Abstract Data Type

Representation and Implementation

in Java

Data Types

- A data type is characterized by:
  - a set of *values*
  - a *data representation,* which is common to all these values, and
  - a set of *operations,* which can be applied uniformly to all these values

Abstract Data Types

- An Abstract Data Type (ADT) is:
  - a set of *values*
  - a set of *operations,* which can be applied uniformly to all these values
- To *abstract* is to leave out information, keeping (hopefully) the more important parts
  - What part of a Data Type does an ADT leave out?

Abstract Data Types

- **<u>Data type</u>**: set of values and operations on those values.
- Ex: int, String, ComplexNumber, BigNumber,Tour, . . .

- **<u>Abstract data type</u>** is a **<u>data type</u>** whose internal representation is hidden. They are language independent.

- Separate implementation from design specification.

- CLASS: provides data representation and code for operations.
- CLIENT: uses data type as black box.
- INTERFACE: contract between client and class.

Intuition

Client

Interface
- volume
- change channel
- adjust picture
- decode NTSC, PAL signals

Implementation
- cathode ray tube
- electron gun
- Sony Wega 36XBR250
- 241 pounds, $2,699

client needs to know how to use interface

implementation needs to know what interface to implement

Implementation and client need to agree on interface ahead of time.

Intuition

Client

Interface
- volume
- change channel
- adjust picture
- decode NTSC, PAL signals

Implementation
- gas plasma monitor
- Pioneer PDP-502MX
- wall mountable
- 4 inches deep
- $19,995

client needs to know how to use interface

implementation needs to know what interface to implement

Can substitute better implementation without changing the client.

ADT Implementation in Java

Java ADTs.

- Keep data representation hidden with `private` access modifier.
- Define interface as operations having `public` access modifier.

```
public class Complex {
   private double re;
   private double im;

   public Complex(double re, double im) { . . . }
   public double abs()                  { . . . }
   public String toString()             { . . . }
   public Complex conjugate()           { . . . }
   public Complex plus(Complex b)       { . . . }
   public Complex times(Complex b)      { . . . }
}
```

Advantage: can switch to polar representation without changing client.

Note: all of the data types we have created are actually ADTs!

Modular Programming and Encapsulation

ADTs enable modular programming.

- Split program into smaller modules.
- Separate compilation.
- Different clients can share the same ADT.

ADTs enable encapsulation.

- Keep modules independent (include `main` in each class for testing).
- Can substitute different classes that implement same interface.
- No need to change client.

Classes in Java

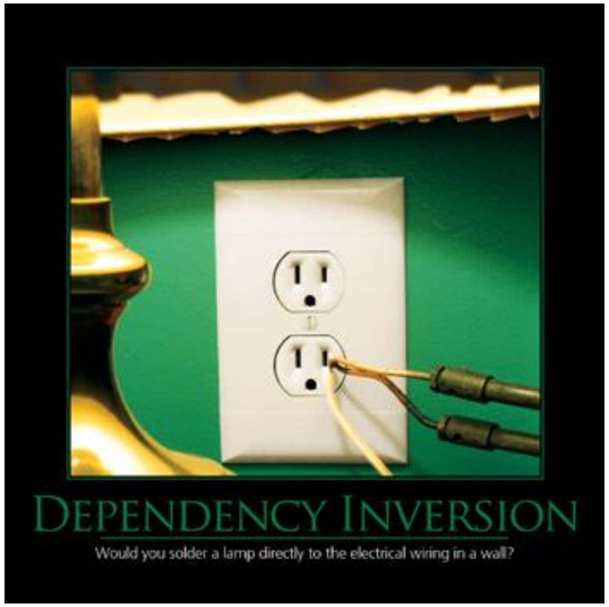
- A class defines a *data type*
  - The possible *values* of a class are called objects
  - The *operations* on the objects are called methods
  - The *data representation* is all the fields that are contained within the object
- If there is no external access to the data representation, you have an *abstract data type*
- Sun' s Java classes are (almost all) abstract data types

ADTs are better than DTs

- It is the responsibility of a class to protect its own data, so that objects are always in a valid state
    - Invalid objects cause program bugs
    - By keeping the responsibility in one place, it is easier to debug when invalid objects are present

ADTs are better than DTs

- The less the user has to know about the implementation, the easier it is to use the class
- The less the user *does* know about the implementation, the less likely s\he is to write code that depends on it
- "Less is more"


DEPENDENCY INVERSION
Would you solder a lamp directly to the electrical wiring in a wall?

DEPENDENCY INVERSION PRINCIPLE
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Data representation in an ADT

- An ADT must obviously have *some* kind of representation for its data
  - The user need not know the representation
  - The user should not be allowed to tamper with the representation
  - Solution: Make all data private
- But what if it's really more convenient for the user to have direct access to the data?
  - Solution: Use *setters* and *getters*

Example of setters and getters

```
class Pair {
  private int first, last;

  public getFirst() { return first; }
  public setFirst(int first) { this.first = first; }

  public getLast() { return last; }
  public setLast(int last) { this.last = last; }
}
```

Aside: Naming setters and getters

- Setters and getters should be named by:
  - Capitalizing the first letter of the variable (first becomes First), and
  - Prefixing the name with get or set (setFirst)
  - For boolean variables, you can replace get with is (for example, isRunning)

What’s the point?

- We can *claim* that a class is an abstract data type if we make all data private
- However, if we then provide unlimited access by providing simple getters and setters for everything, we are only fooling ourselves

What’s the point?

Rules:

- Only supply getters and setters if there is a clear use for them
- Do not supply getters and setters that depend on the particular implementation you are using
- Never, *never* write a setter that could be used to create an invalid object!
  - Your setters should do all possible error checking before they change the object

And another thing...

- Setters and getters allow you to keep control of your implementation
- For example, you decide to define a Point in a plane by its x-y coordinates:
  - `class Point { public int x; public int y; }`
- Later on, as you gradually add methods to this class, you decide that it's more efficient to represent a point by its angle and distance from the origin, θ and ρ
- Sorry, you can't do that—you'll break too much code that accesses x and y directly
- If you had used setters and getters, you could redefine them to *compute* x and y from θ and ρ

Contracts

Every ADT should have a contract (or specification) that:

- Specifies the set of valid values of the ADT
- Specifies, for each operation of the ADT:
  - Its name
  - Its parameter types
  - Its result type, if any
  - Its observable behavior
- Does *not* specify:
  - The data representation
  - The algorithms used to implement the operations

Importance of the contract

- A contract is an agreement between two parties; in this case
  - The implementer of the ADT, who is concerned with making the operations correct and efficient, *and also* with preserving the flexibility to make changes later
  - The applications programmer, who just wants to *use* the ADT to get a job done
- It doesn't matter if you are *both* of these parties; the contract is *still essential* for good code
- This separation of concerns is essential in any large project

Promise no more than necessary

- For a general API, the implementer should provide as much generality as feasible
    - This is the way Java classes are (mostly) written
- But for a specific program, the class author should provide only what is essential at the moment
    - In Extreme Programming terms, "You ain't gonna need it!"
    - Your documentation should not expose anything that the application programmer does not need to know
- If you design for generality, it's easy to add functionality later—but removing it may have serious consequences

Implementing an ADT

To implement an ADT, you need to choose:

- a data representation that
  - must be able to represent all possible values of the ADT
  - should be private
- a set of methods that support normal use of the ADT
  - The user must be able to create, possibly modify, and examine the values of the ADT
- an algorithm for each of the possible operations that
  - must be consistent with the chosen representation
  - all auxiliary (helper) operations that are not in the contract should be private

Writing the contract

- In most cases, the Javadoc for a class *is* the contract
- This means:
  - The Javadoc documentation should describe what the class is for and how it should be used
  - The Javadoc documentation should ***not*** describe implementation details
  - Also, now is a good time to read **Documentation Comments** (rules 38 to 58) in *The Elements of Java Style*

Writing the contract

- Sometimes, however...
  - The particular implementation makes certain operations efficient at the cost of making others inefficient
  - For example, Java provides both ArrayList (fast random access) and LinkedList (fast insertions and deletions)
  - The user needs to know this information, but doesn’t need detailed implementation information

Interfaces

- In many cases, an interface makes a better contract than a concrete implementation
- Why? Because the interface can only describe what the ADT does
    - It cannot describe variables
    - It cannot provide an implementation
    - Unfortunately, Java interfaces are not allowed to have constructors, but (as of Java 8) they are allowed to contain static methods. So we can implement creator operations as static methods. This design pattern, using a static method as a creator instead of a constructor, is called a factory method.

Interfaces

simplified version of the Set interface:

```
public interface Set<E> {

// example of creator method

/** Make an empty set.
* @return a new set instance, initially empty
*/
public static Set<E> make() { ... }
```

Interfaces

```
// examples of observer methods

/** Get size of the set.
* @return the number of elements in this set. */
public int size();

/** Test for membership.
* @param e an element
* @return true iff this set contains e. */
public boolean contains(E e);
```

Interfaces

```
// examples of mutator methods

/** Modifies this set by adding e to the set.
* @param e element to add. */
public void add(E e);

/** Modifies this set by removing e, if found.
* If e is not found in the set, has no effect.
* @param e element to remove.*/
public void remove(E e);
```

Interfaces

- Design principle: "Program to an interface, not an implementation"
  - Always avoid getting locked in to a specific implementation
  - Example: You need a list?
    - List myList = new ArrayList(); is better than
      ArrayList myList = new ArrayList();
    - This makes it much easier if you later decide a LinkedList is better, because it prevents you from using implementation-specific methods

Real life example

```
/**
* This data type represents a tweet from Twitter.
*/
public class Tweet {
    private final String author;
    private final String text;
    private final Date timestamp;
/**
* Make a Tweet.
* @param author Twitter user who wrote the tweet.
* @param text text of the tweet
* @param timestamp date/time when the tweet was sent
*/
public Tweet(String author, String text, Date timestamp) {
    this.author = author;
    this.text = text;
    this.timestamp = timestamp;
}
}
```

Real life example

```
/** @return Twitter user who wrote the tweet */
public String getAuthor() {
return author;
}

/** @return text of the tweet */
public String getText() {
return text;
}

/** @return date/time when the tweet was sent */
public Date getTimestamp() {
return timestamp;
}
```

Summary

- A Data Type describes values, representations, and operations
- An Abstract Data Type describes values and operations, but *not* representations
  - An ADT should *protect its data* and *keep it valid*
    - All, or nearly all, data should be private
    - Access to data should be via getters and setters
  - An ADT should provide:
    - A *contract*
    - A *necessary and sufficient* set of operations for the intended use of the ADT
- Depend on the *abstraction*, not the particular *implementation*

ADT Examples

Fraction

Consider an example:

We would like to creat an ADT that represents *fractions*;

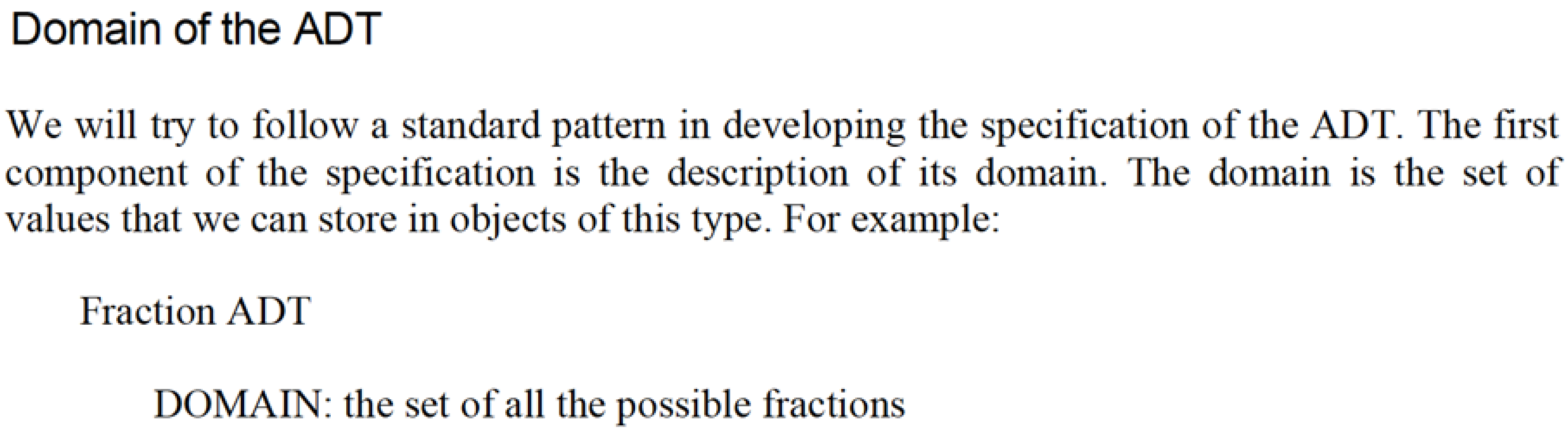
The domain is the set of all possible fractions (fractions are the entities we are interested in manipulating).

The operations we would like to perfom on fractions are:

- check if two fractions are equal
- read the numerator and denominator of the fraction
- simply a fraction
- add and multiply fraction

We desire a new data type (called **Fraction**) so that we can declare variables of this type and use them in our program.

Fraction



Domain of the ADT

We will try to follow a standard pattern in developing the specification of the ADT. The first component of the specification is the description of its domain. The domain is the set of values that we can store in objects of this type. For example:

Fraction ADT

DOMAIN: the set of all the possible fractions

Fraction

Constructor: we can assume one constructor that generates a brand new object of type fraction; the operation requires as input the numerator and denominator of the fraction we wish to create. The result should be the new fraction created. We will denote this as follows:

```
Fraction createFraction (int num, int den)

// creates a new fraction, having num as numerator and den as denominator

// the result is an object of type Fraction
```

Destructors: we will ignore this for the moment

Fraction

Inspectors: we will consider the following inspectors:

int getNumerator()

// the operation, applied to a fraction, returns the numerator of the fraction

int getDenominator()

// the operation, applied to a fraction, returns the denominator of the fraction

boolean isEqualTo (Fraction f)

// the operation, applied to a fraction, return true if the fraction has the same

// value as the fraction f, false otherwise

Fraction

Modifiers: we will use the following modifiers:

void simplifyFraction()

// this operation, applied to a fraction, simplifies the fraction to its normal form

void incrementFraction (Fraction f)

// this operation, applied to a fraction, change its value by adding to it the value