Inheritance

Exam-Level 04

Agenda

- 9:10 - 9:15 ~ announcements
- 9:15 - 9:30 ~ content review
- 9:40 - 1:55 ~ question 2

Announcements

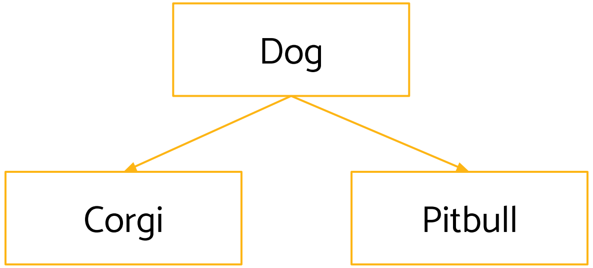
- Midterm 1 on Thursday 9/21 7-9 PM
    - Review Session Friday 9/15 11-1PM in Soda labs
- No lab assignment this week (Project 1 Workday)
- Project 1A due this Monday 9/11
- Project 1B due next Monday 9/18
- Project 1C due next Monday 9/25
- Weekly Survey 3 due this Monday 9/11

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:

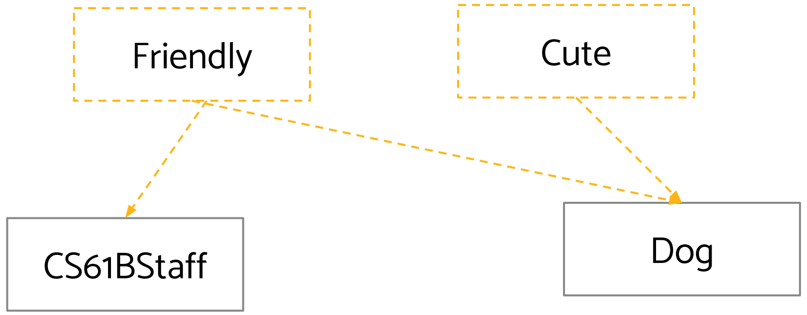
```
@Override
public void speak() { System.out.print("Woof, I'm a corgi!"); }
```

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)

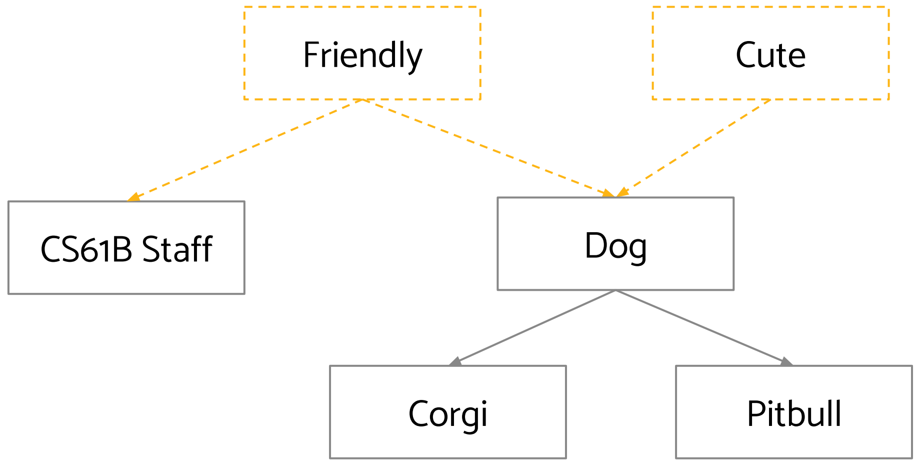


Interfaces vs. Classes

- 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).

```
Dog d = new Corgi();
```

Static type of `d` is `Dog`

Dynamic type of `d` is `Corgi`

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. . .

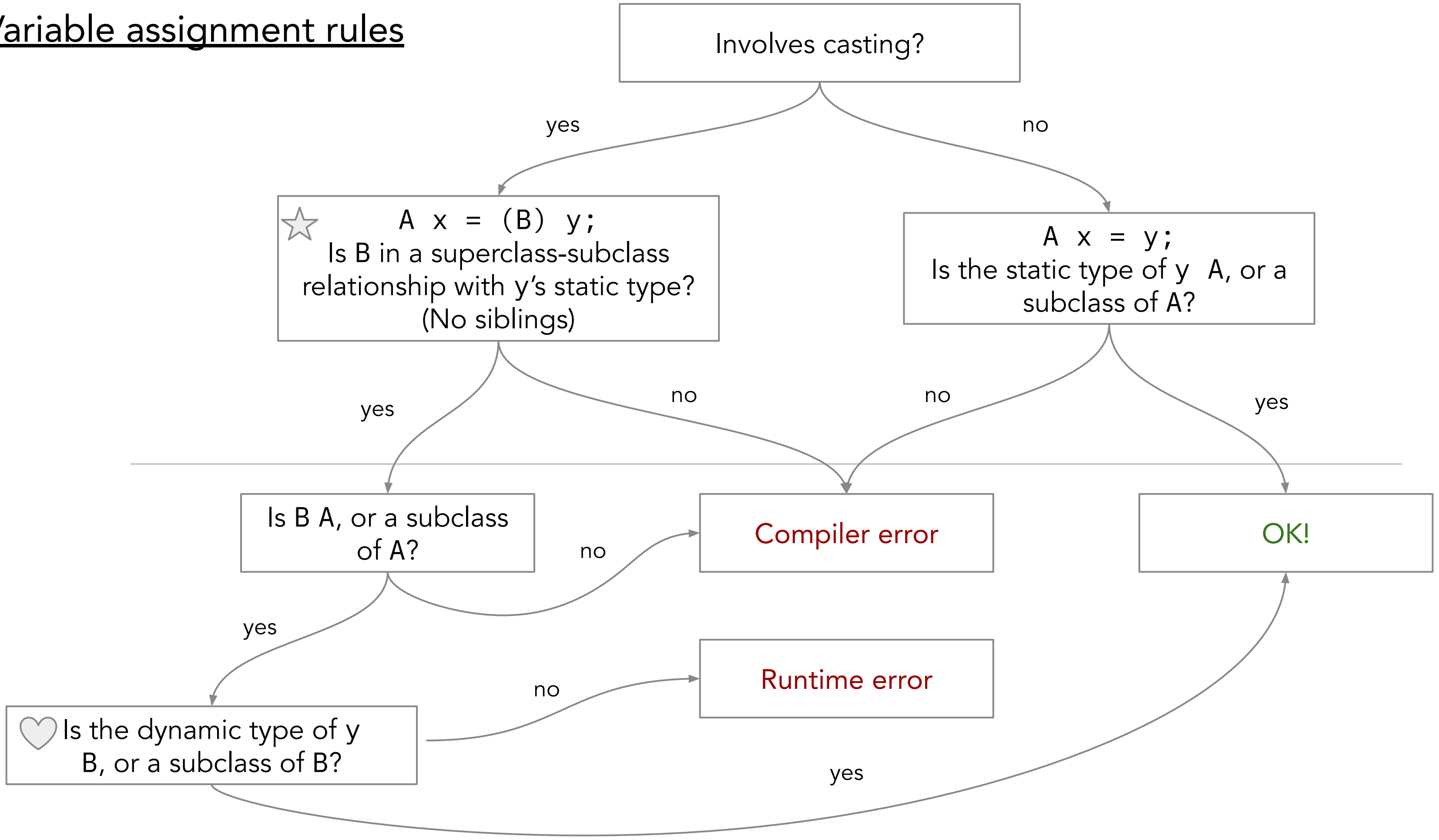
@ Compile Time, we only care about static type of the invoking / calling instance:

1. Check for valid variable assignments
2. Check for valid method calls (<u>only considering static type and static types superclass(es)</u>)
    a. Lock in exact method signature as soon as we find an adequate one, traversing parent classes
3. If nothing found, compiler error

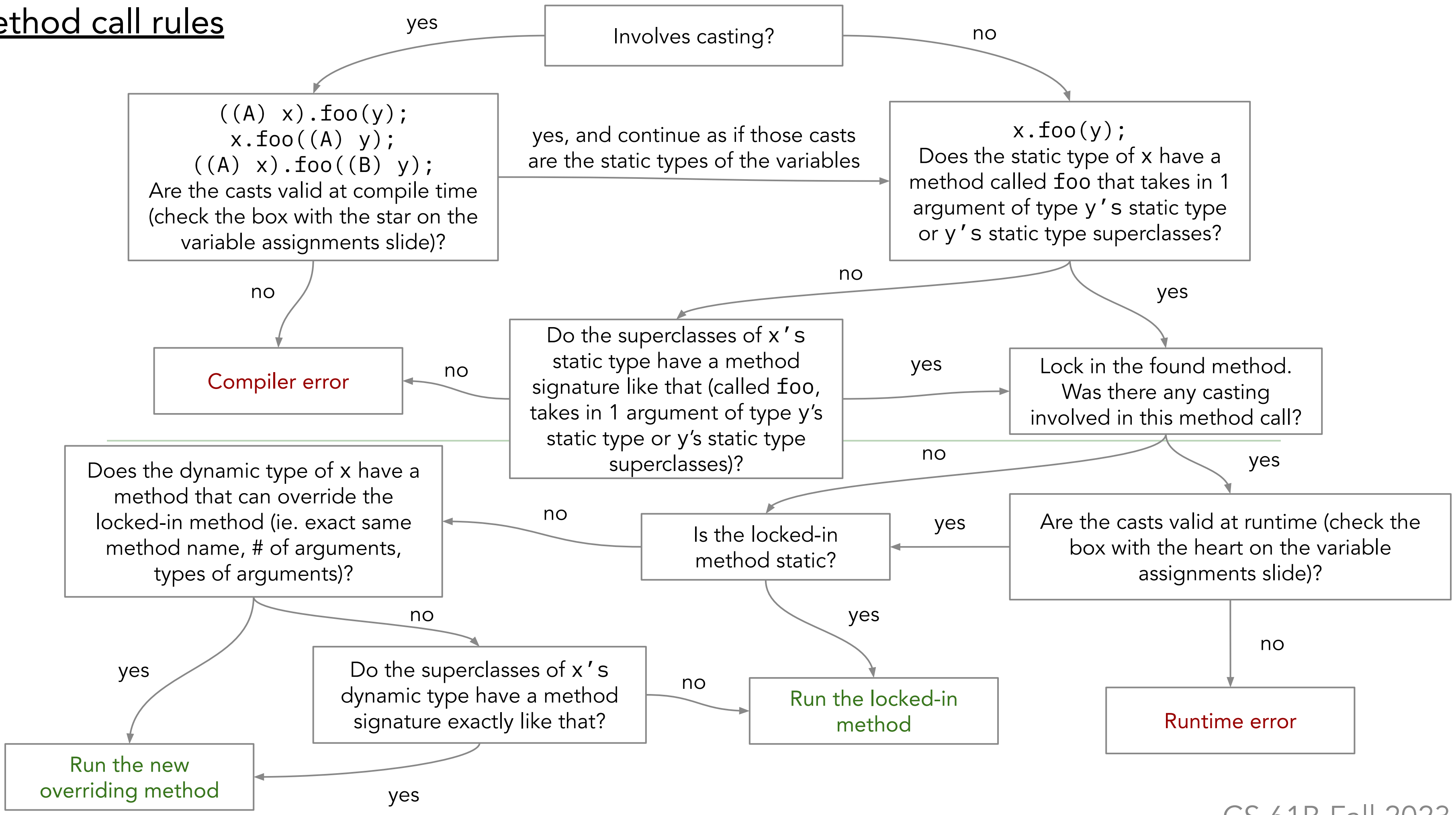
@ Run Time, we care about dynamic type of the invoking / calling instance:

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


Involves casting?
yes
no
A x = (B) y;
Is B in a superclass-subclass relationship with y's static type? (No siblings)
A x = y;
Is the static type of y A, or a subclass of A?
yes
no
no
yes
Is B A, or a subclass of A?
Compiler error
OK!
no
yes
Runtime error
no
Is the dynamic type of y B, or a subclass of B?
yes

Method call rules


Involves casting?
yes
no
((A) x).foo(y); x.foo((A) y); ((A) x).foo((B) y); Are the casts valid at compile time (check the box with the star on the variable assignments slide)?
yes, and continue as if those casts are the static types of the variables
x.foo(y); Does the static type of x have a method called foo that takes in 1 argument of type y's static type or y's static type superclasses?
no
Compiler error
no
yes
Do the superclasses of x's static type have a method signature like that (called foo, takes in 1 argument of type y's static type or y's static type superclasses)?
no
yes
Lock in the found method. Was there any casting involved in this method call?
no
yes
Does the dynamic type of x have a method that can override the locked-in method (ie. exact same method name, # of arguments, types of arguments)?
no
Is the locked-in method static?
yes
Are the casts valid at runtime (check the box with the heart on the variable assignments slide)?
no
yes
yes
no
Do the superclasses of x's dynamic type have a method signature exactly like that?
no
Run the locked-in method
Runtime error
Run the new overriding method
yes

Worksheet

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() {
        super(0, null);
     }

     @Override
     public int max() {
        return 0;
     }
}
```

1 Forget It, We Ball

```java
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

```java
Person ayati = new Person();

Athlete aniruth = new SoccerPlayer();

SoccerPlayer vanessa = aniruth;

Person eric = new Athlete();

Athlete shreyas = new Athlete();

SoccerPlayer yaofu = new SoccerPlayer();
```

1 Forget It, We Ball

```
Person ayati = new Person(); // CE

Athlete aniruth = new SoccerPlayer(); // <nothing>

SoccerPlayer vanessa = aniruth; // CE

Person eric = new Athlete(); // <nothing>

Athlete shreyas = new Athlete(); // <nothing>

SoccerPlayer yaofu = new SoccerPlayer(); //
<nothing>
```

1 Forget It, We Ball

```
Athlete aniruth = new SoccerPlayer();

Person eric = new Athlete();

Athlete shreyas = new Athlete();

SoccerPlayer yaofu = new SoccerPlayer();
```

```
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);
```

1 Forget It, We Ball

```
eric.watch(aniruth); // Person.watch, Athlete.watch → ball is life

shreyas.speakTo(yaofu); // Athlete.speakTo, Athlete.speakTo → i love sports

yaofu.speakTo(eric); // SoccerPlayer.speakTo, SoccerPlayer.speakTo → join 61ballers

((Athlete) yaofu).speakTo(eric); // Athlete.speakTo, SoccerPlayer.speakTo → join 61ballers

((Person) yaofu).speakTo(eric); // Person.speakTo, SoccerPlayer.speakTo → join 61ballers

((Athlete) eric).speakTo(shreyas); // Athlete.speakTo, Athlete.speakTo → i love sports

((SoccerPlayer) eric).watch(yaofu); // RE
```

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.
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.

   ```
   eric.watch(yaofu);
   ```

CS 61B
Fall 2023

Inheritance

Exam-Level 4: September 11, 2023

1 Forget It, We Ball

The 61Ballers are organizing the best IM team at Cal, but they first need your help with some inheritance issues...

Suppose we have the `Person` interface and the `Athlete`, and `SoccerPlayer` classes defined below.

```java
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"); }
}
```

Read the code below and fill in the table on the next page.

For lines 1-11, write down the static type of the object being created in the "Compile Time (Static)" column, the dynamic type in the "Runtime (Dynamic)" column. For the output, write nothing if there are no errors, write CE if there's a compiler error, and write RE if there's a runtime error.

For lines 13-25, identify the method that's been saved during compile time, and write down its name and the class it belongs to in the "Compile Time (Static)" column. Identify the method executed at runtime, and write down its information in the "Runtime (Dynamic)" column. Write output in the "Output" column, if anything. 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.

```java
1  Person ayati = new Person();
2
3  Athlete aniruth = new SoccerPlayer();
4
5  SoccerPlayer vanessa = aniruth;
6
7  Person eric = new Athlete();
8
9  Athlete shreyas = new Athlete();
```

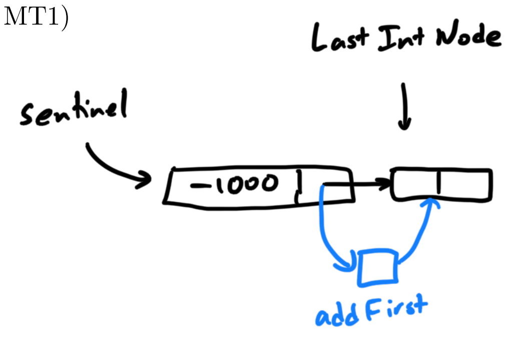
```
10
11 SoccerPlayer yaofu = new SoccerPlayer();
12
13 eric.watch(aniruth);
14
15 shreyas.speakTo(yaofu);
16
17 yaofu.speakTo(eric);
18
19 ((Athlete) yaofu).speakTo(eric);
20
21 ((Person) yaofu).speakTo(eric);
22
23 ((Athlete) eric).speakTo(shreyas);
24
25 ((SoccerPlayer) eric).watch(yaofu);
```

Variables

Line	Compile Time (Static)	Runtime (Dynamic)	Output
1 ayati	CE (can't instantiate interface)		CE
3 aniruth	Athlete	SoccerPlayer	
5 vanessa	CE (compiler uses static types, would need cast)		CE
7 eric	Person	Athlete	
9 shreyas	Athlete	Athlete	
11 yaofu	SoccerPlayer	SoccerPlayer	
13 eric	Person.watch	Athlete.watch	ball is life
15 shreyas	Athlete.speakTo	Athlete.speakTo	i love sports
17 yaofu	SoccerPlayer.speakTo	SoccerPlayer.speakTo	join 61 ballers
19 yaofu	Athlete.speakTo	SoccerPlayer.speakTo	join 61ballers
21 yaofu	Person.speakTo	SoccerPlayer.speakTo	join 61ballers
23 eric	Athlete.speakTo	SoccerPlayer.speakTo	i love sports
25 eric	SoccerPlayer.watch	RE	RE

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`. You may not change anything in the given code. You may only fill in blanks. You may not need all blanks. (Spring '16, MT1)



```java
public class DMSList {
    private IntNode sentinel;

    public DMSList() {
        sentinel = new IntNode(-1000, new LastIntNode());
    }

    public class IntNode {
        public int item;
        public IntNode next;
        public IntNode(int i, IntNode h) {
            item = i;
            next = h;
        }

        public int max() {
            return Math.max(item, next.max());
        }
    }

    class LastIntNode extends IntNode {
        public LastIntNode() {

            super(0, null);
        }

        @Override
        public int max() {

            return 0;
        }
    }

    /* Returns 0 if list is empty. Otherwise, returns the max element. */
    public int max() {
        return sentinel.next.max();
    }

    public void insertFront(int x) { sentinel.next = new IntNode(x, sentinel.next); }
}
```

like the "base case"

no more nodes after this one

Key Insight:
Last int node is the last int node in the list.
It prevents an error when calling max on an empty DMS list (instead returns 0).