Inheritance & Polymorphism

Topical Review Session 03

CS 61B // Spring 2021

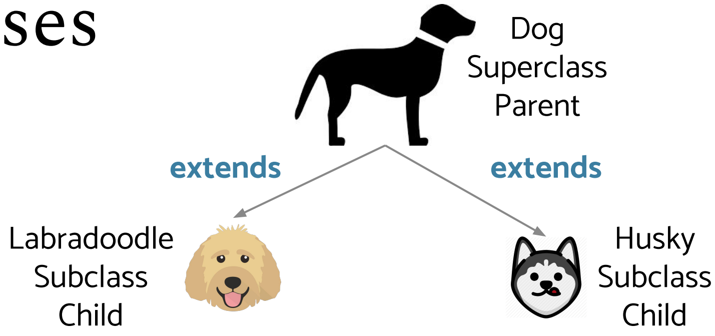
Agenda

Topic

- ❏ Superclasses
- ❏ Interfaces
- ❏ Dynamic Method Selection

Superclasses

Extending Classes



- Subclasses use the **extends** keywords to inherit attributes and methods from the superclass (“is a” relationship)

  ex. `class Labradoodle extends Dog { . . . }` → *Labradoodle is a Dog*
- What gets inherited?
  - All instance and static variables (static variables are shared between instances of parent and child)
  - All methods (constructors don’t count!)
- You can only extend **ONE** class
- All subclass constructors implicitly call superclass’s constructor at the beginning, which can be overridden to include parameters

  ex. `super();` → `super(parameter);`

Extending Practice

```java
public class Dog {
      public int age, weight;
      public Dog() {
            age = 0;
            weight = 5;
      }
      public void bark() {
            System.out.println(“bark”);
      }
}

public class Labradoodle extends Dog {
      public boolean happy;
      public Labradoodle() {
            happy = true;
      }
      public Labradoodle(boolean happy) {
            super();
            this.happy = happy;
      }
      public void ageBackwards() { age -= 4; }
}
```

Questions:

1. `> Dog marty = new Dog();`

   What are `marty.age` and `marty.weight`?

2. What is the output of `marty.bark()`?

3. `> Labradoodle lab = new Labradoodle();`

   What are `lab.age`, `lab.weight` and `lab.happy`?

4. `> lab.ageBackwards();`

   What is `lab.age`?

5. `> Labradoodle coco = new Labradoodle(false);`

   What are `coco.age`, `coco.weight` and `coco.happy`?

Extending Practice

```
public class Dog {
    public int age, weight;
    public Dog() {
        age = 0;
        weight = 5;
    }
    public void bark() {
        System.out.println(“bark”);
    }
}
public class Labradoodle extends Dog {
    public boolean happy;
    public Labradoodle() {
        happy = true;
    }
    public Labradoodle(boolean happy) {
        super();
        this.happy = happy;
    }
    public void ageBackwards() { age -= 4; }
}
```

Questions:

1. `> Dog marty = new Dog();`

   What are `marty.age` and `marty.weight`?

   `age = 0, weight = 5`

2. What is the output of `marty.bark()`?

   `bark`

3. `> Labradoodle lab = new Labradoodle();`

   What are `lab.age`, `lab.weight` and `lab.happy`?

   `age = 0, weight = 5, happy = true`

4. `> lab.ageBackwards();`

   What is `lab.age`?

   `age = -4`

5. `> Labradoodle coco = new Labradoodle(false);`

   What are `coco.age`, `coco.weight` and `coco.happy`?

   `age = 0, weight = 5, happy = false`

Overriding

- Inherit methods from your superclass, but what if you want to modify them in your subclass?
  - Redefine them in your subclass with the SAME function signature exactly (name + parameters)
- Bark method in Dog class

```
public void bark() {
     System.out.println(“Bark”);
}
```

- Bark method in Labradoodle class where Labradoodle extends Dog

```
@Override // @Override is optional but used as sanity check and for readability
public void bark() {
     System.out.println(“BARK!!!!!!!!!”);
}
```

- What will this print?

```
Labradoodle lab = new Labradoodle();
lab.bark();
```

Overriding

- Inherit methods from your superclass, but what if you want to modify them in your subclass?
  - Redefine them in your subclass with the SAME function signature exactly (name + parameters)
- Bark method in Dog class

```
public void bark() {
     System.out.println(“Bark”);
}
```

- Bark method in Labradoodle class where Labradoodle extends Dog

```
@Override // @Override is optional but used as sanity check and for readability
public void bark() {
     System.out.println(“BARK!!!!!!!!!”);
}
```

- What will this print?

```
Labradoodle lab = new Labradoodle();
lab.bark();
```

```
BARK!!!!!!!!!
```

Overloading

- Methods in a class have same name, different variables
- Bark methods in Dog class

```
bark();

bark(int number);

bark(int number, int decibels);

bark(string noise);
```

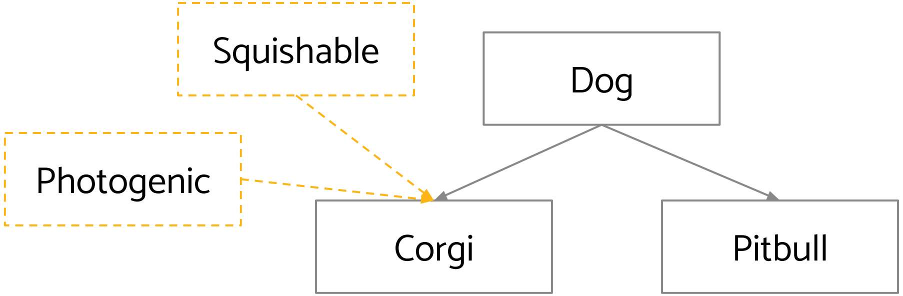
- Compiler is smart! It chooses based on parameters passed in

- Overloading and overriding are COMPLETELY DIFFERENT, no overlap whatsoever
    - Overload: 2 or more methods in a class have same method name but different parameters
    - Override: Child class can provide specific implementation of method already provided in parent class

Interfaces

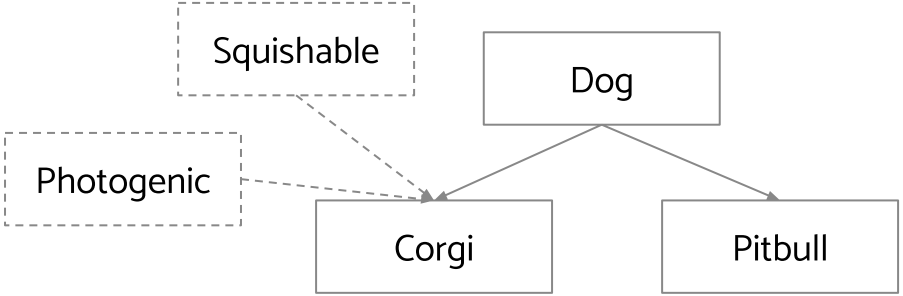
Interfaces

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. One class can implement many interfaces. *Ex. Comparable*



Implementation

```
interface Squishable {...}

interface Photogenic {...}

class Dog {...}

class Pitbull extends Dog {...}

class Corgi extends Dog implements Squishable, Photogenic {...}
```



Default Methods

The way we have dealt with interfaces, there is no content in them. We only define a certain set of operations that need to be fulfilled by anything that implements the interface. However, we can create **default** methods that take the following form:

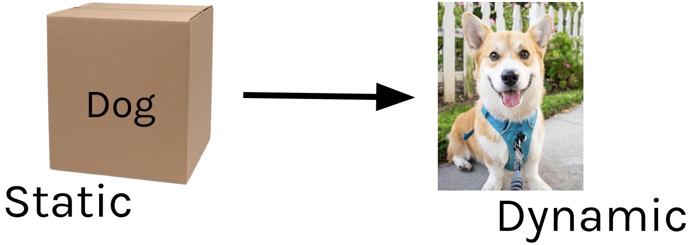
```
default void methodName(){...}
```

Normal interface methods have no body and only state what needs to be defined. Default methods on the other hand provide how a method by providing a method body. Default methods must have the modifier *default* and a method body. They can be defined only inside an interface not in any classes. They cannot be used to override *java.lang.Object* class' methods.

- Variables can exist in interfaces but they are `public static final`.
- Classes can implement more than 1 interface
- Methods are public unless stated otherwise
- Interfaces cannot be instantiated, but rather are implemented

Dynamic Method Selection

Static type vs Dynamic type



- The Java Code you write is scanned line-by-line twice, at compilation time and at runtime
    - javac filename.java (compilation time) & java filename (runtime)
- As a result, variables also have two distinct types: Static type & Dynamic Type

Static Type Variable (Compilation time)

- Specified during *declaration*
- Check type during compilation

Example:

```
List arbitraryList
```

Dynamic Type Variable (Runtime)

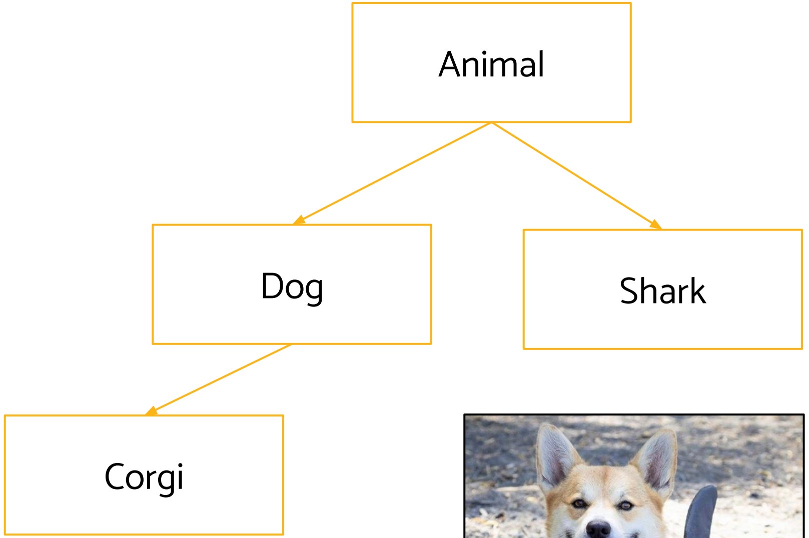
- Specified during *instantiation*
- Check type during runtime

Example:

```
new AList();
```

Example: Which of these are legal?

```
Animal ari = new Animal();

Animal benny = new Corgi();

Shark chum = benny;

Shark anchor = new Animal();

Shark bruce = new Shark();

Corgi david = new Shark;
```



Solutions: Which of these are legal?

Animal ari = new Animal(); ✅

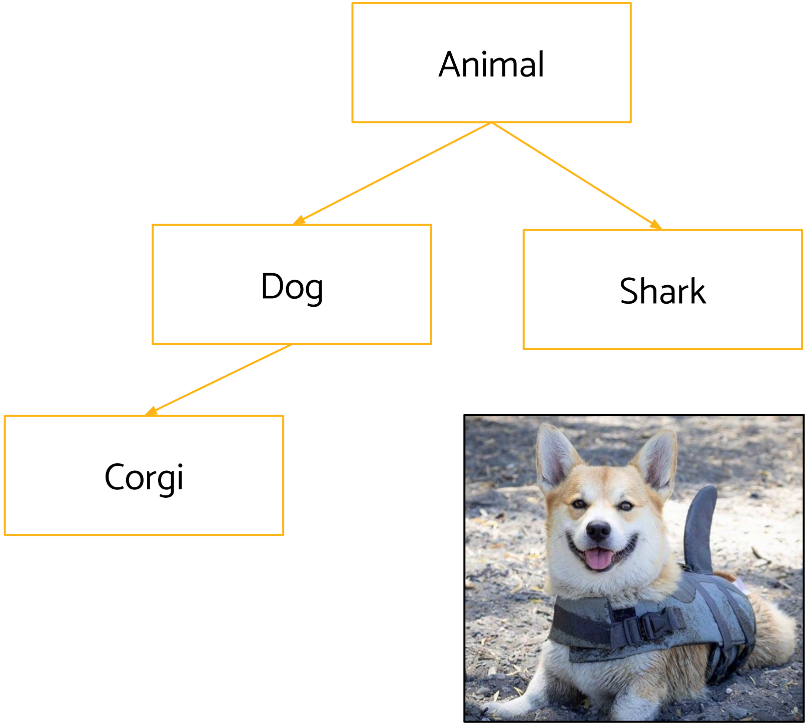
Animal benny = new Corgi(); ✅

Shark chum = benny; ❌

Shark anchor = new Animal(); ❌

Shark bruce = new Shark(); ✅

Corgi david = new Shark; ❌



Method Selection Overview

Dynamic method selection is a process by which Java determines which method to call, given various inheritance relationships.

@ Compile Time

1. Check for valid variable assignments
2. Check for valid method calls (only considering static types)

@ Run Time

1. Check for overridden methods
2. Ensure casted objects can be assigned to their variables

Compile Time → Does the method even exist?

1. Look in the class corresponding to the static type for the method signature.
2. If the signature exists, the compiler takes a snapshot and **LOCKS-IN** on that **EXACT SIGNATURE**.
3. If the signature does not exist, keep checking parent classes.
4. Finally, if no signature exists, it is a Compile Time Error.

```
class A {
    void f(A obj) { 1 }
}

class B extends A {
    void f(A obj) { 2 }

    void f(B obj) { 3 }
}
```

```
A peach = new B();
B banana = new B();
peach.f(peach);
peach.f(banana);
```

Run Time

→ Can we do better?

1. Look in the class corresponding to the *dynamic* type for the **EXACT SIGNATURE** we **LOCKED IN** during compile.
2. If the signature exists, use it.
3. Otherwise, keep checking superclasses of the dynamic type until **IDENTICAL** signatures exist.
4. If we can not do better, use the method found during compile.

```
class A {
    void f(A obj) { 1 }
}

class B extends A {
    void f(A obj) { 2 }

    void f(B obj) { 3 }
}
```

```
A peach = new B();
B banana = new B();
peach.f(peach);
peach.f(banana);
```

Example: Practice With DMS

```
public class Dog {
     public void bark() {
          System.out.println(“yip”);
     }
}


public class Corgi extends Dog {
     public void bark() {
          System.out.println(“woof woof”);
     }
     public void bark(int n) {
          System.out.print(n + “ sheep”);
     }
}
```

```
public static void main(String[] args) {
     Dog dog = new Dog();
     Corgi corgi = new Corgi();
     Dog benny = new Corgi();

     dog.bark();
     corgi.bark();
     benny.bark();
     dog.bark(5);
     corgi.bark(5);
     benny.bark(5);
}
```

Example: Practice With DMS

```
public class Dog {
     public void bark() {
          System.out.println(“yip”);
     }
}


public class Corgi extends Dog {
     public void bark() {
          System.out.println(“woof woof”);
     }
     public void bark(int n) {
          System.out.print(n + “ sheep”);
     }
}
```

```
public static void main(String[] args) {
     Dog dog = new Dog();
     Corgi corgi = new Corgi();
     Dog benny = new Corgi();

     dog.bark();
     corgi.bark();
     benny.bark();
     dog.bark(5);
     corgi.bark(5);
     benny.bark(5);
}
```

Name	Static	Dynamic
dog	Dog	Dog
corgi	Corgi	Corgi
benny	Dog	Corgi

Solutions: Practice With DMS

```
public class Dog {
    public void bark() {
        System.out.println(“yip”);
    }
}


public class Corgi extends Dog {
    public void bark() {
        System.out.println(“woof woof”);
    }
    public void bark(int n) {
        System.out.print(n + “ sheep”);
    }
}
```

```
public static void main(String[] args) {
    Dog dog = new Dog();
    Corgi corgi = new Corgi();
    Dog benny = new Corgi();

    dog.bark();     yip
    corgi.bark();   woof woof
    benny.bark();   woof woof
    dog.bark(5);    compile-time error
    corgi.bark(5);  5 sheep
    benny.bark(5);  compile-time error
}
```

Name	Static	Dynamic
dog	Dog	Dog
corgi	Corgi	Corgi
benny	Dog	Corgi

Casting

Casting tells the compiler to treat the variable as the type casted to, at that line only.
Casting tricks the compiler into thinking the static type of a variable is the casted type. This casted type must be a class that inherits from the original static type.

Compile Time

1. Casts must be along the same branch in inheritance hierarchy.
2. Check cast is legal according to static type.

Run Time

1. Check that cast was legal according to dynamic type.

Example: DMS with Casting

```
public class Tampa {
     public void winning(int x) {}
}


public class Bay extends Tampa {
     public void winning(boolean y) {}
}


public class Buccaneers extends Tampa {
     public void winning(boolean z) {}
}
```

```
public static void main(String[] args) {
     Tampa t = new Tampa();
     Bay b = new Bay();

     ((Tampa) b).winning(31);
     ((Tampa) b).winning(true);

     ((Bay) t).winning(true);
     ((Bay) t).winning(31);

     ((Buccaneers) b).winning(true);
```

For the lines above, write CE if there's a compiler error or RE if there's a runtime error.

Example: DMS with Casting

```
public class Tampa {
     public void winning(int x) {}
}


public class Bay extends Tampa {
     public void winning(boolean y) {}
}


public class Buccaneers extends Tampa {
     public void winning(boolean z) {}
}
```

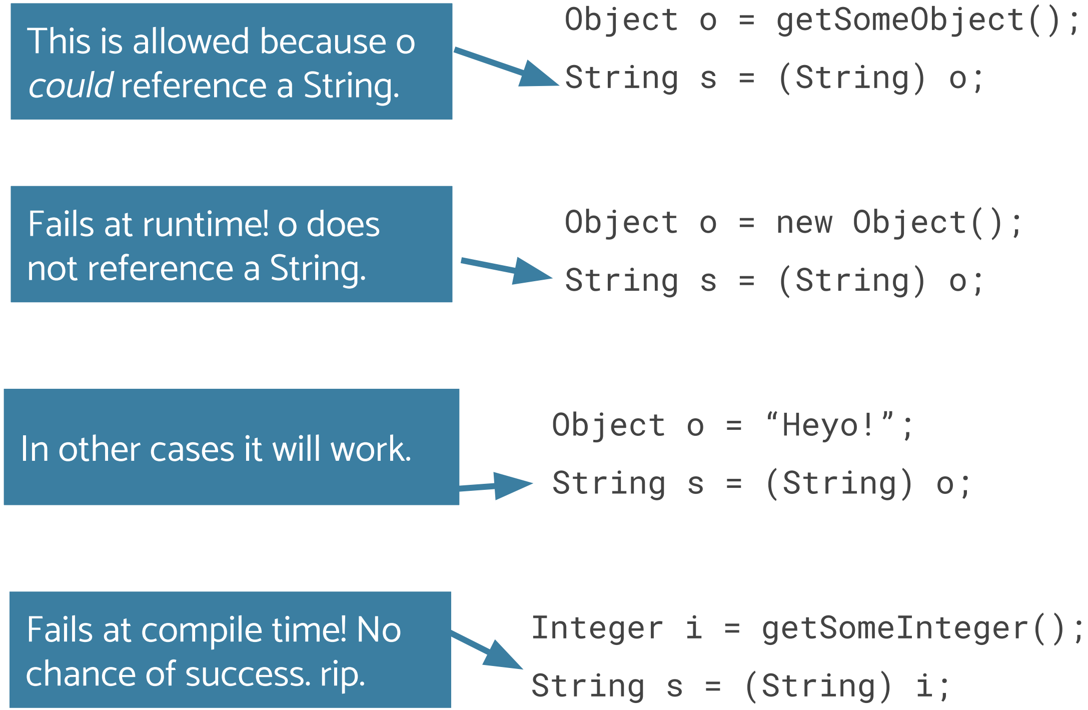
```
public static void main(String[] args) {
     Tampa t = new Tampa();
     Bay b = new Bay();

     ((Tampa) b).winning(31); nothing
     ((Tampa) b).winning(true); COMPILE TIME ERROR

     ((Bay) t).winning(true); RUN TIME ERROR
     ((Bay) t).winning(31); RUN TIME ERROR

     ((Buccaneers) b).winning(true); COMPILE TIME ERROR
}
```

For the lines above, write CE if there's a compiler error or RE if there's a runtime error.

Intuition Behind Downcasting

Downcasting is allowed when there is a possibility that it succeeds at runtime.

The compiler only throws an error when a cast has absolutely no probability of succeeding.



This is allowed because o *could* reference a String.

```
Object o = getSomeObject();
String s = (String) o;
```

Fails at runtime! o does not reference a String.

```
Object o = new Object();
String s = (String) o;
```

In other cases it will work.

```
Object o = “Heyo!”;
String s = (String) o;
```

Fails at compile time! No chance of success. rip.

```
Integer i = getSomeInteger();
String s = (String) i;
```

https://discord.gg/RPJcRmwG

Thank you for listening! Please head over to Discord for group work time.