1.3 Mutable and Immutable Objects

- Consider this Java class — an alternative to `Point`.

IPoint.java

```
public class IPoint {
   // Instance fields.
   private double x;        // x-coordinate of point.
   private double y;        // y-coordinate of point.

   // Constructors.
   public IPoint() { x = 0; y = 0;}
   public IPoint( double x, double y) {
      this.x = x; this.y = y;}
   public IPoint( IPoint p) {
      x = p.x; y = p.y;}

   // Instance methods (non-mutating).
   public double x() { return x;}
   public double y() { return y;}
   public double dist( IPoint p) {
      return Math.sqrt( (x-p.x)*(x-p.x)
                      + (y-p.y)*(y-p.y));}
   public String toString() {
      return "(" + x + "," + y + ")";}

   public IPoint right( double dx) {
      return new IPoint(x+dx,y);}

   public IPoint up( double dy) {
      return new IPoint(x,y+dy);}
}
```

- Then the only differences between `Point` and `IPoint` lie in methods `right()` and `up()`.

```
 // Class Point:  Mutating methods right() and up().
 public void right( double dx) { x += dx;}
 public void up( double dy) {y += dy;}
```

```
// Class IPoint:  Non-mutating methods  right() and up().
public Point right( double dx) {
   return new Point(x+dx,y);}
public Point up( double dy) {
   return new Point(x,y+dy);}
```

- Class `IPoint` has no mutating methods.
  - What is the consequence?
  - Unlike a `Point` object, an `IPoint` <u>object</u> cannot be modified, once it has been created and initialized.
    - In other words, an `IPoint` <u>object</u> is constant during its lifetime.
      - ◇ *Lifetime* means from just after construction, to just before destruction.
  - How do we know that an `IPoint` object cannot be modified?
    - Given

      `IPoint b = new IPoint(2,5);`

      how might we attempt to modify our `IPoint` object `b`?

i) *By manipulating a field (data member) directly, e.g.,* `b.x += 1`*?*

- ○ The compiler won't allow this because all fields of `IPoint` are private. (This is typical of classes)

ii) *By assignment, e.g,* `b = c`*?*

- ○ But assignment modifies only object references, not objects.

iii) *By invoking an* `IPoint` *method, e.g,* `b.right(8)`*?*

- ○ But all methods of `IPoint` are non-mutating.

iii) *By invoking an* `IPoint` *constructor?*

- ○ Constructors are invoked only during initialization — this doesn't count as part of the lifetime.

> In a reference semantics language, we say that the objects of a class are immutable if objects of that class cannot be modified during their lifetimes.

- An `IPoint` is immutable, but a `Point` is not.
- Typically we talk about "immutable objects", but note immutability is determined by the class.
    - Roughly, the objects of a class are immutable if the class provides no mutating methods.

- We must be careful not read more into the phrase

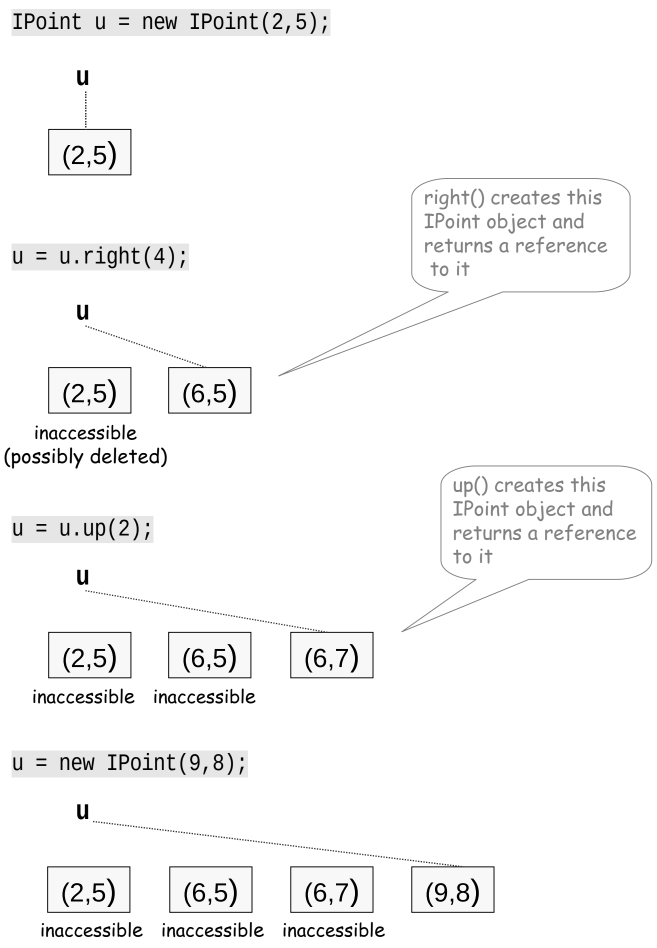
  "objects cannot be modified"

  than is really there.

- Consider this Java code

```
IPoint u = new IPoint(2,5);
System.out.println("u="+u);    // Displays: u = (2,5)
u = u.right(4);
System.out.println("u="+u);    // Displays: u = (6,5)
u = u.up(2);
System.out.println("u="+u);    // Displays: u = (6,7)
u = new IPoint(9,8);
System.out.println("u="+u);    // Displays: u = (9,8)
```

- How can the "value of `u`" change, if an `IPoint` is immutable?
    - An `IPoint` object is immutable.
    - But `u` is an `IPoint` reference.
    - Here is what happens.



`IPoint u = new IPoint(2,5);`

`u = u.right(4);`

right() creates this IPoint object and returns a reference to it
u
(2,5)
(6,5)
inaccessible (possibly deleted)

`u = u.up(2);`

up() creates this IPoint object and returns a reference to it
u
(2,5)
(6,5)
(6,7)
inaccessible
inaccessible

`u = new IPoint(9,8);`

u
(2,5)
(6,5)
(6,7)
(9,8)
inaccessible
inaccessible
inaccessible

- Note <u>no `IPoint` object was modified</u> during its lifetime.

- If we think of the "value" of an object reference `r` as the value of the object that `r` refers to, then there are two ways to change to "value" of `r`.

  i) Modify the object that to which `r` refers.

  ◇ For an immutable object, this is not possible.

  ii) Assign to `r`, so that `r` refers to a different object .

  ◇ This can be done even if objects of the class are immutable.

- The designers of the Java Library chose to make certain library class objects immutable. e.g.,

`String`	Perhaps the single most important class type in Java. (Python designers also made strings immutable)

`Integer`
`Double`
`Character`
etc
	“Wrapper classes” for primitive types

- Why design a class like `String` so `String` objects are immutable?

  i) The compiler can arrange that `String` objects be shared.

  - Two `String` objects having the same initial value can be stored at the same address.

  ii) Having objects be immutable may help with program verification, and may allow the compiler to do better code optimization.

- On the other hand,

  iii) Constructing small variations of a large objects becomes expensive, and sometimes awkward.

  - For example, replace character 10 of string `s` (length n, $n \geq 10$) by a blank.
    - ◇ C++: `s[10] = ' ';`
    - ◇ Java:
      ```
      s = s.substring(0,10) + " " +
          s.substring(11);
      ```
    - ◇ Note the C++ code is far more efficient — O(1) time vs O(n) time.

- Consider this C++ code.

```
// Efficient C++ code to read standard input into a string s.
//  Takes advantage of mutability of strings by using s+= ch to
//  append ch to s.  The running time is roughly linear in the
//  size of standard input.
string s;
char ch;
while ( cin.get(ch) )
   s += ch;
```

```
// Grossly inefficient C++ code to perform the same task.
//  Essentially treats strings as immutable, appending ch to s
//  using s = s + ch.  (Actually, due to the value-semantics
//  assignment, string objects are modified.)  The running time
//  is roughly quadratic  in the size of the standard input..
string s;
char ch;
while ( cin.get(ch) )
   s = s + ch;
```

- Time to read in a 200K file on a certain Sun workstation:

Efficient code:	< 0.1 sec
Inefficient code:	129.0 sec

- Since Java `String`s are immutable, only the inefficient code could be translated directly into Java, with `s` having type `String`.

  Note: Java permits `s += ch`, but treats it as equivalent to `s = s + ch`; no string object is modified.

- However, Java provides an alternate class: `StringBuffer`.
  - Unlike `String` objects, `StringBuffer` objects are mutable.
  - Otherwise, the capabilities of the two classes are somewhat similar.
  - Our efficient C++ code could be translated into Java with `s` becoming a `StringBuffer`.