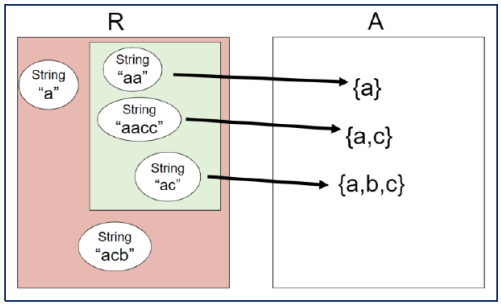
representation exposure: code outside the class can modify the representation directly.

immutable wrappers:
Collections.unmodifiableList() takes a (mutable) List and wraps it with an object that looks like a List , but whose mutators are disabled set()set(), add()add(), remove() throw exceptions.

- Consider the rep of `CharSet` in Example 3:

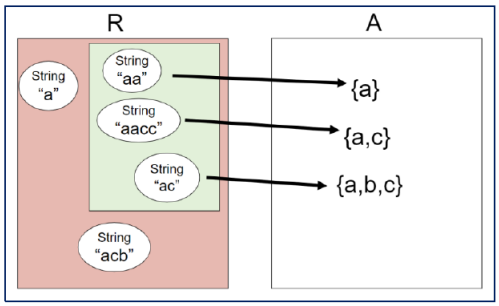
```
public class CharSet {
   private String s;
   // Rep invariant:
   //   s.length() is even
   //   s[0] <= s[1] <= ... <= s[s.length()-1]
   // Abstraction function:
   //   represents the union of the ranges
   //   {s[i]...s[i+1]} for each adjacent pair
   //   of characters in s
   ...
}
```



- Which of the following values of s satisfy this rep invariant?
  - ❑ `"abc"`
  - ❑ `"abcd"`
  - ❑ `"eeee"`
  - ❑ `"ad"`
  - ❑ `"adad"`
  - ❑ `""`

- Consider the rep of `CharSet` in Example 3:

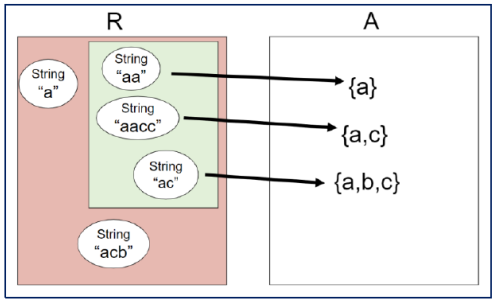
```
public class CharSet {
   private String s;
   // Rep invariant:
   //   s.length() is even
   //   s[0] <= s[1] <= ... <= s[s.length()-1]
   // Abstraction function:
   //   represents the union of the ranges
   //   {s[i]...s[i+1]} for each adjacent pair
   //   of characters in s
   ...
}
```



- Which of the following does AF("acfg") map to?
  - ○ `{a,b,c,d,e,f,g}`
  - ○ `{a,b,c,f,g}`
  - ○ `{a,c,f,g}`
  - ○ some other abstract value
  - ○ no abstract value, because `"acfg"` does not satisfy the rep invariant

- Consider the rep of `CharSet` in Example 3:

```
public class CharSet {
   private String s;
   // Rep invariant:
   //   s.length() is even
   //   s[0] <= s[1] <= ... <= s[s.length()-1]
   // Abstraction function:
   //   represents the union of the ranges
   //   {s[i]...s[i+1]} for each adjacent pair
   //   of characters in s
   ...
}
```



- Which of these values does the abstraction function map to the same abstract value as it maps `"tv"`?
  - ❑ `"ttv"`
  - ❑ `"ttuv"`
  - ❑ `"ttuuvv"`
  - ❑ `"tuv"`

To make an invariant hold, we need to:
○ make the invariant true in the initial state of the object; and
○ ensure that all changes to the object keep the invariant true.

Translating this in terms of the types of ADT operations, this means:
○ creators and producers must establish the invariant for new object instances; and
○ mutators and observers must preserve the invariant.

- Consider the following rep for an abstract data type:

```
/** Immutable type representing a set of letters, ignoring case */
class LetterSet {
   private String s;
   // Abstraction function:
   //    AF(s) = the set of the letters that are found in s
   //            (ignoring non-letters and alphabetic case)
   // Rep invariant:
   //    true
   /**
    * Make a LetterSet consisting of the letters found in chars
    * (ignoring alphabetic case and non-letters).
    */
   public LetterSet(String chars) {
       s = chars;
   }
   ... // observer and producer operations
}
```

- Using the *abstraction function* definition of equality for `LetterSet`, which of the following should be considered equal to `new LetterSet("abc")`?
  - ❑ `new LetterSet("aBc")`
  - ❑ `new LetterSet("")`
  - ❑ `new LetterSet("bbbbbc")`
  - ❑ `new LetterSet("1a2b3c")`

observational equality means that two references cannot be distinguished now , in the current state of the program.

Behavioral equality means that two references cannot be distinguished now or in the future, even if a mutator is called to change the state of one object but not the other.