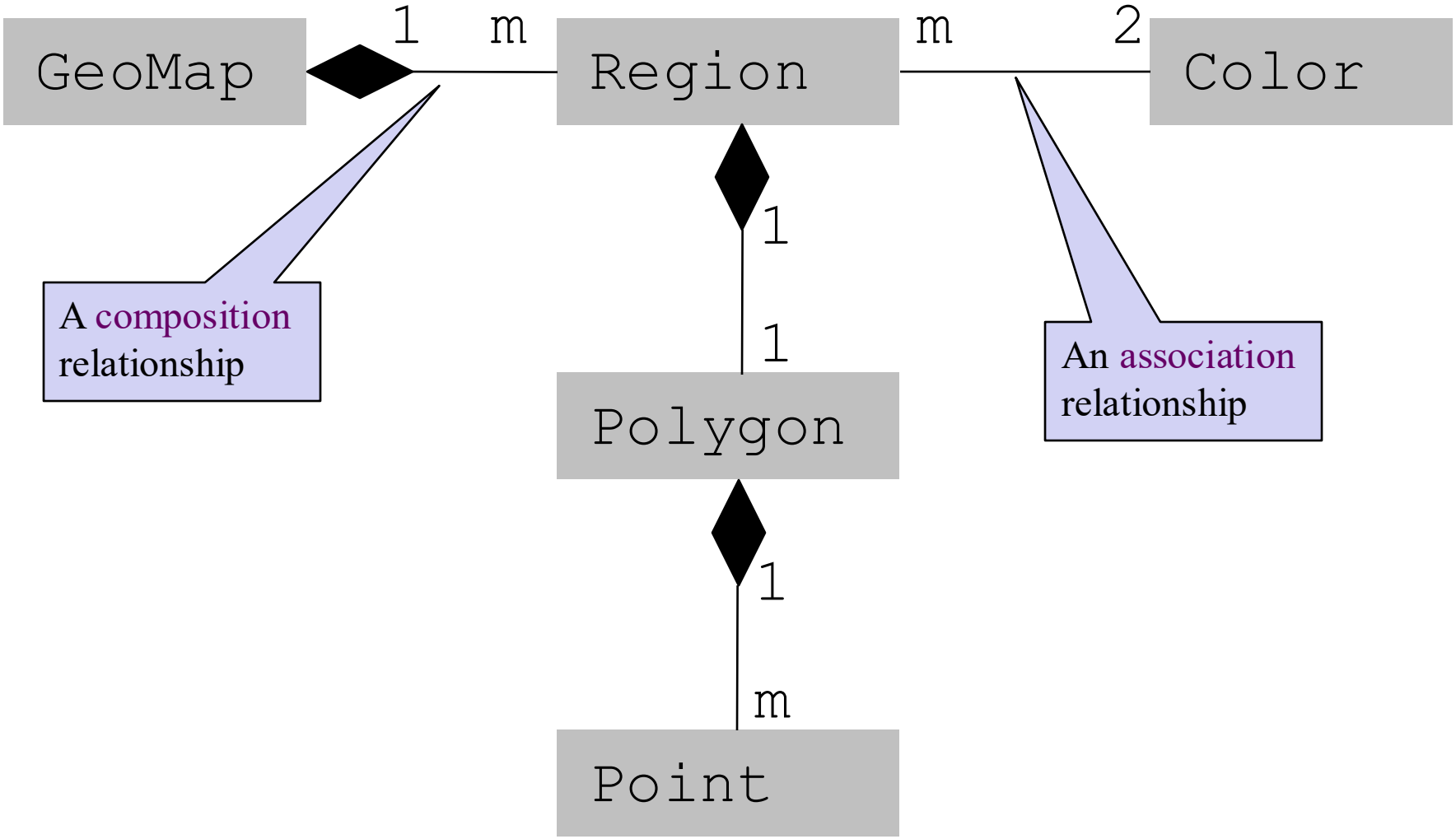
12/17/2025

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.

Outline

- Admin and recap
  - Final Exam: 2025-12-30 10:30-12:30 学武楼(1号楼)A206
- Object-oriented design
  - *Inheritance(继承) relationship*

Recap: The GeoVisualization Domain and OO Composition/Association Relationships



Inheritance

- **Inheritance**: Reuse classes by deriving a new class from an existing one
  - The existing class is called the parent class, or superclass, or base class
  - The derived class is called the *child class* or *subclass*.

- As the name implies, the child inherits characteristics of the parent
  - The child class *inherits* every method and every data field defined for the parent class

Deriving Subclasses: Syntax

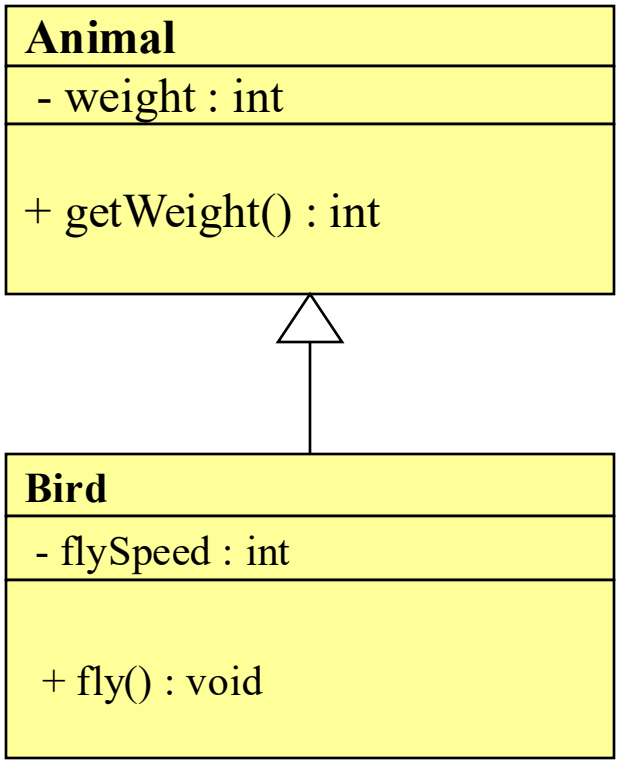
```
public class <name> extends <superclass> {

}
```

For example:

```
class Animal {
  // class contents
  private int weight;
  public int getWeight() {…}
}

class Bird extends Animal {
  private int flySpeed;
  public void fly() {…};
}
```

Visualize Inheritance

- The child class *inherits all* methods and data defined for the parent class



an animal object

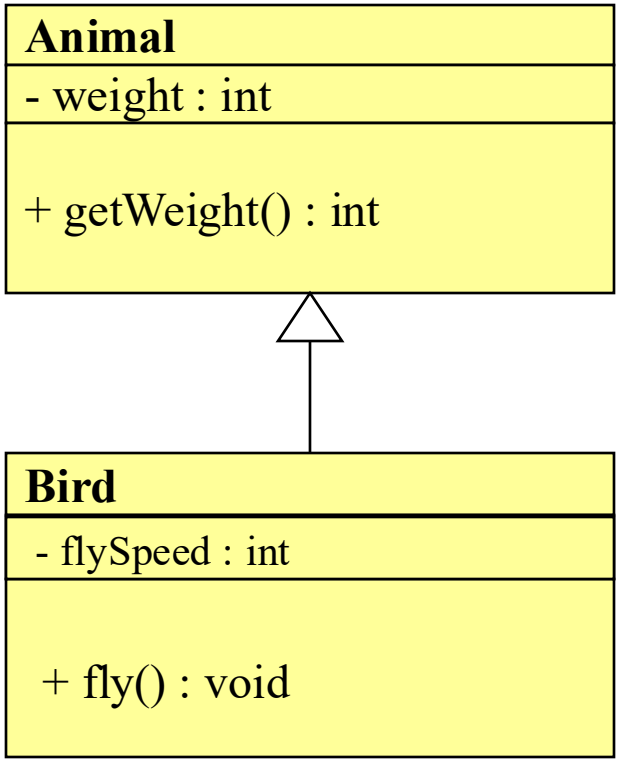
weight = 120
getWeight()

a bird object

weight = 100
flySpeed = 30
getWeight()
fly()

Visualize Inheritance

- Shown graphically in a *class diagram*, with the arrow pointing to the parent class



Inheritance should create an *is-a relationship*: the child *is a* more specific version of the parent

Example: The Law Firm(律师事务所)

- The firm has 5 types of employees
  - Standard employee
  - Secretary
    - prepare ordinary documents
  - Legal secretary
    - prepare both documents and legal documents. ordinary
  - Marketer
    - advertise
  - Lawyer
    - sue(起诉)

The Law Firm

- Work time policy: Employees work 40 hours / week.
- Pay policy: Employees, base salary of $50,000 per year, except that
  - legal secretaries: 10% extra over base per year,
  - marketers: 20% extra over base per year,
  - lawyers who reach partner level get bonus.
- Vacation policy: Employees have 2 weeks of paid vacation leave per year, except that
  - lawyers: an extra week on top of base,
  - employees: use a yellow form to apply for leave, except for lawyers who use a pink form.

The Employee class

```
public class Employee {
    public int hours() {
        return 40;              // works 40 hours / week
    }
    public double pay() {
        return 50000.0;         // $50,000.00 / year
    }
    public int vacationDays() {
        return 10;              // 2 weeks' paid vacation
    }
    public String vacationForm() {
        return "yellow";        // use the yellow form
    }
    public String toString() {
        String result = "Hours: " + hours() + "\n";
        result += "Pay: " + pay() + "\n";
        result += "Vacation days: " + vacationDays() + "\n";
        result += "Vacation Form: " + vacationForm() + "\n";
        return result;
    }   }
```

Secretary without Reuse

```
public class Secretary {
  public int hours() {
     return 40;            // works 40 hours / week
  }
  public double pay() {
     return 50000.0;       // $50,000.00 / year
  }
  public int vacationDays() {
     return 10;            // 2 weeks' paid vacation
  }
  public String vacationForm() {
     return "yellow";      // use the yellow form
  }
  public String toString() {
     String result += "Hours: " + hours() + "\n";
     result += "Pay: ”  + pay() + "\n";
     result += "Vacation days: " + vacationDays() + "\n";
     result += "Vacation Form: " + vacationForm() + "\n";
     return result;
  }
  public void prepareDoc(String text) {
     System.out.println(“Working on Document: " + text);
  }
}
```

Improved Secretary code

```
// A class to represent secretaries.
public class Secretary extends Employee {

    public void prepareDoc(String text) {
        System.out.println("Working on document: " + text);
    }

}
```

- By extending `Employee`, each `Secretary` object now:
  - receives methods `hours`, `pay`, `vacationDays`, `vacationForm`, `toString` from `Employee`'s definition automatically
  - can be treated as an `Employee` by client code (seen later)
- Now we only write the parts unique to each type.

Outline

- Class inheritance
  - why and how?
  - inheritance and object construction

Object Construction Example

```
public class Secretary extends Employee {
    public Secretary() {
      System.out.println("In Secretary()");
    }
    …
}

public class Employee {
    public Employee() {
       System.out.println("In Employee()");
    }
    …
}

public static void main(String[] args) {
    Secretary seth = new Secretary();
}
```

Output:

```
In Employee()
In Secretary()
```

Object Construction Example

```
public class Secretary extends Employee {
  public Secretary() {
     System.out.println("In Secretary()");
  }
  …
}

public class Employee {
   private String name;
   public Employee(String name) {
     System.out.println("In Employee()");
     this.name = name;
   }
   public Employee() {
     System.out.println("In Employee()");
   }
   …
}
```

Puzzle: This program will not compile.

Inheritance and Constructor

- Java object construction can appear to be complex
- Rules:
  1. When an object is created, the constructor identified by new is invoked.
  2. If a class does not define any constructor, Java automatically defines a default constructor (class name w/o any parameters).
  3. Constructors are not inherited.
  4. In a child class, the constructor of the parent class is first called. If the programmer does not invoke the parent's constructor, Java automatically inserts a call to the parent's default constructor.

```
public class Secretary extends Employee {
    public Secretary () {
      // super() is automatically inserted
      System.out.println("In Secretary()");
    }
    …}
```

Object Construction Example

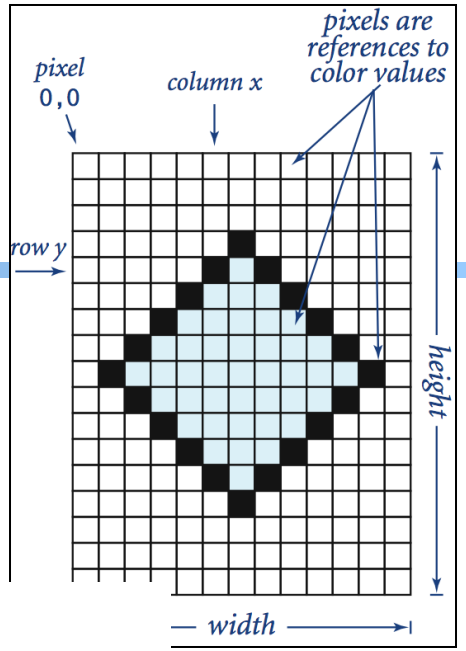
```
public class Secretary extends Employee {
  public Secretary() {
// super() is automatically inserted, but not defined
    System.out.println("In Secretary()");
  }
  …
}

public class Employee {
  private String name;
  public Employee(String name) {
    System.out.println("In Employee()");
    this.name = name;
  }
  …
}
```

super and Constructor

- If you insert super(…) as the first statement in child's constructor, Java will not insert the default parent constructor:

```
public class Secretary extends Employee {
    public Secretary(String name) {
      super(name);
      System.out.println("In Secretary()");
    }
    …
}

public class Employee {
    private String name;
    public Employee(String name) {
       System.out.println("In Employee()");
       this.name = name;
    }
    …
}
```

Exercise: Reuse Existing Class



- Assume you find a Picture class (assume no source file, only .class file)

```
public class Picture
```

	Picture(String filename)	*create a picture from a file*
	Picture(int w, int h)	*create a blank* w*-by-*h *picture*
int	width()	*return the width of the picture*
int	height()	*return the height of the picture*
Color	get(int x, int y)	*return the color of pixel (x, y)*
void	set(int x, int y, Color c)	*set the color of pixel (x, y) to* c
void	show()	*display the image in a window*
void	save(String filename)	*save the image to a file*

- Assume you want to have a Picture class w/ more functions, such as converting to gray. What are your design options?

Two Options to Reuse an Existing Code

- Wrapper (delegation design)

```
public class BtrPicture {
  private Picture p;

  public BtrPicture(String file Name) {
    p = new Picture( fileName );
  }
  public int width() { return p.width();}
  ..

  public BtrPicture gray() {
     // create a new picture as gray
  }
}
```

- Inheritance design

```
public class BtrPicture extends Picture {


  public BtrPicture(String file Name) {
    super( fileName );
  }
  ..


  public BtrPicture gray() {
     // create a new picture as gray
  }
}
```

Both designs have pros (positive) and cons (Negative) in terms of amount of coding, control.

Outline

- Admin and recap
- Class inheritance
  - why and how?
  - inheritance and object construction
  - inheritance and "mutation" (overriding)

Motivation: Implementing the `Lawyer` class: Attempt 1

```
// A class to represent lawyers.
public class Lawyer extends Employee {
    public Lawyer(String name) {
        super(name);
    }
    public void sue() {
        System.out.println("I'll see you in court!");
    }
}
```

```
public static void main(String[] args) {
  Lawyer larry = new Lawyer("larry");
  System.out.println ( larry.vacationDays() );
}
```

Does the design work?

Motivation: Implementing the `Lawyer` class: Attempt 1

```
// A class to represent lawyers.
public class Lawyer extends Employee {
    public Lawyer(String name) {
        super(name);
    }
    public void sue() {
        System.out.println("I'll see you in court!");
    }
}
```

```
public static void main(String[] args) {
  Lawyer larry = new Lawyer("larry");
  System.out.println ( larry.vacationDays() );          // 10 not 15
}
```

Does the design work?

Problem

- We want lawyers to inherit *most* behaviors from employee, but we want to *replace* parts with new behavior:
  - Lawyers get an extra week of paid vacation over base vacation (a total of 3).
  - Lawyers use a pink form when applying for vacation leave.

Defining Methods in the Child Class: Overriding Methods

- A child class can (have the option to) *override* the definition of an inherited method in favor of its own
  - that is, a child can redefine a method that it inherits from its parent
  - the new method must have the same signature as the parent's method, but can have different code in the body

- The method invoked is always the one defined in the child class, if the child class refines (overrides) a method

Lawyer class

```
// A class to represent lawyers.
public class Lawyer extends Employee {
    public Lawyer(String name) {
        super(name);
    }

    // overrides getVacationDays from Employee class
    public int vacationDays() {
        return 15;              // one more week vacation
    }

    // overrides getVacationForm from Employee class
    public String vacationForm() {
        return "pink";
    }

    public void sue() {
        System.out.println("I'll see you in court!");
    }
}
```

Overriding and the @Override annotation

```
// A class to represent lawyers.
public class Lawyer extends Employee {
    public Lawyer(String name) {
        super(name);
    }

    @Override // optional hint to compiler to check spelling
    public int vacationDays() {
        return 15;            // one more week vacation
    }

    @Override
    public String vacationForm() {
        return "pink";
    }

    public void sue() {
        System.out.println("I'll see you in court!");
    }
}
```

Overloading vs. Overriding

- Overloading deals with multiple methods in the same class with the same name but different signatures
- Overloading lets you define a similar operation in different ways for different data

- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overriding lets you define a similar operation in different ways for different object types

Outline

- Admin and recap
- Class inheritance
  - why and how?
  - inheritance and object construction
  - inheritance and "mutation" (overriding)
    - Good overriding design

Marketer class

```
// A class to represent marketers.
public class Marketer extends Employee {
    public Marketer(String name) {
        super(name);
    }

    public void advertise() {
        System.out.println("Act while supplies last!");
    }

    // override
    public double pay() {
        return 60000.0;
    }
}
```

- marketers: 20% extra over base per year,

```
// $60,000 = +20% of 50,000
```

Anything you do not like about the design?

A Problem

```
public class Marketer extends Employee {
    public double pay() {
        return 60000.0;
    }
    ...
}
```

- Problem: The policy is that Marketer's salaries are based on the `Employee`'s base salary (20% more than base), but the `pay` code does not reflect this.

Motivation: Changes to Common Behavior

- Imagine a company-wide change affecting all employees.

- Example: Everyone is given a $10,000 raise due to inflation.
  - The base employee salary is now $60,000.
  - We modify Employee's pay method to reflect this policy change.

Modifying the superclass

```
// A class to represent employees in general (20-page manual).
public class Employee {
    public int hours() {
        return 40;              // works 40 hours / week
    }

    public double pay() {
        return 60000.0;         // $60,000.00 / year
    }

    ...
}
```

- Issue: the `Marketer` subclass is still incorrect.
  - It has overridden `pay` to return another value.
- Good design: derived behavior is based on base behavior

Calling overridden methods

Subclasses can call overridden methods with `super`

`super.`***`<method>`***`(`***`<parameters>`***`)`

- Exercise: Modify `Marketer` to derive pay for marketers from base pay.

Improved subclasses

```
public class Marketer extends Employee {
    public void advertise() {
        System.out.println("Act now while supplies last!");
    }

    // override and invoke the parent's version
    public double pay() {
        return super.pay() * 1.2;
    }
    …
}
```

Outline

- Admin and recap
- Class inheritance
  - why and how?
  - inheritance and object construction
  - inheritance and "mutation" (overriding)
  - inheritance and field access

Inheritance and Fields

- Setting: To retain their lawyers, the firm changes pay policy so that a lawyer gets the base and $5000 for each year in the firm

```
public class Lawyer extends Employee {
     …
     public double pay() {
        return super.pay() + 5000 * years;
     }

     …
}

//years is a private field in Employee.
```

https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcR3InPYVkDuRLOzveTXxW0YyTTZi1b5Uo0pD8_Jkodow0IlPY9a

Problem

- Fields declared `private` cannot be accessed from subclasses
  - Reason: subclassing cannot break encapsulation
  - Q: how to get around this limitation?

Solution 1

- Add an accessor for any field needed by the subclass

```
public class Employee {
    private String name; private int years;

    public Employee(String name, int initialYears) {
        this.name = name; years = initialYears;
    }

    public int getYears() {
        return years;
    }
    ...
}

public class Lawyer extends Employee {
    public Lawyer(String name, int years) {
        super(name, years);
    }

    public double pay() {
        return super.pay() + 5000 * getYears();
    }
    ...
}
```

Solution 2

- Java provides a third visibility modifier to denote fields/methods to be accessible by **only** child classes: `protected`

```
public class Employee {
    private String name;
    protect int years;

    public Employee(String name, int years) {
        this.name = name;
        this.years = years;
    }

…
}
```

Discussion

- How to choose between the two designs?
  - Design 1: Add `public getYear()`
  - Design 2: make year `protected`

- Use Design 1, unless the method is an implementation method (not accessable, not service method)
  - Adding `public getYear()` makes it available to not only child class, but also all other classes.
  - If you do not want this, use Design 2

Outline

- Admin and recap
- Class inheritance
  - why and how?
  - inheritance and object construction
  - inheritance and "mutation" (overriding)
  - inheritance and field access
  - inheritance hierarchy

Levels of inheritance

- Multiple levels of inheritance in a hierarchy are allowed.
  - Example: A legal secretary is the same as a regular secretary but makes more money (10% more) and can file legal briefs.
  - Exercise: Implement the `LegalSecretary` class.

Example: LegalSecretary class

```
// A class to represent legal secretaries.
public class LegalSecretary extends Secretary {
    public void fileLegalBriefs() {
        System.out.println("I could file all day!");
    }

    public double pay() {
        return super.pay() * 1.1 ;
    }
}
```

Example: Partner class

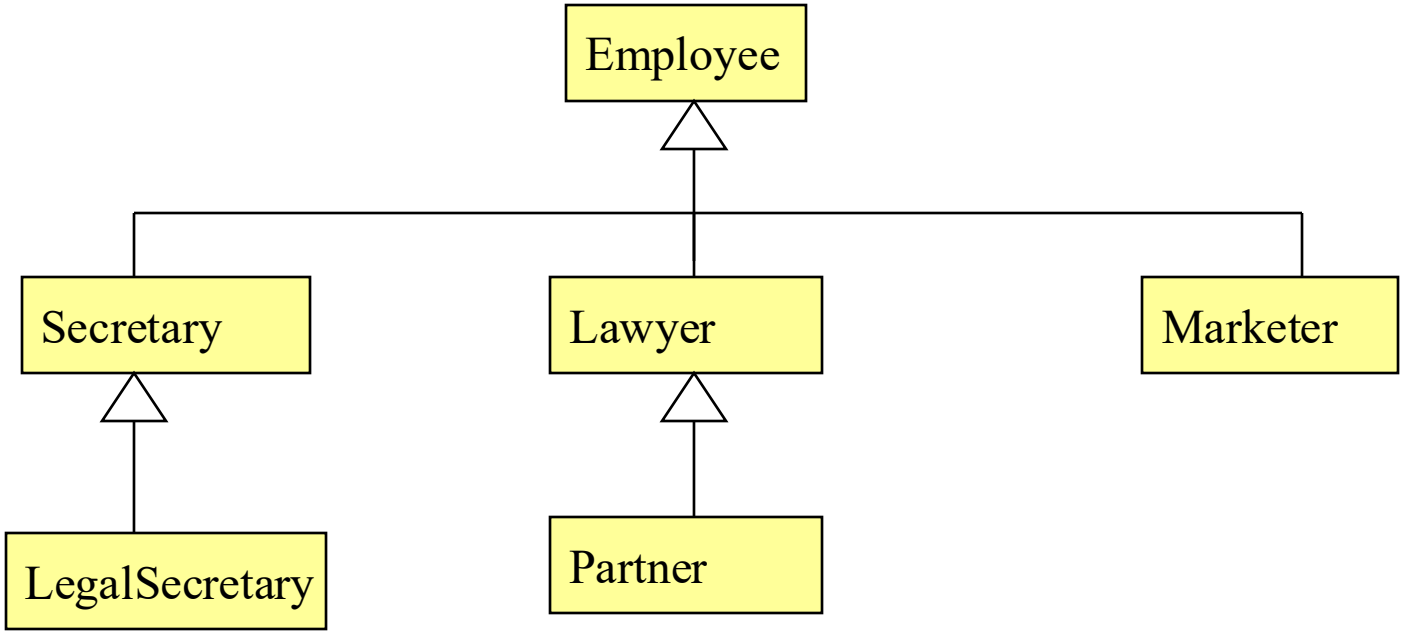
- Partner is a senior lawyer that can get bonus. Thus it supports:
  `awardBonus(double bonus)`

Example: Partner class

```
// A class to represent partner.
public class Partner extends Lawyer {
    private double bonus;
     public void awardBonus(double bonus) {
         this.bonus = bonus;
     }

     public double pay() {
         return super.pay() + bonus ;
     }
}
```

Class Hierarchies

- Many large-scale software systems define *class hierarchies*, where the root defines the common behaviors

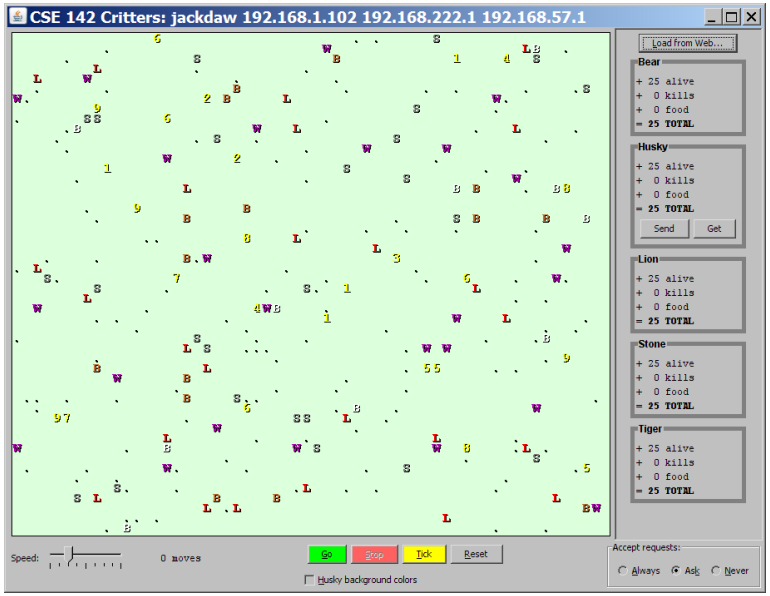


Outline

- Admin and recap
- Class inheritance
  - why and how?
  - inheritance and object construction
  - inheritance and "mutation" (overriding)
  - inheritance and field access
  - inheritance hierarchy
  - inheritance hierarchy of Critters and event-driven programming

Critters

- A simulation (game) world of animal objects (e.g., Ants, Birds, Cougars) with common behaviors such as
  - `eat` eating food
  - `fight` animal fighting
  - `getColor` color to display
  - `getMove` movement
  - `toString` letter to display



The `Critter` Class

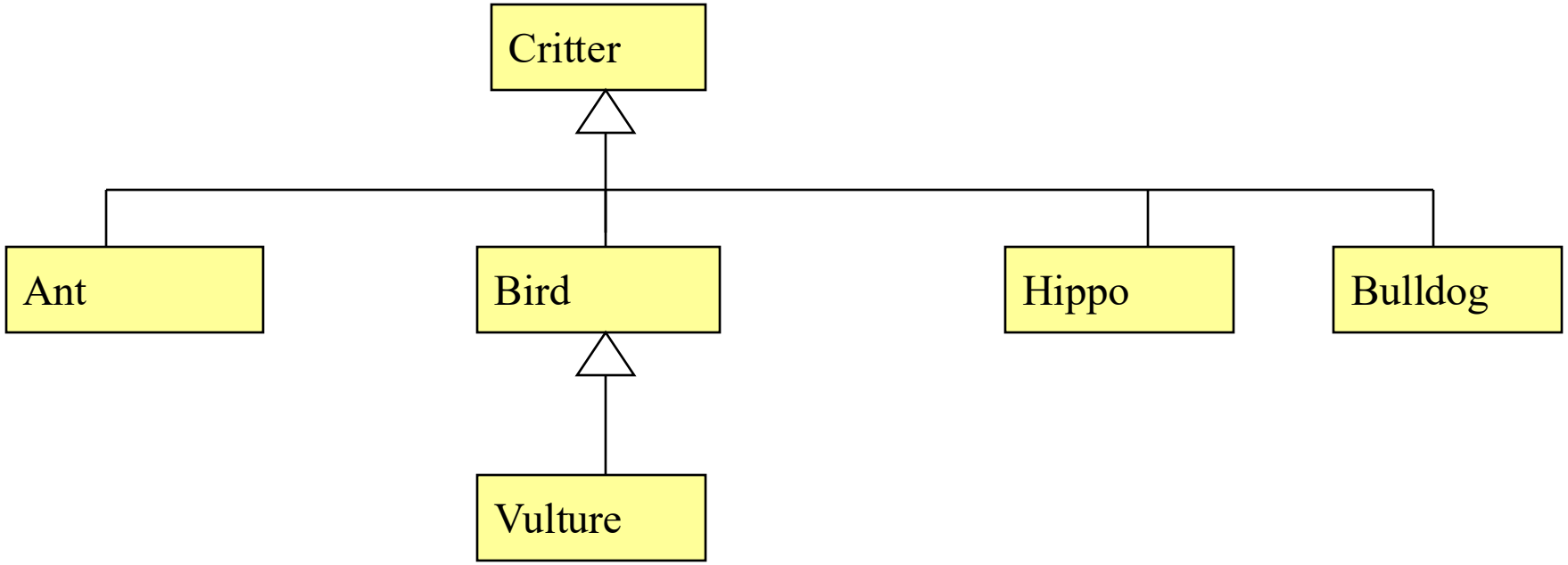
```
// abstract class means not implement every method
public abstract class Critter {
    public boolean eat()
    public Attack fight(String opponent)
            // ROAR, POUNCE, SCRATCH, FORFEIT
    public Color getColor()
    public Direction getMove(String[][] grid)
            // NORTH, SOUTH, EAST, WEST, CENTER
    public String toString()
     …
    // read the class for other methods available
}
```

Defining a `Critter` subclass

```
public class name extends Critter {
    ...
}
```

- `extends Critter` tells the simulator your class is a critter
  - an example of *inheritance*
- Override methods to give each new type of animal distinct behaviors.

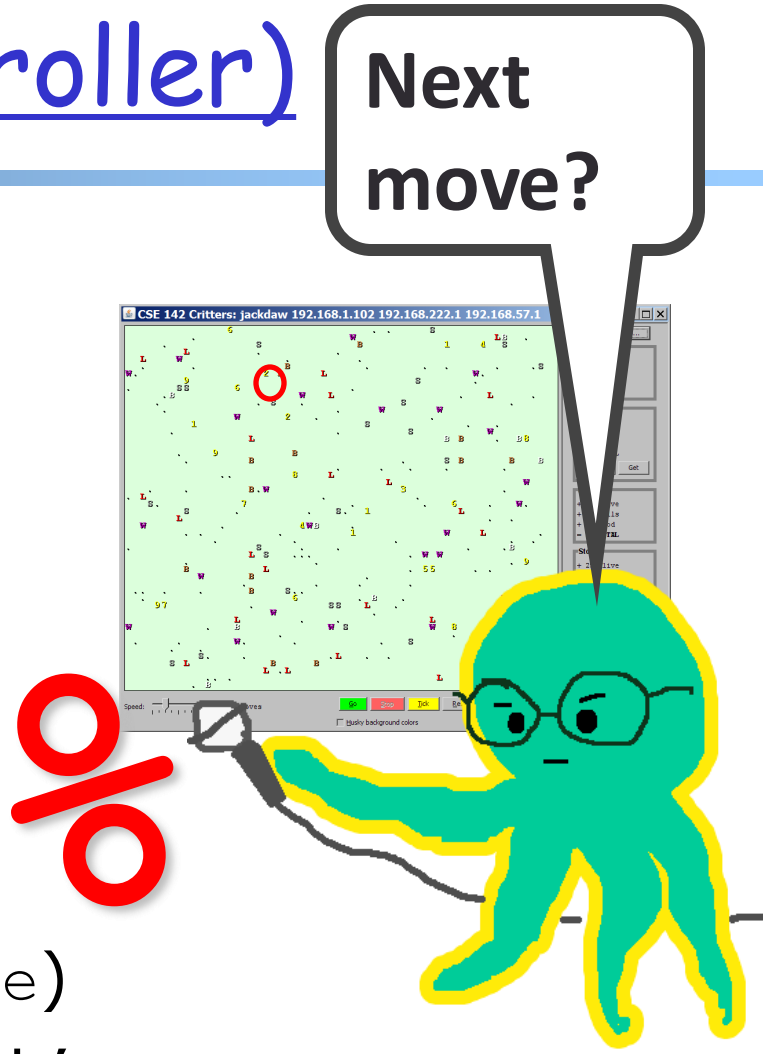
Example Critter World Class Hierarchy



Vulture秃鹫
Hippo河马
Bulldog斗牛犬

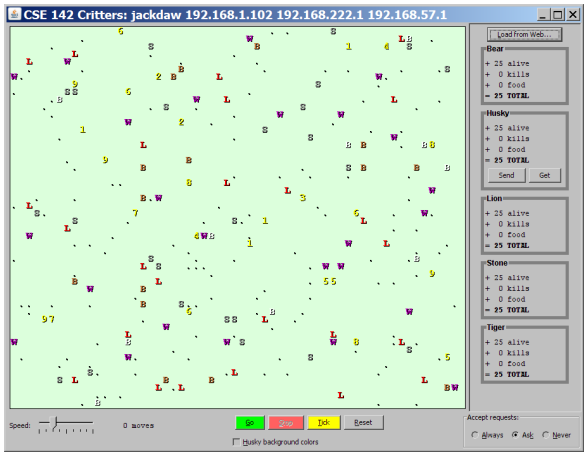
The Simulator (Controller)

- The simulator is in CritterMain.java
- It searches local dir for all critters types
- The simulator creates an array of critters
- "Go" → loop, e.g.,
  - move each animal (`getMove`)
  - if two collide(碰撞), call each's `fight` method on its behavior
  - if is over food, call `eat`



Simulator Pseudo-code

```
Critter[] critters = new Critter[N];
critters[0] = new Ant();
critters[1] = new Bird();
…

loop
  foreach critter i in critters
    call getMove of critter i if it can move

  foreach critter i in critters
     if new pos of critter i results in fight
       ask how critter i will fight
     else if new pos finds food
        ask critter i whether it will eat
      else if new pos results in mate possibility
        ask if critter i will mate

   compute new state of critters
```



Critter Example: Stone

```
import java.awt.*;

public class Stone extends Critter {
      public Attack fight(String opponent) {
         return Attack.ROAR;     // ROAR(咆哮)... nothing beats that!
      }

      public Color getColor() {
         return Color.GRAY;      // stones are gray in color
      }

      public String toString() {
        return "St";             // the game displays a stone
      }
}
```

Event-Driven Programming

- Key concept: The simulator is in control, NOT your animal.
  - Example: `getMove` can return only one move at a time.
    `getMove` can't use loops to return a sequence of moves.
    - It wouldn't be fair to let one animal make many moves in one turn!
  - Your animal must keep state (as fields) so that it can make a single move, and know what moves to make later.
  - We say that you focus on writing the callback functions of objects

Critter exercise: `Cougar`

- Write a critter class `Cougar` (among the dumbest of all animals):

Method	Behavior
constructor	`public Cougar()`
`eat`	Always eats.
`fight`	Always roars.
`getColor`	Blue if the `Cougar` has never fought; red if he has.
`getMove`	Walks west until he finds food; then walks east until he finds food; then goes west and repeats.
`toString`	`"C"`

Implement Cougar's eat, fight, toString.

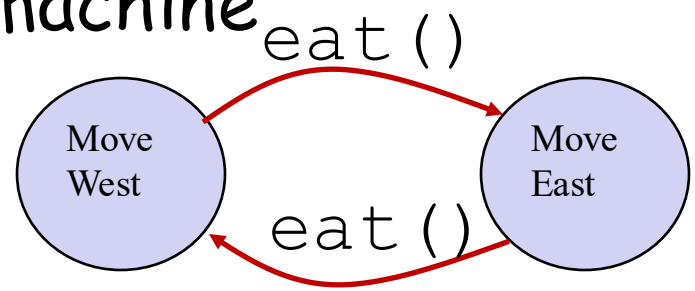
getMove

- How can a critter move west until it finds food and then moves to east until find food and repeat?

```
public Direction getMove(String[][] grid) {
    initial currentDirect = WEST
    loop
      if (eat) {
         reverse currentDirect;
      print currentDirection;
}
```

getMove for Cougar

- State machine

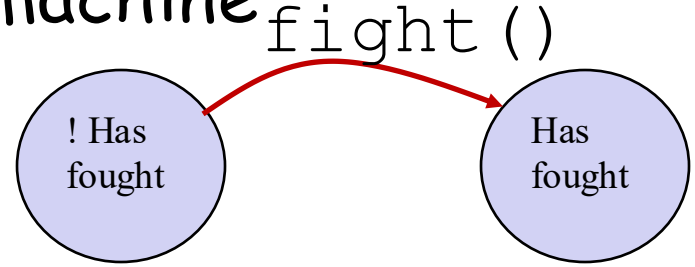


- How to remember the state?
  - a boolean instance variable: `boolean west`
- What is initial state and where to set it?
  - In constructor: `west = true;`
- Who/when updates the state?
  - In `eat()`: `reverse state`

getColor for Cougar

Blue if the `Cougar` has never fought; red if he has.

- State machine



- How to remember the state?
  - A boolean instance variable:
    `boolean fought`
- What is initial state and where to set it?
  - In constructor: `fought = false;`
- Who/when updates the state?
  - In `fight()`: `fought = true`

Cougar solution

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

public class Cougar extends Critter {
    private boolean west;
    private boolean fought;

    public Cougar() {
        west = true;
        fought = false;
    }

    public boolean eat() {
        west = !west;
        return true;
    }

    public Attack fight(String opponent) {
        fought = true;
        return Attack.POUNCE;
    }

    ...
```

Cougar solution

```
    ...

    public Color getColor() {
        if (fought) {
            return Color.RED;
        } else {
            return Color.BLUE;
        }
    }

    public Direction getMove(String[][] grid) {
        if (west) {
            return Direction.WEST;
        } else {
            return Direction.EAST;
        }
    }

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

Comment: PS10 Development Strategy

- Do one species at a time
  - in ABC order from easier to harder
  - debug println's
- Simulator helps you debug
  - smaller width/height
  - fewer animals
  - **"Tick"** instead of "Go"
  - **"Debug"** checkbox
  - drag/drop to move animals

Testing critters

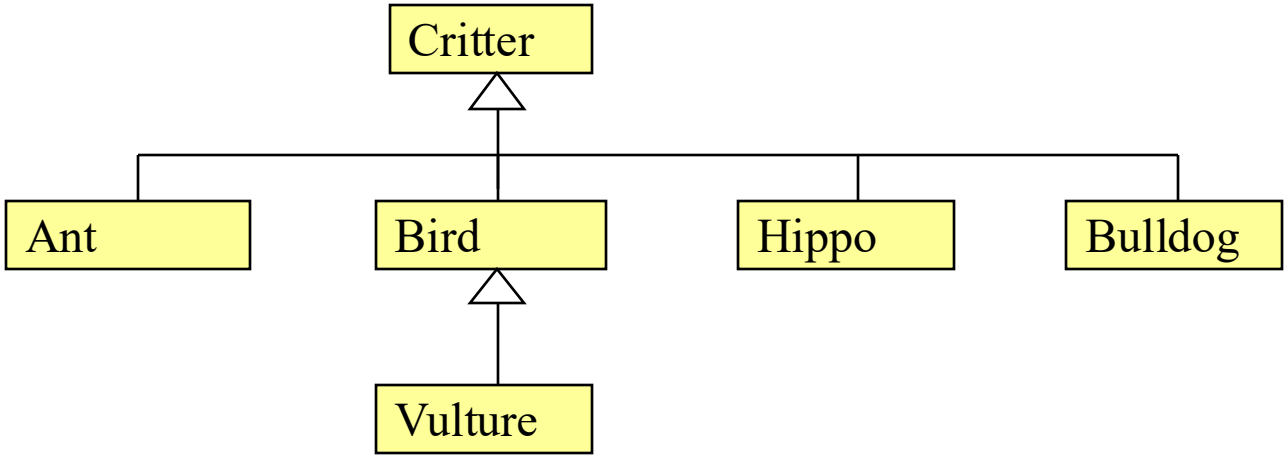
- Focus on one specific critter of one specific type
  - Only spawn 1 of each animal, for debugging
- Make sure your fields update properly
  - Use `println` statements to see field values
- Look at the behavior one step at a time
  - Use "Tick" rather than "Go"

Recap: Field/Method Access

Access Modifier	Who can access
`private`	Only within the defining class
`public`	Everywhere
`protected`	The defining class and its descendent classes
-	With the same package

- Two approaches to access a field/method defined in parent class
  - Parent class defines it as `public`
  - Parent class defines it as `protected`

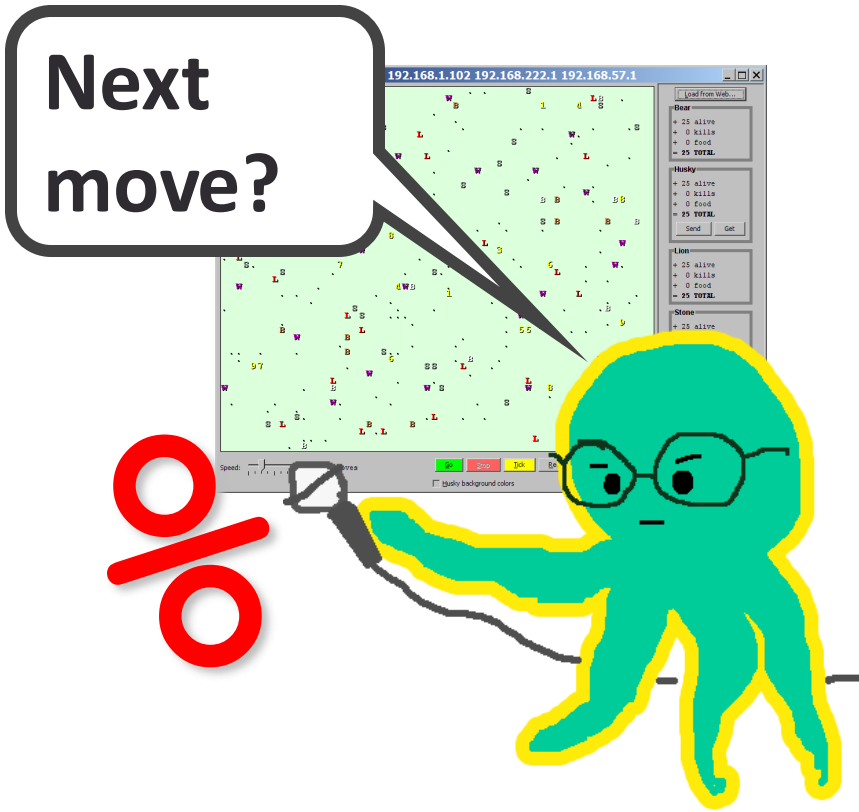
Recap: The `Critter` Class Hierarchy




```
// abstract class means not implement every method
public abstract class Critter {
    public boolean eat()
    public Attack fight(String opponent)
            // ROAR, POUNCE, SCRATCH, FORFEIT
    public Color getColor()
    public Direction getMove(String[][] grid)
            // NORTH, SOUTH, EAST, WEST, CENTER
    public String toString()
     …
    // read the class for other methods available
}
```

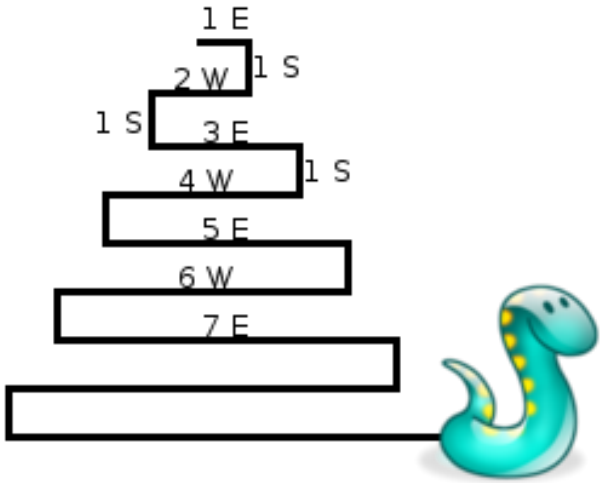
Critters and Event-Driven Programming

- 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
- We will discuss how an EDP framework is designed.



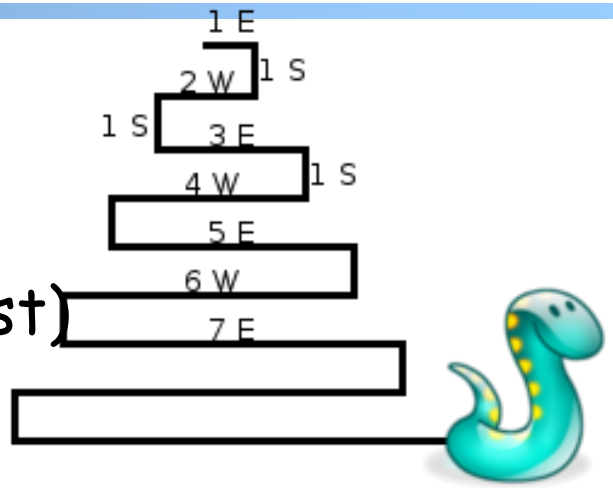
Critter : Snake

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"`



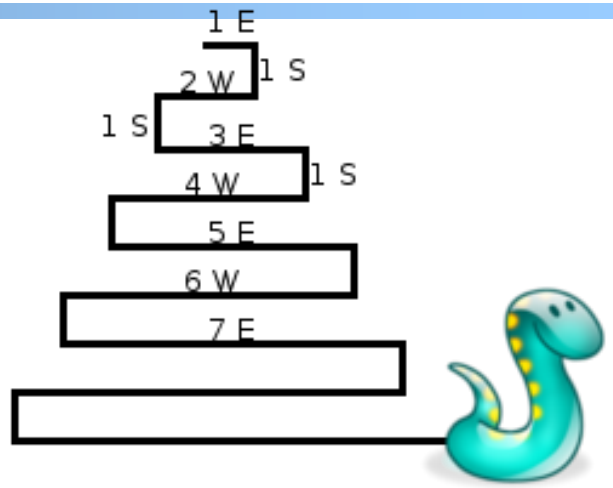
EDP for getMove

- 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



Non-EDP Version

A non-event driven version

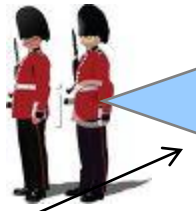



```
cycleLength = 1; steps = 0;
do {
   while (steps < cycleLength)
       if cycleLength % 2 == 1
              go East
       else
             go West
       steps ++;

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

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
    private int steps;          // # of cycle's steps al      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";
    }
}
```

steps < cycleLength

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

steps == cycleLength

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

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?