CS 420 Advanced Programming Languages
Fall Semester, 2022
Doc 6 Assignment 1, Generics, Closure, Box
Sep 13, 2022

Copyright ©, All rights reserved. 2022 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/opl.shtml) license defines the copyright on this document.

A verses B

```
struct A {
    int x;
    int y;
    int z;
}
```

```
public class B {
    public int x;
    public int y;
    public int z;
}
```

A verses B

```
public class A {
    public int x;
    public int y;
    public int z;
}
```

```
public class B {
    private int x;
    private int y;
    private int z;

    public int getX() { return x;}
    public int getY() { return y;}
    public int getZ() { return z;}
    public void setX(int value) {x = value;}
    public void setY(int value) {y = value;}
    public void setZ(int value) {z = value;}
}
```

A verses B

```
public class Stack {
  ArrayList elements;

  public void push(Object item) {
    elements.add(item)
  }

  public Object pop() {
    if elements.isEmpty()
      throw new NoSuchElement;
    return element.remove(elements.length -1)
  }
```

```
struct Stack {
  ArrayList elements;
}


void push(Stack a, Object item) {
  add(a, item);
}

public Object pop(Stack a) {
  if isEmpty(a)
    throw new NoSuchElement;
  return remove(a, length(a) -1);
}
```

```
struct Foo {
    value: i32,
}

impl Foo {
    fn new(value: i32) -> Foo {
        Foo { value: value }
    }

    fn get_value(&self) -> i32 {
        self.value
    }

    fn get_value2(from: &Foo) -> i32 {
        from.value
    }
}

fn get_value3(from: &Foo) -> i32 {
    from.value
}
```

```
fn main() {
    let foo = Foo::new(42);
    let a = foo.get_value();
    let b = Foo::get_value2(&foo));
    let c = get_value2(&foo);
}
```

```
// Increase function
pub fn Increase(number: i64){
    // Takes number from function and adds 1
    let mut number = number;
    number += 1;

    // Prints the number
    println!("{}", number);
 }

fn main() {

    // Example number
    let input: i64 = 10;

    // Passes input through the Increase function
    Increase(input);
}
```

```
// Increase function
pub fn Increase(number: i64){

    let mut number = number;
    number += 1;


    println!("{}", number);
 }

fn main() {


    let input: i64 = 10;


    Increase(input);
}
```

Comments

There's a fine line between comments that illuminate and comments that obscure.

Are the comments necessary?

Do they explain "why" and not "what"?

Can you refactor the code so the comments aren't required?

And remember, you're writing comments for people, not machines.

http://blog.codinghorror.com/code-smells/

```
// Increase function
pub fn Increase(number: i64){

    let mut number = number;
    number += 1;



    println!("{}", number);
 }

fn main() {



    let input: i64 = 10;



    Increase(input);
}
```

```
/// Increase function
pub fn Increase(number: i64){
    println!("{}", number + 1);
 }

fn main() {
   Increase(10);
}
```

```
/// Increase function
pub fn increase(number: i64){
    println!("{}", number + 1);
 }

fn main() {
   Increase(10);
}
```

Printing to Standard out

Debugging

Error messages
- But does not work for
  - Servers
  - Applications
  - Web pages

```
/// Increase function
pub fn increase(number: i64){
    println!("{}", number + 1);
 }

fn main() {
   Increase(10);
}
```

Can not use increase function

Rust Naming Conventions

Modules	snake_case
Types	UpperCamelCase
Traits	UpperCamelCase
Enum variants	UpperCamelCase
Functions	snake_case
Methods	snake_case
General constructors	new or with_more_details
Conversion constructors	from_some_other_type
Macros	snake_case!
Local variables	snake_case
Statics	SCREAMING_SNAKE_CASE
Constants	SCREAMING_SNAKE_CASE
Type parameters	concise UpperCamelCase, usually single uppercase letter: T
Lifetimes	short lowercase, usually a single letter: 'a, 'de, 'src

Reading Verses Writing Programs

Code

- Written once
- Read many times

Use names that help the reader understand the code

Avd brvtns

brvtns r hrd t rd

n brvtn cn stnd fr dffrnt thngs

- tmp - tmprr r tmprtr

Dffrnt ppl wll brvt dffrntl

Ds tcmplt s dn't hv t typ lng nms

Avoid Abbreviations

Abbreviations are hard to read

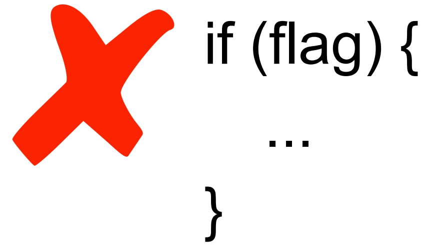
An abbreviation can stand for different things

- tmp - temporary or temperature

Different people will abbreviate differently

IDEs autocomplete so don't have to type long names

Describe What "flag" is Used For



Do not help understand code

Variables 1 through N

```
String s1;
String s2;
```

```
String fileContents;
String pattern;
```

Who can remember the difference between s1 and s2?

Avoid Names With No Meaning

Who are you?

What makes your LinkedList different?

All variables are temporary

```
swap = a;
a = b;
b = swap
```

```
(a, b) = (b, a)
```

Guidelines - Class Names

Use nouns

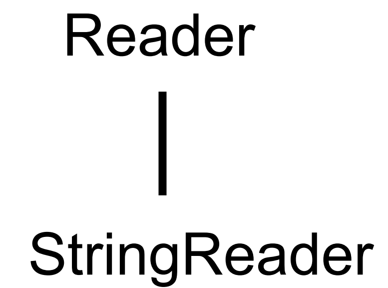
No abbreviations

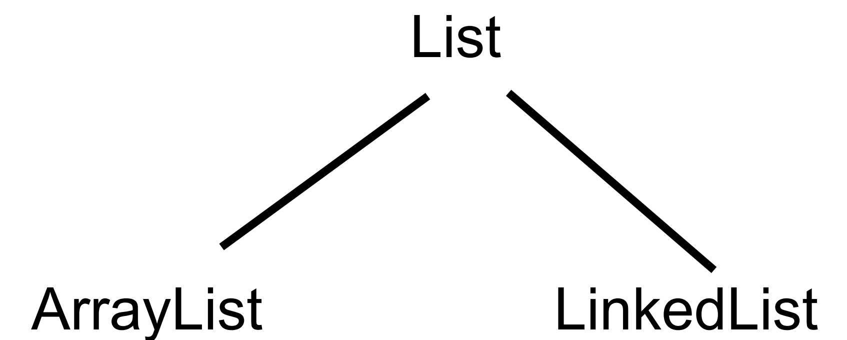
Superclass

  Single word to convey its purpose

Subclass

  Prepend adjective to superclass name





Guidelines - Method/Function/Procedure Names

Describe what method does

Use verb to describe an action

add(int index, E element)
clear()

If returns a value name what it returns

iterator()
subList(int fromIndex, int toIndex)

If returns boolean value make it true/false statement

isEmpty()
contains(Object o)

Guidelines - Variables, Fields, Parameters

Use names that indicate role variable is playing

If declare variable types don't use type as name

Use plurals to indicate collections

Make boolean variable names true/false statement

isVisible, hasMultipleParts,

```
public void execute(Vector vector) {
    Stack s;
}
```

```
public void execute(Vector commands) {
    Stack commandsExecuted;
}
```

Summary

Use names to help the reader understand the code

Follow language conventions

Avoid abbreviations

Use names that indicate role item is playing

```
fn main() {
    println!("Hello, world!");
    println!("{}", increase(22));
}

fn increase(num :i64) -> i64 {
    return num + 1;
}

fn main() {
    let x = increase(3);
    println!("The value of x is: {x}");
}

fn increase(x: i64) -> i64{
    x + 1
}
```

```
fn main() {
    println!("{}", increase(22));
}

fn increase(num :i64) -> i64 {
    return num + 1;
}
```

```
//function to increase x by 1
fn increase(x: i64) -> i64
{
    return x + 1;

}

fn main()
{
    //print result
    println!("{}", increase(10));

    //test if function worked
    assert_eq!(11, increase(10));

}
```

```
///Increase x by 1
fn increase(x: i64) -> i64
{
    return x + 1;
}

fn main()
{
    assert_eq!(11, increase(10));
}
```

```
fn increase(x: i64) -> i64
{
    x + 1
}
```

```
fn increase(n: i64) -> i64 {
    n + 1
}

#[test]
fn test_increase() {
    assert_eq!(increase(6), 5);
}
```

```
#[test]
fn test_increase() {
    let input = [-1, 0, 1, 2, 3, 4, 5];
    for i in input.iter() {
        assert_eq!(increase(*i), *i + 1);
    }
}

use itertools::izip;

#[test]
fn test_increase() {
    let inputs = [-1, 0, 5, 41];
    let answers = [0, 1, 6, 42];
    for (input, answer) in izip!(&inputs, &answers) {
        assert_eq!(increase(*input), *answer);
    }
}
```

```
fn multiple_tests(input:Vec<i64>, expected:Vec<i64>, test_fn:fn(i64)->i64) {
    for (input, expected) in izip!(input, expected) {
        assert_eq!(test_fn(input), expected);
    }
}

#[test]
fn test_increase() {
    let inputs = [-1, 0, 5, 41];
    let answers = [0, 1, 6, 42];
    multiple_tests(inputs.to_vec(), answers.to_vec(), increase);
}
#[test]
fn test_decrease() {
    let inputs = [-1, 0, 5, 41];
    let answers = [-2, -1, 4, 40];
    multiple_tests(inputs.to_vec(), answers.to_vec(), decrease);
}
```

Generics

```
fn multiple_tests<T>(input:Vec<T>, expected:Vec<T>, test_fn:fn(T)->T) {
    for (input, expected) in izip!(input, expected) {
        assert_eq!(test_fn(input), expected);
    }
}
```

error[E0369]: binary operation `==` cannot be applied to type `T`

Restricting the Generic

```
fn multiple_tests<T, U>(input:Vec<T>, expected:Vec<U>, test_fn:fn(T)->U)
    where T:Clone+PartialEq+core::fmt::Debug,
          U:Clone+PartialEq+core::fmt::Debug {
    for (input, expected) in izip!(input, expected) {
        assert_eq!(test_fn(input), expected);
    }
}

#[test]
fn test_increase2() {
    let inputs = [-1, 0, 5, 41];
    let answers = [0, 1, 6, 42];
    multiple_tests(inputs.to_vec(), answers.to_vec(), increase);
}
```

Monomorphization

The compiler generates concrete copies for each use of a generic

```
#[test]
fn test_increase() {
    multiple_tests(vec![1.0, 2.2, 3.0], vec![2.0, 3.2, 4.0],   |x | x + 1.0);
    multiple_tests(vec![1, 2, 3], vec![2, 3, 4],   |x | x + 1);
}
```

How to Make this Work?

```
#[test]
fn test_increase() {
    multiple_tests(vec!["cat"], vec!["CAT"],   |x| &x.to_uppercase());
}
```

Closure Syntax

```
fn  add_one_v1   (x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|             { x + 1 };
let add_one_v4 = |x|               x + 1  ;
```

Filter

```
let a = [0i32, 1, 2];

let mut iter = a.iter().filter(|x| x.is_positive());

assert_eq!(iter.next(), Some(&1));
assert_eq!(iter.next(), Some(&2));
assert_eq!(iter.next(), None);
```

Map & Collect

```
let a = [1, 2, 3];

let mut doubled= a.iter()
    .map(|&x| x * 2);

assert_eq!(2, doubled.next().unwrap());
assert_eq!(4, doubled.next().unwrap());
assert_eq!(6, doubled.next().unwrap());
```

```
let a = [1, 2, 3];

let doubled: Vec<i32> = a.iter()
                         .map(|&x| x * 2)
                         .collect();

assert_eq!(vec![2, 4, 6], doubled);
```

Using Collect - Type inference Failed

```
let a = [0i32, 1, -3, 2];
let mut positive = a.iter().filter(|x| x.is_positive()).collect::<Vec<_>>();

assert_eq!(positive.len(), 2);
assert_eq!(positive, [&1,&2]  );
assert_eq!(positive, vec![&1,&2]  );
```

fold

```
let a = [1, 2, 3];

let sum = a.iter().fold(0, |acc, x| acc + x);

assert_eq!(sum, 6);


let a = [1, 3, 2];

let largest = a.iter().fold(0, |acc, x| cmp::max(acc, *x));

assert_eq!(largest, 3);
```

Any & All

```
let a = [1, 2, 3];

assert!(a.iter().any(|&x| x > 0));

assert!(!a.iter().any(|&x| x > 5));
```

```
let a = [1, 2, 3];

assert!(a.iter().all(|&x| x > 0));

assert!(!a.iter().all(|&x| x > 2));
```

Iterator Methods

advance_by
all
any
by_ref
chain
cloned
cmp
cmp_by
collect
collect_into
copied
count
cycle
enumerate
eq
eq_by
filter
filter_map
find
find_map
flat_map
flatten
fold
for_each
fuse
ge
gt
inspect
intersperse
intersperse_with
is_partitioned
is_sorted
is_sorted_by
is_sorted_by_key
last
le
lt
map
map_while
max
max_by
max_by_key
min
min_by
min_by_key
ne
nth
partial_cmp
partial_cmp_by
partition
partition_in_place
peekable
position
product
reduce
rev
rposition
scan
size_hint
skip
skip_while
step_by
sum
take
take_while
try_collect
try_find
try_fold
try_for_each
try_reduce
unzip
zip

Closure & Ownership

```
fn create_adder(n:i32) -> impl Fn(i32) -> i32 {
    |x| x + n
}
```

```
|x| x + n
 |     ^^^     - `n` is borrowed here
 |     |
 |     may outlive borrowed value `n`
```

Closures can outlive the values it borrows

move - steals the value

```
fn create_adder(n:i32) -> impl Fn(i32) -> i32 {
    move |x| x + n
}
```

move
converts any variables captured to variables captured by value.

```
#[test]
fn test_create_adder() {
    let adder1 = create_adder(1);
    let adder5 = create_adder(5);
    assert_eq!(adder1(2), 3);
    assert_eq!(adder5(2), 7);
}
```

Mutable Borrow

```
let mut list = vec![1, 2, 3];
println!("Before defining closure: {:?}", list);

let mut borrows_mutably = || list.push(7);

borrows_mutably();
println!("After calling closure: {:?}", list)
```

```
let mut list = vec![1, 2, 3];
println!("Before defining closure: {:?}", list);

let mut borrows_mutably = || list.push(7);
println!("Before calling closure: {:?}", list);
borrows_mutably();
println!("After calling closure: {:?}", list)
```

```
 let mut borrows_mutably = || list.push(7);
   |                                   -- ---- first borrow occurs due to use
`list` in closure
   |                                   |
   |                                   mutable borrow occurs here
24 |     println!("Before calling closure: {:?}", list);
   |                                                ^^^^ immutable borrow
occurs here
25 |     borrows_mutably();
   |     --------------- mutable borrow later used here
```

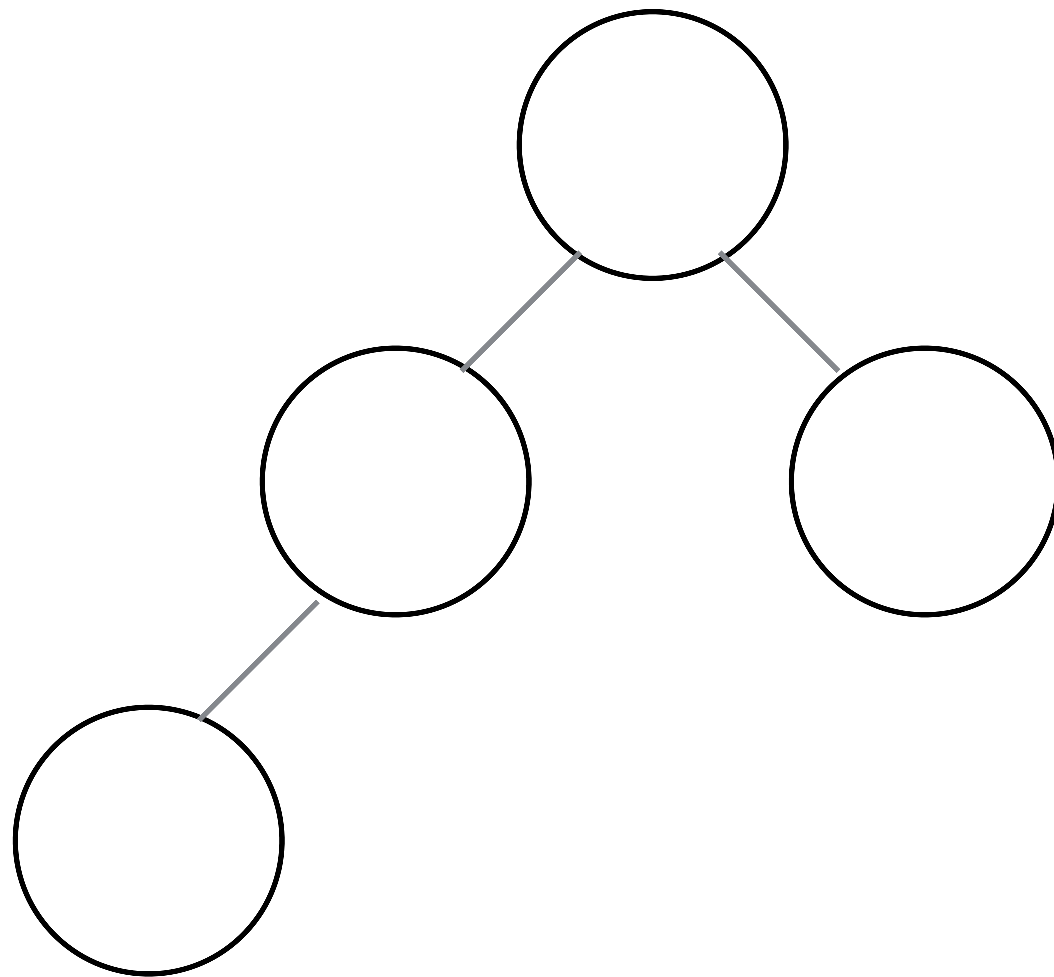
What Happens Here?

```
let mut list = vec![1, 2, 3];
println!("Before defining closure: {:?}", list);

let mut borrows_mutably = || list.push(7);
borrows_mutably();
println!("After calling closure: {:?}", list);
borrows_mutably();
```

Recursive Structures

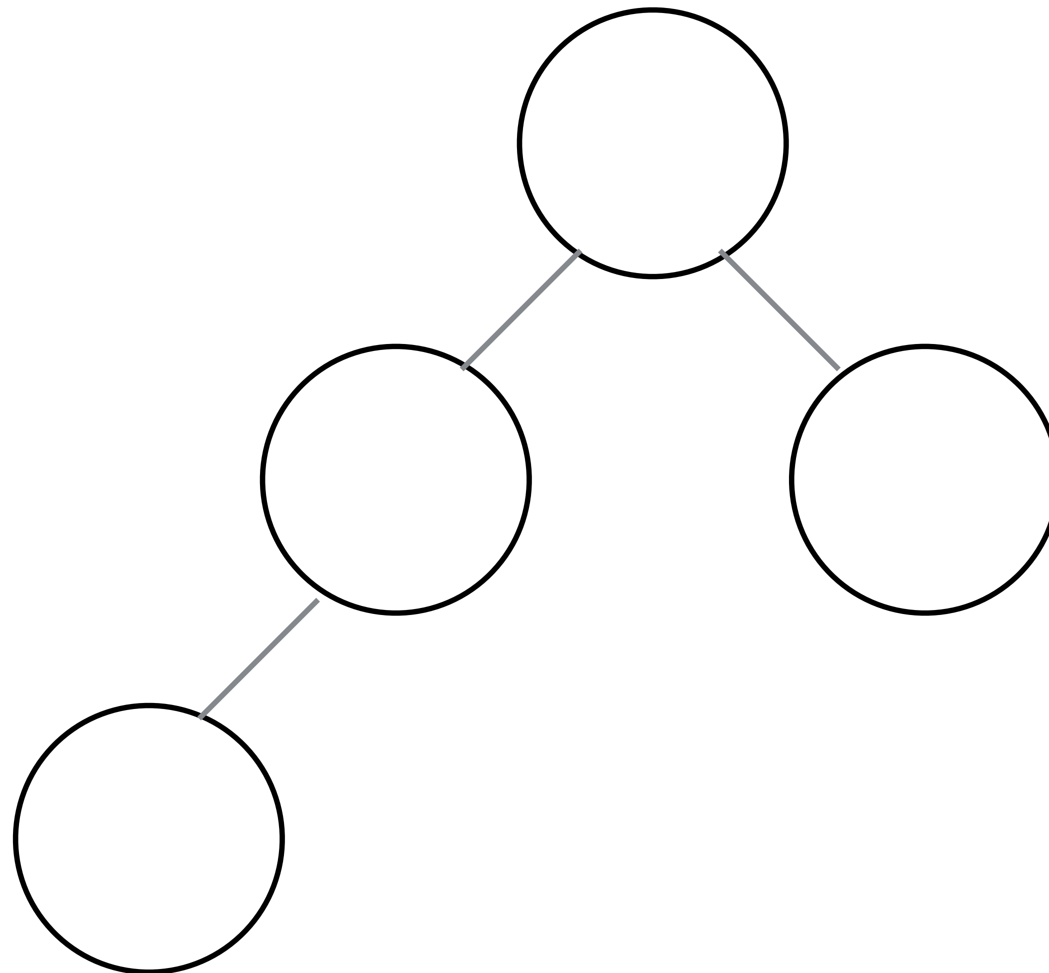
Rust compile needs to know the size of data structures



Recursive Structures

```
struct BST_Node {
    val: i32,
    left: BST,
    right: BST,
}
```

```
| struct BSTNode {
 | ^^^^^^^^^^^^^^ recursive type has infinite size
```



Using Option

```
struct BSTNode {
    val: i32,
    left: Option<BSTNode>,
    right: Option<BSTNode>,
}
```

```
 4 | struct BSTNode {
   | ^^^^^^^^^^^^^^ recursive type has infinite size
 5 |     val: i32,
 6 |     left: Option<BSTNode>,
   |           --------------- recursive without indirection
 7 |     right: Option<BSTNode>,
   |            --------------- recursive without indirection
```

Box

Box

- Allocates data on Heap
- Returns reference

```
let mut b : Box<i32> = Box::new(5);
assert_eq!(*b,5);
let a = *b;
assert_eq!(a,5);
*b = 6;
assert_eq!(*b,6);
let c = &mut b;
assert_eq!(**c,6);
**c = 7;
println!("b = {}", b);
```

```
fn increase(x: &mut i32) {
    *x += 1;
}
```

```
let mut b = Box::new(5);
increase(&mut b);
assert_eq!(*b,6);
```

Don't need to dereference b

The Recursive

```
struct BSTNode {
    val: i32,
    left: Option<Box< BSTNode >>,
    right: Option<Box< BSTNode >>,
}
```