# Object-Oriented Programming in a Functional Language

*October 16, 2018*

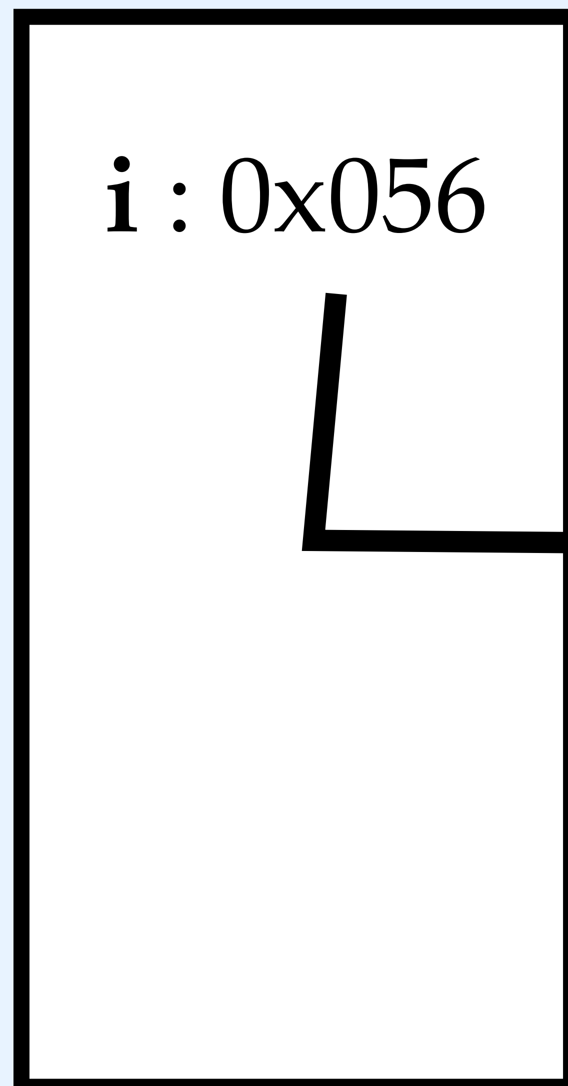
# Variable assignment

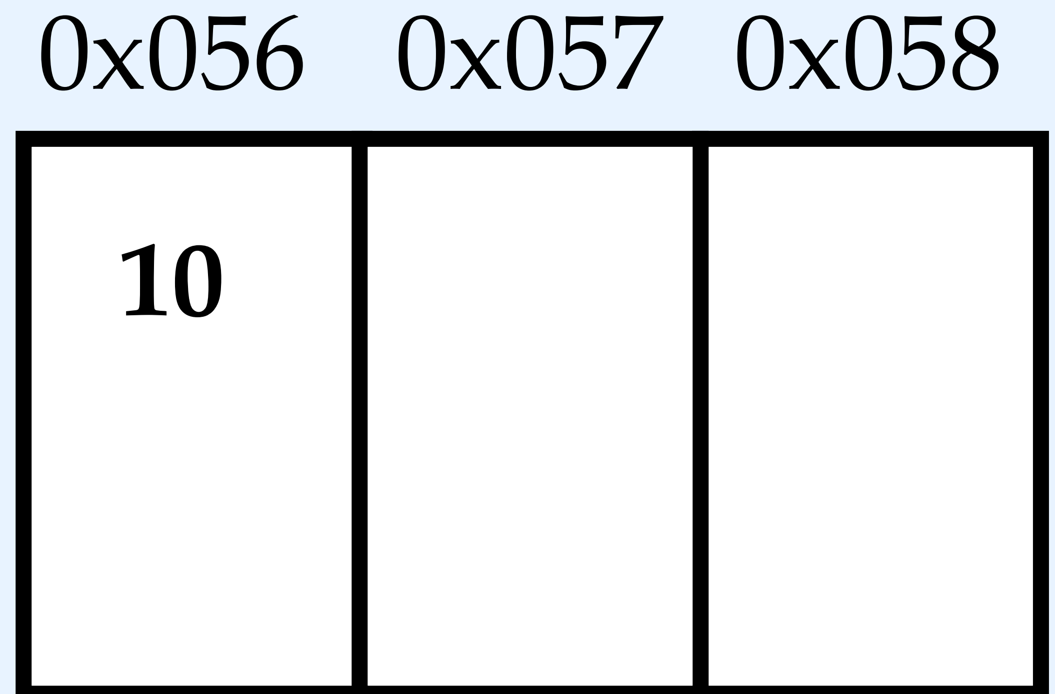
Java

int i = 10;

**Environment**

**Memory**



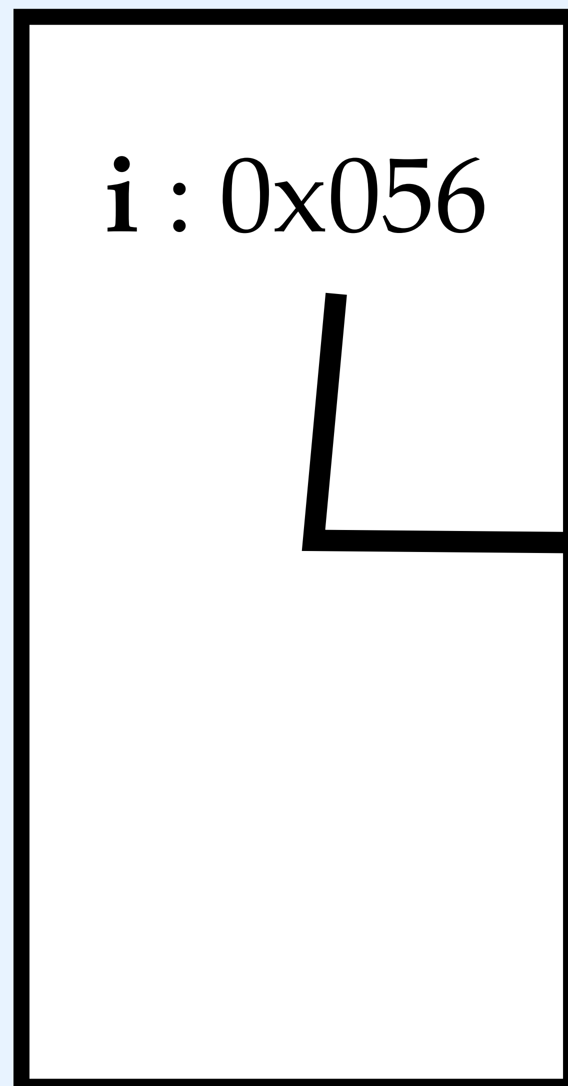
# Variable update

Java

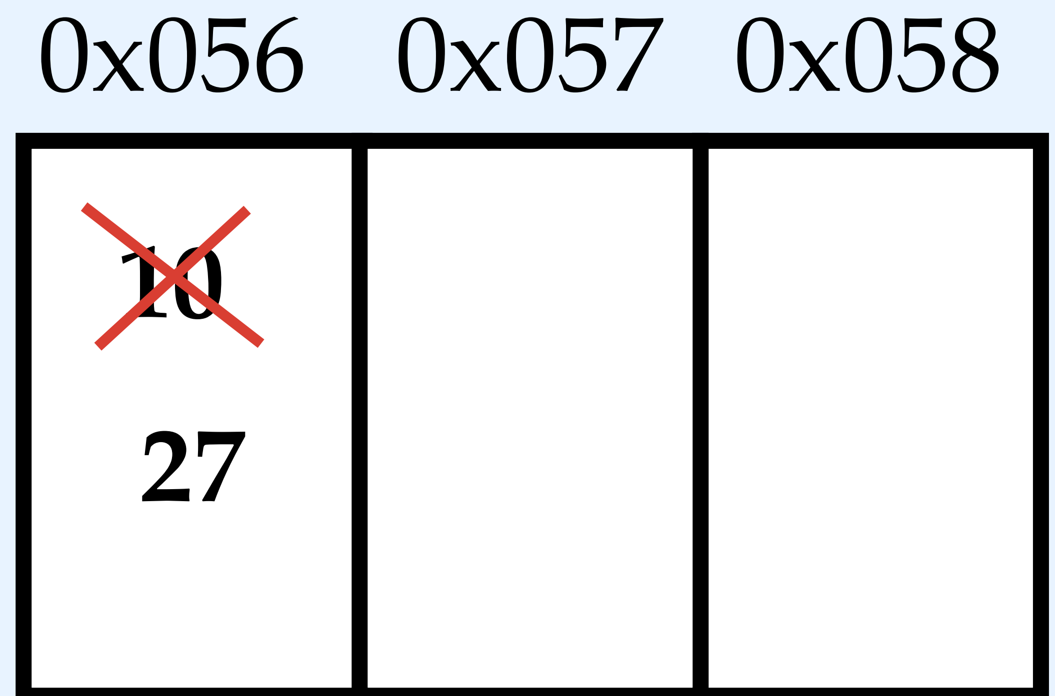
```
int i = 10;

i = 27;
```

**Environment**




**Memory**



# State

account123: $100

```
> (withdraw  account123  25)

> (balance?  account123)
$75

> (transfer  account123  account987  30)

> (balance?  account123)
$45
```

# State

account123: $100

```
> (withdraw  account123  25)
> (balance?  account123)
$75
> (transfer  account123  account987  30)
> (balance?  account123)
$45
```

# Referential transparency

In a language like Racket, which is *referentially transparent*, a variable always means the same thing regardless of where it occurs in the program.

This means that its value can be freely substituted in without regard to the context it appears in.

# Simulating mutation

Mutation gives us a way of updating what is stored in memory at a given address.

Can we simulate mutation?

<u>Components</u>

- Declare a variable name and a location for its value
- Update the value stored at that location

# Store-passing style

Store-passing style is a way of simulating mutation by simulating updates to memory.

Instead of directly addressing into memory to get the value of a variable, we look up the value of a variable in a **store.**

# Store-passing style

**Store**

**1** : \$50

**2** : \$10

**3** : \$25

**Program**

```
> (define account1 (store 1))
> (define account2 (store 2))
> (define account3 (store 3))
> account2
$10
```

# Store-passing style

We still don't have a way of modifying the value of an entry in the store.

But… we can just return a different store!

In other words, we can fake a mutable store by having different versions of the store (copies).

This works for stores, but not for memory (because we can't tell functions which version of memory to use).