Unit 27: Type Inference

After this unit, students should:

- be familiar how Java infers missing type arguments

We have seen in the past units the importance of types in preventing run-time errors. Utilizing types properly can help programmers catch type mismatch errors that could have caused a program to fail during run-time, possibly after it is released and shipped.

By including type information everywhere in the code, we make the code explicit in communicating the intention of the programmers to the readers. Although it makes the code more verbose and cluttered -- it is a small price to pay for ensuring the type correctness of the code and reducing the likelihood of bugs as the code complexity increases.

Java, however, allows the programmer to skip some of the type annotations and try to infer the type argument of a generic method and a generic type, through the *type inference* process.

The basic idea of type inference is simple: Java will looking among the matching types that would lead to successful type checks, and pick the most specific ones.

Diamond Operator

One example of type inference is the diamond operator `<>` when we `new` an instance of a generic type:

```
Pair<String,Integer> p = new Pair<>();
```

Java can infer that `p` should be an instance of `Pair<String,Integer>` since the compile-time type of `p` is `Pair<String,Integer>`. The line above is equivalent to:

```
Pair<String,Integer> p = new Pair<String,Integer>();
```

Type Inferencing

We have been invoking

```java
class A {
  // version 0.7 (with wild cards array)
  public static <S> boolean contains(Array<? extends S> array, S obj) {
    for (int i = 0; i < array.getLength(); i++) {
      S curr = array.get(i);
      if (curr.equals(obj)) {
        return true;
      }
    }
    return false;
  }
}
```

by explicitly passing in the type argument `Shape` (also called *type witness* in the context of type inference).

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

We could remove the type argument `<Shape>` so that we can call `contains` just like a non-generic method:

```java
    A.contains(circleArray, shape);
```

and Java could still infer that `S` should be `Shape`. The type inference process looks for all possible types that match. In this example, the type of the two parameters must match. Let's consider each individually first:

- An object of type `Shape` is passed as an argument to the parameter `obj`. So `S` might be `Shape` or, if widening type conversion has occurred, one of the other supertypes of `Shape`. Therefore, we can say that `Shape <: S <: Object`.
- An `Array<Circle>` has been passed into `Array<? extends S>`. A widening type conversion occurred here, so we need to find all possible `S` such that `Array<Circle>` <: `Array<? extends S>`. This is true only if `S` is `Circle`, or another supertype of `Circle`. Therefore, we can say that `Circle <: S <: Object`.

Solving for these two constraints on `S`, we get the following:

```
Shape <: S <: Object
```

We therefore know that `S` could be `Shape` or one of its supertypes: `GetAreable` and `Object`. We choose the lower bound, so `S` is inferred to be `Shape`.

Type inferencing can have unexpected consequences. Let's consider an older version of `contains` that we wrote:

```java
class A {
    // version 0.4 (with generics)
    public static <T> boolean contains(T[] array, T obj) {
      for (T curr : array) {
        if (curr.equals(obj)) {
          return true;
        }
      }
      return false;
    }
}
```

Recall that we want to prevent nonsensical calls where we are searching for an integer in an array of strings.

```java
String[] strArray = new String[] { "hello", "world" };
A.<String>contains(strArray, 123); // type mismatch error
```

But, if we write:

```java
A.contains(strArray, 123); // ok!  (huh?)
```

The code compiles! Let's go through the type inferencing steps to understand what happened. Again, we have two parameters:

- `strArray` has the type `String[]` and is passed to `T[]`. So `T` must be `String` or its superclass `Object` (i.e. `String <: T <: Object`). The latter is possible since Java array is covariant.
- `123` is passed as type `T`. The value is treated as `Integer` and, therefore, `T` must be either `Integer`, or its superclasses `Number`, and `Object` (i.e. `Integer <: T <: Object`).

Solving for these two constraints:

```
T <: Object
```

Therefore `T` can only have the type `Object`, so Java infers `T` to be `Object`. The code above is equivalent to:

```java
A.<Object>contains(strArray, 123);
```

And our version 0.4 of `contains` actually is quite fragile and does not work as intended. We were bitten by the fact that the Java array is covariant, again.

Target Typing

The example above performs type inferencing on the parameters of the generic methods. Type inferencing can involve the type of the expression as well. This is known as *target typing*. Take the following upgraded version of `findLargest`:

```
// version 0.6 (with Array<T>)
public static <T extends GetAreable> T findLargest(Array<? extends T>
array) {
  double maxArea = 0;
  T maxObj = null;
  for (int i = 0; i < array.getLength(); i++) {
    T curr = array.get(i);
    double area = curr.getArea();
    if (area > maxArea) {
      maxArea = area;
      maxObj = curr;
    }
  }
  return maxObj;
}
```

and we call

```
Shape o = A.findLargest(new Array<Circle>(0));
```

We have a few more constraints to check:

- Due to target typing, the returning type of `T` must be a subtype of `Shape` (i.e. `T <: Shape`)
- Due to the bound of the type parameter, `T` must be a subtype of `GetAreable` (i.e. `T <: GetAreable`)
- `Array<Circle>` must be a subtype of `Array<? extends T>`, so `T` must be a supertype of `Circle` (i.e. `Circle <: T <: Object`)

Solving for all three of these constraints:

```
Circle <: T <: Shape
```

The lower bound is `Circle`, so the call above is equivalent to:

```
Shape o = A.<Circle>findLargest(new Array<Circle>(0));
```

Further Type Inference Examples

We now return to our `Circle` and `ColoredCircle` classes and the `GetAreable` interface. Recall that `Circle` implements `GetAreable` and `ColoredCircle` inherits from `Circle`.

Now lets consider the following method signature of a generic method `foo`:

```
public <T extends Circle> T foo(Array<? extends T> array)
```

Then we consider the following code excerpt:

```
ColoredCircle c = foo(new Array<GetAreable>());
```

What does the java compiler infer `T` to be? Lets look at all of the constraints on `T`.

- First we can say that the return type of `foo` must be a subtype of `ColoredCircle`, therefore we can say `T <: ColoredCircle`.
- `T` is also a bounded type parameter, and therefore we also know `T <: Circle`.
- Our method argument is of type `Array<GetAreable>` and must be a subtype of `Array<? extends T>`, so `T` must be a supertype of `GetAreable` (i.e. `GetAreable <: T <: Object`).

We can see that there no solution to our contraints, `T` can not be both a subtype of `ColoredCircle` and a supertype of `GetAreable` and therefore the Java compiler can not find a type `T`. The Java compiler will throw an error stating the inference variable `T` has incompatible bounds.

Lets consider, one final example using the following method signature of a generic method `bar`:

```
public <T extends Circle> T bar(Array<? super T> array)
```

Then we consider the following code excerpt:

```
GetAreable c = bar(new Array<Circle>());
```

What does the java compiler infer `T` to be? Again, lets look at all of the constraints on `T`.

- We can say that the return type of `bar` must be a subtype of `GetAreable`, therefore we can say `T <: GetAreable`.
- Our method argument is of type `Array<Circle>` and must be a subtype of `Array<? super T>`, so `T` must be a subtype of `Circle` (i.e. `T <: Circle`).

Solving for these two constraints:

```
T <: Circle
```

Whilst `ColoredCircle` is also a subtype of `Circle` it is not included in the above statement and therefore the compiler does not consider this class during type inference. Indeed, the compiler cannot be aware[1] of all subtypes of `Circle` and there could be more than one subtype. Therefore `T` can only have the type `Circle`, so Java infers `T` to be `Circle`.

Rules for Type Inference

We now summarize the steps for type inference. First, we figure out all of the type constraints on our type parameters, and then we solve these constraints. If no type can satisfy all the constraints, we know that Java will fail to compile. If in resolving the type constraints for a given type parameter `T` we are left with:

- `Type1 <: T <: Type2`, then `T` is inferred as `Type1`
- `Type1 <: T`[2], then `T` is inferred as `Type1`
- `T <: Type2`, then `T` is inferred as `Type2`

where `Type1` and `Type2` are arbitrary types.

---

1. Due to evolving specifications of software, at the time of compilation, a subtype may not have even been conceived of or written yet! ↩
2. Note that `T <: Object` is implicit here. We can see that this case could also be written as `Type1 <: T <: Object`, and would therefore also be explained by the previous case (`Type1 <: T <: Type2`). ↩