

Objectives

- Jar Files
- Interfaces
- Abstract Classes
- Collections – case study of interfaces, abstract classes

JAR FILES

Jar (Java Archive) Files

- Archives of Java files
- Package code into a neat bundle to distribute
 - Easier, faster to download
 - Easier for others to use
- **jar** command: create, view, and extract Jar files
 - Works similarly to **tar**
- Run jar file using java

```
java -jar myapplication.jar
```

Jar/Tar Commands

- Common options:

Option/ Operations	Meaning
f	The name of the archive file
c	Create an archive file
x	Extract the archive file
v	Verbose
z	Zip (compress)
t	Table of contents (list contents)

- Common use:

- `jar cf code.jar class_files_directory`
- `jar xfz code.jar.gz`

Jar file in Assignment 3

- In the demo directory: Game.jar
- To see an example of the final product
 - Run `java -jar Game.jar`
- To list the contents of the jar file:
 - `jar tf Game.jar`
 - .class files, images, etc.
- Packages up code in a neat little bundle!

Another Use Case for Jar Files

- “I want to use this third-party (not part of Java library) library in my code”
- You have a *jar* file of the code
- Add the jar file to your ***classpath***

More on this later...

Review

1. How do we make a class inherit from a parent class?
2. How does a class refer to its parent class?
3. What does a class inherit from its parent class?
 - What is *not* inherited?
4. What does it mean for a *class* to be *final*?
 - For a *method* to be *final*?
5. What are the access modifiers, ordered from least restrictive to most restrictive?
 1. What should you consider when considering which modifier to use when you declare a field? A method?
6. How does Java decide which method to call on an object?
 - Example: `chicken[1].feed()`;
 - Give name of how decision is made and explain how it works
7. What is polymorphism?
8. Review from before last class:
 - How can we check that an object variable is a certain type?
 - How can we specify that an object variable has a different type (e.g., a derived type)?
9. Review from Python/112
 - What are abstract classes and interfaces?
 - How are they useful?

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** if override a method with a more restricted access modifier

Preventing Overriding: `final` Method

- If you don't want child classes to override a method, you can make that method `final`

```
class Chicken {  
    . . .  
    public final String getName() { . . . }  
    . . .  
}
```

Why would we want to make a method `final`?

What are possible benefits to developers, the compiler, ...?

Motivating Abstraction...

- We have a Customer Service Driver program
- It currently uses FIFO to schedule customers
- Depending on the circumstances, we may want to use different algorithms to determine the service order, e.g.,
 - HighestPayingFirst
 - CriticalProblemFirst
 - ShortestJobFirst

Consider Original FIFO Implementation

```
while( agent.isAvailable() ) {  
    if( customerList.size() > 0 ) {  
        Customer next = customerList.remove(0);  
        agent.handle( next );  
    }  
}
```

- If you wanted to change the algorithm, how much code would you need to change?
- What if later you wanted to change back to FIFO or you wanted to change between algorithms frequently?
- Consider what is common among the algorithms

Design Solution

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

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 with *polymorphism*

Interfaces and Abstract Classes

Provide abstraction

→ Makes code easier to change, extend, maintain

Note it's not that they make code easier to implement or understand.
You need some more experience on that front.

INTERFACES

Interfaces

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

Interface Definitions

- Example: define an interface for an object that is capable of moving:

```
public interface Movable {  
    void move(double x, double y);  
}
```

- Interface methods are **public** by default
 - Do not *need* to specify methods as **public**

Interface Definitions and Inheritance

- Can extend interfaces

- Allows a chain of interfaces that go from general to more specific

- Example:

```
public interface Powered extends Movable {  
    double milesPerGallon();  
}
```

- A class that implements the Powered interface must have a milesPerGallon and move method

Class Implements Interface

- Class needs to implement all methods declared in the interface

```
public class Car implements Powered { ...  
    public double milesPerGallon() {  
        return mpg;  
    }  
  
    public void move(double x, double y) {  
        xcoord += x;  
        ycoord += y;  
    }  
    ...  
}
```

Car.java

Constants in an Interface

- If a variable is specified in an interface, it is automatically a constant:

➤ `public static final variable`

```
public interface Powered extends Movable {  
    double SPEED_LIMIT = 95;  
    double milesPerGallon();  
}
```

- Example: An object that implements Powered interface has a constant SPEED_LIMIT defined

Multiple Interfaces

- A class can implement *multiple* interfaces
 - Must fulfill the requirements of each interface

```
public final class String implements  
    Serializable, Comparable, CharSequence { ...
```

- Recall: NOT possible with inheritance
 - A class can only extend (or inherit from) **one** class

Testing for Interfaces

- Can also use the `instanceof` operator to see if an object implements an interface
 - e.g., to determine if an object is movable

```
if (obj instanceof Movable) {  
    // runs if obj is an object variable of a class  
    // that implements the Movable interface  
}  
else {  
    // runs if obj does not implement the interface  
}
```

Interface Object Variables

- Can use an *object variable* to refer to an object of *any* class that implements an interface
- Using this object variable, can only access the interface's methods
- For example...

```
public void aMethod(Object obj) {  
    ...  
    if (obj instanceof Movable) {  
        Movable mover = (Movable) obj;  
        mover.move(x, y);  
    }  
}
```

Comparable Interface

- Implemented by String and many other classes
- Uses **Generics**!
- Interface declaration:

```
public interface Comparable<T>
```

- Declared method:

```
int compareTo(T o)
```

The type it compares



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/25/docs/api/java.base/java/lang/Comparable.html>

java.lang.Comparable

```
public interface Comparable<T> {  
    int compareTo(T other);  
}
```

- A class that implements **Comparable** must have a method named **compareTo**
- Behavior:
 - Returns a negative integer if this object is less than the object passed as a parameter
 - Returns a positive integer if this object is greater than the object passed as a parameter
 - Returns a 0 if the two objects are equal

Example Use of an Interface

- Recall: `Arrays.sort(array)`
- `public static void sort(Object[] a)`
 - “Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. **All elements in the array must implement the Comparable interface.**”
- Overloaded method, so can also pass in arrays of primitive types

Implementing an Interface with Generics

```
public interface Comparable<T> {  
    int compareTo(T other);  
}
```

Interface definition is *generic* to the type. T is a placeholder.

Implementing an Interface with Generics

```
public interface Comparable<T> {  
    int compareTo(T other);  
}
```

Interface definition is *generic* to the type. T is a placeholder.

- In the implementing class definition, specify
 - the class will implement the interface
 - the *type* it will accept/operate on

```
public class Chicken implements Comparable<Chicken>
```

Chicken Implementing Comparable

With Generics

```
public int compareTo(Chicken other) {  
    if (height < other.getHeight() )  
        return -1;  
    if (height > other.getHeight())  
        return 1;  
    return 0;  
}
```

Without Generics

```
public int compareTo(Object otherObject) {  
    if( ! (otherObject instanceof Chicken) )  
        return 1;  
    Chicken other = (Chicken) otherObject;  
    if (height < other.getHeight() )  
        return -1;  
    if (height > other.getHeight())  
        return 1;  
    return 0;  
}
```

Interface Summary

- Contain only object (*not class*) methods
- All methods are **public**
 - Implied if not explicit
- Fields are constants that are **static** and **final**
- A class can implement multiple interfaces
 - Separated by commas in definition

Benefits of Interfaces

- Abstraction
 - Separate the *interface* from the *implementation*
- Allow easier type substitution
- Classes can implement multiple interfaces

ABSTRACT CLASSES

Abstract Classes

- Classes that are not fully implemented are *abstract classes*
 - Often: some methods defined, others not defined
 - Purpose: provide some common, shared, base code
 - Example: **public abstract class ZooAnimal**
 - Declared but not implemented methods are marked 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 Class

Why would we define a constructor if we can't create objects?

- An abstract class **cannot** 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 Rules

static, private, and final methods cannot be abstract

- Because cannot be overridden by a child class

final class cannot contain abstract methods

- Because a child class cannot inherit from a final class

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 use *dynamic dispatch* to call appropriate method

## Summary: Defining Abstract Classes

- ➔ Define a class as **abstract** when have *partial implementation*
  - Typically used as a base class for other, similar classes

# Comparing Concreteness

Concrete Class

Abstract Class

Interface



Increase in abstraction,  
resilience to change

# Comparing Interfaces and Abstract Class

## Interfaces

- No implementation
- Any class can use
  - (b/c classes can implement multiple interfaces)
- May need to implement methods multiple times
- Adding a method to interface will break classes that implement interface

## Abstract Classes

- Contain partial implementation
- Child classes can't extend/subclass multiple classes
- Can add non-abstract methods without breaking child classes

## One Option: Use Both!

- Define interface, e.g., `MyInterface`
- Define abstract class, e.g., `AbstractMyInterface`
  - Implements interface
  - Provides implementation for some methods

## Example: CustomerServiceOrder

- Interface: CustomerServiceOrderable
- Abstract Class: CustomerServiceOrder
  - Has state: nextCustomer, numWaiting
- Concrete Classes: FIFOOrder, ShortestJobFirst, ...

# Abstract Classes and Interfaces

- Important structures in Java
  - Make code easier to change
- Will return to/apply these ideas throughout the course
- Concepts are used in many languages besides Java

# Looking Ahead

- Assignment 3 is due next Wednesday
  - You know all you need to know to complete the assignment