Objectives

- Jar Files

- Classpaths

- Abstract Classes

- Interfaces

- Collections

Capture the Flag 3

Launch party: Sunday, 8-10 p.m.
- Meet, greet, eat
- Training and practice sessions
- No prior CTF experience necessary

Main CTF: Sunday, 10 p.m. to following Sunday at 10 p.m.

JAR FILES

Jar (**J**ava **Ar**chive) 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 **f**ile
c	**C**reate an archive file
x	E**x**tract the archive file
v	**V**erbose
z	**Z**ip (compress)
t	**T**able of contents (list contents)

- Common use:
  - ➤ jar cfz code.jar.gz class_files_directory
  - ➤ jar xfz code.jar.gz

Typical Use Case with 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

- You then add the jar file to your ***classpath***

CLASSPATH

Classpath

- Tells the compiler or JVM where to look for user-defined classes and packages (jar files)
  - ➢ Often when using third-party libraries
- Similar to PYTHONPATH
- Typically know it needs to be set when there are "Class not found" error messages in your code but you have the appropriate import

Setting the Classpath

- Can specify classpath in command line

```
javac -cp path/to/myjavaclasses MyClass.java
java -cp path/to/myjavaclasses MyClass
```

Can be .class files or jar files

- Can specify the classpath environment variable

➢ Edit your `.bash_profile` (or similar) OR

➢ Set in terminal

```
CLASSPATH=$CLASSPATH:path/to/myjavaclasses
echo $CLASSPATH
```

Current value of CLASSPATH

Review

- 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 does it mean for a class to be final?
  - For a method to be final?
- What are the access modifiers, ordered from least restrictive to most restrictive?
  - What should you consider to know which modifier to use when you make a field? A method?

- 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
- What is polymorphism?
- Not from last class, before that:
  - 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)?
- Review from Python
  - What are abstract classes and interfaces?
  - How are they useful?

A Scenario

- 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 Version

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

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

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:

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

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

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

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

`Car.java`

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

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...

```java
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>
```

The type it compares

- Declared method:

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

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/21/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()
- 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

Example Use of an Interface

- Recall: `Arrays.sort(array)`
  - ➤ `Arrays.sort` sorts arrays of *any* Object class that implements the `Comparable` interface
    - Overloaded method, so can also pass in arrays of primitive types

Implementing an Interface with Generics

- In the class definition, specify that the class will implement the interface and specify its type it will accept/operate on.

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

Generics in Comparable

With Generics

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

Without Generics

```java
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;
}
```

Chicken.java

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
        - Partial implementation
    - `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 Classes

- `static`, `private`, and `final` methods cannot be abstract
  - Because cannot be overridden by a child class
- `final` class cannot contain abstract methods
  - Because class cannot be inherited
- 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* for methods

Summary: Defining Abstract Classes

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

- Typically used as a base class for a bunch of 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

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

COLLECTIONS

Collections

- Sometimes called *containers*
- Group multiple elements into a single unit
- Store, retrieve, manipulate, and communicate aggregate data
- Represent data items that form a natural group
  - Poker hand (a collection of cards)
  - Mail folder (a collection of messages)
  - Telephone directory (a mapping of names to phone numbers)

Java Collections Framework

- *Unified architecture* for representing and manipulating collections

- More than arrays

  ➢ More flexible, functionality, dynamic sizing

- In `java.util` package

Collections Framework

- **Interfaces**
  - ➤ Abstract data types that represent collections
  - ➤ Collections can be manipulated *independently* of implementation
- **Implementations**
  - ➤ Concrete implementations of collection interfaces
  - ➤ Reusable data structures
- **Algorithms**
  - ➤ Methods that perform useful computations on collections, e.g., searching and sorting
  - ➤ Reusable functionality
  - ➤ *Polymorphic*: same method can be used on many different implementations of collection interface

Core Collection Interfaces

- Encapsulate different types of collections
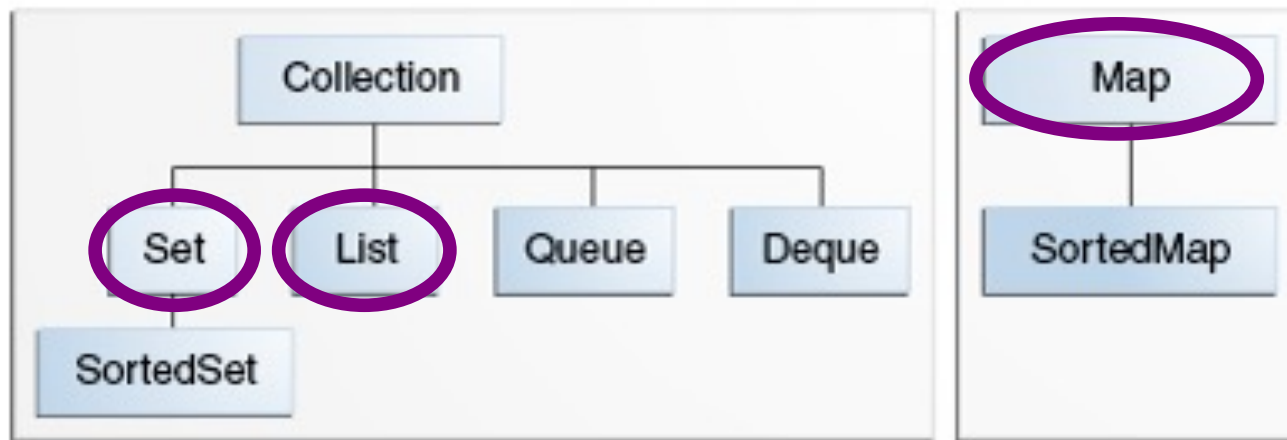
LISTS

List Interface

- An *ordered* collection of elements
- Can contain duplicate elements
- Has control over where objects are stored in the list

List Interface

- **boolean add(<E> o)**
  - ➤ Returns boolean so that List can refuse some elements
    - e.g., refuse adding null elements
- **<E> get(int index)**
  - ➤ Returns element at the position index
  - ➤ Different from Python: no shorthand
    - Can't write ~~list[pos]~~
- **int size()**
  - ➤ Returns the number of elements in the list
- And more!
  - ➤ contains, remove, toArray, …

List Interface

- `boolean` add(`<E>` o)
  - Returns boolean so that List can refuse some elements
    - e.g., refuse adding `null` elements
- `<E>` get(`int` index)
  - Returns element at the position index
  - Different from Python: no shorthand
    - Can't write ~~list[pos]~~
- `int` size()
  - Returns the number of elements in the list
- And more!
  - `contains, remove, toArray`, …

Common `List` Implementations

- **`ArrayList`**
  - ➤ Resizable array

- **`LinkedList`**

> When should you use one vs the other?

Common `List` Implementations

● `ArrayList`

- ➢ Resizable array
- ➢ Used most frequently
- ➢ Fast

● `LinkedList`

- ➢ Use if adding elements to ends of list
- ➢ Use if often delete from middle of list
- ➢ Implements `Deque` and other methods so that it can be used as a stack or queue

How would you find the other implementations of `List`?

API Notes

- `ArrayList` and `LinkedList` extend from `AbstractList`, which implements `List` interface

GENERICS

Generic Collection Interfaces

- Declaration of the `Collection` interface:

Type parameter

```
public interface Collection<E> …
```

- ➢ `<E>` means interface is generic for **e**lement class
- When declare a `Collection`, **specify type** of object it contains
  - ➢ Allows compiler to verify that object's *type* is correct
    - Reduces errors at runtime

Always declare type contained in collections

- Example, a hand of cards:

```
List<Card> hand = new ArrayList<Card>();
```

Added in Java 7:

```
List<Card> hand = new ArrayList<>();
```

Comparing: Before & After Generics

- **Before Generics**

```
List myList = new LinkedList();
myList.add(new Card(4, "clubs"));
…
Card x = (Card) myList.get(0);
```

- List of `Objects`
- Need to cast to the desired child class

Comparing: Before & After Generics

- **Before Generics**

```
List myList = new LinkedList();
myList.add(new Card(4, "clubs"));
…
Card x = (Card) myList.get(0);
```

- List of Objects
- Need to cast to the desired child class

- **After Generics**

```
List<Card> myList = new LinkedList<>();
myList.add(new Card(4, "clubs"));
…
Card x = myList.get(0);
```

- If you try to add not-a-Card, compiler gives an error

✓ Improved readability and robustness

Types Allowed with Generics

- Can only contain `Objects`, not primitive types

- Autoboxing and Autounboxing to the rescue! (for next time!)

Assignment 3

- Start of a simple video game
  - ➢ Game class to run
  - ➢ GamePiece is parent class of other moving objects
- Can complete everything now
- Due Wednesday before class