Abstraction, Interfaces, Inheritance Outline

- Abstract Data Types, ADTs
- Java Interfaces
- Class hierarchy and inheritance

Data Types

- Data Types you already know in Java include boolean, int, String, and arrays
  - arrays contain elements of other data types
- You create new data types in Java by defining new classes
- Such a data type defines data and operations

  The type defines the operations that can be done on the data, the meaning of the data, and the way values of that type can be stored. (from wikipedia, “data type”)
- an Abstract Data Type (ADT) is similar to a data type, but is a design tool and cannot be used directly as code

Abstract Data Types

- you don't need to know how a phone works in order to use one!
  - same for cars, fridges, programming libraries, and many other things
- all you need to know is how to operate it:

  what operations it supports and how to use those operations
  - this knowledge is important when using a library in a program
- abstract data types (ADTs) group data with the operations on that data
- the data is usually protected as fields of the class
- many of the operations are public and provide access to the data
- simple example:
  - a water bottle has four operations: drink, fill, remove cap, place cap on
  - the "data" of a water bottle is the water it contains, and whether the cap is on

using Abstract Data Types

- an ADT can be particularly effective when it models something in the real world
- does a car ADT have to have a method that returns the car's color?
- that depends on what we have in mind for the ADT:
  - for a police or a showroom application, we might want such a method
  - if our only interest is to model driving time, the color is not necessary
- objects in the real world are generally more complex than the models we make of them
- the features of an object that are represented in the ADT as data are the object's **attributes**
- the actions of an object that are operations of the ADT are the object's **behaviors**

Abstract Data Type implementation

- when we do write code to implement an ADT, we might want code that specifies the attributes and behaviors, but is not executable
- in Java, we can use **interfaces** to define attributes and behaviors
- more in general, an interface has a collection of methods and constants of a class
  - but provides no implementation of these methods
- classes **satisfy** an interface if they implement those methods and constants

Example interface

```
public interface Teacher {
  /* each of these methods is public and abstract, as is the default
for interfaces */
  /* @param start The time at which teaching begins
     @param finish The time at which teaching stops
     @return The contents of the teaching
   */
  Voice teach(Time start, Time finish);
  /* @param question The subject that needs to be clarified
     @return The clarification
   */
  Voice askClarification(Voice question);
  int numberOfExams = 3;   /* implicitly "public abstract final" */
}
```

Using the example interface

- a Java class can then claim to implement such an interface:

```
public class Edo implements Teacher { ...
public class Kristy implements Teacher { ...
```

  The compiler will report an error if the class fails to provide the needed methods and constants.

- code can then use these classes interchangeably, if all that is needed is the methods and constants specified by the interface:

```
Teacher lectureTeacher = new Edo();
Teacher labTeacher = new Kristy();
```

- wherever `lectureTeacher` and `labTeacher` are used, their type is `Teacher`, not the type of the underlying object

Using the example interface, cont'd

```
Student I = new ...();
try {
  try {
     I.listen(lectureTeacher.teach(10:30, 11:45));
    } catch (NotUnderstood something) {
    I.listen(lectureTeacher.askClarification(something));
    }
  } catch (MissingInformation somethingElse) {
    /* MissingInformation could be from the teaching or
the clarification */
Teacher myLabTeacher = labTeacher;
I.listen(myLabTeacher.askClarification(somethingElse));
```

Java Interfaces to implement ADTs

- a Java interface can be used to describe an ADT:
- the data itself is not public, so

    the code that uses

        the class that implements the interface

    has no direct access to the class's internal data structures
- the methods each represent an operation on the data
- in-class exercise (all together): design an ADT for a door
- in-class exercise (groups of 2-3 people): design an ADT for a fridge
- how would you implement ArbitraryPrecisionInterface.java?

Java classes and objects

- Java classes can implement abstract data types
- the order of data field and method declarations within the class does not matter
  - a class variable is also known as a **field** or **data field**
- an object is an instance (a specific example) of a class:

```
String first = new String("1st");
String upper = first.toUpperCase();
```

  first and upper are objects belonging to the class `String`
- objects in a program are generally referenced by a variable: the same variable can reference different objects at different times
- a **variable** is a memory location that can store an object reference, or a value of a simple type such as boolean, int, etc.
- a variable (of an object type) that does not reference an object has the special value `null`. Attempting to access (reference) a method or field of this object will throw a `NullPointerException`

Special Methods

- a **constructor** initializes new objects of the given class
  - there may be more than one constructor, with different parameters
  - the no-arguments constructor is called by Java when no other constructor is called by the programmer
  - each constructor must set the values of fields to a valid initial value. Such values depend on the class and sometimes on the constructor
  - interfaces do not list constructors
- sometimes programmers make the data of a class protected or private, but provide **accessor** methods to read the value, or **mutator** methods to modify the value
- accessor and mutator methods hide the implementation details
- a `toString` method is invoked automatically by Java when the object is passed to a method or operator that needs a string

Inheritance

- a class (the subclass) is usually derived from another class (the superclass)
  - when every B is an A, class B should be a subclass of class A
- for example, every cat is an animal, so class Cat should be a subclass of class Animal
- this is described as inheritance, or B extends A

```
public class Cat extends Animal
```

  - everything in Java ultimately inherits from `Object`
- for example, exceptions such as `IndexOutOfBoundsException` are derived from `RuntimeException`, which is derived from `Exception`
- so `IndexOutOfBoundsException` is a subclass of `RuntimeException`;
- and `RuntimeException` is the superclass of `IndexOutOfBoundsException` (and of many other exceptions)
- `IndexOutOfBoundsException` inherits all the methods and data fields of `RuntimeException`
- I could create a class `ICS211` which extends `ICSCourse`

Special class variables: `this` and `super`

- the predefined field `this` in the code of any class refers to the object of this class
- the predefined field `super` in the code of a subclass refers to the object of the superclass:

```
public class Child extends Parent {
  private String myName;
  public Child (String name) {
    super ();       // call no-argument constructor for parent
    this.myName = name; // initialize local data
  }
  // omitting this hasFeature method would give the same result
  public boolean hasFeature(String feature){
    return super.hasFeature(feature);
  }
  public String name(){
    return myName;
  }
}
```