CS310 – Advanced Data Structures and Algorithms

APIs and their Interfaces

Spring 2021

Note: Project 0 (pa0) is now available at our website
www.cs.umb.edu/cs310
API and Encapsulation

- **Application Programming Interface (API):** a group of function specifications that are meant to provide some service or services to programs, clients of the API.

- **Data encapsulation:** Hiding the *data* of the implementation of the API functions so well that the client can’t get at it by normal programming methods.

- The clients **have** to use the API functions to get their work done.
API – application programming interface

Example: C string library functions: strcpy, strcmp
Example: The Flips.java example from section 1.2 in S&W: Client code for Counter.java
API and Encapsulation

Implementation details and class variables

API
Advantages of Encapsulation

- Clear statement of functionality in use – what is intended/provided by the software.
- Partition of responsibility/code. Important if many programmers.
- Can share General Purpose objects across many apps. Saves coding. At worst have to make minor improvements later.
- Safety of contents. Primitive functions can check arguments, etc.,
- Debugability. No mystery changes to data – can breakpoint the functions that change data.
That clear statement of functionality in use – the API helps with this

It describes the calls into the system

Note that the term "API" is often used for a set of network calls that also specify what a system can do.

-- That use of API is more particularly called a “REST API”

We will use API to mean a set of specific function calls, for us Java method calls and constructor calls

Let’s look at a simple examples to start...
Java provides **interfaces** as a way to specify the methods (and constants) for an API, separately from classes used to implement the API, i.e., provide code for it.

**geeksforgeeks** intro:
A Vehicle can be a car or a bike. Here is its interface:

```java
interface Vehicle {
    void changeGear(int a);
    void speedUp(int a);
    void applyBrakes(int a);
}
```

This means the Vehicle API has those methods. To use the API, we need code for it, i.e., an implementation—

```java
// implementation of Vehicle, for bicycles:
class Bicycle implements Vehicle{ ... }  // code for API
```

Now we can use Vehicle and Bicycle as two related types.
Java Interfaces for expressing APIs

geeksforgeeks intro: A Vehicle can be a car or a bike.

interface Vehicle {
    void changeGear(int a);
    void speedUp(int a);
    void applyBrakes(int a);
}

// implementation of Vehicle, for bicycles:
class Bicycle implements Vehicle { ... }  // code for API

Now we can use Vehicle and Bicycle as two related types. But note that we can “new” only Bicycle, not Vehicle:

    // creating an instance of Bicycle
    // doing some operations
    Bicycle bicycle = new Bicycle();
    bicycle.changeGear(2);
    bicycle.speedUp(3);
    bicycle.applyBrakes(1);

But this code doesn’t use Vehicle at all...
geeksforgeeks intro: A Vehicle can be a car or a bike.

interface Vehicle {
    void changeGear(int a);
    void speedUp(int a);
    void applyBrakes(int a);
}

// implementation of Vehicle, for bicycles:
class Bicycle implements Vehicle {
    … } //code for API

Now we can use Vehicle and Bicycle as two related Java types.

// create an instance of Bicycle
// type a variable as Vehicle, do some operations
Vehicle bicycle = new Bicycle();
bicycle.changeGear(2); // call the API via interface
bicycle.speedUp(3); // call the API via interface
bicycle.applyBrakes(1);

We see that we can call the interface methods using a variable of the interface type. Those calls execute code in Bicycle.java.
intro: A Vehicle can be a car or a bike.

```java
interface Vehicle {
    void changeGear(int a);
    void speedUp(int a);
    void applyBrakes(int a);
}

// implementation of Vehicle, for bicycles and racecars:
class Bicycle implements Vehicle{
    // code for Bicycle
}
class RaceCar implements Vehicle{
    // code for RaceCar
}

Vehicle v = null; // v not yet assoc. with an object
v = new Bicycle(); // now v refs a Bicycle object
v.changeGear(2);  // executes Bicycle.changeGear
v.speedUp(3);

v = new RaceCar(); // switch v to ref a RaceCar object
v.speedup(100);    // executes RaceCar.speedup
```

These two Vehicle objects could be in an array of Vehicle.
Our textbook doesn’t use Java interfaces as much as it could (and should, in my opinion).

It does use a few, as outlined on page 100: Comparable, Comparator, Iterator, and Iterable, all defined in the JDK.

Even though it describes six classes that implement the symbol table API (page 363), it doesn’t set up a Java interface to describe the API methods get, put, contains, keys, etc.

The JDK itself uses interfaces extensively, so we will too. For example, HashMap and TreeMap implement Map, the JDK name for a symbol table.

Let’s look at an S&W class and write an interface for it...
Example – 2-dimensional point \((x, y)\)

Construct two points:
Point2D \(p1 = \text{new Point2D}(3,4)\);
Point2D \(p2 = \text{new Point2D}(5,7.4)\);

Get back x-y coordinates:
\[
\text{double } x1 = p1.x(); \quad \text{double } y1 = p1.y();
\]

Compute distance between them:
\[
\text{double } \text{distance} = p1.\text{distanceTo}(p2);
\]

Convert to polar coordinates:
\[
\text{double } \text{angle} = p1.\text{theta}();
\text{double } \text{radius} = p1.\text{r}();
\]

The API specifies the methods that can be used with a certain class. An API lies between two bodies of code, the client code and the described class implementation.
We can figure out that this class must* contain the x and y coordinates of a point, so it looks like this:

```java
public class Point2D {
    private double x, y; // private data
    // constructor - create a Point2D
    public Point2D(double x, double y) {...}
    // public methods: for the API
    public double x() { return x; }
    public double y() { return y; }
    public double distanceTo(Point2D p) { ... }
    ...
    // maybe private methods too...
}
```

*Well, it could contain radius and theta instead…
In our textbook, its API is written out as follows: See page 77 (I’ve dropped the draw() method to simplify).

These are the public methods and constructor of the class, without “public” on each:

```java
public class Point2D

Point2D(double x, double y) // create a point
double x() // x-coordinate
double y() // y-coordinate
double r() // radius (polar coordinates)
double theta() // angle (polar coordinates)
double distanceTo(Point2D that) // Euclidean distance from this point to that
```

Note that this is just documentation, not compilable by Java
We can extract the API methods of this class into a Java interface.

We’ll call it Pont2Dim. It is compilable.

```
public interface Point2Dim {
    double x();   // x-coordinate
    double y();   // y-coordinate
    double r();   // radius (polar coordinates)
    double theta(); // angle (polar coordinates)
    double distanceTo(Point2Dim that); // Euclidean distance from this point to that
}
```

We end up with a minimalistic summary of the actions ("behavior") of the class.

Note that the constructor isn’t (and can’t be) in the interface, just the public methods (and constants)
To make this interface usable, a concrete class has to acknowledge that Point2Dim in fact expresses its interface like this:

```java
public class Point2D implements Point2Dim {
    private double x, y;
    // constructor - create a Point2D
    public Point2D(double x, double y) {...}
    // public methods: for the API
    public double x() { return x;}
    public double y() { return y;}
    // use if type in param now--
    public double distanceTo(Point2Dim p) {...}
    ...
}
```

This shows that all the methods of the interface are methods of the class. The class can have additional methods, public or not.
How can we use this interface?

We can define variables of this interface type and use them to call the methods of the interface.

But to get started, we need to use the concrete class to create an object.

```java
public class TestPoint {
    public static void main(String[] args) {
        // create object using concrete class
        // -- then use interface variable for actions:
        // -- we’ll be doing this a lot...
        Point2Dim p1 = new Point2D(2, 3);
        // call a method via interface: get back 2.0
        double x1 = p1.x();
    }
}
```
Using Point2Dim with multiple concrete classes

- Why bother?
- Suppose you are interfacing to another Java class that has a Point class too
- Say PointXY, but different details in the concrete class
- If they both implement Point2Dim, you can write code that works for both!

```java
public class TestPoints {
    public static void main(String[] args) {
        // create objects using concrete classes
        // --then use interface variables for them
        Point2Dim p1 = new Point2D(2, 3);
        Point2Dim p2 = new PointXY(4, 5);
        //
        double dist = p1.distanceTo(p2); // interoperability!
    }
}
```
Of course you probably don’t often need to use 2 kinds of 2D points

We don’t bother with interfaces for many detail objects

We do want to use them for serious reusable classes

Notably container classes and symbol table (AKA map) classes

There generics help a lot, but can’t do the whole job

Quick generics example: public class Queue<Item> can use any Item class.
When there are competing implementations with the same API, an interface is called for.

For example, a symbol table can be implemented with an array, ArrayST in cs210 project 5, or by a binary tree structure, RedBlackBST in project 5, or a Kd-tree.

This cries out for a unifying interface, to make them interchangeable, and document their important behavior.

In cs210, an interface was presented, BasicST, and ArrayST implements BasicST: this is a start. BasicST is not in the textbook, however.

We’ll continue this effort.
Above we did: `Point2Dim p1 = new Point2D(2,3);`
That’s assigning a `Point2D` object to a `Point2Dim` variable
How come Java allows this, given its famous “strong typing”?
  – Because it recognizes `Point2D` as a subtype of `Point2Dim`
(Java can see they are locked together in this way by the interface arrangement)
In OO language, we can say `Point2D ISA Point2Dim`
  – and thus it can be the value of a `Point2Dim` variable
If you have been using our textbook (recent cs210, etc.), you have seen some cases of type compatibility without much fanfare (too little in my opinion). Some examples:

- General sort functions for elements implementing Comparable:
  Chap. 2
- Implementing keys() for a symbol table by creating a Queue and returning it as an Iterable: pg. 382. Works because a Queue ISA Iterable because Queue implements Iterable (page 121).
- BST based on a key that itself ISA Comparable, Section 3.2

Note these are JDK-defined interfaces (Iterable, Comparable), but we'll see there are many more in the JDK, and we can create them ourselves.
Example – bank account.

Construct an account:
BankAccnt ba = new BankAccnt("JJ", 234, 100);

Modify balance:
ba.withdraw(20); ba.deposit(10);

Access balance:
ba.balance; // Incorrect!!
ba.getBalance(); // Correct!!

The API specifies the methods that can be used with a certain class. An API lies between two bodies of code, the client code and the class implementation.
Java interface:

- All its methods are public, and it can have constants as well.
- It can’t be instantiated, i.e., we can’t use “new Account()”
- A class is said to implement this interface if it provides definitions (i.e. code) for all these methods.

```java
public interface Account {
    // withdraw amt from this BA
    int withdraw(int amount);
    // deposit amt to this BA
    void deposit(int amount);
    // return balance for this BA
    int getBalance();
}
```
Bank Account Example

Any class that has balance going up and down can implement this Account interface. Here BankAccount is a subtype of Account, i.e., BankAccount ISA Account

```java
public class BankAccount implements Account {
    // constructor - create a BA
    public BankAccount(String nm, int _id, int bal) {...}
    // Account API functions
    public int withdraw(int amt) {...}
    public void deposit(int amt) {...}
    public int getBalance() {...}
    // Fields - all private
    private int id;
    private String name;
    private int balance;
}
```
Client code: a BankAccount ISA Account

// client code example, v1:
public class TestBankAccount {
    public static void main(String [] args)
    {
        BankAccount ba = new BankAccount("JJ", 234, 100);
        ba.withdraw(20);
        ba.deposit(10);
    }
}

// client code example, v2: ba can have type Account here
public class TestBankAccount {
    public static void main(String [] args)
    {
        Account ba = new BankAccount("JJ", 234, 100);
        ba.withdraw(20);
        ba.deposit(10);
    }
}
A class A ISA B if A is the same type as B, or A’s class implements B, an interface*.

Note that it is an asymmetric relationship: A ISA B does not mean B ISA A

If A ISA B, then we say A is type-compatible with B. We can assign \( b = a \), where \( a \) is an A and \( b \) is a B object.

-- i.e., an object \( b \) can refer to a B or any type that ISA B

Object references can be tested for type compatibility with instanceof: if “a instanceof B” is true, then \( b = a \) will work.

-- But we try to write code that doesn’t use instanceof: the types should be clear from context.

* We are not covering class inheritance to start with, but later we’ll see that a class can be a subtype (subclass) of another class.
Every class is a subclass of Object.*

So class A ISA Object, for every A.

We can say Object x = a, for any a of class A.

-- But this loses the useful typing information, so we use it sparingly.

More commonly, we use the inherited Object methods: these are available on *any object*.

Most common Object methods: equals, hashCode, toString

Another we’ll use: getClass, useful for implementing equals correctly.

* We said we won’t cover “class A subclass of class B” until later, but in the case that B is Object, we need this info.
public class BankAccount implements Account {
    // constructor - create a BA
    public BankAccount(String nm, int _id, int bal) { ... }
    // Account API functions
    public int withdraw(int amt) { ... }
    public void deposit(int amt) { ... }
    public int getBalance() { ... }
    // Fields - all private
    private int id;
    private String name;
    private int balance;
}

Note that there are no methods named equals, hashCode, toString, and that’s perfectly OK.
public class TestBankAccount {
    public static void main(String [] args) {
        Account ba = new BankAccount("JJ", 234, 100);
        ba.withdraw(20);
        ba.deposit(10);
        System.out.println(ba.toString());  //call toString on ba
    }
}

C:\cs310\bankaccount>javac *.java
C:\cs310\bankaccount>java TestBankAccount
BankAccount@24d46ca6

- We see that Object.toString() is not very useful: it just supplies the memory address of the object.
- Note this code is available for download: see link under today’s class on the class webpage.
Our example has:
Account ba = new BankAccount("JJ", 234, 100);
So that ba is of type Account, the interface.

Yet ba.toString() works fine, even though toString() is
not part of the interface, the normal requirement of calls
on interface objects. Java simply calls toString of the
referenced object, which must exist since any object
ISA Object.

See StackOverflow 1214163 for discussion of this
obscure point of Java.

So the full rule on what you can call on an interface
object is any method of the interface or any Object
method.
Not everything is an object in Java.

Primitive types: int, double, boolean, char
(also long, short, float, and byte, but we won’t use them)

Each of these has a corresponding object type: Integer, Double, Boolean, Character, and conversion is automatic—see Autoboxing page 132-133.

Integer i = 10;

Here 10 is a literal int constant that is “autoboxed” into the object Integer i.