11.2 Subtype Polymorphism vs Explicit Higher Order Functions

We've seen how inheritance lets us reuse existing code in a superclass while implementing small modifications by overriding a superclass's methods or writing brand new methods in the subclass. Inheritance also makes it possible to design general data structures and methods using *polymorphism*.

Polymorphism, at its core, means 'many forms'. In Java, polymorphism refers to how objects can have many forms or types. In object-oriented programming, polymorphism relates to how an object can be regarded as an instance of its own class, an instance of its superclass, an instance of its superclass's superclass, and so on.

Consider a variable `deque` of static type Deque. A call to `deque.addFirst()` will be determined at the time of execution, depending on the run-time type, or dynamic type, of `deque` when `addFirst` is called. As we saw in the last chapter, Java picks which method to call using dynamic method selection.

Suppose we want to write a python program that prints a string representation of the larger of two objects. There are two approaches to this.

1. Explicit HoF Approach

```python
def print_larger(x, y, compare, stringify):
    if compare(x, y):
        return stringify(x)
    return stringify(y)
```

2. Subtype Polymorphism Approach

```python
def print_larger(x, y):
    if x.largerThan(y):
        return x.str()
    return y.str()
```

Using the explicit higher order function approach, you have a common way to print out the larger of two objects. In contrast, in the subtype polymorphism approach, the object *itself* makes the choices. The `largerFunction` that is called is dependent on what x and y actually are.

Previous
11.1 A Review of Dynamic Method Selection

Next
11.3 Comparables

Last updated 1 year ago