

Inheritance

Exam-Level 03

Example Agenda

- 1:10 - 1:15 ~ announcements
- 1:15 - 1:30 ~ content review
- 1:30 - 1:40 ~ question 1
- 1:40 - 1:55 ~ question 2

Announcements

- Midterm 1 on Thursday 2/15 7-9 PM
 - Review Session Friday 2/9 11-1PM in Soda labs
- No lab assignment this week (Project 1 Workday)
- Project 1A due this Monday 2/5
- Project 1B due next Monday 2/12
- Project 1C due Tuesday 2/20
- Weekly Survey 3 due this Monday 2/5

Content Review

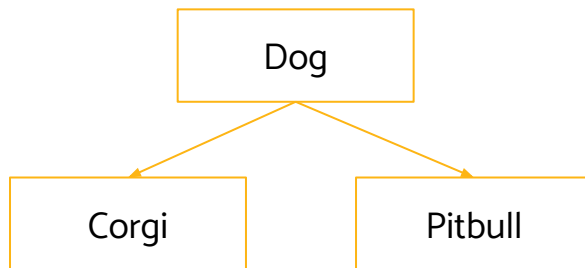
Classes

Subclasses (or child classes) are classes that inherit from another class. This means that they have access to all of the non-private functions and variables of their parent class in addition to any functions and variables defined in the child class.

Example: Corgi, Pitbull

Superclasses or parent classes are classes that are inherited by another class.

Example: Dog



Fun with Methods

Method Overloading is done when there are multiple methods with the same name, but different parameters.

```
public void barkAt(Dog d) { System.out.print("Woof, it's another dog!"); }  
public void barkAt(CS61BStaff s) { System.out.print("Woof, what is this?"); }
```

* Food for thought: what is an advantage of method overloading? Hint: think about `System.out.print`

Method Overriding is done when a subclass has a method with the exact same function signature as a method in its superclass. It is usually marked with the `@Override` tag.

In Dog class:

```
public void speak() { System.out.print("Woof, I'm a dog!"); }
```

In Corgi Class, which inherits from Dog:

```
@Override
```

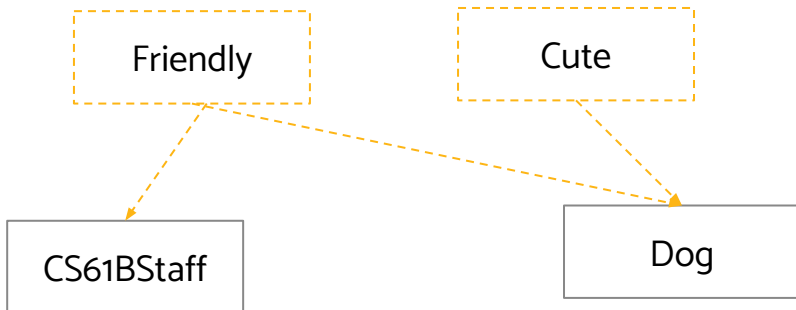
```
public void speak() { System.out.print("Woof, I'm a corgi!"); }
```

Interfaces

Interfaces are **implemented** by classes. They describe a narrow ability that can apply to many classes that may or may not be related to one another.

They do not usually implement the methods they specify, but can do so with the `default` keyword. Interface methods are inherently **public**, which must be specified in the subclass that implements them (subclasses must override and implement non-default interface methods).

Interfaces cannot be instantiated. (ie. `Friendly f = new Friendly();` does not compile)



Interfaces vs. Classes

- A class can **implement** many interfaces and **extend** only one class
- Interfaces tell us what we want to do but not how; classes tell us how we want to do it
- Interfaces can have empty method bodies (that must be filled in by subclasses) or default methods (do not need to be overridden by subclasses)
- With extends, subclasses inherit their parent's instance and static variables, methods (can be overridden), nested classes
 - But not constructors!
 - Use **super** to refer to the parent class

Implementation

```
interface Cute {...}
```

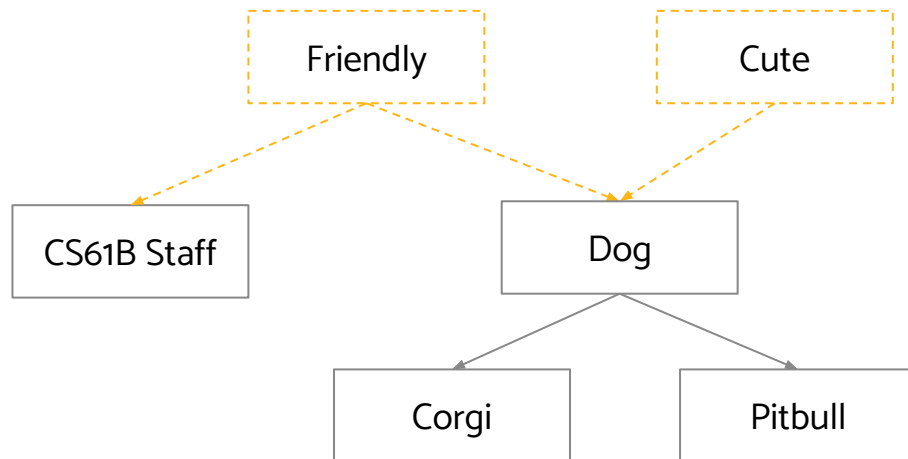
```
interface Friendly {...}
```

```
class CS61BStaff implements Friendly {...}
```

```
class Dog implements Cute, Friendly {...}
```

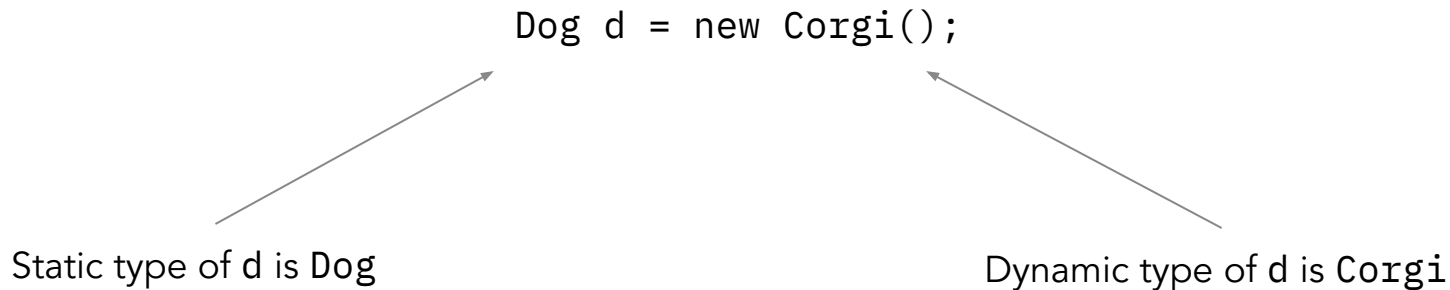
```
class Corgi extends Dog {...}
```

```
class Pitbull extends Dog {...}
```



Static vs. Dynamic Type

A variable's **static type** is specified at declaration, whereas its **dynamic type** is specified at instantiation (e.g. when using `new`).



The static and dynamic type of a variable have to complement each other or else the code will error. For example, a `Dog` is not necessarily a `Corgi`, so `Corgi c = new Dog();` will not compile.

General rule of thumb: Given `LHS = RHS`, is `RHS` guaranteed to be a `LHS`?

Though interfaces cannot be instantiated, they can be static types (ie. `Cute c = new Corgi();`)

Casting

Casting allows us to tell the compiler to treat the static type of some variable as whatever we want it to be (need to have a superclass/subclass relationship). If the cast is valid, for that line only we will treat the static type of the casted variable to be whatever we casted it to.

```
Animal a = new Dog();  
Dog d = a;      // Compiler error: an animal is not a dog  
Dog d = (Dog) a;    // Valid cast: an animal could reasonably be a dog  
d = new Dog();  
a = (Animal) d    // Valid cast: a dog definitely is-a animal  
Cat c = new Cat();  
d = (Dog) c;      // Compiler error: a cat is definitely not a dog  
a = c;  
d = (Dog) a;      // Cast compiles because an animal could reasonably be a dog.  
                During runtime, errors
```

All these concepts - What's the point?

It allows for **Subtype Polymorphism**. (You'll also see this in lecture this week).

Polymorphism means “providing a single interface to entities of different types”

Example:

Consider a variable deque of static type Deque:

When you call `deque.addFirst()`, the actual behavior is based on the dynamic type.

```
Deque deque = new LinkedListDeque();// Runs LinkedListDeque's addFirst
```

```
Deque deque = new ArrayDeque();// Runs ArrayDeque's addFirst
```

Java automatically selects the right behavior using what is sometimes called “dynamic method selection”.

Dynamic Method Selection

Your computer. . .

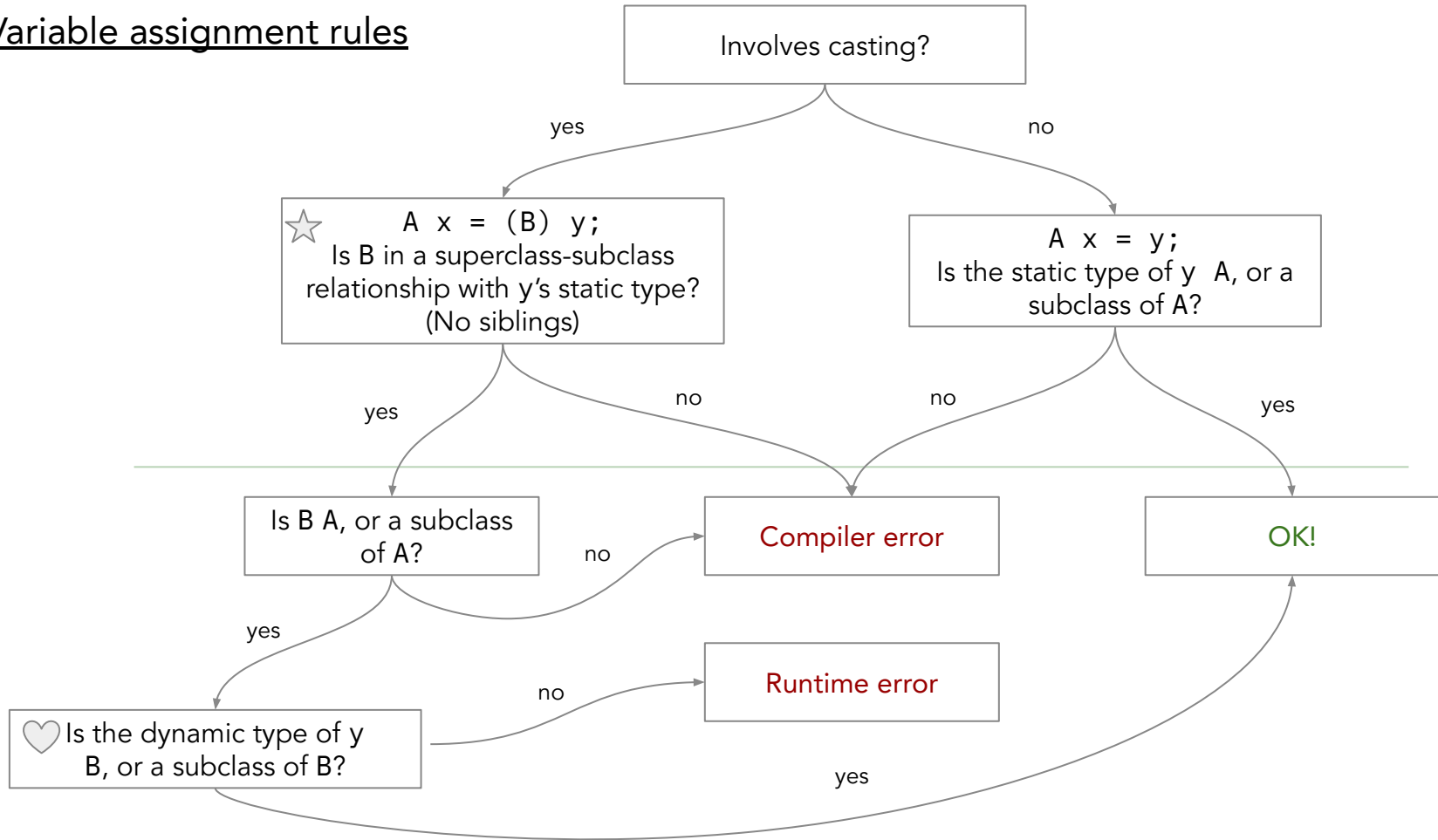
@ Compile Time, we only care about static type of the invoking / calling instance:

1. Check for valid variable assignments
2. Check for valid method calls (only considering static type and static types superclass(es))
 - a. Lock in exact method signature as soon as we find an adequate one, traversing parent classes
 - b. “adequate” means (1) same name, (2) argument type (actual) is-a parameter type (defined).
3. If nothing found, compiler error

@ Run Time, we care about dynamic type of the invoking / calling instance:

1. If the locked-in method is static, skip the step below and just run that method
2. Check for overridden methods
 - a. Does the locked-in method signature have an identical one in the dynamic class or the dynamic class's parent classes?
3. Ensure casted objects can be assigned to their variables

Variable assignment rules



Worksheet

1 Forget It, We Ball

```
interface Person {  
    void speakTo(Person other);  
    default void watch(Athlete other) { System.out.println("wow"); }  
}  
  
public class Athlete implements Person {  
    @Override  
    public void speakTo(Person other) { System.out.println("i love sports"); }  
    @Override  
    public void watch(Athlete other) { System.out.println("ball is life"); }  
}  
  
public class SoccerPlayer extends Athlete {  
    @Override  
    void speakTo(Person other) { System.out.println("join 61ballers"); }  
}
```

For each line, write what, if anything, is printed after its execution. Write CE if there is a compiler error and RE if there is a runtime error. If a line errors, continue executing the rest of the lines.

1 Forget It, We Ball

```
1 Person ayati = new Person();  
  
3 Athlete aniruth = new SoccerPlayer();  
  
5 SoccerPlayer vanessa = aniruth;  
  
7 Person eric = new Athlete();  
  
9 Athlete shreyas = new Athlete();  
  
11 SoccerPlayer yaofu = new SoccerPlayer();
```

1 Forget It, We Ball

```
1 Person ayati = new Person(); // CE - can't new interface  
  
3 Athlete aniruth = new SoccerPlayer();  
  
5 SoccerPlayer vanessa = aniruth;  
  
7 Person eric = new Athlete();  
  
9 Athlete shreyas = new Athlete();  
  
11 SoccerPlayer yaofu = new SoccerPlayer();
```

1 Forget It, We Ball

```
1 Person ayati = new Person(); // CE - can't new interface

3 Athlete aniruth = new SoccerPlayer(); // Athlete, SoccerPlayer

5 SoccerPlayer vanessa = aniruth;

7 Person eric = new Athlete();

9 Athlete shreyas = new Athlete();

11 SoccerPlayer yaoфу = new SoccerPlayer();
```

1 Forget It, We Ball

```
1 Person ayati = new Person(); // CE - can't new interface

3 Athlete aniruth = new SoccerPlayer(); // Athlete, SoccerPlayer

5 SoccerPlayer vanessa = aniruth; // CE

7 Person eric = new Athlete();

9 Athlete shreyas = new Athlete();

11 SoccerPlayer yaoфу = new SoccerPlayer();
```

1 Forget It, We Ball

```
1 Person ayati = new Person(); // CE - can't new interface

3 Athlete aniruth = new SoccerPlayer(); // Athlete, SoccerPlayer

5 SoccerPlayer vanessa = aniruth; // CE

7 Person eric = new Athlete(); // Person, Athlete

9 Athlete shreyas = new Athlete();

11 SoccerPlayer yaofu = new SoccerPlayer();
```

1 Forget It, We Ball

```
1 Person ayati = new Person(); // CE - can't new interface

3 Athlete aniruth = new SoccerPlayer(); // Athlete, SoccerPlayer

5 SoccerPlayer vanessa = aniruth; // CE

7 Person eric = new Athlete(); // Person, Athlete

9 Athlete shreyas = new Athlete(); // Athlete, Athlete

11 SoccerPlayer yaofu = new SoccerPlayer();
```

1 Forget It, We Ball

```
1 Person ayati = new Person(); // CE - can't new interface

3 Athlete aniruth = new SoccerPlayer(); // Athlete, SoccerPlayer

5 SoccerPlayer vanessa = aniruth; // CE

7 Person eric = new Athlete(); // Person, Athlete

9 Athlete shreyas = new Athlete(); // Athlete, Athlete

11 SoccerPlayer yaoфу = new SoccerPlayer(); // SoccerPlayer,
    SoccerPlayer
```

1 Forget It, We Ball

```
13 eric.watch(aniruth);  
  
15 shreyas.speakTo(yaofu);  
  
17 yaofu.speakTo(eric);  
  
19 ((Athlete) yaofu).speakTo(eric);  
  
21 ((Person) yaofu).speakTo(eric);  
  
23 ((Athlete) eric).speakTo(shreyas);  
  
25 ((SoccerPlayer) eric).watch(yaofu);
```

```
Athlete aniruth = new SoccerPlayer();
```

```
Person eric = new Athlete();
```

```
Athlete shreyas = new Athlete();
```

```
SoccerPlayer yaofu = new SoccerPlayer();
```

1 Forget It, We Ball

```
13 eric.watch(aniruth); // Person.watch, Athlete.watch → ball is life
15 shreyas.speakTo(yaofu);
17 yaofu.speakTo(eric);
19 ((Athlete) yaofu).speakTo(eric);
21 ((Person) yaofu).speakTo(eric);
23 ((Athlete) eric).speakTo(shreyas);
25 ((SoccerPlayer) eric).watch(yaofu);
```

1 Forget It, We Ball

```
13 eric.watch(aniruth); // Person.watch, Athlete.watch → ball is life
15 shreyas.speakTo(yaofu); // Athlete.speakTo, Athlete.speakTo → i love
sports
17 yaofu.speakTo(eric);
19 ((Athlete) yaofu).speakTo(eric);
21 ((Person) yaofu).speakTo(eric);
23 ((Athlete) eric).speakTo(shreyas);
25 ((SoccerPlayer) eric).watch(yaofu);
```

1 Forget It, We Ball

```
13 eric.watch(aniruth); // Person.watch, Athlete.watch → ball is life
15 shreyas.speakTo(yaofu); // Athlete.speakTo, Athlete.speakTo → i love
sports
17 yaofu.speakTo(eric); // SoccerPlayer.speakTo, SoccerPlayer.speakTo →
join 61ballers
19 ((Athlete) yaofu).speakTo(eric);
21 ((Person) yaofu).speakTo(eric);
23 ((Athlete) eric).speakTo(shreyas);
25 ((SoccerPlayer) eric).watch(yaofu);
```

1 Forget It, We Ball

```
13 eric.watch(aniruth); // Person.watch, Athlete.watch → ball is life
15 shreyas.speakTo(yaofu); // Athlete.speakTo, Athlete.speakTo → i love
sports
17 yaofu.speakTo(eric); // SoccerPlayer.speakTo, SoccerPlayer.speakTo →
join 61ballers
19 ((Athlete) yaofu).speakTo(eric); // Athlete.speakTo,
SoccerPlayer.speakTo → join 61ballers
21 ((Person) yaofu).speakTo(eric);
23 ((Athlete) eric).speakTo(shreyas);
25 ((SoccerPlayer) eric).watch(yaofu);
```

1 Forget It, We Ball

```
13 eric.watch(aniruth); // Person.watch, Athlete.watch → ball is life
15 shreyas.speakTo(yaofu); // Athlete.speakTo, Athlete.speakTo → i love
sports
17 yaofu.speakTo(eric); // SoccerPlayer.speakTo, SoccerPlayer.speakTo →
join 61ballers
19 ((Athlete) yaofu).speakTo(eric); // Athlete.speakTo,
SoccerPlayer.speakTo → join 61ballers
21 ((Person) yaofu).speakTo(eric); // Person.speakTo, SoccerPlayer.speakTo
→ join 61ballers
23 ((Athlete) eric).speakTo(shreyas);
25 ((SoccerPlayer) eric).watch(yaofu);
```

1 Forget It, We Ball

```
13 eric.watch(aniruth); // Person.watch, Athlete.watch → ball is life
15 shreyas.speakTo(yaofu); // Athlete.speakTo, Athlete.speakTo → i love
sports
17 yaofu.speakTo(eric); // SoccerPlayer.speakTo, SoccerPlayer.speakTo →
join 61ballers
19 ((Athlete) yaofu).speakTo(eric); // Athlete.speakTo,
SoccerPlayer.speakTo → join 61ballers
21 ((Person) yaofu).speakTo(eric); // Person.speakTo, SoccerPlayer.speakTo
→ join 61ballers
23 ((Athlete) eric).speakTo(shreyas); // Athlete.speakTo, Athlete.speakTo →
i love sports
25 ((SoccerPlayer) eric).watch(yaofu);
```

1 Forget It, We Ball

```
13 eric.watch(aniruth); // Person.watch, Athlete.watch → ball is life
15 shreyas.speakTo(yaofu); // Athlete.speakTo, Athlete.speakTo → i love sports
17 yaofu.speakTo(eric); // SoccerPlayer.speakTo, SoccerPlayer.speakTo → join
61ballers
19 ((Athlete) yaofu).speakTo(eric); // Athlete.speakTo, SoccerPlayer.speakTo →
join 61ballers
21 ((Person) yaofu).speakTo(eric); // Person.speakTo, SoccerPlayer.speakTo →
join 61ballers
23 ((Athlete) eric).speakTo(shreyas); // Athlete.speakTo, Athlete.speakTo → i
love sports
25 ((SoccerPlayer) eric).watch(yaofu); // Athlete.watch, RE
```

1 Forget It, We Ball

(Optional Challenge) Describe how you would use casting to fix the following lines as described, or explain why it is not possible.

1. Allow `SoccerPlayer vanessa = aniruth;` to compile.
2. `((SoccerPlayer) eric).watch(yaofu);` so that `wow` is printed.
3. `((SoccerPlayer) eric).watch(yaofu);` so that `i love sports` is printed.

1 Forget It, We Ball

Describe how you would use casting to fix the following lines as described, or explain why it is not possible.

1. Allow `SoccerPlayer vanessa = aniruth;` to compile.

`SoccerPlayer vanessa = (SoccerPlayer) aniruth;`

2. `((SoccerPlayer) eric).watch(yaofu);` so that `wow` is printed.
3. `((SoccerPlayer) eric).watch(yaofu);` so that `i love sports` is printed.

1 Forget It, We Ball

Describe how you would use casting to fix the following lines as described, or explain why it is not possible.

1. Allow `SoccerPlayer vanessa = aniruth;` to compile.

`SoccerPlayer vanessa = (SoccerPlayer) aniruth;`

2. `((SoccerPlayer) eric).watch(yaofu);` so that `wow` is printed.

Not possible, since at runtime `eric` is an `Athlete`, which overrides the `watch` method.

3. `((SoccerPlayer) eric).watch(yaofu);` so that `i love sports` is printed.

1 Forget It, We Ball

Describe how you would use casting to fix the following lines as described, or explain why it is not possible.

1. Allow `SoccerPlayer vanessa = aniruth;` to compile.

`SoccerPlayer vanessa = (SoccerPlayer) aniruth;`

2. `((SoccerPlayer) eric).watch(yaofu);` so that `wow` is printed.

Not possible, since at runtime `eric` is an `Athlete`, which overrides the `watch` method.

3. `((SoccerPlayer) eric).watch(yaofu);` so that `i love sports` is printed.

Not possible. “i love sports” is only defined in the `speakTo` method, but we never interact with it in any way.

2 List Inheritance

Modify the code below so that the max method of `DMSList` works properly. Assume all numbers inserted into `DMSList` are positive, and we only insert using `insertFront`.

```
public class DMSList {
    private IntNode sentinel;
    public DMSList() {
        sentinel = new IntNode(-1000, _____);
    }
    public class IntNode {
        // IntNode definition here
    }
    class LastIntNode extends IntNode {
        // LastIntNode definition here
    }
    public int max() {
        return sentinel.next.max();
    }
    public void insertFront(int x) {
        // insert code
    }
}
```

```
class LastIntNode extends IntNode {
    public LastIntNode() {
        _____;
    }

    @Override
    public int max() {
        _____;
    }
}
```

2 List Inheritance

Modify the code below so that the max method of `DMSList` works properly. Assume all numbers inserted into `DMSList` are positive, and we only insert using `insertFront`.

```
public DMSList() {  
    sentinel = new IntNode(-1000,  
new LastIntNode());  
  
    // other code here  
}
```

```
class LastIntNode extends IntNode {  
    public LastIntNode() {  
        -----;  
    }  
  
    @Override  
    public int max() {  
        -----;  
    }  
}
```

2 List Inheritance

Modify the code below so that the max method of `DMSList` works properly. Assume all numbers inserted into `DMSList` are positive, and we only insert using `insertFront`.

```
public DMSList() {  
    sentinel = new IntNode(-1000,  
new LastIntNode());  
  
    // other code here  
}
```

```
class LastIntNode extends IntNode {  
    public LastIntNode() {  
        super(0, null);  
    }  
  
    @Override  
    public int max() {  
        -----;  
    }  
}
```

2 List Inheritance

Modify the code below so that the max method of `DMSList` works properly. Assume all numbers inserted into `DMSList` are positive, and we only insert using `insertFront`.

```
public DMSList() {  
    sentinel = new IntNode(-1000,  
new LastIntNode());  
  
    // other code here  
}
```

```
class LastIntNode extends IntNode {  
    public LastIntNode() {  
        super(0, null);  
    }  
  
    @Override  
    public int max() {  
        return 0;  
    }  
}
```