Welcome to CS2030 Lab 6!

18 March 2022 [10J]

Only scan if you're physically present!
LOGIN TO PLAB NOW!

`scp` command synopsis:

`scp [OPTIONS] [user@]SRC_HOST:]file1 [user@]DEST_HOST:]file2`

Example:

`scp lyskevin@stu.comp.nus.edu.sg:file.txt /Users/lyskevin/Desktop`

- In the above command, we send a copy of `file.txt` in the main directory of a stu account to the `/Users/lyskevin/Desktop` directory of the device (i.e. someone's laptop).

- `[OPTIONS]` was not used (left empty), and the command was run that person's laptop.

Wildcard Types

You can use **unbounded wildcards, bounded wildcards**, or **lower bound wildcards** to specify a range for a generic type.

1. `?`: unbounded wildcard, is the same as `? extends Object`.

2. `? extends T`: (upper) bounded wildcard, represents T or a **subtype** of T.

3. `? super T`: lower bound wildcard, denotes T or a **supertype** of T.

Note that `GenericClass<Integer>` is not a subtype of `GenericClass<Object>` even though `Integer` is a subtype of `Object`.

Bounded Wildcards

```
Box<? extends T> // Box can have any subclass of T
Box<? super T> // Box can have any superclass of T
Box<?> // Synonymous with Box<? extends Object>
```

Get and Put Principle

- Use an **extends** wildcard when you only **get** values out of a structure.

- Use a **super** wildcard when you only **put** values into a structure.

- And **don't use a wildcard** when you support **both get and put**.

2 Exceptions:

1. You cannot put anything into a type declared with an extends wildcard except for the value `null`, which **belongs to every reference type**.

2. You cannot get anything out from a type declared with a super wildcard except for a value of type `Object`, which is a **super type of every reference type**.

Producer Extends

```
List<? extends Number> foo3 = new ArrayList<Number>();
// Number "extends" Number (in this context)
List<? extends Number> foo3 = new ArrayList<Integer>();
// Integer extends Number
List<? extends Number> foo3 = new ArrayList<Double>();
// Double extends Number
```

- The above assignments are all valid

- Getting from foo3 (must be legal for **all** possible foo3 assignments):

  - You can get a `Number` because any of the lists that could be assigned to foo3 contain a `Number` or a subclass of `Number` (You are sure that foo3 contains `Numbers`)

  - You cannot get an `Integer` because foo3 could be pointing to a `List<Double>`

  - You cannot get a `Double` because foo3 could be pointing to a `List<Integer>`

```
List<? extends Number> foo3 = new ArrayList<Number>();
// Number "extends" Number (in this context)
List<? extends Number> foo3 = new ArrayList<Integer>();
// Integer extends Number
List<? extends Number> foo3 = new ArrayList<Double>();
// Double extends Number
```

- Adding to foo3 (must be legal for **all** possible foo3 assignments):

  - **You cannot add anything into it**

  - Cannot add `Integer` because foo3 could be pointing to `List<Double>`

  - Cannot add `Double` because foo3 could be pointing to a `List<Integer>`

  - Cannot add `Number` because foo3 could be pointing to a `List<Integer>`

```
List<? extends Number> foo3 = new ArrayList<Number>();
// Number "extends" Number (in this context)
List<? extends Number> foo3 = new ArrayList<Integer>();
// Integer extends Number
List<? extends Number> foo3 = new ArrayList<Double>();
// Double extends Number
```

- You can't add any object to `List<? extends T>` because you can't guarantee what kind of List it is really pointing to, so you can't guarantee that the object is allowed in that List. The only "guarantee" is that you can only get from it and you'll get a T or subclass of T.

- Conclusion: **Don't use** `extends` **if you want to add to a collection**

Consumer Super

```
List<? super Integer> foo3 = new ArrayList<Integer>();
// Integer is a "superclass" of Integer (in this context)
List<? super Integer> foo3 = new ArrayList<Number>();
// Number is a superclass of Integer
List<? super Integer> foo3 = new ArrayList<Object>();
// Object is a superclass of Integer
```

- The above assignments are all valid.

- Getting from foo3 (must be legal for **all** possible foo3 assignments):

  - You cannot get an `Integer` because foo3 could be pointing to a `List<Number>` or `List<Object>`

  - You cannot get a `Number` because foo3 could be pointing to a `List<Object>`

  - You can only get an `Object` but the subclass (if any) is unknown (redundant to read an `Object`; every class is a subclass of `Object`)

```
List<? super Integer> foo3 = new ArrayList<Integer>();
// Integer is a "superclass" of Integer (in this context)
List<? super Integer> foo3 = new ArrayList<Number>();
// Number is a superclass of Integer
List<? super Integer> foo3 = new ArrayList<Object>();
// Object is a superclass of Integer
```

- Adding to foo3 (must be legal for **all** possible foo3 assignments):

  - You can add an `Integer` (allowed by all 3 lists)

  - You can add an instance of a subclass of `Integer` (allowed by all 3 lists)

  - You cannot add a `Double` because foo3 might point to a List<Integer>

  - You cannot add a `Number` because foo3 might point to a List<Integer>

  - You cannot add an `Object` because foo3 might point to a List<Integer>

```
List<? super Integer> foo3 = new ArrayList<Integer>();
// Integer is a "superclass" of Integer (in this context)
List<? super Integer> foo3 = new ArrayList<Number>();
// Number is a superclass of Integer
List<? super Integer> foo3 = new ArrayList<Object>();
// Object is a superclass of Integer
```

- You can't get from a `List<? super T>` because you do not know what kind of `List` it points to (unless you assign the item to an `Object` instance; redundant as mentioned before). You can only add items of type `T` or a subclass of `T` to the list.

- Conclusion: **Use** `super` **to add objects into your collection**

Single Abstract Method (SAM)/ Functional Interfaces

Single Abstract Method (SAM)/Functional Interfaces

- `Function<T, R>` is a SAM interface (i.e. it is an interface containing only one abstract method for implementing classes to implement)

- Only (abstract) method in Function interface:

  ```
  public R apply(T t) // Takes in a T and returns an R
  ```

- 3 Ways to implement:

```
1. Create a **separate class** that implements the function
2. Create an **anonymous class** to implement it
3. Use a **lambda** expression
```

Anonymous Classes

- You can instantiate a class without defining it proper.

- Eg. Anonymous class that implements `Function`:

```
Function<Integer, Integer> addOne = new Function<>() {
    @Override
    public Integer apply(Integer x) {
        return Integer.valueOf(x.intValue() + 1);
    }
};
```

Lambda Expressions

- A shorter way of creating a `Function`.

  `Function<Integer, Integer> addOne = x → x + 1;`

- Takes in an `(Integer) x` and returns an `(Integer) x+1`

- How does the Java compiler know which method the lambda expression is implementing? Well, `Function` is a SAM interface, so there is only one method that it can implement (the `apply` method)

Other Functional Interfaces

Observe the input/output carefully

Interface	Function Signature	Example
Predicate<T>	boolean test(T t)	Collection::isEmpty
Function<T,R>	R apply(T t)	Arrays::asList
Supplier<T>	T get()	Math::random
Consumer<T>	void accept(T t)	System.out::println
UnaryOperator<T>	T apply(T t)	String::toLowerCase
BinaryOperator<T>	T apply(T t1, T t2)	BigInteger::add

`java.util.Optional<T>`

- `Optional` class is a useful abstraction to deal with `null` values

- `Optional<T>` creates an `Optional` that wraps around type T (e.g. `Optional<Integer>` creates an `Optional` that wraps around an `Integer`)

- Sounds familiar? (*Hint: refer to one of the recent recitation questions*)

If you find yourself writing a method that can't always return a value and you believe it is important that users of the method consider this possibility every time they call it, then you should probably return an optional

- `Optional<T>` represents an immutable container that can hold either a single non-null T reference or nothing at all.

- An optional that contains nothing is said to be empty.

- A value is said to be present in an optional that is not empty.

- An optional is essentially an immutable collection that can hold at most one element.

Return Type	Method	Description
T	orElse(T other)	yields the value of this `Optional`, or `other` if this `Optional` is empty.
T	orElseGet(Supplier<? extends T> other)	yields the value of this `Optional`, or the result of invoking `other` if this `Optional` is empty.
<X extends Throwable> T	orElseThrow(Supplier<? extends X> exceptionSupplier)	yields the value of this `Optional`, or throws the result of invoking `exceptionSupplier` if this `Optional` is empty.
void	ifPresent(Consumer<? super T> action)	if this `Optional` is nonempty, passes its value to `action`.
void	ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)	if this `Optional` is nonempty, passes its value to action, else invokes `emptyAction`.
<U> Optional<U>	map(Function<? super T,? extends U> mapper)	yields an `Optional` whose value is obtained by applying the given function to the value of this `Optional` if present, or an empty `Optional` otherwise.
Optional<T>	filter(Predicate<? super T> predicate)	yields an `Optional` with the value of this `Optional` if it fulfills the given predicate, or an empty `Optional` otherwise.
Optional<T>	or(Supplier<? extends Optional<? extends T>> supplier)	yields this `Optional` if it is nonempty, or the one produced by the supplier otherwise.
T	get()	
T	orElseThrow()	yields the value of this `Optional`, or throws a `NoSuchElementException` if it is empty.
boolean	isPresent()	returns `true` if this `Optional` is not empty.
static <T> Optional<T>	of(T value)	
static <T> Optional<T>	ofNullable(T value)	yields an `Optional` with the given value. If `value` is `null`, the first method throws a `NullPointerException` and the second method yields an empty `Optional`.
static <T> Optional<T>	empty()	yields an empty `Optional`.
<U> Optional<U>	flatMap(Function<? super T,? extends Optional<? extends U>> mapper)	yields the result of applying `mapper` to the value in this `Optional` if present, or an empty optional otherwise.
<U> Optional<U>	flatMap(Function<? super T,Optional<U>> mapper)	yields the result of applying `mapper` to the value of this `Optional`, or an empty `Optional` if this `Optional` is empty.

Optional - PECS in context

- The `Optional` class uses PECS to support passing super/subclasses of wrapped types into methods

- For instance, `Function<? super T, ? extends U> mapper` is passed into the map method

- In context, it means that `mapper` can take any superclass of T (stored in `Optional`) as input and have any subclass of U as output (U's type depends on the `Function` being used)

- Many other classes that support generics also use PECS to make their methods more general/accepting of types

```
<U> Optional<U> map(Function<? super T, ? extends U> mapper)
```

yields an `Optional` whose value is obtained by applying the given function to the value of this `Optional` if present, or an empty `Optional` otherwise.

```
jshell> Optional.of(1).map(x → x + 1)
$1 ⟹ Optional[2]

jshell> Optional.of(1).map(x → "1" + x)
$2 ⟹ Optional[11]

jshell> Optional.of(1).map(x → Optional.of(x))
$3 ⟹ Optional[Optional[1]]
```

```
<U> Optional<U> flatMap(Function<? super T,
                        ? extends Optional<? extends U>
                        > mapper)


<U> Optional<U> flatMap(Function<? super T,
                        Optional<U>
                        > mapper)
```

yields the result of applying `mapper` to the value in this `Optional` if present, or an empty optional otherwise.

```
jshell> Optional.of(1).flatMap(x → Optional.of(x + 1))
$4 ⟹ Optional[2]

jshell> Optional.of(1).flatMap(x → Optional.of(x))
$5 ⟹ Optional[1]

jshell> Optional.of(Optional.of(1)).flatMap(x → Optional.of(x))
$6 ⟹ Optional[Optional[1]]
```

Invariance, Contravariance, Covariance (Further Reading)

- A and B are types.

- f is a type transformation.

- $\leq$ is the subtype relation (i.e. $A \leq B$ means that A is a subtype (subclass) of B).

- f is **covariant** if $A \leq B$ implies that $f(A) \leq f(B)$.

- f is **contravariant** if $A \leq B$ implies that $f(A) \geq f(B)$.

- f is **invariant** if neither of the above holds.

Invariance, Contravariance, Covariance (Further Reading)

Prefer lists to arrays

Arrays are *covariant*, i.e.

$$\text{Sub} \leq \text{Super} \implies \text{Sub[]} \leq \text{Super[]}$$

```
jshell> Object[] objectArray = new Long[1]
objectArray ⟹ Long[1] { null }

jshell> objectArray[0] = "I don't fit in"
|  Exception java.lang.ArrayStoreException: java.lang.String
|       at (#8:1)
```

Finds out mistake at runtime. ❌

Generics are *invariant*.

```
jshell> List<Object> ol = new ArrayList<Long>()
|  Error:
|  incompatible types: java.util.ArrayList<java.lang.Long>
cannot be converted to java.util.List<java.lang.Object>
|  List<Object> ol = new ArrayList<Long>();
|                    ^------------------^
```

Finds out mistake at compile time. ✅

Invariance, Contravariance, Covariance (Further Reading)

- Therefore, `List<?>` is not the same as `List<Object>`

- `List<?>` is more general than `List<Object>` because `<?>` is synonymous with `<? extends Object>`

- `List<?>` can store instances of type `Object` or any subclass of `Object`, i.e. `List<?>` can store anything since all other classes are subclasses of `Object`