

Inheritance

Exam-Level 03

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

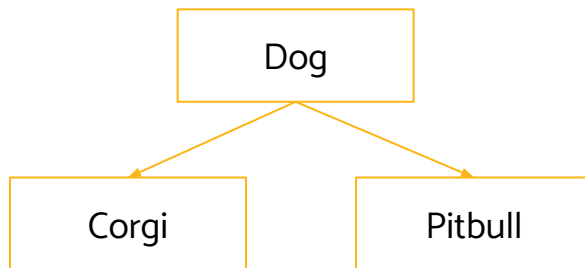
Classes

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`

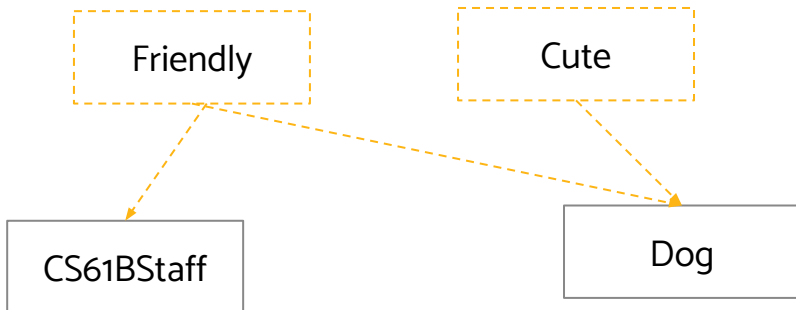
```
public void speak() { System.out.print("Woof, I'm a corgi!"); }
```

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. This is incredibly useful if you're building for other people.

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 {...}
```

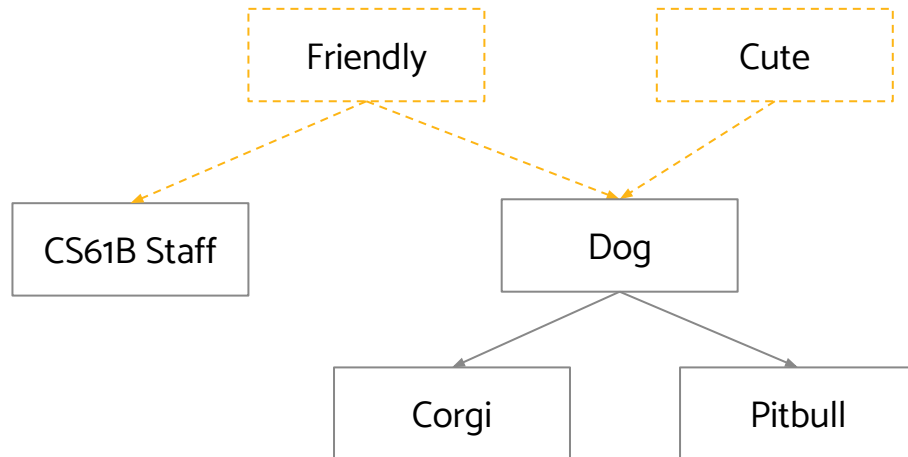
```
interface Friendly {...}
```

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

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

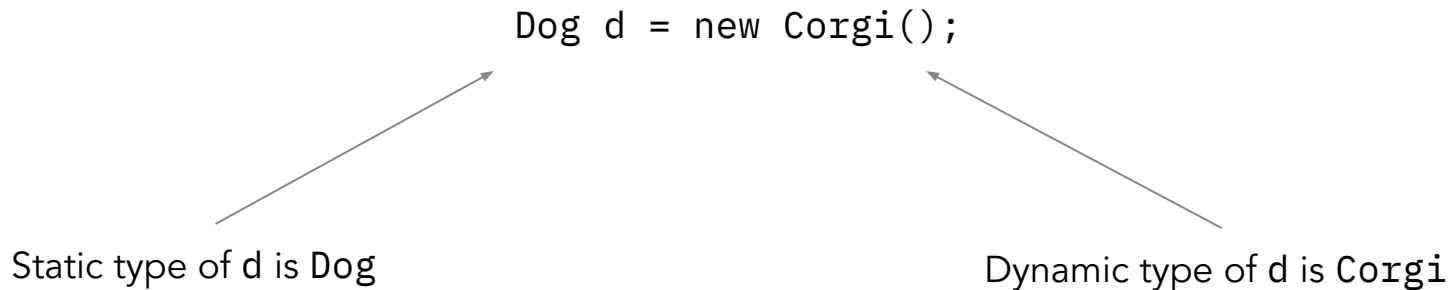
```
class Corgi extends Dog {...}
```

```
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. . .

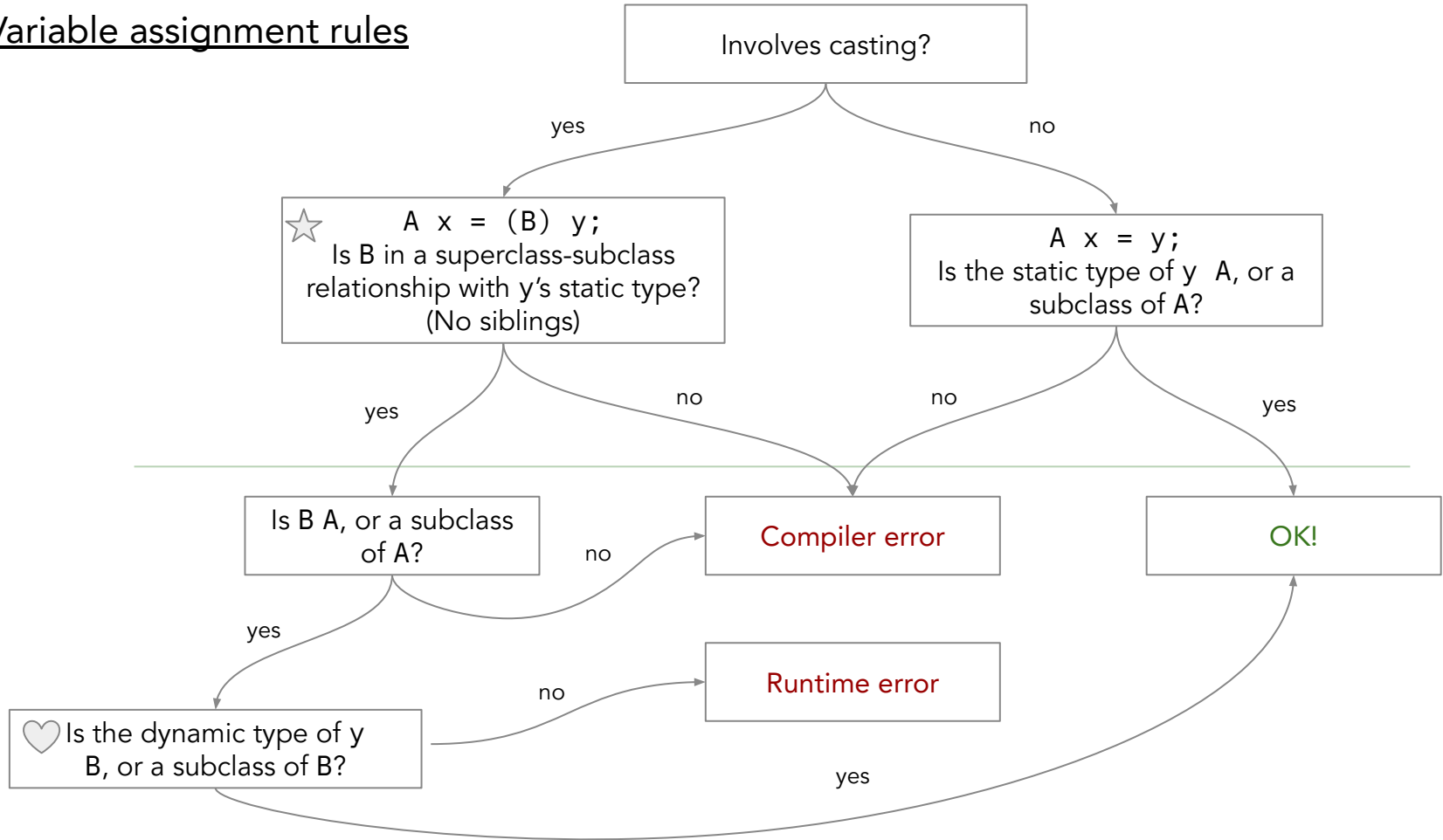
@ Compile Time, we only care about static type of the invoking / calling instance:

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
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

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.

```
1 interface Person {
2     void speakTo(Person other);
3     default void watch(Athlete other) { System.out.println("wow"); }
4 }
5
6 public class Athlete implements Person {
7     @Override
8     public void speakTo(Person other) { System.out.println("i love sports"); }
9     @Override
10    public void watch(Athlete other) { System.out.println("ball is life"); }
11 }
12
13 public class SoccerPlayer extends Athlete {
14     @Override
15     void speakTo(Person other) { System.out.println("join 61ballers"); }
16 }
```

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.

```
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();
```

Person is an interface, no instances via constructor

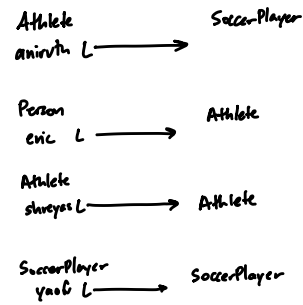
without casting, this will not work due to static type

Can fix by doing (SoccerPlayer) aniruth

```

10
11 SoccerPlayer yaofu = new SoccerPlayer();
12
13 Person (Athlete)    Athlete
  eric.watch(aniruth);
14
15 Athlete (Athlete)    SoccerPlayer
  shreyas.speakTo(yaofu);
16
17 SoccerPlayer (SoccerPlayer) Person ← dynamic type of parameter
  yaofu.speakTo(eric);           is irrelevant
18
19 Athlete (SoccerPlayer)    Person
  ((Athlete) yaofu).speakTo(eric);
20
21 Person (SoccerPlayer)    Person
  ((Person) yaofu).speakTo(eric);
22
23 Athlete (Athlete) ← eric's static type is ignored!
  ((Athlete) eric).speakTo(shreyas);
24
25 SoccerPlayer ← Class Cast Exception, dynamic type <= Static type
  ((SoccerPlayer) eric).watch(yaofu);

```



Line	Compile Time (Static)	Runtime (Dynamic)	Output
1	CE	n/a	CE
3	Athlete	SoccerPlayer	n/a
5	CE	n/a	CE
7	Person	Athlete	n/a
9	Athlete	Athlete	n/a
11	Soccer Player	Soccer Player	n/a
13	Person.watch(Athlete other)	Athlete.watch(Athlete other)	ball is life
15	Athlete.speakTo(Person other) ←		i love sports
17	SoccerPlayer.speakTo(Person other) ←		join 6lbatters
19	Athlete.speakTo(Person other)	SoccerPlayer.speakTo(Person other)	join 6lbatters
21	Athlete.speakTo(Person other)	SoccerPlayer.speakTo(Person other)	join 6lbatters
23	Athlete.speakTo(Person other) ←		i love sports
25	Athlete.watch(Athlete other)	RE ← Class Cast Exception	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)

```

1  public class DMSList {
2      private IntNode sentinel;
3
4      public DMSList() {
5          sentinel = new IntNode(-1000, new LastIntNode());
6      }
7
8      public class IntNode {
9          public int item;
10         public IntNode next;
11         public IntNode(int i, IntNode h) {
12             item = i;
13             next = h;
14         }
15
16         public int max() {
17             return Math.max(item, next.max());
18         }
19     }
20
21     class LastIntNode extends IntNode {
22         public LastIntNode() {
23
24             super(0, null);
25         }
26
27         @Override ← necessary since there is no next node on this last node, or NullPointerException
28         public int max() {
29
30             return 0;;
31         }
32     }
33
34     /* Returns 0 if list is empty. Otherwise, returns the max element. */
35     public int max() {
36         return sentinel.next.max();
37     }
38
39     public void insertFront(int x) { sentinel.next = new IntNode(x, sentinel.next); }
40 }

```

no edge case of LastIntNode not being at the end

↑ constructor w/o arguments

recursively compares item with the next one

↑ since all values are positive