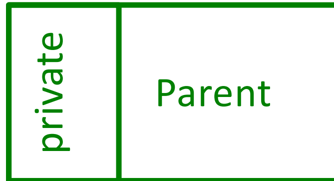
Objectives

- Polymorphism
- Dynamic Dispatch
- Abstract Classes
- Interfaces

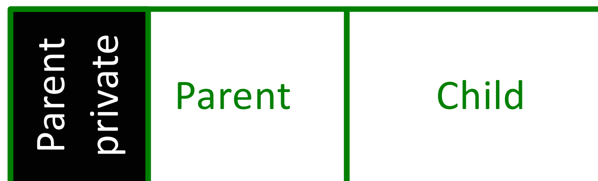
Review

- How does Java handle memory management?
  - What are the benefits and limitations of that approach?
- How do we make a class inherit from a parent class?
- How does a class refer to its parent class?
- What does a class inherit from its parent class?
  - What is *not* inherited?
- What are the access modifiers, ordered from least restrictive to most restrictive?
- How can we check that an object variable is a certain type?
  - (not from last class; before that)

Inheriting Private Variables



Parent has private variables,
which objects of Parent class can access



Child class inherits the private variables from Parent but cannot directly access them.

3

Review

- Designing classes: When should you make a variable/field
  - Local vs instance vs static?
  - Private vs protected vs public?

- Inheritance in game code
  - Javadocs

4

POLYMORPHISM & DISPATCH

Polymorphism

- ***Polymorphism*** is an object's ability to vary behavior based on its type
- You can use a child class object whenever the program expects an object of the parent class
- Object variables are ***polymorphic***
- A `Chicken` object variable can refer to an object of class `Chicken`, `Rooster`, `Hen`, or any class that *inherits from* `Chicken`

```
Chicken[] chickens = new Chicken[3];
chickens[0] = momma;
chickens[1] = foghorn;
chickens[2] = baby;
```

We can guess the actual types
But compiler can't

Somewhere Else…

```
Rooster foghorn = new Rooster(…);
Hen momma = new Hen(…);
Chicken baby = new Chicken(…);
```

- These objects were instantiated at some point in time …

Compiler's Behavior

```
Chicken[] chickens = new Chicken[3];
chickens[0] = momma; // a Hen
chickens[1] = foghorn; // a Rooster
chickens[2] = baby; // a Chicken
```

- We know `chickens[1]` is probably a `Rooster`, but to *compiler*, it's a `Chicken` so

`chickens[1].crow();` will not compile

Compiler's Behavior

- When we refer to a `Rooster` object through a `Rooster` object variable, compiler sees it as a `Rooster` object
- If we refer to a `Rooster` object through a `Chicken` object variable, compiler sees it as a `Chicken` object.

→ Object *variable* determines how compiler sees object.

- We cannot assign a parent class object to a child class object variable
  - Ex: `Rooster` is a `Chicken`, but a `Chicken` is not necessarily a `Rooster`

```
Rooster r = chicken;
```


Polymorphism

```
Chicken[] chickens = new Chicken[3];
chickens[0] = momma;
chickens[1] = foghorn;
chickens[2] = baby;
```

```
chickens[1].feed();
```

Compiles because `Chicken` has a `feed` method.

But, which `feed` method is called – `Chicken's` or `Rooster's`?

Polymorphism

- Which method do we call when we call `chicken[1].feed()?` Rooster's or Chicken's?
- In Java: Rooster's!
  - Object is a `Rooster`
  - JVM figures out object's class at runtime and runs the appropriate method
- ***Dynamic dispatch***
  - At runtime, the object's class is determined
  - Appropriate method for that class is dispatched

Feed the Chickens!

Think on your own for 1 minute

Recall:

```
Chicken[] chickens = new Chicken[3];
chickens[0] = momma;
chickens[1] = foghorn;
chickens[2] = baby;
```

```
for( Chicken c: chickens ) {
        c.feed();
}
```

How to read this code?
What happens in execution?

- ***Dynamic dispatch*** calls the method corresponding to the actual class of each object
  - This is the power of polymorphism and dynamic dispatch!

Dynamic Dispatch vs. Static Dispatch

- Dynamic dispatch is not necessarily a property of statically typed object-oriented programming languages in general
- Some OOP languages use ***static dispatch***
  - Type of the object variable that the method is called on determines which version of method gets run
- The primary difference is ***when*** **decision on which method to call is made…**
  - ***Static*** dispatch (C#) decides at ***compile*** time
  - ***Dynamic*** dispatch (Java) decides at ***run*** time
- Dynamic dispatch is slower
  - In mid to late 90s, active research on how to decrease time

What Will This Code Output?

Think on your own for 1 minute

```
class Parent {
    public Parent() {}

    public void method1() {
        System.out.println("Parent: method1");
    }

    public void method2() {
        System.out.println("Parent: method2");
        method1();
    }
}

class Child extends Parent {
    public Child() {}

    public void method1() {
        System.out.println("Child: method1");
    }
}
```

```
public class DynamicDispatchExample {
    public static void main(String[] args) {
        Parent p = new Parent();
        Child c = new Child();

        p.method1();
        System.out.println("");

        c.method1();
        System.out.println("");

        p.method2();
        System.out.println("");

        c.method2();
        System.out.println("");
    }
}
```

See handout

What Will This Code Output?

```
class Parent {
    public Parent() {}

    public void method1() {
        System.out.println("Parent: method1");
    }

    public void method2() {
        System.out.println("Parent: method2");
        method1();
    }
}

class Child extends Parent {
    public Child() {}

    public void method1() {
        System.out.println("Child: method1");
    }
}
```

```
public class DynamicDispat
    public static void mai
        Parent p = new Par
        Child c = new Chil

        p.method1();
        System.out.println("");

        c.method1();
        System.out.println("");

        p.method2();
        System.out.println("");

        c.method2();
        System.out.println("");
    }
}
```

```
Parent: method1

Child: method1

Parent: method2
Parent: method1

Parent: method2
Child: method1
```


Inheritance Rules: Access Modifiers

Access modifiers in child classes
- Can make access to child class ***less*** restrictive but not more restrictive

- **Why?**
- What would happen if a method in the parent class is `public` but the child class's method is `private`?

Inheritance Rules: Access Modifiers

Access modifiers in child classes
- Can make access to child class ***less*** restrictive but not more restrictive

- If a `public` method could be overridden as a `protected` or `private` method, child objects would not be able to respond to the same method calls as parent objects
- When a method is declared `public` in the parent, the method remains `public` for all that class's child classes
- Remembering the rule: compiler error to override a method with a more restricted access modifier

Summary of Inheritance

- Remove repetitive code by modeling the "is-a" hierarchy
  - Move "common denominator" code up the inheritance chain
- Don't use inheritance unless *all* inherited methods make sense
- Use polymorphism

CSCI112 Review

- What are ***abstract classes***?
- What are ***interfaces***?

ABSTRACT CLASSES

Abstract Classes

- Classes in which not all methods are implemented are *abstract classes*
  - Some methods defined, others not defined
    - Partial implementation
  - `public abstract class ZooAnimal`
- Declared but not implemented methods are labeled as `abstract`

```
public abstract void exercise(Environment env);
```


Abstract Classes

- An abstract class can***not*** be instantiated
  - i.e., can't create an object of that class
  - But can have a constructor!
- Child class of an abstract class can only be instantiated if it overrides and implements **every abstract method** of parent class
  - If child class does not override *all* abstract methods, it is **also abstract**

Abstract Classes

- `static`, `private`, and `final` methods cannot be `abstract`
  - B/c cannot be overridden by a child class
- `final` class cannot contain abstract methods

Why?

- A class can be abstract even if it has no abstract methods
  - Use when implementation is incomplete and is meant to serve as a parent class for class(es) that complete the implementation
- Can have array of objects of abstract class
  - JVM will do dynamic dispatch for methods

Summary: Defining Abstract Classes

➡ Define a class as `abstract` when have *partial implementation*

INTERFACES

Interfaces

- Pure specification, no implementation
  - A set of requirements for classes to conform to
- Classes can `implement` one *or more* interfaces

A Scenario

- We have a Customer Service Driver program
- Depending on the circumstances, we may want to use different algorithms to determine the service order
- Possible algorithms
  - FIFO
  - HighestPayingFirst
  - CriticalProblemFirst
  - ShortestJobFirst

Design Solution

- Interface `CustomerServiceOrder`
  - `public Customer getNextCustomer();`
  - `public boolean hasNext();`
- Driver program snippet

```
CustomerServiceOrder customerOrder = …;
while( agent.isAvailable() ) {
    if( customerOrder.hasNext() ) {
        Customer next = customerOrder.getNextCustomer();
        agent.handle( next );
    }
}
```

Design Solution

- Classes adhere to (i.e., *implement*) the interface, implementing different algorithms
  - FIFOOrder
  - HighestPayingFirstOrder
  - CriticalProblemFirstOrder
  - ShortestJobFirstOrder
- Assign objects of any of these types to the interface variable

```
CustomerServiceOrder customerOrder = new FIFOOrder();
while( agent.isAvailable() ) {
    if( customerOrder.hasNext() ) {
        Customer next = customerOrder.getNextCustomer();
        agent.handle( next );
    }
}
```

Easily change program behavior with only one change in code

Example of an Interface

- Recall: `Arrays.sort(array)`
  - `Arrays.sort` sorts arrays of *any* object class that implements the `Comparable` interface
- Classes that implement the `Comparable` interface must provide a way to decide if one object is less than, greater than, or equal to another object

java.lang.Comparable

```
public interface Comparable {
        int compareTo(Object other);
}
```

- Any object that implements Comparable must have a method named compareTo()
- Returns:
  - Return a negative integer if this object is less than the object passed as a parameter
  - Return a positive integer if this object is greater than the object passed as a parameter
  - Return a 0 if the two objects are equal

Comparable Interface API/Javadoc

- Specifies what the compareTo() method should do
- Says which Java library classes implement Comparable

https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/lang/Comparable.html

Looking Ahead

- Assignment 5: Goblin Game
  - Can now do the refactoring part
  - Due Tuesday at 11:59 p.m.
- Exam: Next Friday
  - Online, timed exam (70 minutes)
    - No class next Friday
    - Start time 8:30 a.m. Friday, due time Sun 11:59 p.m.
  - Open book/notes/slides – but **do not** rely on that
    - NOT open internet
  - Prep document online
  - 3 sections:
    - Very Short Answer, Short Answer, Coding