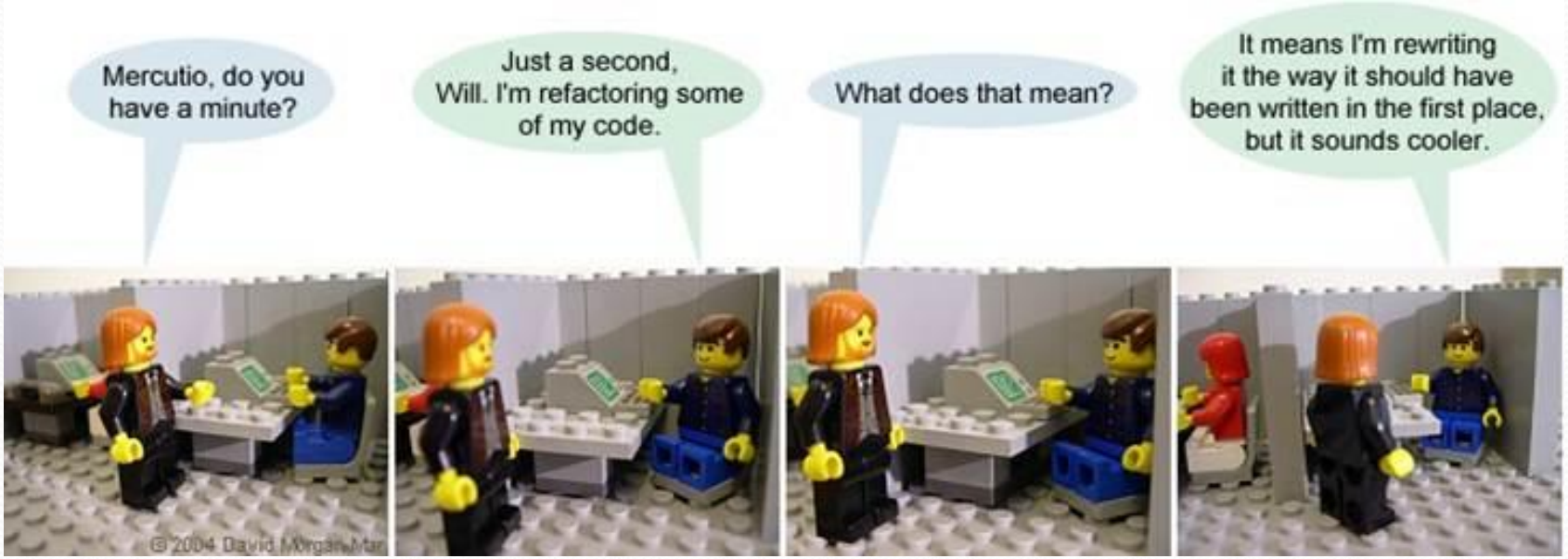
CS 142

Lecture 11: Polymorphism



Thanks to Marty Stepp and Stuart Reges for parts of these slides.

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.
  - `CritterMain` can interact with any type of critter.
    - Each one moves, fights, 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`.

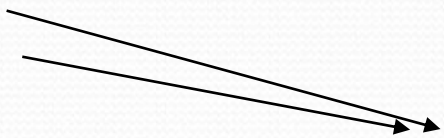
- When a method is called on `ed`, it behaves as a `Lawyer`.

```
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
v.days: 15
v.form: pink
```

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

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
```

"Polymorphism mystery"

- Figure out the output from all methods of these classes:

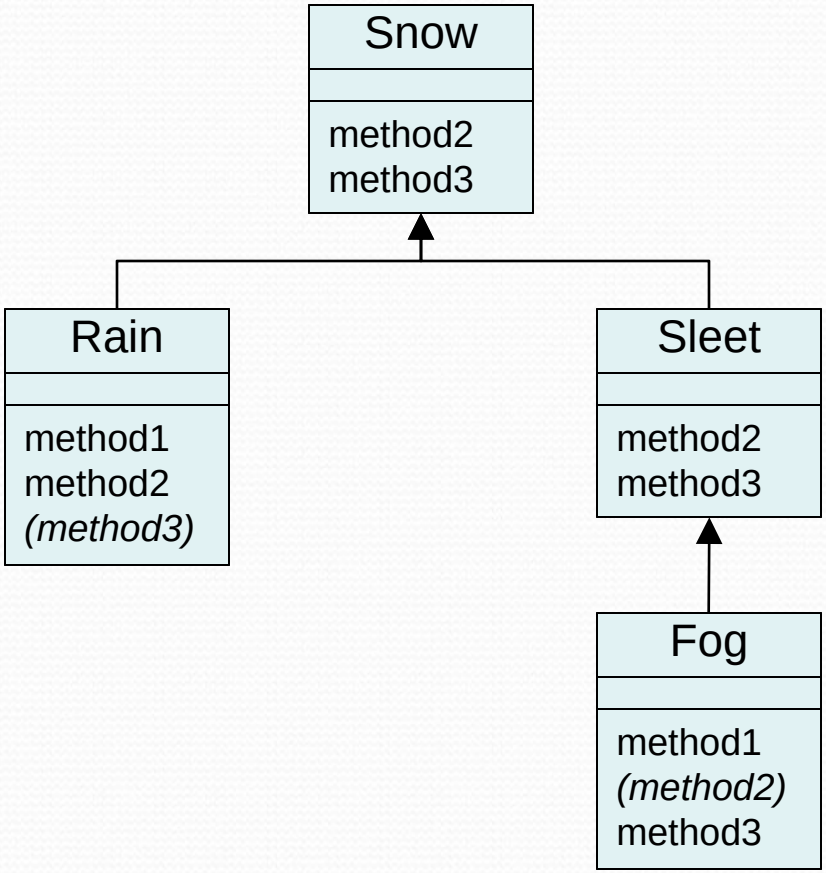
```
public class Snow {
    public void method2() {
        System.out.println("Snow 2");
    }

    public void method3() {
        System.out.println("Snow 3");
    }
}

public class Rain extends Snow {
    public void method1() {
        System.out.println("Rain 1");
    }

    public void method2() {
        System.out.println("Rain 2");
    }
}
```

"Polymorphism mystery"

```
public class Sleet extends Snow {
    public void method2() {
        System.out.println("Sleet 2");
        super.method2();
        method3();
    }

    public void method3() {
        System.out.println("Sleet 3");
    }
}

public class Fog extends Sleet {
    public void method1() {
        System.out.println("Fog 1");
    }

    public void method3() {
        System.out.println("Fog 3");
    }
}
```

Technique 1: diagram

- Diagram the classes from top (superclass) to bottom.



Technique 2: table

method	Snow	Rain	Sleet	Fog
method1		Rain 1		Fog 1
method2	Snow 2	Rain 2	Sleet 2 Snow 2 **method3()**	*Sleet 2* *Snow 2* ***method3()***
method3	Snow 3	*Snow 3*	Sleet 3	Fog 3

Italic - inherited behavior
Bold - dynamic method call

Mystery problem, no cast

```
Snow var3 = new Rain();
var3.method2();              // What's the output?
```

- If the problem does *not* have any casting, then:
  1. Look at the variable's type.
     If that type does not have the method: ERROR.
  2. Execute the method, behaving like the object's type.
     (The variable type no longer matters in this step.)

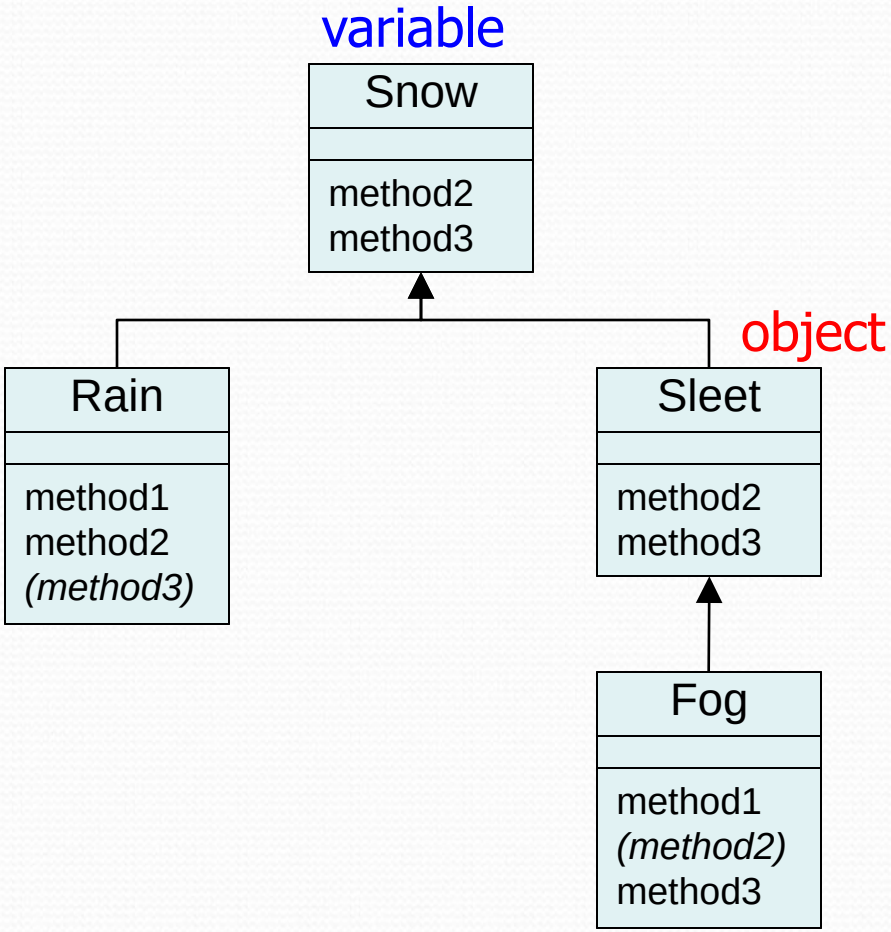
Example 1

- What is the output of the following call?

```
Snow var1 = new Sleet();
var1.method2();
```

- Answer:

```
Sleet 2
Snow 2
Sleet 3
```



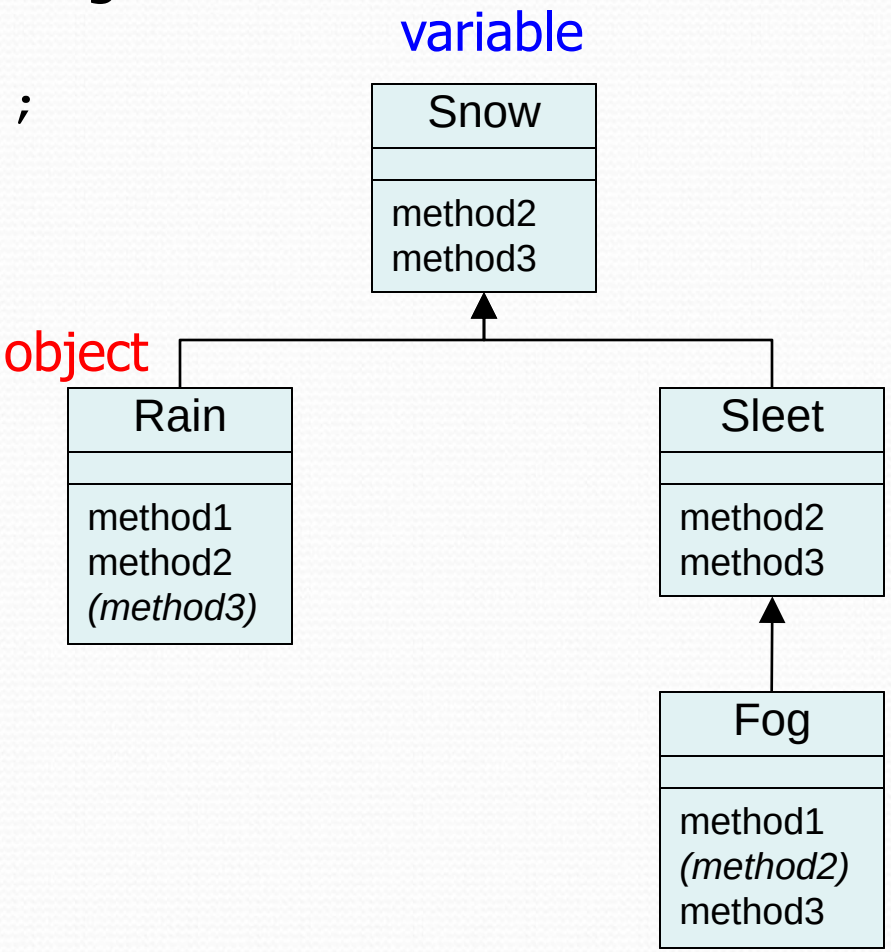
Example 2

- What is the output of the following call?

```
Snow var2 = new Rain();
var2.method1();
```

- Answer:

  ERROR
  (because `Snow` does not have a `method1`)



Mystery problem with cast

```
Snow var2 = new Rain();
((Sleet) var2).method2();    // What's the output?
```

- If the problem *does* have a type cast, then:
  1. Look at the cast type.
     If that type does not have the method: ERROR.
  2. Make sure the object's type is the cast type or is a subclass of the cast type. If not: ERROR. (No sideways casts!)
  3. Execute the method, behaving like the object's type.
     (The variable / cast types no longer matter in this step.)

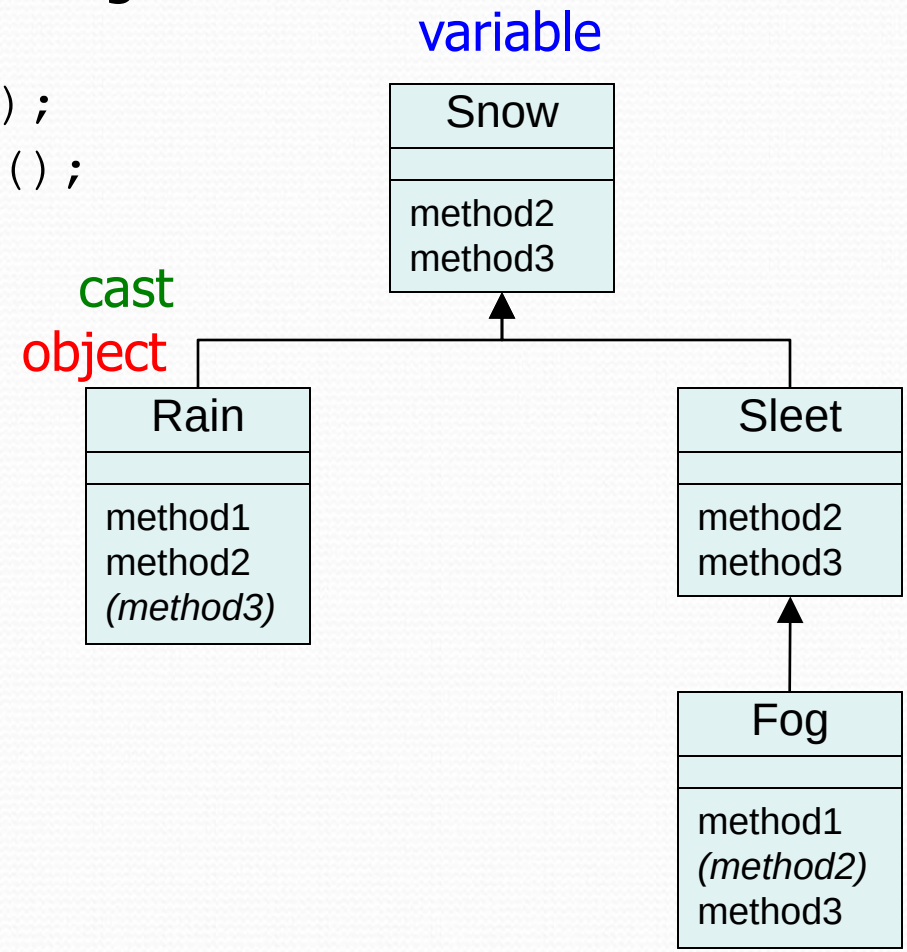
Example 3

- What is the output of the following call?

```
Snow var2 = new Rain();
((Rain) var2).method1();
```

- Answer:

```
Rain 1
```



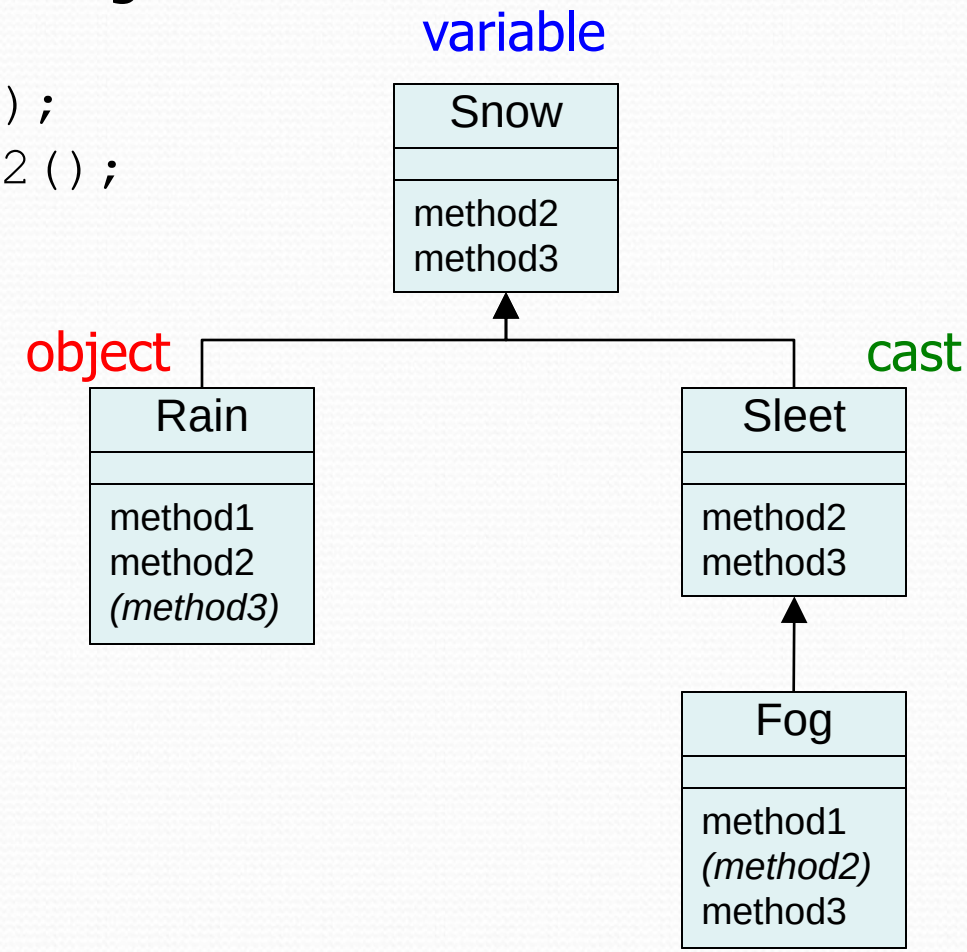
Example 4

- What is the output of the following call?

```
Snow var2 = new Rain();
((Sleet) var2).method2();
```

- Answer:

  ERROR
  (because the object's type, `Rain`, cannot be cast into `Sleet`)



Casting references

- A variable can only call that type's methods, not a subtype's.

```
Employee ed = new Lawyer();
int hours = ed.getHours();  // ok; this is in Employee
ed.sue();                   // compiler error
```

  - The compiler's reasoning is, variable `ed` could store any kind of employee, and not all kinds know how to `sue` .

- To use `Lawyer` methods on `ed`, we can type-cast it.

```
Lawyer theRealEd = (Lawyer) ed;
theRealEd.sue();                    // ok

((Lawyer) ed).sue();                // shorter version
```

More about casting

- The code crashes if you cast an object too far down the tree.

```
Employee eric = new Secretary();
((Secretary) eric).takeDictation("hi");      // ok
((LegalSecretary) eric).fileLegalBriefs();   // exception

// (Secretary object doesn't know how to file briefs)
```

- You can cast only up and down the tree, not sideways.

```
Lawyer linda = new Lawyer();
((Secretary) linda).takeDictation("hi");     // error
```

- Casting doesn't actually change the object's behavior.

  It just gets the code to compile/run.

```
((Employee) linda).getVacationForm()     // pink (Lawyer's)
```

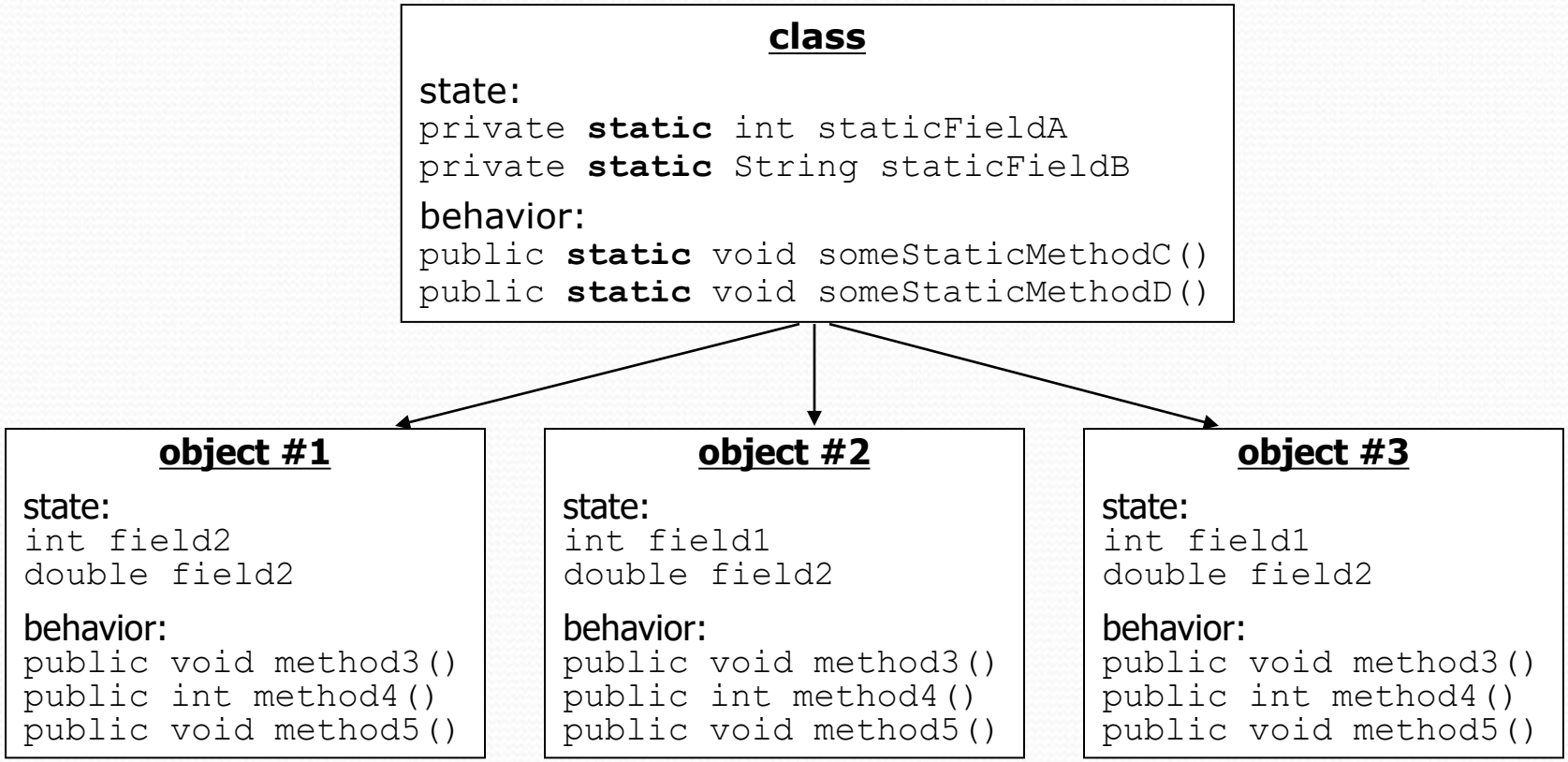
Exercise: BankAccount

How can we make sure that every bank account gets a unique id?

```
public class BankAccount {
    private String name;
    private int id;

    public BankAccount(int accountNumber) {
        id = accountNumber;  // give number to account
    }

    ...

    public int getID() {    // return this account's id
        return id;
    }
}
```

Static members

- **static**: Part of a class, rather than part of an object.
  - Object classes can have static methods *and fields*.
  - Not copied into each object; shared by all objects of that class.



Static fields

```
private static type name;
```

or,

```
private static type name = value;
```

- Example:

```
private static int theAnswer = 42;
```

- **static field**: Stored in the class instead of each object.
  - A "shared" global field that all objects can access and modify.
  - Like a class constant, except that its value can be changed.

Accessing static fields

- From inside the class where the field was declared:

```
fieldName                          // get the value
fieldName = value;                 // set the value
```

- From another class (if the field is `public`):

```
ClassName.fieldName                // get the value
ClassName.fieldName = value;       // set the value
```

  - generally static fields are not `public` unless they are `final`

- Exercise: Write the working version of `Giraffe`.

BankAccount solution

```
public class BankAccount {
    // static count of how many accounts are created
    // (only one count shared for the whole class)
    private static int objectCount = 0;

    // fields (replicated for each object)
    private String name;
    private int id;

    public BankAccount() {
        objectCount++;      // advance the id, and
        id = objectCount;   // give number to account
    }

    ...

    public int getID() {    // return this account's id
        return id;
    }
}
```

Static methods

```
// the same syntax you've already used for methods
public static type name(parameters) {
    statements;
}
```

- **static method**: Stored in a class, not in an object.
  - Shared by all objects of the class, not replicated.
  - Does not have any *implicit parameter*, `this`; therefore, cannot access any particular object's fields.

- Exercise: Make it so that clients can find out how many total `BankAccount` objects have ever been created.

BankAccount solution

```
public class BankAccount {
    // static count of how many accounts are created
    // (only one count shared for the whole class)
    private static int objectCount = 0;

    // clients can call this to find out # accounts created
    public static int getNumAccounts() {
        return objectCount;
    }

    // fields (replicated for each object)
    private String name;
    private int id;

    public BankAccount() {
        objectCount++;      // advance the id, and
        id = objectCount;   // give number to account
    }

    ...
    public int getID() {    // return this account's id
        return id;
    }
}
```