1 Administrivia

- Project 2 first deliverable due tonight
- Project 2 group meetings tomorrow during lab

2 Visitors

2.1 Overview

Visitors are a powerful design pattern that come in very useful when operating on tree-like data, such as the recursive data types that we have been discussing. The visitor pattern can be thought of as a direct counterpart to the interpreter pattern:

<table>
<thead>
<tr>
<th>Visitor</th>
<th>Interpreter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations decoupled from data</td>
<td>Operations embedded inside data</td>
</tr>
<tr>
<td>Adding new operations is easy</td>
<td>Adding new operations is hard</td>
</tr>
<tr>
<td>Adding new data types is hard</td>
<td>Adding new data types is easy</td>
</tr>
</tbody>
</table>

2.2 Implementation

To properly implement the visitor pattern, there are a few important components and methods each component should support:

- **Data elements** — Any element that could be traversed by a visitor should present a uniform interface to the visitor, either by extending from a common root type (e.g. `Music`, `Node`), or implementing an interface (e.g. `Visitable`). Any visitable data element must provide an `accept` method:
  
  $$ <T> T \text{ accept}(Visitor<T> \ \text{visitor}) $$ — The `accept` method on visitable elements is typically only responsible for calling the correct specific `visit` method on the visitor.

- **Visitor interface** — In order to properly decouple the data from the specific operations, we'll need an interface that the concrete visitors implement. The `accept` methods in the visitable elements will depend on this interface, instead of specific concrete visitors. For an interface defined as `Visitor<T>`, we will also need `visit` methods:
  
  $$ T \text{ visit}(SpecificType \ \text{e}) $$ — The visitor interface should have a `visit` method for each supported data type. We will abuse Java's method overloading to allow all of these methods to share the same name, but this is only a convenience. These methods could also be called something like `visitSpecificType`.

- **Concrete visitors** — These are the actual implementations of the visitor interface where we actually write the code to perform the operations that we want, e.g. `EvaluateVisitor` or `SizeVisitor`.

- **Entry point** — There must also eventually be some place where we actually create a visitor and make it execute. This is as simple as constructing a visitor and calling the `accept` method on the data with the visitor as an argument.

2.3 Double Dispatch

If you do it right, you should feel like the flow of your program is ping-ponging back and forth between your visitable elements and your visitor, back and forth through `accept` and `visit`. This is called *double dispatch*, and it is a clever way to avoid type casts and unsafe operations.

In plain English, here are what `accept` and `visit` really mean:

- `accept` — Visitor says, “Hey data, I don’t know what runtime type you actually are, but I’m going to pass myself to you, and hopefully you will know which of my `visit` methods to call.”

- `visit` — Data says, “Hey visitor, I’m actually this specific type, so run the code you have for my specific type please, and any of my child data types would be happy to `accept` you too.”
3 Generics

3.1 Overview

Generics are a language feature of Java that were added (or bolted) on in Java 1.5. They are a useful tool that allow you to define classes and methods that are flexible yet still type-safe with full compile-time type-checking. Let’s look at what we had to do before generics:

```java
List intList = new LinkedList();
intList.add(new Integer(0));
Integer x = (Integer) intList.get(0);
```

Look at that ugly cast. It is not only inconvenient, but code like this can often be prone to runtime casting exceptions that cannot be caught at compile-time. What if we had added a String in line 2 instead, and then tried to cast it to Integer?

With generics, we could rewrite that code and add a layer of compile-time checking to prevent those types of errors, and remove that ugly cast:

```java
List<Integer> intList = new LinkedList<Integer>();
intList.add(new Integer(0));
Integer x = intList.get(0);
```

3.2 Generic Classes

```java
public interface List<E> {
    void add(E e);
    int size();
}

public class ClassList implements List<Student> {
    void add(Student s) { ... }
    int size() { ... }
}
```

3.3 Generic Methods

```java
public static <T extends Number> T square(T x) { ... }

public static String join(String s, List<?> list) { ... }

public <T> T accept(Visitor<T> visitor) { ... }

public static <T extends Comparable<? super T>> void sort(List<T> list) { ... }
```

3.4 Subtyping

If $U$ is a subtype $V$, then for some generic type $C$, it is usually not the case that $C<U$ is a subtype of $C<V$.

Listing 1: Example of improper generics subtyping which will not compile

```java
List<String> stringList = new ArrayList<String>();
// we will create another variable that points to the same object
List<Object> objectList = stringList;
objectList.add(new Object());
String s = stringList.get(0); // how could this possibly make sense?
```