Generics in CSE 332

In CSE 332, we will *use* generics, but they are not an important topic. This guide is intended to walk you through the icky parts from a "usage" perspective. CSE 331 will cover exactly how these things work and why they are the way they are, but CSE 332 is not a "Java" course; so, we'll learn just enough to get by.

Creating Generic Arrays

Suppose we have a type parameter `E` and we would like to create a new array. The standard way we would do this would be:

```
E[] array = new E[SIZE];
```

Unfortunately, in Java, this will not compile. The underlying reason is how generics are implemented in Java. In particular, Java cannot figure out what constructor to call for `E`, because it does not know at runtime what `E` is. The workaround is to create an array of a base type (`Object`, `Comparable`, etc.) and cast it.

For `ArrayStack`, you will use the following declaration:

```
E[] array = (E[])new Object[SIZE];
```

For `CircularArrayFIFOQueue` and `MinFourHeap` in P1, you will use

```
E[] array = (E[])new Comparable[SIZE];
```

In P2, you will have to edit `MinFourHeap` to use `new Object[SIZE]` instead.

There is a similar issue if we would like to create an array of a parametrized type. For example, if we have a class (*not a type parameter*) called `Thing` which takes a type parameter `E`, and we would like to create an array of type `Thing<E>[]`, we would do the following:

```
Thing<E>[] array = (Thing<E>[])new Thing[SIZE];
```

That is, here, we are able to create the array, because the type is a real type, but the type `E` is still not known. So, we must cast it after creating the array.
In all of the above cases, you can add `@SuppressWarnings("unchecked")` above the method performing the cast to prevent compiler warnings.

Casting Generic Nodes to Specific Nodes

For all the trees (Tries, BSTs, AVL Trees) we will deal with this quarter, there will be a hierarchy of node types. The field `root` will usually be of the most general version. For example, for `AVLNode`, it will be `BSTNode`, and for `HashTrieNode`, it will be `TrieNode`. This means that you will always have to cast the root immediately after getting it. That is, anywhere you do `this.root` in your `HashTrieMap`, you should cast it like this before use:

```
(HashTrieNode)this.root
```

Type Parameters with Restrictions

You may not be familiar with type parameters that have restrictions. In CSE 332, we will never ask you to write these yourself, but it can be useful to be able to read them. Consider the type definition of `TrieMap`

```
TrieMap<A, K extends BString<A>, V> extends Dictionary<K, V>
```

This should be read in the following way:

- `A` is any type
- `K` is a subclass of `BString<A>`
- `V` is any type
- `TrieMap` must be a `Dictionary` that uses the above definitions of `K` and `V`