Subtype and Parametric Polymorphism

Definition: Subtype

Given two Java classes **A** and **B**, where **B extends A**, we say that **B** is a subtype of **A** and we write **B <: A** (if **I** is an interface and **B** implements **I**, directly or indirectly, we also say that **B <: I**)

> Example: For Java classes **class Shape {...}** and **class Circle extends Shape {...},** we can write **Circle <: Shape** and say that **Circle** is a subtype of **Shape**

Definition: Subtype (Inclusion) Polymorphism

If **B <: A**, then an object of type **B** can be safely used anytime an object of type **A** is needed.

> Example: Let us suppose that you have a **circle = new Circle()** and you need a **shape** object. Everything you can do with a **shape** object you can do with a **circle** object, perhaps even more. Why? Because a **circle** is a **shape**

Definition: Parametric Polymorphism

Parametric polymorphism is the property of a programming language that allows the creation of abstract types. In Java the parametric polymorphism is called generics. Generics allows abstracting over types.

> Example: Let us suppose that we want to build a **Drawing** class where we can draw or erase shapes. We have two options: (1) define a field in our class of type **Shape**, or (2) we could use a generic type **T**. Generics allow us to abstract over the shape type.

```
public class Drawing<T> {

     private T t; //type parameter

     public void draw(T t) { this.t = t; }

     public T erase() { return t; }

}
```

To create a **Drawing** object you must perform a *generic type invocation* which replaces **T** with some concrete object: **Drawing<Circle> dc = new Drawing<>();**

T in **Drawing<T>** is called *type parameter* and **Circle** in **Drawing<Circle>** is called *type argument*.

QUESTION:

Since Circle is a subtype of Shape type (Circle <: Shape), is Circle[] a subtype of Shape[]

(Circle[] <: Shape[]) ???

Would the following code compile? Would it run?

```
class Shape {}

class Circle extends Shape {}

class Rectangle extends Shape {}


public class Question {

  public static void main(String[] args) {

     Circle[] myCircles = {new Circle(), new Circle()};

     Shape[] myShapes = myCircles;


    myShapes[0] = new Rectangle(); // line 10

  }

}
```

Answer:

The code compile correctly, but when it runs we get an error:
java.lang.ArrayStoreException: Rectangle

```
at Question.main(Question.java:10)
```

Why ???

Arrays are in Java a *reifiable type*. This means that at run-time Java knows that myShapes array was actually instantiated as an array of Circles which simply happens to be accessed through a reference of type Shape[].

Lessons to be learned:

There is a difference between:

(1) The actual type of the object we have (array of circles), and
(2) The reference type we use to access that object (array of shapes)

Read next tutorial about variance, to learn how you can use compiler to help you caching these types of errors.

Covariance and Contravariance

Fundamental Question: If two types are in a subtype relationship and we have two collections of these types, what is the relationship between collections?

For example: If we have a painting of shapes and a painting of circles, could we think that the painting of shapes could be replaced by the painting of circles, since a circle is a shape. You may think that a painting of shapes is nicer that a painting that contains only circles and you do not accept the replacement.

What if you want a set of numbers and I will give you a set of integers? Would you accept it? After all an integer is a number. Let us try to define some concepts.

Definition: Variance

Variance studies how simple type relationships are translated into relationships of collections of simple type.

Definition: Covariant

Two types **A** and **B** are in a covariant relationship, if **A <: B** then **C(A) <: C(B)**, i.e., the ordering of types is preserved, where **C(X)** is a complex type of **X**.

Example 1: Arrays in Java are covariant

```
class Shape {}

class Circle extends Shape {}

Circle[] myCircles = {new Circle(), new Circle()};

Shape[] myShapes = myCircles; //this line compiles
```

Since **Circle[]** is a subtype of **Shape[]**, as **Circle** is a subtype of **Shape**, it means that arrays in Java are covariant. (If **A<:B** then **A[]<:B[]**)

Remember the question from the previous note?

class Rectangle extends Shape {}

myShapes[0] = new Rectagle(); //this line compiles, although it is a heap pollution

At runtime we get a **ArrayStoreException** - we try to put a rectangle into an array of circle, through a shape reference. Arrays in Java are reifiable type: runtime environment knows that **myShapes** variable references an array of circles.

Example 2: Covariant method return type

```
class Painting {

   Shape eraseShape() {...}

   void drawShape(Shape shape) {...}

}


class CirclePainting extends Painting {

 Circle eraseShape() { return new Circle(); }

 ...

}
```

The subclass **CirclePainting** overrides the **eraseShape** method from the superclass **Painting**, although it returns a different type. This is possible, because Java supports covariant return types (**CirclePainting <: Painting** and the return types maintain the same relationship – **Circle <: Shape**).

Definition: Contravariant

Two types **A** and **B** are in a contravariant relationship if the ordering of types is reversed, i.e., if **A <: B** then **C(B) <: C(A)**.

Example 3: Contravariant method argument type

```
class Painting {

    Shape eraseShape() {...}

    void drawShape(Shape shape) {...}

}


class CirclePainting extends Painting {

    ...

    void drawShape(Object shape) { }

}
```

There are languages that allow the subclass **CirclePainting** to override the **drawShape** method from the superclass **Painting** through a more general argument type. This would be called contravariant method argument type, since **Shape <: Object** and **CirclePainting <: Painting**. Java **does not support contravariant method argument type** (**drawShape** in **CirclePainting** is an overloaded method)

Definition: Bivariant

A relationship is bivariant if it is covariant and contravariant.

Definition: Invariant (Nonvariant)

A relationship is invariant if it is neither covariant, nor contravariant.

Read next tutorial about subtype variance in Java, to learn how you can use wildcards.

Wildcards - Variance and Generics

Fundamental Question: What kind of variance does exist between generic types?

Consider the classes:

```
public class Drawing<T>

  private T t; //type parameter

  public void draw(T t) { this.t = t; }

  public T take() { return t; }

}

class Shape {}

class Circle extends Shape {}

class Rectangle extends Shape {}
```

Consider the next two lines:

Drawing<Circle> circles = …;

Drawing<Shape> shapes = circles; // Is it right???

(1) It is not. If it were allowed, then **shapes.draw(new Rectagle())** would draw on a painting of circles, something different such as a rectangle.

What about this?

Drawing<Shape> shapes = …;

Drawing<Circle> circles = shapes;// Is it right???

(2) This is not right either. In a drawing of shapes, we can have all kinds of shapes not only circles.

Observation 1: **Generic types are invariant** (see 1 & 2 & read previous note about variance).

Problem: How could we have subtype relationship between generic types in Java?

What about a drawing of unknown types? Something defined like **Drawing<?>**

Definition: Wildcard Type

A wildcard type is a parameterized type, where **?** is the type argument.

Example: **Drawing<?>** is a wildcard type, and can be thought of a drawing of some type, where the wildcard is hiding that type, drawing of unknown type.

Definition: Upper-bounded Wildcard Type

An upper-bounded wildcard type is a parameterized type, where **? extends T** is the type argument (**? <: T**).

Example: Let us suppose that we want in our drawing only elements that are subtypes of **Circles**, but we do not know, what concrete type of circles. We can define a generic drawing type like this with: **Drawing<? extends Circle>**, where **Circle** is the upper bound.

Definition: Lower-bounded Wildcard Type

A lower-bounded wildcard type is a parameterized type, where **? super T** is the type argument (**T <: ?**).

Example: Again, let us suppose that we want in our drawing only elements that are supertypes of **Rectangle (**i.e. some unknown type **T**, where **Rectangle extends T)**. We can define a generic drawing type like this with: **Drawing<? super Rectangle >**, where **Rectangle** is the lower bound.

Note: The bounding types may also have wildcards as type arguments. For example, how would you read this statement: **Drawing<? extends Drawing<?>>**

It is a painting composed of any painting that is derived from a painting of anything.

Observation 2: Wildcard types allow variant subtyping

Observation 3: **Upper-bound wildcard types are covariant**

If **Circle <: Shape** then **Drawing(? extends Circle) <: Drawing(? extends Shape)**

Example:

Drawing<? extends Shape> ds = new Drawing<Circle>();

Shape shape = ds.take(circle); //it is allowed, after all a circle is a shape

ds.draw(new Rectangle()); //it is NOT allowed – compiler error

WHY? Since draw method requires a **T** parameter and **T** in our case is **? extends Shape**, we do not know what kind a shape is coming. If it is rectangle or one of rectangle supertype it is ok, since we expect rectangle, but if it is not, we are in trouble. So the compiler flags this as an error.

We can read from an upper-bound wildcard type, but we cannot write to it(i.e., upper-bound wildcard type are immutable).

Observation 4: **Lower-bound wildcard types are contravariant**

If **Circle <: Shape** then **Drawing(? super Shape) <: Drawing(? super Circle).**

Example:

Drawing<Shape> ds= new Drawing<Shape>();

Drawing<? super Circle> dc = ds;

dc.draw(new Circle()); // safe, to draw a circle , when a shape is needed

Circle c = dc.take(…); // it is NOT allowed – compiler error

WHY? Here the take method argument is **T**, which in our case is **? super Circle**, so since it is a superclass of **Circle**, it could be something different from a **Circle** and we could get a **ClassCastException**.

We can write to a lower-bound wildcard type, but we cannot read from it.

Observation 5: **Unbounded wildcard types are bivariant**

Drawing<? extends Shape> is a subtype of **Drawing<?>** and **Drawing<? super Shape>** is a supertype of **Drawing<?>**

Moreover, **Drawing<?> <: Drawing<?>**, an unbounded wildcard is a subtype of itself.

Unbounded wildcards are useful writing generic code that is completely independent of parameterized type

Lessons to be learned:

1. We use covariance (upper-bounded wildcard types), when we want only to read from covariant types.

2. We use contravariance (lower-bounded wildcard types), when we want only to write to contravariant types.

Exercise:

Define the signature of a method that copies a drawing of shapes.

Solution:

void copy(<? extends Shape> source, <? super Shape> destination);

Jordan Anastasiade