Data Abstraction

Data Abstraction

- **Allows us to extend the programming language in use (e.g. Java) with new data types.**

- **What new types are needed depends on the application domain.**
  - **compiler: tables, stacks, …**
  - **banking: accounts, …**

Data abstraction consists of a set of objects, e.g. accounts, and a set of methods

- **In Java: data abstraction is implemented as a class or an interface**

Data Abstraction (contd.)

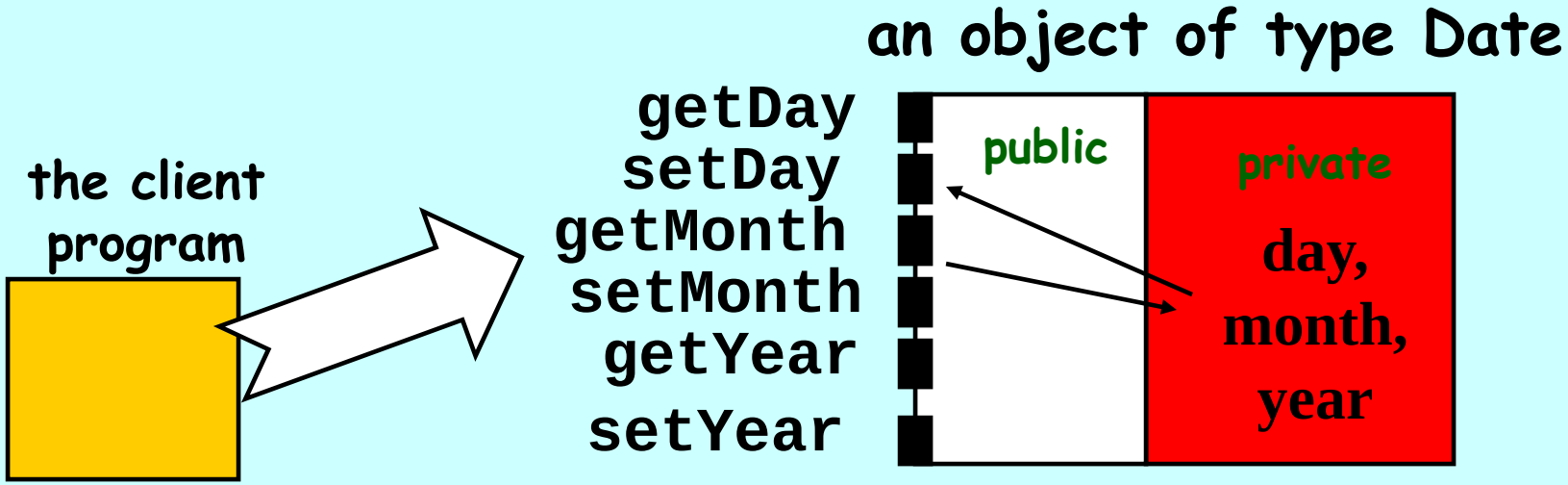
- **Why methods are needed to be the part of the data type ?**

- **Allows us to defer decisions about data structures until the uses of the data are fully understood.**

- **Data Abstraction is valuable during program modification and maintenance.**

When we use Data Abstraction, we do not need to care about representation of objects

- details about how the objects are represented are hidden. They can be accessed only through the methods.



Data Abstraction (contd.)

data abstraction = <objects, instance methods>

How to specify Data Abstraction in Java

- **In Java, new data types are defined by classes where each class defines a name for the type, a set of constructors and a set of methods.**

- **In Java, new data types can also be defined by interfaces.**

- **Specification consists of explaining what the methods do. It takes the form of comments in the code.**

Form of a Data Abstraction Specification

```
<visibility> class dname {
   // OVERVIEW: brief description of the behavior of the
   //                    data type

   // specs for constructors
   // constructors

   // specs for methods
   // instance methods
}
```

Data Abstraction Specification has three parts

- **OVERVIEW: gives a brief decription of the data abstraction, including a way of viewing the abstract objects in terms of "well-understood" concepts. It also states whether objects of the type are mutable or not.**

- **constructors: defines how new objects are initialized**

- **instance methods: defines the methods that allow access to objects once they have been created.**

Example: Specification of IntSet data abstraction

```
public class IntSet {
  // OVERVIEW: IntSets are mutable, unbounded
  //           sets of integers.
  //           A typical IntSet is {x1,..., xn}


  // constructors
  public IntSet ()
     // EFFECTS: Initializes this to be empty
```

```
// methods
public void insert (int x)
   // MODIFIES: this
   // EFFECTS: Adds x to the elements of  this,
   //          i.e. this_post = this + {x}

public void remove (int x)
    // MODIFIES: this
    // EFFECTS: Removes x from this,
    //           i.e. this_post = this - {x}
```

```
    public boolean isIn (int x)
       // EFFECTS: if x is in this returns true
       //          else returns false

    public int size ( )
       // EFFECTS: Returns the cardinality of this

    public int choose ( ) throws EmptyException
      // EFFECTS: if this is empty, throws EmptyException
      //          else returns an arbitrary element of
      //          this
}
```

Example: Specification of Poly data abstraction

```
public class Poly {
  // OVERVIEW: Polys are immutable polynomials with
  //integer coefficients. A typical Poly is:
  // c0 + c1x + …

  // constructors
  public Poly()
  //EFFECTS: Initializes this to be the zero polynomial

  public Poly(int c, int n) throws NegativeExponentException
  //EFFECTS: If n < 0 throws NegativeExponentException else
  //  initializes this to be the Poly cx^n
```

```
// methods
public int degree()
//EFFECTS: Returns the degree of this, i.e. the largest
//  exponent with a non-zero coefficient. Returns 0 if
//  this is the zero Poly.

public int coeff(int d)
//EFFECTS: Returns the coefficient of the term of this whose
//  exponent is d.

public Poly add(Poly q) throws NullPointerException
//EFFECTS: If q is null throws NullPointerException else
//  returns the Poly this + q.
```

```
  public Poly mul(Poly q) throws NullPointerException
  //EFFECTS: If q is null throws NullPointerException else
  //  returns the Poly this * q.

  public Poly sub(Poly q) throws NullPointerException
  //EFFECTS: If q is null throws NullPointerException else
  //  returns the Poly this - q.

  public Poly minus(Poly q)
  //EFFECTS: Returns the Poly -this.
}
```

Using IntSet data abstraction: Example

```
public static IntSet getElements (int[] a)
                        throws NullPointerException {
  //EFFECTS: If a is null throws NullPointerException
  //  else returns a set containing an entry for each
  //  distinct element of a

  IntSet s = new IntSet();
  for (int i = 0; i < a.length(); i++) s.insert(a[i]);
  return s;
}
```

This method returns an IntSet containing the integers in its array argument a; there are no duplicates in the returned IntSet.

class not shown

Using Poly data abstraction: Example

```
public static Poly diff (Poly p) throws NullPointerException {
  //EFFECTS: If p is null throws NullPointerException
  //  else returns the Poly obtained by differentiating p

  Poly q = new Poly ();
  for (int i = 1; i <= p.degree(); i++) {
    q = q.add( new Poly(p.coeff(i) * i, i - 1) );
  }
  return q;
}
```

This method returns a new Poly that is the result of differentiating its argument Poly.

class not shown

Implementing IntSet data abstraction

```
public class IntSet {
  // OVERVIEW: IntSets are mutable, unbounded
  //           sets of integers.
  //           A typical IntSet is {x1,..., xn}

  private Vector els; // the representation

  // constructors
  public IntSet () {
     // EFFECTS: Initializes this to be empty
    els = new Vector();
  }
```

```
// methods
public void insert (int x) {
   // MODIFIES: this
   // EFFECTS: Adds x to the elements of  this,
   //          i.e. this_post = this + {x}

  Integer y = new Integer(x);
  if(getIndex(y) < 0) els.add(y);
}
```

```
public void remove (int x) {
    // MODIFIES: this
    // EFFECTS: Removes x from this,
    //          i.e. this_post = this - {x}

  int i = getIndex(new Integer(x));
  if (i < 0) return;
  els.set(i, els.lastElement());
  els.remove(els.size() -1);
}
```

```
private int getIndex (Integer x) {
  //EFFECTS: if x is in this returns index where x
  //  appears else -1

  for (int i = 0; i < els.size(); i++)
  if (x.equals(els.get(i))) return i;
  return -1;
}
public boolean isIn (int x) {
  // EFFECTS: if x is in this returns true
  //          else returns false
  return getIndex(new Integer(x)) >= 0;
}
```

```
  public int size ( ) {
    // EFFECTS: Returns the cardinality of this
    return els.size();
  }

  public int choose ( ) throws EmptyException {
    // EFFECTS: if this is empty, throws EmptyException
    //          else returns an arbitrary element of
    //          this
    if (els.size() == 0)
       throw new EmptyException("IntSet.choose");
    return els.lastElement();
  }
}
```

Additional Methods inherited from Object class

- **public boolean equals(Object o): the default implementation provided by the Object class tests whether two objects have same identity or not.**

- **public String toString()**

- **public Object clone()**

Abstraction Function and Representation invariant

- **both are useful in understanding an implementation of a data abstraction**

- ***Abtraction function*:**
  - **captures the designer's intent in choosing a particular representation: what instance variables to use and how they relate to the abstract object they are intended to represent**

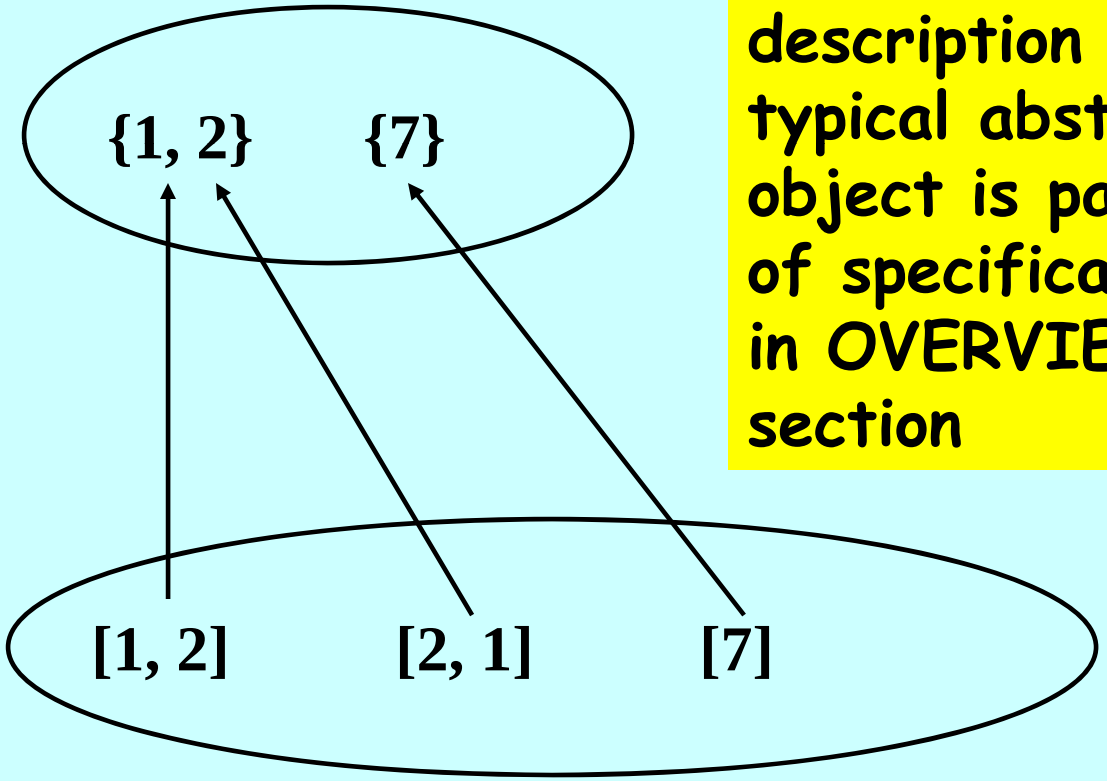
Abstraction Function and Representation invariant

- ***Rep Invariant*** **:**
  - **captures the common assumptions on which the implementations of constructors and methods are based.**

- ***Both the Abstraction function and the Rep invariant should be included as comment in the code. They should be included in the specification.***

Abstraction Function

- **Abstraction function AF : C → A maps the representation of an object (instance variables) to the abstract object being represented**



many-to-one

description of a typical abstract object is part of specification; in OVERVIEW section

Abstraction Function for IntSet Data Abstraction

```
AF(c) = {c.els[i].intValue | 0 <= i < c.els.size}
```

{x|p(x)} describes the set of all x such that the predicate p(x) is true

Representation Invariant

- A statement of a property that all legitimate objects satisfy is called a *representation invariant* or *rep invariant*

- A *representation invariant is a predicate*

  J: C → boolean

  that is true of legitimate objects.

Rep invariant for IntSet Data Abstraction

```
// The rep invariant is:
// c.els != null &&
//for all integers i, c.els[i] is an Integer &&
//for all integers i, j,
//(0 <= i < j < c.els.size =>
// c.els.[i].intValue != c.els.[j].intValue)
```

Informally:

```
// The rep invariant is:
// c.els != null &&
// all elements of c.els are Integers &&
// there are no duplicates in c.els
```

Implementing the Abstraction Function and the Representation invariant

- ***Abstraction function is implemented by toSring() method***

- ***Representation invariant is implemented by repOk() method***

Specification of repOk() method

```
public boolean repOk()
//EFFECTS: Returns true if the rep
// invariant holds for this;
// otherwise returns false
```

repOk() method for IntSet data abstraction

```
public boolean repOk ( ) {
  if (els == null) return false;
  for (int i = 0; i < els.size ( ); i++) {
    Object x = els.get(i);
    if (! (x instanceof Integer)) return false;
    for (int j = i + 1; j < els.size ( ); j++){
      if (x.equals(els.get(j))) return false;
    }
  }
  return true;
}
```

repOk() method is used in two ways

- **Test programs can call it to check whether an implementation of a data abstraction is preserving the Rep invariant or not.**

- **It can be used inside method and constructor implementations**

Properties of Data Abstraction Implementations

- ***Benevolent Side Effects*: an implementation performs *benevolent side effect* if it modifies the rep without affecting the abstract state of the object. Only possible when AF is many-to-one.**

- ***Exposing the rep* : an implementation *exposes the rep* if it provides users of its objects with a way of accessing some mutable components of the rep.**

Benevolent Side Effects: Example

Rational Number:

```
Abstraction function is:
//  A typical rational number is n/d
// The AF is: AF(c) = c.num/c.denom


// The rep invariant:
// c.denom > 0
```

```
public boolean equals( Rational r) {
    if (r == null) return false;
    if (num == 0) return r.num == 0;
    if (r.num == 0) return false;
    reduce();
    r.reduce();
    return (num == r.num && denom == r.denom);
}
private void reduce() {
    ….
    num = num/g;
    denom = denom/g;
}
```

Exposing the Rep

- **Means the implementation makes mutable components of the rep (i.e. instance variables) accessible to the outside the class.**
  - **have instance variables not private**

Exposing the Rep

- **even if all instance variables private, still expose the rep ??**

e.g. in IntSet data abstraction:

```
public Vector allEls() { return els; }
```

Method returns a mutable object in the rep

e.g. in IntSet data abstraction:

```
public IntSet( Vector v ) {
    if( v != null ) els = v;
    else els = new Vector();
}
```

Constructor or a Method makes a mutable argument a part of the rep.

Design Issues

1. **Mutability: A data abstraction is mutable is it has any mutator methods; otherwise it is immutable.**
   - **Immutable abstractions are safer than mutable ones**
   - **New Objects may be created and discarded more frequently for immutable ones than the mutable ones**

Design Issues

2. Kinds of Operations: There are four kinds:

- Creators: produce new objects "from scratch"
- Producers: produce new objects given existing objects as arguments (take objects of their type as inputs and create other objects of their type)
- Mutators: modify the state of their object
- Observers: provide information about the state of their object

Design Issues

3. Adequacy: A data type is *adequate* if it provides enough methods so that whatever users need to do with ts objects can be done conveniently and with reasonable efficiency.