Contracts for objects -- 0

Clear Box Specification

Contracts for classes

- Now we extend the idea of contracts to classes.
- As an example, we consider a class for representing rational numbers.
- We use a simple data structure:

Rational

```
class Rational {
    private double numerator ;
    private double denominator ;

    // requires d != 0.0
    // ensures denominator' != 0.0
    public Rational( double n, double d ) {
        numerator = n ; denominator = d ; }
```

Rational

```
// requires denominator != 0.0
// ensures result == numerator / denominator
public double toDouble() {
    return numerator / denominator ; }
```

- Does it make sense to require the client to ensure that the denominator is not 0 before calling toDouble?
- We should not force the client to reason in terms of the private fields of an object.
- To do so is contrary to the principles of information hiding and abstraction.

- Objects are meant to represent *things.*
- There are certain states of the objects that are sensible and certain states that --while representable by the fields– should not be reachable. These states do not represent things.

Invariants

- It is the job of the implementer of a class (not its clients) to ensure that the objects of the class do not reach states that are not sensible.
- An object invariant is a description of the states that of an object that are sensible.
- We start again. This time we state the invariant

Invariants

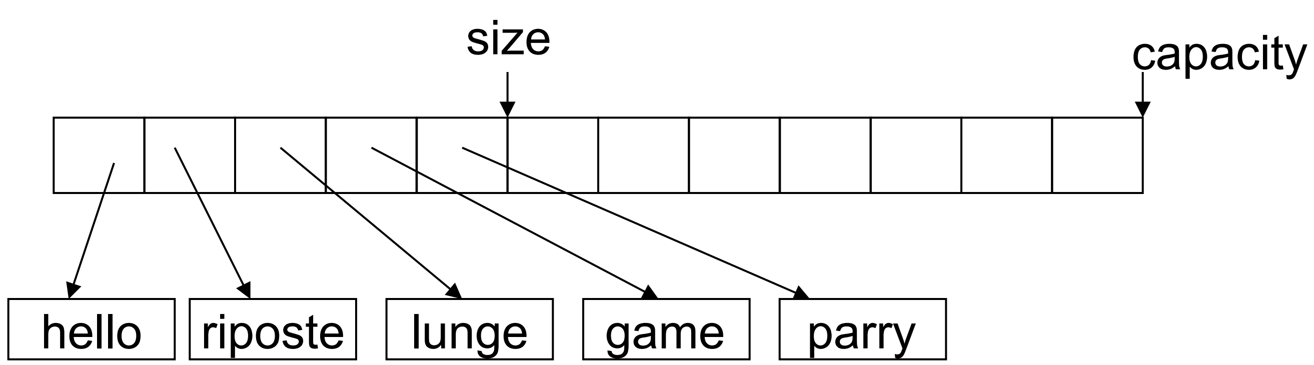
```
class Rational {
    // invariant denominator != 0.0
    protected double numerator ;
    protected double denominator ;

    // requires d != 0.0
    public Rational( double n, double d ) {
        numerator = n ; denominator = d ; }
    // ensures result == numerator / denominator
    public double toDouble() {
        return numerator / denominator ; }
```

Invariants

- The client coder does not need to think about the invariant.
- The implementer may assume that the invariant is true at the start of each method.
- But the implementer must also ensure that the each method and constructor of the class establishes the invariant at its end.
- Thus each method should preserve the invariant.

Another example

- As a second example, we use a dictionary that creates and records an association between strings and small integers.
- We use a simple data structure:



Data structure

```
class Dictionary {
    public final static INIT_CAPACITY = 10 ;
    protected int size = 0 ;
    protected String[] a = new String[INIT_CAPACITY ] ;

    // modifies size, a
    // ensures size' == 0 and a' != null
    public Dictionary() { … }

    // ensures result == size
    public int getSize() { … }

    // requires a != null
    // ensures result == a.length
    public int getCapacity() { … }
```

getInt

```
// requires str != null
//     and a != null and 0 <= size and size <= a.length
//     and (for all i in {0,1,…,size-1}, a[i]!=null)
//     and (for all i,j in {0,1,…,size-1}, a[i]==a[j] implies i==j)
// ensures
//  if( there is an i in {0,1,…,size-1}, str.equals(a[i]) )
//  then  0 <= result and result < size
//            and str.equals( a[result] ) )
//  else result == -1
public int getInt(String str ) { … }
```

putString

```
// requires str != null and size < a.length
//     and a != null and 0 <= size and size <= a.length
//     and (for all i in {0,1,…,size-1}, a[i]!=null)
//     and (for all i,j in {0,1,…,size-1}, a[i]==a[j] implies i==j)
// modifies a[size], size
// ensures 0 <= result and result <= size’
//    and str.equals(a[result]) and (size’ in {size, size+1})
//    and (for all i in {0,1,…,size-1}, a[i]’.equals(a[i])
//    and a != null and 0 <= size’ and size’ <= a’.length’
//    and (for all i in {0,1,…,size’-1}, a[i]’!=null)
//    and (for all i,j in {0,1,…,size’-1}, a[i]’==a[j]’ implies i==j)
int putString( String str ) { … }
```

Invariants

- Notice that certain facts about the fields are required by almost all methods.
- Thus these facts must be established by each constructor and *preserved* by each method
- These facts essentially define what it means for the state of the object to be *sensible*.

Invariants

- In this example, we require
  - That a points to an array:
    - a != **null**
  - That size is a valid index or equals the capacity:
    - 0 <= size and size <= a.length
  - That the first size items of the array are not null:
    - (**for all** i **in** {0,1,…,size-1}, a[i]!=**null**)
  - That the first size items of a be unique:
    - (**for all** i,j **in** {0,1,…,size-1}, a[i]==a[j] **implies** i==j)
  - If any of these “facts” is false, then the data structure is corrupt.

Invariants

- We call these facts the *object invariant* (sometimes called *class invariant*)
- The object invariant must be ensured by each constructor and each method of the class.
- The invariant may thus be assumed at the start of each method.

Rewriting the class

- Now we rewrite the Dictionary class, factoring out the invariant.

class Dictionary {

public final static INIT_CAPACITY = 10 ;

protected int size = 0 ;

protected String[] a = **new** String[INIT_CAPACITY] ;

// **invariant** a != **null**

// invariant 0 <= size **and** size <= a.length

// **invariant** (**for all** i **in** {0,1,…,size-1}, a[i]!=**null**)

// **invariant** (**for all** i, j **in** {0,1,…,size-1}, a[i]==a[j] **implies** i==j)

Rewriting the class

```
// modifies size, a
// ensures size == 0
public Dictionary() { … }

// ensures result == size
public int getSize() { … }

// ensures result == a.length
public int getCapacity() { … }
```

Rewriting the class

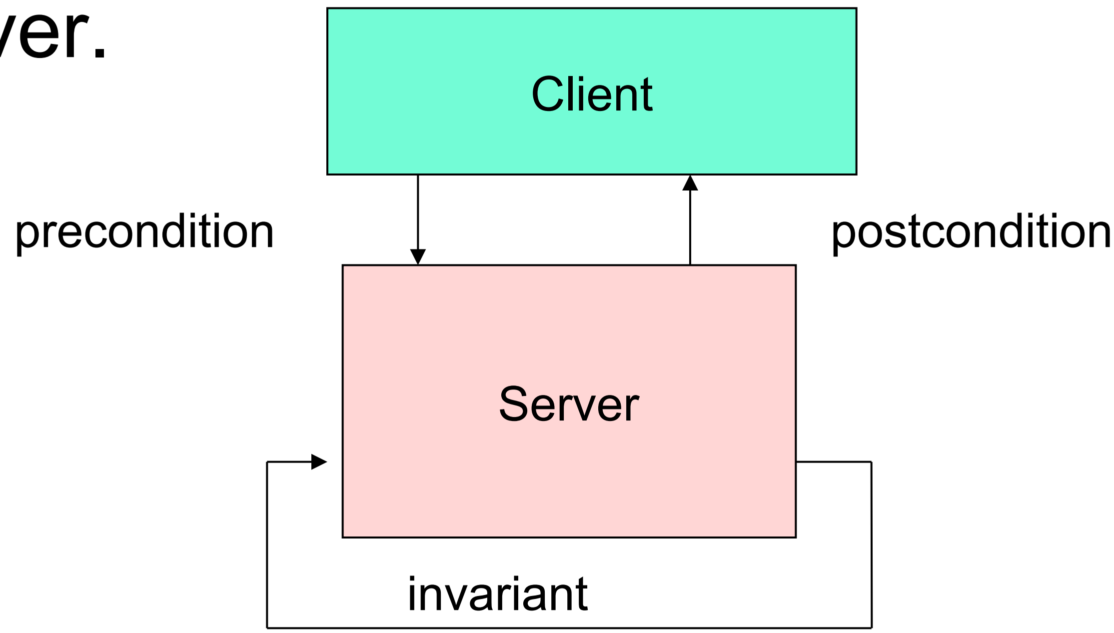
```
// requires str != null
// ensures
//  if( there is an i in {0,1,…,size-1}, str.equals(a[i]) )
//  then  0 <= result and result < size
//          and str.equals( a[result] ) )
//  else result == -1
public int getInt(String str ) { … }
```

Rewriting the class

```
// requires str != null and size < a.length
// modifies a[size], size
// ensures 0 <= result and result <= size'
//     and str.equals(a[result])' and (size' in {size, size+1})
//     and (for all i in {0,1,…,size-1}, a[i]'.equals(a[i])
int putString( String str ) { … }
```

Summary

- Note that the precondition now contains only things that the client actually has control over.



Flow of obligations

Invariants and defensive checks

- We can typically write the invariant as a method that is called at the end of each constructor and mutator (method that changes state). The check can be partial or full.
- To be extra careful, also call it at the start of each method.

```
protected void  invariant () {
    assert a != null ;
    assert 0 <= size && size <= a.length ; … }
```

Invariants and callbacks

- As mentioned, it is ok for the invariant to become untrue during the execution of a method, as long as it is restored by the end.
- Of course the invariant must be true also before any call that might cause a method invocation on the same object.
- In particular you have to be careful about calling other objects that might call back

E.g.

```
void someMutator() {
    …make some changes...
    invariant() ; // invariant should be true here
    notifyAllObservers() ;
    … do something else…
    invariant() ;
}
```

Invariants and shared objects

- Recall that in concurrent programming we should ensure that *shared objects* are never “owned” (aka “occupied”) by more than one thread at a time.
- The invariant of a shared object should be true whenever no thread owns it.
- It may be assumed at the start of synchronized methods.
- It should be true on return from synchronized methods.
- It should be true before any call to wait().
- It may be assumed after any call to wait().
- I.e. it is both a pre- and a postcondition of wait().