# Polymorphism and interfaces

AP Computer Science

# Substitutability

```
ActorWorld world = new ActorWorld();
SpiralBug alice = new SpiralBug(6);
BoxBug bob = new BoxBug(3);
world.add(alice);
world.add(bob);
world.show();
```

But aren’t bob and alice two different types?

- **Substitutability** is the ability for an object of a subclass to be used successfully anywhere the object of the superclass is used.

# Polymorphism

- **polymorphism**: Ability for the same code to be used with different types of objects and behave differently with each.
  - `System.out.println` can print any type of object.
    - Each one displays in its own way on the console.
  - `world.add(<actor>)` can take any type of actor.
    - Each one moves, etc. in its own way

# Coding with polymorphism

- A variable of type *T* can hold an object of any subclass of *T*.

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

  - You can call any methods from the `Employee` class on `ed`.

- When a method is called on `ed`, it behaves as a `Lawyer`.

```
System.out.println(ed.getSalary());       // 50000.0
System.out.println(ed.getVacationForm()); // pink
```

# Polymorphism and parameters

- You can pass any subtype of a parameter's type.

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

    public static void printInfo(Employee empl) {
        System.out.println("salary: " + empl.getSalary());
        System.out.println("v.days: " + empl.getVacationDays());
        System.out.println("v.form: " + empl.getVacationForm());
        System.out.println();
    }
}
```

OUTPUT:

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

# Adding actors

- Inheritance relationship

```
class Bug extends Actor {}
class BoxBug extends Bug {}
class SpiralBug extends Bug {}
```

- Each inherits the implementation of `putSelfInGrid` from `Actor`

- In ActorWorld:

```
public void add(Location loc, Actor occupant) {
    occupant.putSelfInGrid(getGrid(), loc);
}
```

# Polymorphism and arrays

- Arrays of superclass types can store any subtype as elements.

```
public class EmployeeMain2 {
    public static void main(String[] args) {
        Employee[] e = { new Lawyer(),   new Secretary(),
                         new Marketer(), new LegalSecretary() };

        for (int i = 0; i < e.length; i++) {
            System.out.println("salary: " + e[i].getSalary());
            System.out.println("v.days: " + e[i].getVacationDays());
            System.out.println();
        }
    }
}
```

Output:

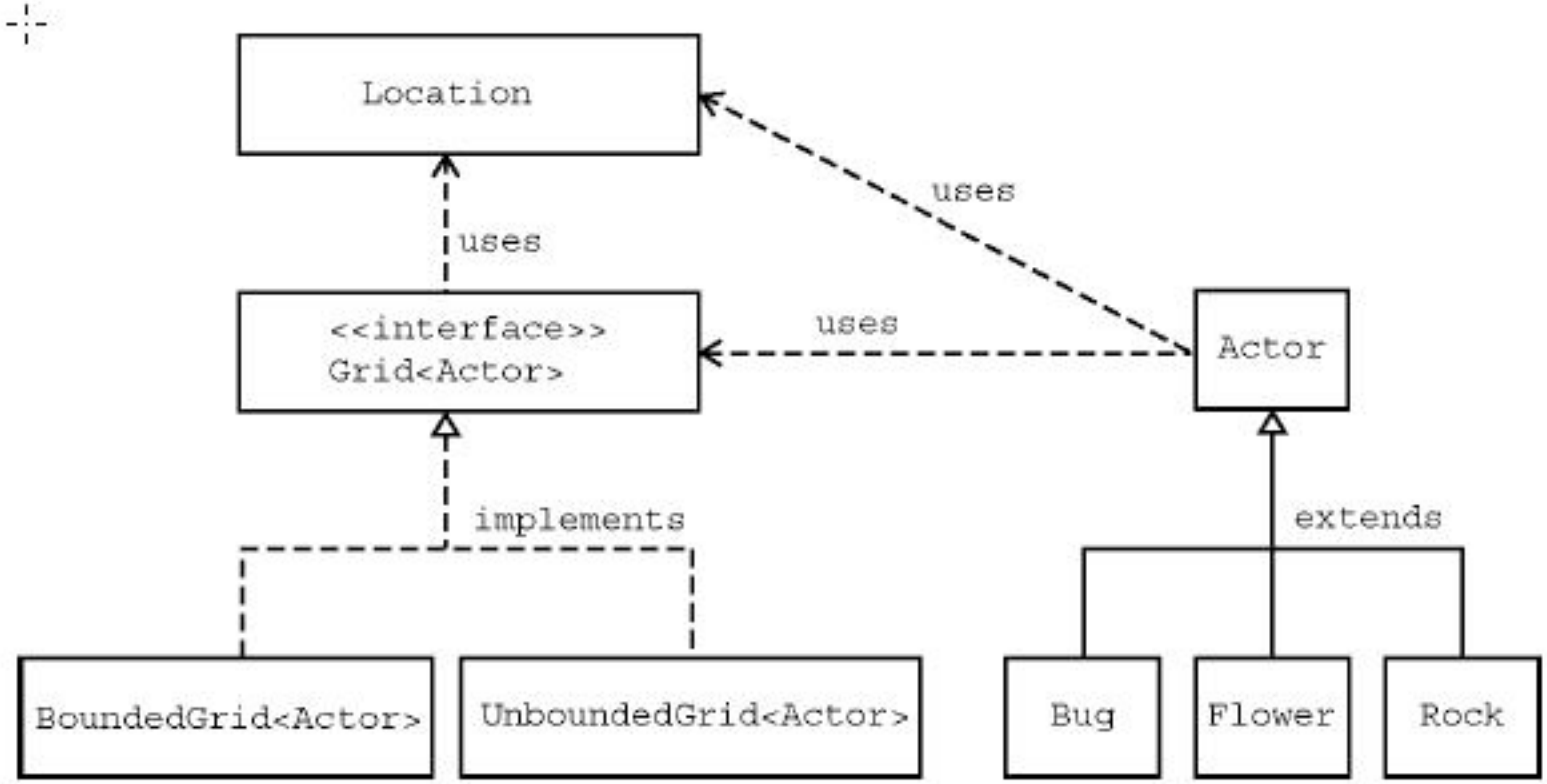
```
salary: 50000.0
v.days: 15

salary: 50000.0
v.days: 10

salary: 60000.0
v.days: 10

salary: 55000.0
v.days: 10
```

# Interfaces


Location
uses
uses
<<interface>>
Grid<Actor>
uses
Actor
implements
extends
BoundedGrid<Actor>
UnboundedGrid<Actor>
Bug
Flower
Rock

# Inheritance limitations

- A class can only extend one superclass
  - what about an employee who is a part time secretary?
- Code is always shared

# Interfaces

- **interface**: A list of methods that a class can implement.

- Inheritance gives you an is-a relationship and code-sharing.
  - A `Lawyer` object can be treated as an `Employee`, and `Lawyer` inherits `Employee`'s code.

- Interfaces give you an is-a relationship *without* code sharing.
  - A `Rectangle` object can be treated as a `Shape`.

- Analogous to the idea of roles or certifications:
  - "I'm certified as a CPA accountant. That means I know how to compute taxes, perform audits, and do consulting."
  - "I'm certified as a Shape. That means I know how to compute my area and perimeter."

# Declaring an interface

```
public interface name {
    public type name(type name, ..., type name);
    public type name(type name, ..., type name);
    ...
}
```

Example:

```
public interface Vehicle {
    public double speed();
    public void setDirection(int direction);
}
```

- **abstract method**: A header without an implementation.
  - The actual body is not specified, to allow/force different classes to implement the behavior in its own way.

# Shape interface

- All shape classes should have methods `perimeter` and `area`.

- Client code should be able to treat different kinds of shape objects in the same way, such as:
  - Write a method that prints any shape's area and perimeter.
  - Create an array of shapes that could hold a mixture of the various shape objects.
  - Write a method that could return a rectangle, a circle, a triangle, or any other shape we've written.
  - Make a `DrawingPanel` display many shapes on screen.

- Exercise: Write an interface for shapes.

# Shape interface

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

- This interface describes the features common to all shapes. (Every shape has an area and perimeter.)

# Implementing an interface

```
public class name implements interface {
    ...
}
```

- Example:

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

- A class can declare that it *implements* an interface.
    - This means the class must contain each of the abstract methods in that interface. (Otherwise, it will not compile.)

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

# Interface requirements

- If a class claims to be a `Shape` but doesn't implement the `area` and `perimeter` methods, it will not compile.

  - Example:

    ```
    public class Banana implements Shape {
        ...
    }
    ```

  - The compiler error message:

    ```
    Banana.java:1: Banana is not abstract and does not
    override abstract method area() in Shape
    public class Banana implements Shape {
                ^
    ```

# Polymorphism

- Interfaces don't benefit the class so much as the *client.*
  - Interface's is-a relationship lets the client use polymorphism.

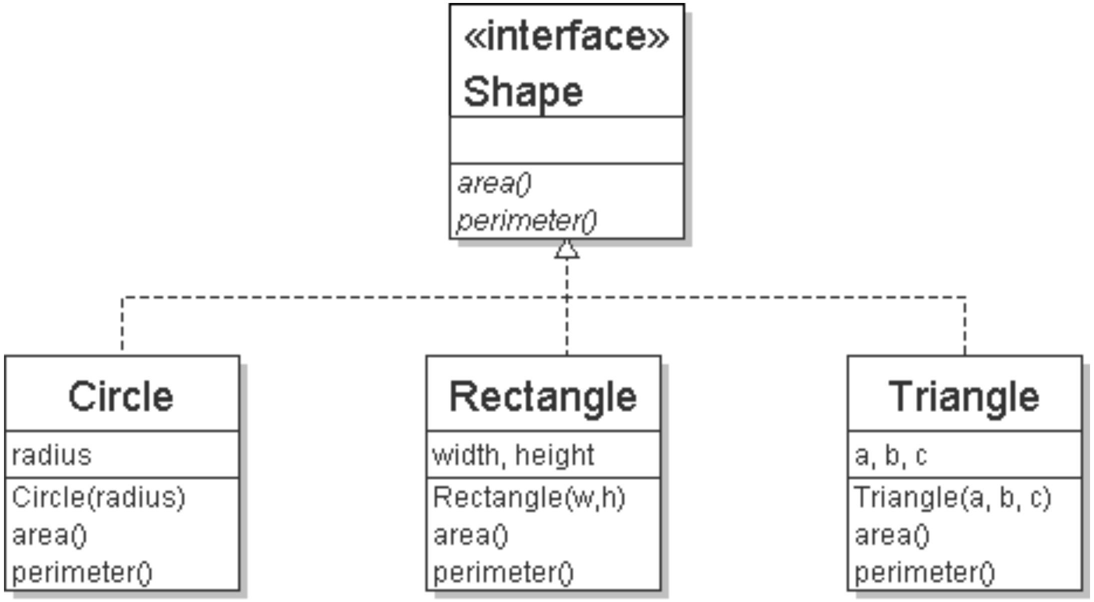
```
public static void printInfo(Shape s) {
    System.out.println("The shape: " + s);
    System.out.println("area : " + s.area());
    System.out.println("perim: " + s.perimeter());
}
```

  - Any object that implements the interface may be passed.

```
Circle circ = new Circle(12.0);
Rectangle rect = new Rectangle(4, 7);
Triangle tri = new Triangle(5, 12, 13);
printInfo(circ);
printInfo(tri);
printInfo(rect);

Shape[] shapes = {tri, circ, rect};
```

# Interface diagram



«interface»
Shape

*area()*
*perimeter()*

Circle

radius

Circle(radius)
area()
perimeter()

Rectangle

width, height

Rectangle(w,h)
area()
perimeter()

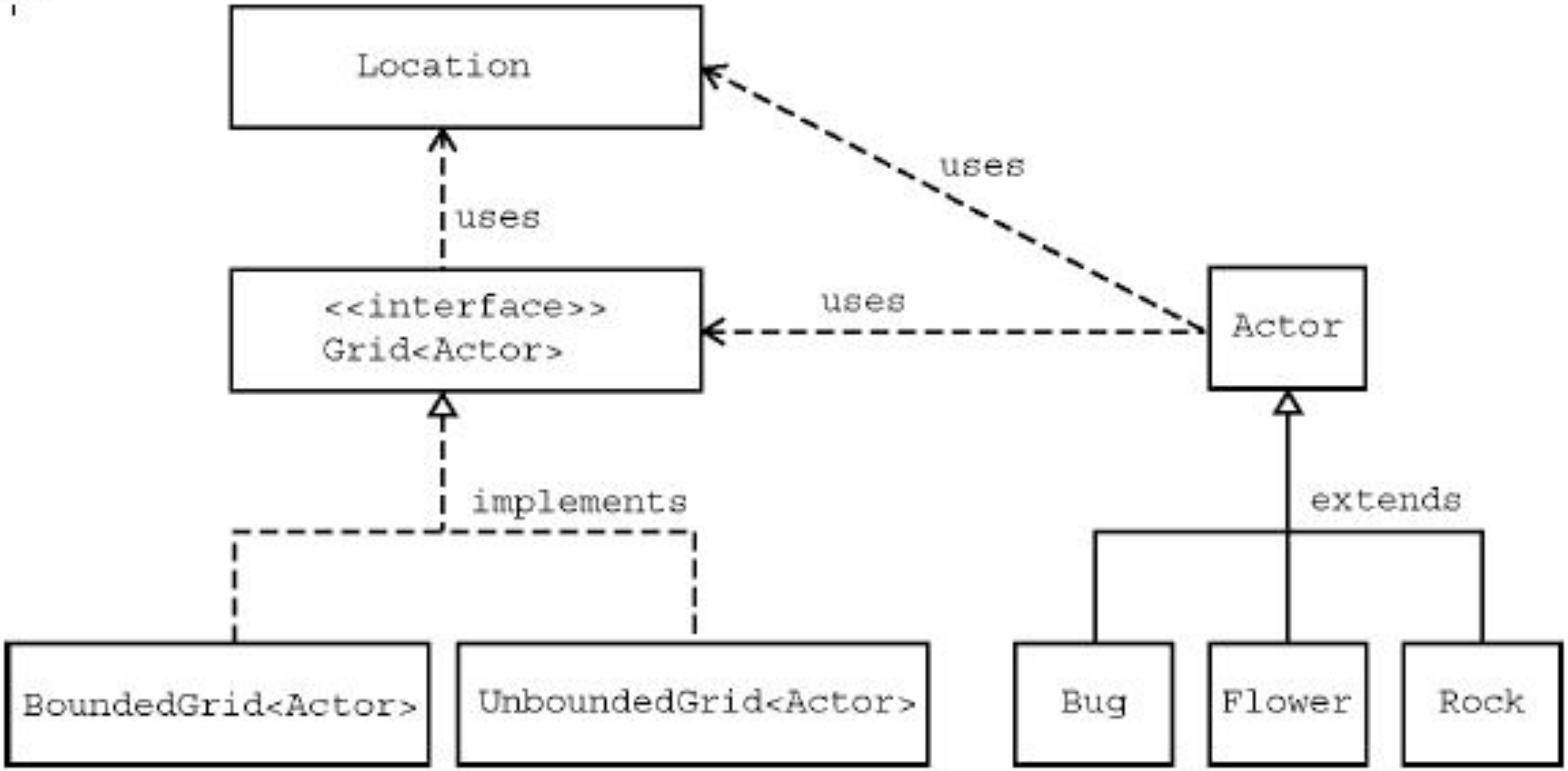
Triangle

a, b, c

Triangle(a, b, c)
area()
perimeter()

# Standard Java Interfaces

- Comparable<T> - requires description of how to compare objects of the type
    - Location implements the Comparable interface (has `equals` and `compareTo`)
- List<E> - used to describe data structures used to store collections of objects
    - ArrayList is-a List!!


Location
uses
uses
<<interface>>
Grid<Actor>
uses
Actor
implements
extends
BoundedGrid<Actor>
UnboundedGrid<Actor>
Bug
Flower
Rock

# Grid Interface

```
package info.gridworld.grid;

import java.util.ArrayList;

public interface Grid<E> {
    int getNumRows();

    int getNumCols();

    boolean isValid(Location loc);

    E put(Location loc, E obj);

    E remove(Location loc);

    E get(Location loc);

    ArrayList<Location> getOccupiedLocations();

    ArrayList<Location> getValidAdjacentLocations(Location loc);

    ArrayList<Location> getEmptyAdjacentLocations(Location loc);

    ArrayList<Location> getOccupiedAdjacentLocations(Location loc);

    ArrayList<E> getNeighbors(Location loc);
}
```

# Bounded v. Unbounded

- Unbounded:

```
public boolean isValid(Location loc)
{
    return true;
}
```

- Bounded:

```
public boolean isValid(Location loc)
{
    return 0 <= loc.getRow() && loc.getRow() < getNumRows()
            && 0 <= loc.getCol() && loc.getCol() < getNumCols();
}
```

- Both fulfill the Grid contract

# Side note: abstract classes

- UnboundedGrid and BoundedGrid extend AbstractGrid
- AbstractGrid implements Grid
- AbstractGrid contains methods common to all implementations
- For example:

```
public ArrayList<E> getNeighbors(Location loc)
{
    ArrayList<E> neighbors = new ArrayList<E>();
    for (Location neighborLoc : getOccupiedAdjacentLocations(loc))
        neighbors.add(get(neighborLoc));
    return neighbors;
}
```