Data Abstraction in Java

Rob Miller
Fall 2007
Abstract data type = type (set of values) + operations

- An ADT expresses a **contract** between implementer and clients

Client uses operations on abstract type:
- constructors
- observers
- producers

Operations have **preconditions** (satisfied by client) and **postconditions** (satisfied by implementer)

**representation type R**

**abstract type A**

**representation invariant RI** describes legal reps

**abstraction function AF** maps legal reps to abstract values

Abstraction barrier preserves **representation independence**
java.lang.String represents an immutable sequence of characters

public class String {
    public String(char[]) ... constructors
    ...
    public char charAt(int)
    public int length()
    public boolean startsWith(String s) ...
    ...
    public String concat(String s)
    public String substring(int start, int end) ...
}

© Robert Miller 2007
String’s Rep

public class String {
    private final char[] chars;
    private final int offset;
    private final int count;

    public String() {
    }

    public String(String s, int start, int end) {
        ... // check that start and end are in range
        return new String(chars, offset+start, end-start);
    }

    // For example:
    String s = "table";
    String t = s.substring(1, 5);

    // This kind of sharing between String instances is only possible because
    String is immutable
java.net.URL represents an immutable URL

http://www.mit.edu:80/~6.005

public class URL {
  public URL(String s)
  public String getProtocol()
  public String getHost()
  public String getPath()
  public int getPort()
  ...
  public InputStream openStream()
  ...
}

InputStream is not immutable (it’s a state machine), but the URL itself is not changed by opening a stream.
URL’s Rep

http://www.mit.edu:80/~6.005

protocol host port path

public class URL {
    private String protocol;
    private String host;
    private int port;
    private String path;
    ... // other fields
}

But is java.net.URL really immutable?

➢ When an (MIT) computer is connected to the network, it returns true; when disconnected, it returns false
  • equals() tries to resolve the domain name (www, www.mit.edu) to an IP address (18.7.22.83) before comparing equality
➢ So the abstraction function is actually changing at runtime
  
  AF(protocol,host,port,path) =
  if DNS(host) == IPaddress, then protocol + ”://” + ipaddress + ...
  if DNS(host) fails, then protocol + ”://” + host + ...

Rl(protocol,host,port,path,...) =
    protocol != null
    && -1 ≤ port ≤ 65535
    && (protocol ∈ {“http”,”https”,”ftp”}
        ⇒ host != null)
    && ... // other fields

© Robert Miller 2007
**ADT Examples from the Java Library**

**java.util.Date** represents a date and time

```java
public class Date {
    public Date()
    public Date(String s)
    ...
    public int getYear()
    public int getMonth()
    public int getDay()
    ...
    public void setYear(int y)
    public void setMonth(int m)
    public void setDay(int d)
    ...
}
```

Date is a **mutable** ADT. What can go wrong?

© Robert Miller 2007
Building an Immutable ADT on Date

Trans represents an immutable transfer of money between two bank accounts

```java
public class Trans{
    private Date date;
    private int amount;
    ...
    public Date getDate() {
        return date;
    }
}
```

rep exposure: returns a reference to a mutable object inside the rep. Client can break the immutability of Trans, even inadvertently:

```java
Date sameTimeNextYear(Trans t) {
    Date d = t.getDate();
    d.setYear(d.getYear() + 1);
    return d;
}
```

To safely return a mutable part of the rep, we have to make a copy of it:

```java
public Date getDate() {
    return new Date(date.getDate());
}
```

Mutable data types like Date can’t be shared as casually as immutable data types like String and URL.

© Robert Miller 2007
More Abstract Data Types

Let’s look at some other examples

- Actions in the Rock Paper Scissors game

\[ A = \{ \text{rock, paper, scissors} \} \]

- Described by enumerating all possible elements

\[ R_1 = \text{int} \]
\[ R_1(n) = \begin{cases} 3 & \text{if } 0 \leq n \leq 2 \\ -1 & \text{otherwise} \end{cases} \]

- One operation:

\[
\text{interface RPS} \{ \\
\text{boolean beats(RPS that)} \\
\} \\
\text{defined such that}
\begin{align*}
\text{rock}.\text{beats}(&\text{scissors}) \\
\text{paper}.\text{beats}(&\text{rock}) \\
\text{scissors}.\text{beats}(&\text{paper})
\end{align*}
\]

© Robert Miller 2007
Typesafe Enumeration

Here’s a rep that has exactly one object for each element of the abstract type

abstract class RPS {
    public abstract boolean beats(RPS that);
    public static final RPS ROCK = Rock.INSTANCE;
    public static final RPS PAPER = Paper.INSTANCE;
    public static final RPS SCISSORS = Scissors.INSTANCE;
}

class Rock extends RPS {
    private Rock() {}  // private constructor guarantees only one Rock
    public static final Rock INSTANCE = new Rock();
    public boolean beats(RPS that) {
        return that == SCISSORS;
    }
}

... // define Paper and Scissors classes similarly

- Unlike int or string rep, this rep has a vacuous rep invariant (true)
Java Enumerations

Java has a special syntax for this pattern

```java
enum RPS {
    ROCK, PAPER, SCISSORS
}
```

➤ RPS is an abstract class (can’t be instantiated), and RPS.ROCK is a singleton object instance of an implicit subclass of RPS

➤ Enum values can even be used in a switch statement (like integers):

```java
public void playSoundEffect(RPS move) {
    switch (move) {
        case ROCK: play("thud.wav"); break;
        case PAPER: play("crumple.wav"); break;
        case SCISSORS: play("snip.wav"); break;
    }
}
```

➤ But note that RPS is an object, not an integer

```java
int move = ROCK; // type error
```

© Robert Miller 2007
Operations on Enums

Enumerations can have fields and methods just like an abstract class

```
enum RPS {
    ROCK {
        public boolean beats(RPS that) {
            return that == SCISSORS;
        }
    },
    PAPER { ... },
    SCISSORS { ... }

    public abstract boolean beats(RPS that);
}
```
More Examples of Enum

enum Day { SUNDAY, MONDAY, ..., SATURDAY }
enum Style { PLAIN, BOLD, ITALIC }
enum DayHalf { AM, PM }

Why are enums better than primitive types?

int day;
p.s.f. int SUN=0;
p.s.f. int MON=1;
...
(representing an enumerated type with integer constants)

Answer #1: static type checking
A bug like this is caught at compile time:
   Style style = SUNDAY;
but this bug may never be caught:
   int style = SUN;

boolean isAM;
_cookie_ representing a two-element type with boolean

Answer #2: code readability
Which is easier to read:
   new Time(3, PM)
or
   new Time(3, false)

© Robert Miller 2007
Pairs are a common pattern in simple ADTs

Also triples, quadruples, or generally $n$-tuples for fixed $n$

Rat = BigInt x BigInt
BigInt = int $^N$
Trans = Date x int

capture the pattern in a reusable, general-purpose ADT

```java
/** Pair represents an immutable pair of objects. */
class Pair {
    private final Object a;
    private final Object b;
    public Pair(Object a, Object b) {
        this.a = a; this.b = b;
    }
    public Object a() { return a; }
    public Object b() { return b; }
}
```

final keyword means the fields can be assigned at most once (so Java compiler helps us ensure that Pair is immutable).

final isn't a panacea. Consider BigInt's representation:
private final int[] digits;
does it really help here?
Using Pairs

Pairs are useful for clumps of data that don’t merit a specialized ADT of their own

Pair divideWithRemainder(BigInt n, BigInt d) {
    ... // divide n by d to find quotient & remainder
    return new Pair(quotient, remainder);
}

Pair findPrimes(int n) {
    ... // find all primes <= n
    return new Pair(largestPrime, numberOfPrimes);
}

Pair trim(String s) {
    ... // remove spaces from start and end of s
    return new Pair(trimmedString,
                    new Pair(numSpacesAtStart,
                             numSpacesAtEnd));
}
Downside of Pairs of Objects

Using Pairs of Objects requires lots of downcasting

Pair p = trim("      hello         ");
String t = (String) p.a();
static type of p.a() is Object, even though trim() guarantees it’s a String

Programmer can easily make a mistake here – compiler accepts any class as a possible cast from Object, so a mistake won’t be found until runtime

Solution: parameterize Pair by types of the two objects

class Pair<A, B> {
    private final A a;
    private final B b;
    public Pair(A a, B b) {
        this.a = a; this.b = b;
    }
    public A a() { return a; }
    public B b() { return b; }
}

Pair is now a generic data type

© Robert Miller 2007
Using Generic Pairs

Now we can declare an accurate static type for each use of Pair

```java
Pair<BigInt,BigInt> divideWithRemainder(BigInt n, BigInt d) {
    ... // divide n by d to find quotient & remainder
    return new Pair<BigInt,BigInt>(quotient, remainder);
}

Pair<Integer,Integer> findPrimes(int n) {
    ... // find all primes <= n
    return new Pair<Integer,Integer>(largestPrime, numPrimes);
}

Pair<String,Pair<Integer,Integer>> trim(String s) {
    ... // remove spaces from start and end of s
    return new Pair<String,Pair<Integer,Integer>>
        (trimmedString, new Pair<Integer,Integer>
            (numSpacesAtStart,
             numSpacesAtEnd));
}
```

© Robert Miller 2007
Using Generic Pairs

The elements of the generic pair now have static types that match their runtime types

```java
Pair<String,Pair<Integer,Integer>> p = trim(" hello ");
```

What is the static type of each of the following expressions?

- `p.a()`
- `p.b()`
- `p.b().a()`
- `p.b().b()`

Long type declaration!

- Alas, Java has no way to give a shorthand name to an instantiated generic type, e.g.: `TrimResult = Pair<String, Pair<Integer,Integer>>`
- To spare clients this pain, long tuples may be better as a specialized ADT, e.g. class `TrimResult { private String s; private int start; private int end; ... }`
Patterns for Implementing ADTs in Java

ADT as class
ADT as interface
ADT as abstract class
ADT as Class

So far we’ve implemented BigInt as a simple class

- We have representation independence because we’re free to change BigInt’s representation internally.
- But if more than one representation might be useful (e.g., BigInt = int for representing small integers), then this pattern forces the client to commit to one or the other – even though both implement the same abstract type.

```java
class BigInt {
    private int[] digits;
    ...
}

static BigInt fact(BigInt n) {
    if (n.isZero()) {
        return new BigInt(1);
    }...
}

class SmallInt {
    private int n;
    ... // same methods as BigInt
}
```

Client can’t use SmallInt rep when appropriate, without lots of change

© Robert Miller 2007
If the abstract type is an interface, then client code doesn’t depend on which class actually implements it.

```java
interface BigInt {
    boolean isZero();
    boolean isOne();
    BigInt incr();
    BigInt decr();
    BigInt plus(BigInt that);
    ...
}

class IntRep implements BigInt {
    private int n;
    ...
}

class ArrayRep implements BigInt {
    private int[] digits;
    ...
}

static BigInt fact(BigInt n) {
    if (n.isZero()) {
        return new BigInt(1);
    }
    ...
}
```

Problem: Java interfaces can’t have constructors.

We could use new IntRep(1) or new ArrayRep(1) here, but that creates the same rep dependence we’re trying to avoid.
**Factory Methods**

**Solution:** instead of a Java constructor, use a static method to make values of the abstract type

A method that makes objects is called a **factory method**

```java
public static BigInt make(int n) {
    return new IntRep(n);
}
public static BigInt make(String s) {
    return s.length() < 10 ? new IntRep(s)
        : new ArrayRep(s);
}
```

Now we’re free to choose the representation class we’re using without changing any client code

**But Java interfaces can’t have static methods either!**

One solution: put the static methods in a utility class (analogous to java.lang.Math or java.util.Collections)

```java
class BigInts {
    private BigInts() {}  
    public static BigInt make(int n) { ... }  
    public static final BigInt ZERO = make(0);  
    public static final BigInt ONE = make(1);
}
```

Private constructor prevents utility class from being instantiated

Also a good place to put constants
Using Factory Methods

```java
import adt2.BigInt;
import adt2.BigInts;
...
static BigInt fact(BigInt n) {
    if (n.isZero()) return BigInts.make(1);
    else return n.times(fact(n.decr()));
}
```

Java also lets you import static methods and constants

```java
import adt2.BigInt;
import static adt2.BigInts.*;
...
static BigInt fact(BigInt n) {
    if (n.isZero()) return make(1);
    else return n.times(fact(n.decr()));
}
```

Or even:

```java
if (n.equals(ZERO)) return ONE;
```
ADT as Abstract Class

A Java abstract class is a class with some methods left unimplemented (abstract)

- Abstract class can have fields, constructors, and static methods and fields (like an ordinary class)
- It can’t be instantiated (like an interface), so need to use a factory method

```java
abstract class BigInt {
    public static BigInt make(int n) {
        return new IntRep(n);
    }
    public abstract boolean isZero();
    public abstract boolean isOne();
    public abstract BigInt incr();
    public abstract BigInt decr();
    ...
}
```

```java
class IntRep extends BigInt {
    private int n;
    ...
}
```

```java
class ArrayRep extends BigInt {
    private int[] digits;
    ...
}
```

Abstract methods are like interface method declarations (in fact, interface methods are implicitly “public abstract”)

© Robert Miller 2007
ADT as Abstract Class

Abstract class can also provide default method implementations

➤ This only works for operations that can be written abstractly (entirely in terms of other operations, without touching the representation directly)

➤ Method is inherited by IntRep and ArrayRep, unless they override it

```java
abstract class BigInt {
  ...
  public BigInt remainder(BigInt that) {
    return this.minus(that.times(this.divide(that)));
  }
}
```
Downside of Abstract Classes

In Java, a class can extend only one abstract class, but it can implement many interfaces
➢ This is called single inheritance

```java
class IntRep extends BigInt {
    ...
}
```

using `abstract class`, the rep class can implement only one ADT

```java
class IntRep implements BigInt, Rat {
    ...
}
```

using `interfaces`, the same rep might implement more than one ADT

Interfaces are usually preferred because of their flexibility
Another Problem: Binary Operations

Binary operations take an object of type BigInt

But what if the method needs to get at the representation?

```java
class IntRep implements BigInt {
    private int n;
    ...
    public BigInt plus(BigInt that) {
        return new IntRep(this.n + that.n);
    }
}
```

why does the compiler reject this?

This is a problem whether BigInt is an interface or an abstract class
Solution: Downcasting

Downcast asserts that the abstract type is actually an instance of the rep type

```java
public BigInt plus(BigInt _that) {
    TwoInts that = (IntRep) _that;
    return new IntRep(this.n + that.n);
}
```

- The compiler checks that IntRep is a subtype of BigInt
- The runtime system checks that the object really is a IntRep, and throws a ClassCastException if it isn’t

Downcasting creates a possible type error at runtime

```java
BigInt x = new IntRep(2);
BigInt y = new ArrayRep(3);
x.plus(y)  // throws exception
```

- Downcasting is avoidable when binary operation can be written abstractly (like remainder() method we saw earlier)

© Robert Miller 2007
Summary

Immutability vs. Mutability
- Sharing is always safe with immutable types
- Uncontrolled sharing is dangerous with mutable types
- Ensuring immutability can be subtle – making your fields final isn’t enough

Patterns for implementing ADTs in Java
- As simple class
- As interface + rep classes + utility class for factory methods
- As abstract class + rep classes

Generic classes
- Using type parameters to make general-purpose classes like Pair typesafe

Enumerations
- Typesafe enumeration pattern, and Java enum implementation of it

© Robert Miller 2007