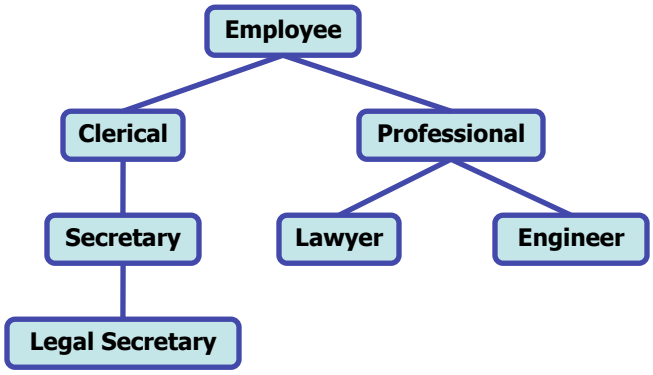
# CSE 143
# Lecture 13

## Inheritance

slides created by Ethan Apter
http://www.cs.washington.edu/143/

## Intuition: Employee Types

- Consider this (partial) hierarchy of employee types:




- What kind of tasks can each employee type perform?

2

## Intuition: Employee Types

- What tasks should *all* employees be able to do?
  - show up for work
  - work
  - collect paychecks
- What tasks can a lawyer do that an engineer cannot?
  - sue
  - file legal briefs
- Which kind of secretary (regular or legal) can accomplish a greater variety of tasks?
  - legal secretaries can do all regular secretarial work and have special training for additional tasks

3

## Intuition: Employee Training

- On your first day at work, you'll likely receive some general training for your new job
- If it's a big company (like Microsoft), you'll likely receive this with many other types of employees
  - engineers, business people, lawyers, etc
- After this general training, you'll probably receive some specialized training
  - "I know yesterday they told you to fill out your time-card on the white sheet, but here we do it online instead"
- We call this kind of replacement **overriding**
  - the new behavior overrides/replaces the old behavior

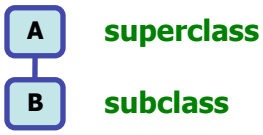
4

# Inheritance Overview

- Java does something similar with inheritance
- If we want to show an inheritance relationship between two classes, we use the **`extends`** keyword:

```
public class B extends A {
    ...
}
```

- Which sets up the following inheritance hierarchy:




5

# Superclasses and Subclasses

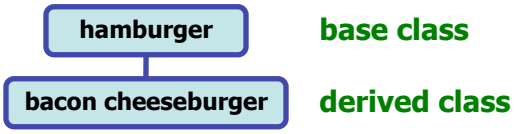
- In the previous example, **`A`** is the **superclass** of **`B`** (**`A`** is above **`B`** on the hierarchy), and **`B`** is a **subclass** of **`A`** (**`B`** is below **`A`** on the hierarchy)
- This wording is somewhat different from standard English:


- So, a "super bacon cheeseburger" is just a hamburger
  - that doesn't seem right: it's missing bacon and cheese!
  - but that's how inheritance works

6

# Base and Derived Classes

- We also say `A` is the **base class** of `B`, and `B` is a **derived class** of `A`
- This makes a little more sense:




- A hamburger provides the basic form of a bacon cheeseburger. Alternatively, a bacon cheeseburger is a hamburger with minor additions

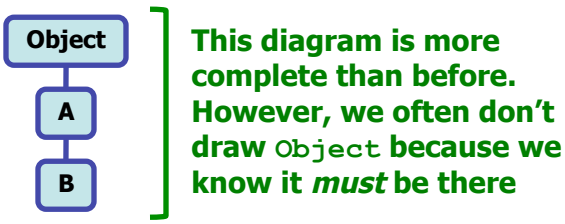
7

# Extending `Object`

- Consider the class `A` that we've been discussing:

```
public class A {
    ...
}
```

- We didn't write that `A` extends anything, but it automatically extends `Object`:




- All the classes you've written so far extend `Object`

8

# Object

- `Object` is a very general class
- Since every class must extend `Object`, either directly like `A` or indirectly like `B`, `Object` must have only state and behavior that is needed by every class:
  - `equals`
  - `toString`
    - this is where the weird default `toString` comes from
  - and more (but we won't bother with the others)

9

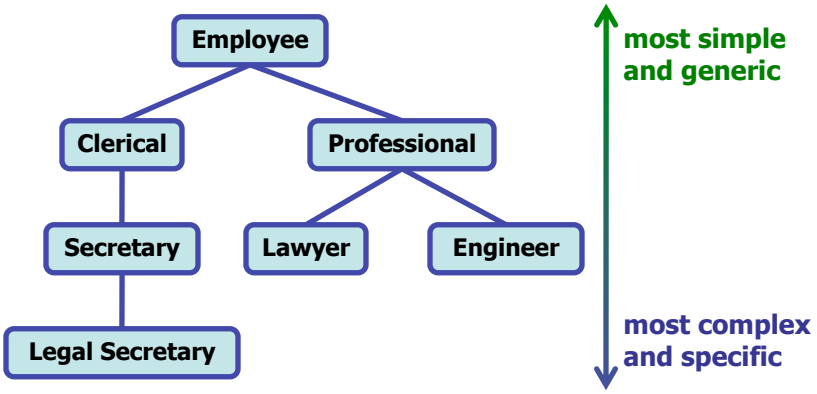
# Why Use Inheritance?

- Inheritance allows us to reuse code we've already written
  - this makes it a powerful tool
- Inheritance also allows us to express the core relationship between different classes
- Subclasses allow us to do two things:
  - add new state and behavior (fields and methods)
    - useful when a superclass has most of the needed state and behavior
  - override inherited methods
    - useful when you need to modify the way an inherited method functions

10

# Substituting

- When can we substitute one object for another?
- Recall our employee hierarchy:




11

# Substituting

- We can always substitute a more specific object for a less specific one. With inheritance, we call this an "is-a" relationship
  - a lawyer **is-a** professional employee **is-an** employee
    - a lawyer can substitute for an employee, etc
  - a legal secretary **is-a** secretary **is-a** clerical employee **is-an** employee
    - a legal secretary can substitute for a secretary, etc
- You can see the is-a relationship by moving up the inheritance hierarchy
- It's *not* ok to substitute across the hierarchy
  - a secretary is NOT a lawyer and can't substitute for one

12

# Substituting

- Recall our classes `B` and `A` (`B extends A`)
- Obviously we can do this:

```
A x = new A();
B y = new B();
```

variable type  object type

- But what if the variable type and object type don't match?

```
A x = new B();
```
perfectly fine

```
B y = new A();
```
not good!

13

# Substituting

- But, what does it mean when the variable type doesn't match the object type?

```
A x = new B();
```

- We are limited to the behaviors of the variable type
  - for `x` above, we are limited to using only methods defined for class `A` (which may be inherited from `Object`)
- When executed, the methods will behave as defined in the object type
  - for `x` above, the methods will execute as defined in `B` (which may be inherited from `A` or `Object`)

14

# Casting

- Suppose that:
  - you're an unemployed legal secretary
  - you know that legal secretaries earn $20 an hour
  - you know that generic secretaries earn $15 an hour
  - you accept a job as a generic secretary for $15 an hour
- So far, this is fine (just not ideal)
- But what if:
  - your employer discovers that you're a legal secretary
  - ...and wants you to do legal secretary work
  - ...for just $15 an hour?
- Is that ok?

15

# Casting

- No, it's not ok!
- If he wants you to do legal secretary work, he can renegotiate your contract to reflect this
  - and pay you $20 an hour
- Java lets us do something similar when we class cast
  - the class cast essentially renegotiates the contract

```
Secretary you = new LegalSecretary();
LegalSecretary youWithRaise = (LegalSecretary)you;
```

**cast**

16

# Exercise: Inheritance Mystery

- 4-5 classes with inheritance relationships are shown
  - the class names *won't* make sense (inheritance *mystery*)
- A client program calls methods on objects of each class
  - some questions involve casting
  - some lines of code are illegal and produce errors
- You must read the code and determine what it does
  - if it produces output, you must be precise
  - if it produces an error, you need to specify what kind of error (either a compiler error or a runtime error)
- A similar problem **will** be on your midterm!

17

# Exercise: Inheritance Mystery

- Assume that the following classes have been declared:

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

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


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

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

18

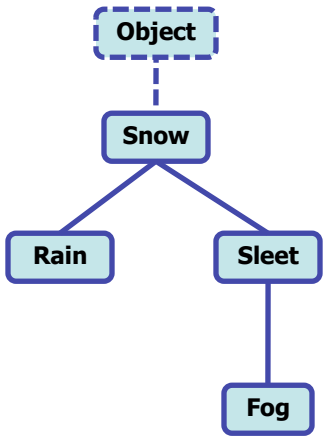
# Exercise: Inheritance Mystery

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

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

19

# Technique: Diagramming

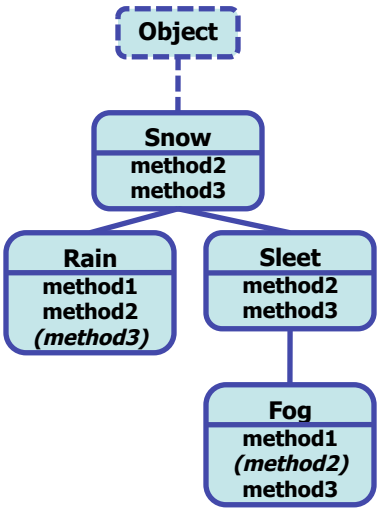
- First, determine the inheritance hierarchy:




20

# Technique: Diagramming

- ...and determine where methods are defined and inherited




21

# Method Calls

- Let's look a little closer at `Sleet`'s `method2()`:

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

- `super` is a Java keyword to look to the super class
  - so `super.method2()` is like saying `Snow.method2()`
  - `super` is static: it ALWAYS refers to `Snow`'s methods

- however, the call on `method3` is dynamic
  - it always runs the current version of `method3`, because it's possible that another subclass can redefine `method3`

22

# Technique: Behavior Table

- Then, figure out the behaviors of each type of object

| method | Snow | Rain | Sleet | Fog |
|---|---|---|---|---|
| method1 | --- | Rain 1 | --- | Fog 1 |
| method2 | Snow 2 | Rain 2 | Sleet 2<br>Snow 2<br>method3() | *Sleet 2*<br>*Snow 2*<br>*method3()* |
| method3 | Snow 3 | *Snow 3* | Sleet 3 | Fog 3 |

*Underlined Italics* - inherited behavior

Circled - dynamic method call

23

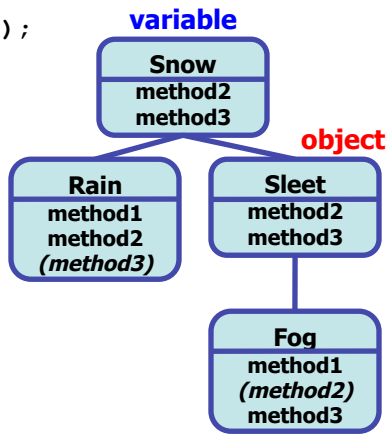
# Example 1

- Problem:

```
Snow var1 = new Sleet();
var1.method3();
```

- Output:

```
Sleet 3
```




24

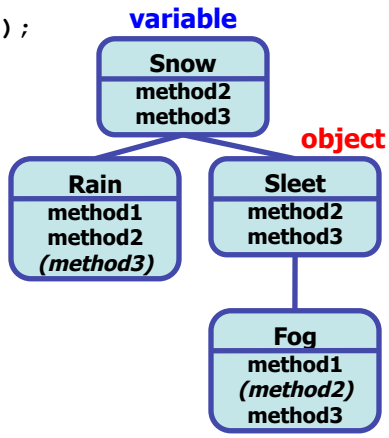
# Example 2

- Problem:

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

- Output:

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






# Example 3

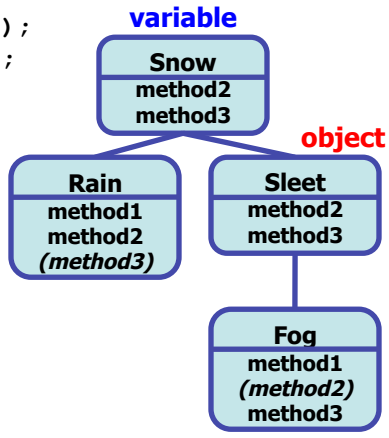
- Problem:

```
Snow var1 = new Sleet();
((Snow)var1).method3();
```

- Output:

```
Sleet 3
```

The actual object is still a `Sleet`!



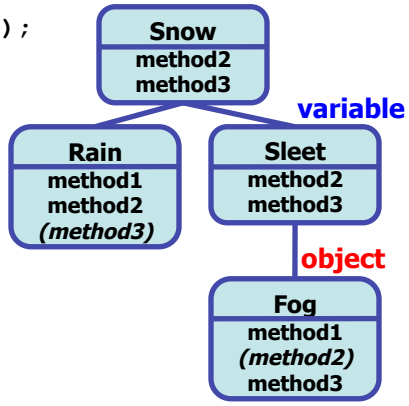
# Example 4

- Problem:

```
Snow var4 = new Fog();
((Sleet)var4).method1();
```

- Output:

```
No output!
Compiler error!
Sleet doesn't have a
method1()!
```




27

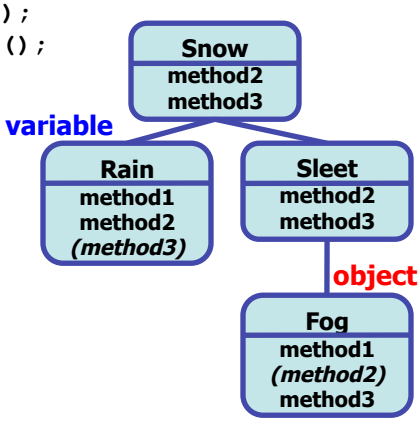
# Example 5

- Problem:

```
Snow var4 = new Fog();
((Rain)var4).method1();
```

- Output:

```
No output!
Runtime error!
A Fog is not a Rain!
```




28

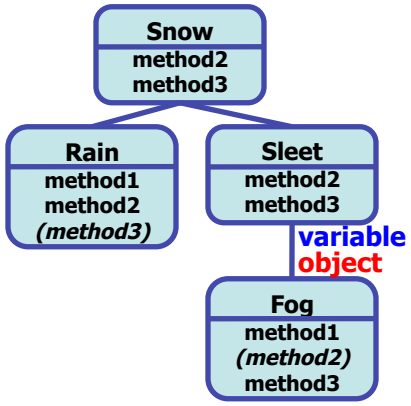
# Example 6

- Problem:

```
Snow var4 = new Fog();
((Fog)var4).method1();
```

- Output:

```
Fog 1
```




29

# Solving Inheritance Mystery

- Steps to solving inheritance mystery:

1. Look at the variable type (if there is a cast, look at the casted variable type). If the variable type does not have the requested method the compiler will report an error.

2. If there was a cast, make sure the casted variable type is compatible with the variable type (i.e. ensure the casted variable type is a superclass or subclass of the variable type). If they are not compatible, the compiler will report an error. If the cast is a subclass of the actual object type, a runtime error (`ClassCastException`) will occur.

3. Execute the method in question, behaving like the object type (the variable type and casted variable type no longer matter at all)

30

# Exercise

- Assume the following declarations:

```
Snow var1 = new Sleet();
Rain var2 = new Rain();
Snow var3 = new Fog();
Object var4 = new Snow();
Sleet var5 = new Fog();
Snow var6 = new Rain();
```



# Exercise

| Letter | Code | Output |
|---|---|---|
| a | `var1.method1();` | `compiler error` |
| b | `var2.method1();` | `Rain 1` |
| c | `var1.method2();` | `Sleet 2/Snow 2/Sleet 3` |
| d | `var2.method2();` | `Rain 2` |
| e | `var3.method2();` | `Sleet 2/Snow 2/Fog 3` |
| f | `var4.method2();` | `compiler error` |
| g | `var5.method2();` | `Sleet 2/Snow 2/Fog 3` |
| h | `var1.method3();` | `Sleet 3` |
| i | `var2.method3();` | `Snow 3` |
| j | `var3.method3();` | `Fog 3` |

# Exercise

| Letter | Code | Output |
|---|---|---|
| k | `var4.method3();` | compiler error |
| l | `var5.method3();` | Fog 3 |
| m | `((Rain)var3).method1();` | runtime error |
| n | `((Fog)var5).method1();` | Fog 1 |
| o | `((Sleet)var3).method1();` | compiler error |
| p | `((Sleet)var3).method3();` | Fog 3 |
| q | `((Fog)var6).method3();` | runtime error |
| r | `((Snow)var4).method2();` | Snow 2 |
| s | `((Sleet)var4).method3();` | runtime error |
| t | `((Rain)var6).method3();` | Snow 3 |

33