CS2030 Notes

Wildcards

Upper-Bounded Wildcards

Let's consider the method `copyFrom`. We should be able to copy from an array of shapes, an array of circles, an array of squares, etc, into an array of shapes. In other words, we should be able to copy from *an array of any subtype of shapes* into an array of shapes. Is there such a type in Java?

The type that we are looking for is `Array<? extends Shape>`. This generic type uses the *wildcard* `?`. Just like a wild card in card games, it is a substitute for any type. A wildcard can be bounded. Here, this wildcard is upper-bounded by `Shape`, i.e., it can be substituted with either `Shape` or any subtype of `Shape`.

The upper-bounded wildcard is an example of covariance. The upper-bounded wildcard has the following subtyping relations:

- If `S` <: `T`, then `A<? extends S>` <: `A<? extends T>` (covariance)
- For any type `S`, `A<S>` <: `A<? extends S>`

For instance, we have:

- `Array<Circle>` <: `Array<? extends Circle>`
- Since `Circle` <: `Shape`, `Array<? extends Circle>` <: `Array<? extends Shape>`
- Since subtyping is transitive, we have `Array<Circle>` <: `Array<? extends Shape>`

Because `Array<Circle>` <: `Array<? extends Shape>`, if we change the type of the parameter to `copyFrom` to `Array<? extends T>`,

```java
public void copyFrom(Array<? extends T> src) {
  int len = Math.min(this.array.length, src.array.length);
  for (int i = 0; i < len; i++) {
      this.set(i, src.get(i));
  }
}
```

We can now call:

```java
shapeArray.copyFrom(circleArray); // ok
```

without error.

Lower-Bounded Wildcards

Let's now try to allow copying of an `Array<Circle>` to `Array<Shape>`.

```
circleArray.copyTo(shapeArray);
```

by doing the same thing:

```
  public void copyTo(Array<? extends T> dest) {
    int len = Math.min(this.array.length, dest.array.length);
    for (int i = 0; i < len; i++) {
        dest.set(i, this.get(i));
    }
  }
```

The code above would not compile. We will get the following somewhat cryptic message when we compile with the `-Xdiags:verbose` flag:

```
Array.java:32: error: method set in class Array<T> cannot be applied to given types;
        dest.set(i, this.get(i));
            ^
  required: int,CAP#1
  found: int,T
  reason: argument mismatch; T cannot be converted to CAP#1
  where T is a type-variable:
    T extends Object declared in class Array
  where CAP#1 is a fresh type-variable:
    CAP#1 extends T from capture of ? extends T
1 error
```

Let's try not to understand what the error message means first, and think about what could go wrong if the compiler allows:

```
        dest.set(i, this.get(i));
```

Here, we are trying to put an instance with compile-time type `T` into an array that contains elements with the compile-time type of `T` or subtype of `T`.

The `copyTo` method of `Array<Shape>` would allow an `Array<Circle>` as an argument, and we would end up putting instance with compile-time type `Shape` into `Array<Circle>`. If all the shapes are circles, we are fine, but there might be other shapes (rectangles, squares) in `this` instance of `Array<Shape>`, and we can't fit them into `Array<Circle>`! Thus, the line

```java
        dest.set(i, this.get(i));
```

is not type-safe and could lead to `ClassCastException` during run-time.

Where can we copy our shapes into? We can only copy them safely into an `Array<Shape>`, `Array<Object>`, `Array<GetAreable>`, for instance. In other words, into arrays containing `Shape` or supertype of `Shape`.

We need a wildcard lower-bounded by `Shape`, and Java's syntax for this is `? super Shape`. Using this new notation, we can replace the type for `dest` with:

```java
  public void copyTo(Array<? super T> dest) {
    int len = Math.min(this.array.length, dest.array.length);
    for (int i = 0; i < len; i++) {
        dest.set(i, this.get(i));
    }
  }
```

The code would now type-check and compile.

The lower-bounded wildcard is an example of contravariance. We have the following subtyping relations:

- If `S` <: `T`, then `A<? super T>` <: `A<? super S>` (contravariance)
- For any type `S`, `A<S>` <: `A<? super S>`

For instance, we have:

- `Array<Shape>` <: `Array<? super Shape>`
- Since `Circle` <: `Shape`, `Array<? super Shape>` <: `Array<? super Circle>`
- Since subtyping is transitive, we have `Array<Shape>` <: `Array<? super Circle>`

The line of code below now compiles:

```java
circleArray.copyTo(shapeArray);
```

Our new `Array<T>` is now

```java
// version 0.5 (with flexible copy using wildcards)
class Array<T> {
  private T[] array;

  Array(int size) {
  // The only way we can put an object into the array is through
  // the method set() and we only put an object of type T inside.
  // So it is safe to cast `Object[]` to `T[]`.
  @SuppressWarnings("unchecked")
    T[] a = (T[]) new Object[size];
  this.array = a;
  }

  public void set(int index, T item) {
    this.array[index] = item;
  }

  public T get(int index) {
    return this.array[index];
  }

  public void copyFrom(Array<? extends T> src) {
    int len = Math.min(this.array.length, src.array.length);
    for (int i = 0; i < len; i++) {
        this.set(i, src.get(i));
    }
  }

  public void copyTo(Array<? super T> dest) {
    int len = Math.min(this.array.length, dest.array.length);
    for (int i = 0; i < len; i++) {
        dest.set(i, this.get(i));
    }
  }
}
```

PECS

Now we will introduce the rule that governs when we should use the upper-bounded wildcard `? extends T` and a lower-bounded wildcard `? super T`. It depends on the role of the variable. If the variable is a producer that returns a variable of type `T`, it should be declared with the wildcard `? extends T`. Otherwise, if it is a consumer that accepts a variable of type `T`, it should be declared with the wildcard `? super T`.

As an example, the variable `src` in `copyFrom` above acts as a *producer*. It produces a variable of type `T`. The type parameter for `src` must be either `T` or a subtype of `T` to ensure type safety. So the type for `src` is `Array<? extends T>`.

On the other hand, the variable `dest` in `copyTo` above acts as a *consumer*. It consumes a variable of type `T`. The type parameter of `dest` must be either `T` or supertype of `T` for it to be type-safe. As such, the type for `dest` is `Array<? super T>`.

This rule can be remembered with the mnemonic PECS, or "Producer Extends; Consumer Super".

Unbounded Wildcards

It is also possible to have unbounded wildcards, such as `Array<?>`. `Array<?>` is the supertype of every parameterized type of `Array<T>`. Recall that `Object` is the supertype of all reference types. When we want to write a method that takes in a reference type, but we want the method to be flexible enough, we can make the method accept a parameter of type `Object`. Similarly, `Array<?>` is useful when you want to write a method that takes in an array of some specific type, and you want the method to be flexible enough to take in an array of any type. For instance, if we have:

```
void foo(Array<?> array) {
}
```

We could call it with:

```
Array<Circle> ac;
Array<String> as;
foo(ac); // ok
foo(as); // ok
```

A method that takes in generic type with unbounded wildcard would be pretty restrictive, however. Consider this:

```
void foo(Array<?> array) {
    :
  x = array.get(0);
  array.set(0, y);

}
```

What should the type of the returned element `x` be? Since `Array<?>` is the supertype of all possible `Array<T>`, the method `foo` can receive an instance of `Array<Circle>`, `Array<String>`, etc. as an argument. The only safe choice for the type of `x` is `Object`.

The type for `y` is every more restrictive. Since there are many possibilities of what type of array it is receiving, we can only put `null` into `array`!

There is an important distinction to be made between `Array`, `Array<?>` and `Array<Object>`. Whilst `Object` is the supertype of all `T`, it does not follow that `Array<Object>` is the supertype of all `Array<T>` due to generics being invariant. Therefore, the following statements will fail to compile:

```
Array<Object> a1 = new Array<String>(0);
Array<Object> a2 = new Array<Integer>(0);
```

Whereas the following statements will compile:

```
Array<?> a1 = new Array<String>(0); // Does compile
Array<?> a2 = new Array<Integer>(0); // Does compile
```

If we have a function

```
void bar(Array<Object> array) {
}
```

Then, the method `bar` is restricted to *only* takes in an `Array<Object>` instance as argument.

```
Array<Circle> ac;
Array<String> as;
bar(ac); // compilation error
bar(as); // compilation error
```

What about raw types? Suppose we write the method below that accepts a raw type

```
void qux(Array array) {
}
```

Then, the method `qux` is also flexible enough to take in any `Array<T>` as argument.

```
Array<Circle> ac;
Array<String> as;
qux(ac);
qux(as);
```

Unlike `Array<?>`, however, the compiler does not have the information about the type of the component of the array, and cannot type check for us. It is up to the programmer to ensure type safety. For this reason, we must not use raw types.

Intuitively, we can think of `Array<?>`, `Array<Object>`, and `Array` as follows:

- `Array<?>` is an array of objects of some specific, but unknown type;
- `Array<Object>` is an array of `Object` instances, with type checking by the compiler;
- `Array` is an array of `Object` instances, without type checking.

Back to `contains`

Now, let's simplify our `contains` methods with the help of wildcards. Recall that to add flexibility into the method parameter and allow us to search for a shape in an array of circles, we have modified our method into the following:

```
class A {
  // version 0.6 (with generic array)
  public static <S,T extends S> boolean contains(Array<T> array, S obj) {
    for (int i = 0; i < array.getLength(); i++) {
      T curr = array.get(i);
      if (curr.equals(obj)) {
        return true;
      }
    }
    return false;
  }
}
```

Can we make this simpler using wildcards? Since we want to search for an object of type `S` in an array of its subtype, we can remove the second parameter type `T` and change the type of array to `Array<? extends S>`:

```
1  class A {
2    // version 0.7 (with wild cards array)
3    public static <S> boolean contains(Array<? extends S> array, S obj) {
4      for (int i = 0; i < array.getLength(); i++) {
5        S curr = array.get(i);
6        if (curr.equals(obj)) {
7          return true;
8        }
9      }
10     return false;
11   }
12 }
```

We can double-check that `array` is a producer (it produces `curr` on Line 5) and this follows the PECS rules. Now, we can search for a shape in an array of circles.

```
        A.<Shape>contains(circleArray, shape);
```

Revisiting Raw Types

In previous units, we said that you may use raw types only in two scenarios. Namely, when using generics and `instanceof` together, and when creating arrays. However, with unbounded wildcards, we can now see it is possible to remove both of these exceptions. We can now use `instanceof` in the following way:

```
a instanceof A<?>
```

Recall that in the example above, `instanceof` checks of the run-time type of `a`. Previously, we said that we can't check for, say,

```
a instanceof A<String>
```

since the type argument `String` is not available during run-time due to erasure. Using `<?>` fits the purpose here because it explicitly communicates to the reader of the code that we are checking that `a` is an instance of `A` with some unknown (erased) type parameter.

Similarly, we can create arrays in the following way:

```
new Comparable<?>[10];
```

Previously, we said that we could not create an array using the expression `new Comparable<String>[10]` because generics and arrays do not mix well. Java insists that the array creation expression uses a *reifiable* type, i.e., a type where no type information is lost during compilation. Unlike `Comparable<String>`, however, `Comparible<?>` is reifiable. Since we don't know what is the type of `?`, no type information is lost during erasure!

Going forward now in the module, we will not permit the use of raw types in any scenario.

Completable Future

The `CompletableFuture` Monad

Let's now examine the `CompletableFuture` monad in more detail. A key property of `CompletableFuture` is whether the value it promises is ready -- i.e., the tasks that it encapsulates has *completed* or not.

Creating a `CompletableFuture`

There are several ways we can create a `CompletableFuture<T>` instance:

- Use the `completedFuture` method. This method is equivalent to creating a task that is already completed and return us a value.
- Use the `runAsync` method that takes in a `Runnable` lambda expression. `runAsync` has the return type of `CompletableFuture<Void>`. The returned `CompletableFuture` instance completes when the given lambda expression finishes.
- Use the `supplyAsync` method that takes in a `Supplier<T>` lambda expression. `supplyAsync` has the return type of `CompletableFuture<T>`. The returned `CompletableFuture` instance completes when the given lambda expression finishes.

We can also create a `CompletableFuture` that relies on other `CompletableFuture` instances. We can use `allOf` or `anyOf` methods for this. Both of these methods take in a variable number of other `CompletableFuture` instances. A new `CompletableFuture` created with `allOf` is completed only when all the given `CompletableFuture` completes. On the other hand, a new `CompletableFuture` created with `anyOf` is completed when any one of the given `CompletableFuture` completes.

Chaining `CompletableFuture`

The usefulness of `CompletableFuture` comes from the ability to chain them up and specify a sequence of computations to be run. We have the following methods:

- `thenApply`, which is analogous to `map`
- `thenCompose`, which is analogous to `flatMap`
- `thenCombine`, which is analogous to `combine`

The methods above run the given lambda expression in the same thread as the caller. There is also an asynchronous version (`thenApplyAsync`, `thenComposeAsync`, `thenCombineAsync`), which may cause the given lambda expression to run in a different thread (thus more concurrency).

`CompletableFuture` also has several methods that takes in `Runnable`. These methods have no analogy in our lab but it is similar to `runAsync` above.

- `thenRun` takes in a `Runnable`. It executes the `Runnable` after the current stage is completed.
- `runAfterBoth` takes in another `CompletableFuture`[1] and a `Runnable`. It executes the `Runnable` after the current stage completes and the input `CompletableFuture` are completed.
- `runAfterEither` takes in another `CompletableFuture`[1] and a `Runnable`. It executes the `Runnable` after the current stage completes or the input `CompletableFuture` are completed.

All of the methods that takes in `Runnable` return `CompletableFuture<Void>`. Similarly, they also have the asynchronous version (`thenRunAsync`, `runAfterBothAsync`, `runAfterEitherAsync`).

Getting The Result

After we have set up all the tasks to run asynchronously, we have to wait for them to complete. We can call `get()` to get the result. Since `get()` is a synchronous call, i.e., it blocks until the `CompletableFuture` completes, to maximize concurrency, we should only call `get()` as the final step in our code.

The method `CompletableFuture::get` throws a couple of checked exceptions: `InterruptedException` and `ExecutionException`, which we need to catch and handle. The former refers to the exception that the thread has been interrupted, while the latter refers to errors/exceptions during execution.

An alternative to `get()` is `join()`. `join()` behaves just like `get()` except that no checked exception is thrown.

Example

Let's look at some examples. Let's reuse our method that computes the i-th prime number.

```
int findIthPrime(int i) {
  return Stream
          .iterate(2, x -> x + 1)
          .filter(x -> isPrime(x))
          .limit(i)
          .reduce((x, y) -> y)
          .orElse(0);
}
```

Given two numbers i and j, we want to find the difference between the i-th prime number and the j-th prime number. We can first do the following:

```
CompletableFuture<Integer> ith = CompletableFuture.supplyAsync(() -> findIthPrime(i));
CompletableFuture<Integer> jth = CompletableFuture.supplyAsync(() -> findIthPrime(j));
```

These calls would launch two concurrent threads to compute the i-th and the j-th primes. The method calls `supplyAsync` returns immediately without waiting for `findIthPrime` to complete.

Next, we can say, that, when `ith` and `jth` complete, take the value computed by them, and take the difference. We can use the `thenCombine` method:

```
CompletableFuture<Integer> diff = ith.thenCombine(jth, (x, y) -> x - y);
```

This statement creates another `CompletableFuture` which runs asynchronously that will compute the difference between the two prime numbers. At this point, we can move on to run other tasks, or if we just want to wait until the result is ready, we call

```
diff.join();
```

to get the difference between the two primes[2].

Handling Exceptions

One of the advantages of using `CompletableFuture<T>` instead of `Thread` to handle concurrency is its ability to handle exceptions. `CompletableFuture<T>` has three methods that deal with exceptions: `exceptionally`, `whenComplete`, and `handle`. We will focus on `handle` since it is the most general.

Suppose we have a computation inside a `CompletableFuture<T>` that might throw an exception. Since the computation is asynchronous and could run in a different thread, the question of which thread should catch and handle the exception arises. `CompletableFuture<T>` keeps things simpler by storing the exception and passing it down the chain of calls, until `join()` is called. `join()` might throw `CompletionException` and whoever calls `join()` will be responsible for handling this exception. The `CompletionException` contains information on the original exception.

For instance, the code below would throw a `CompletionException` with a `NullPointerException` contains within it.

```
CompletableFuture.<Integer>supplyAsync(() -> null)
                 .thenApply(x -> x + 1)
                 .join();
```

Suppose we want to continue chaining our tasks despite exceptions. We can use the `handle` method, to handle the exception. The `handle` method takes in a `BiFunction` (similar to `cs2030s.fp.Combiner`). The first parameter to the `BiFunction` is the value, the second is the exception, the third is the return value.

Only one of the first two parameters is not `null`. If the value is `null`, this means that an exception has been thrown. Otherwise, the exception is `null`[3].

Here is a simple example where we use `handle` to replace a default value.

```
cf.thenApply(x -> x + 1)
  .handle((t, e) -> (e == null) ? t : 0)
  .join();
```