Last Updated: March 25, 2012 by Dan Grossman

Notes on Generic Arrays for CSE 332 Winter 2013

Introduction

CSE332 uses several more advanced features of Java than you may have seen previously. Our focus is not particularly "learning weird Java features" but we use them nonetheless because:

1. They are an excellent match for describing the sort of data structures and algorithms we are using.
2. Picking up programming knowledge "along the way" is a vital skill that you will never stop using.

In particular, we use Java's generic types a lot because they capture exactly the idea that many of our data structures and algorithms do not rely on the type of data being manipulated.

Resources for understanding Java generics include:

1. The Weiss book (required textbook), Chapter 1
2. The CoreJava book (recommended book): it really is worth taking the time to read relevant sections when you need to learn something new. An example that may not be covered in section is methods that take generic parameters. (This is different than a method of a generic class.)
3. The materials from section, particularly in the early weeks of the course

Unfortunately, generics interact pretty poorly with arrays in Java, which are also a crucial feature for data structures. In this document, we quickly cover the 3 bad interactions you are likely to encounter and the workarounds we suggest. We purposely don't go into the reasons that Java is less than ideal here (backward compatibility and type erasure and covariant arrays), since that is probably a better topic for a course on programming languages than one on data abstractions.

A warning on warnings

The workarounds all fundamentally involve writing Java code that creates a warning. This can desensitize you toward the "bad idea" of writing code that generates warnings. You are highly encouraged never to leave in warning-generating code unless you understand exactly why the warning is there and why it is unavoidable. Generics and arrays are one such "justifiable" warning, and you are unlikely to encounter very many others.

Workaround #1: The Basic Generic Array

You can declare a field or variable to have a generic array type:

```
class Foo<E> {
  E[] arr;
  void m() {
     E[] localArr;
     ...
  }
}
```

However, you cannot create a new generic array with:

```
new E[SOME_SIZE];
```

This is because when the code runs, the type `E` is "not known" and so the code has no idea how to identify the type of the elements for the array -- and that's important for checking that the array is used properly. The same idea works in many other languages, but it doesn't work in Java.

One workaround is to create an `Object[]` and then cast it (generating a warning) to `E[]`:

```
arr = (E[])new Object[SOME_SIZE]; // WORK-AROUND #1
```

Now when the code runs, the array that `arr` refers to will allow any `Object` in it, so there won't be any run-time problems. Be wary of casting from `Object[]` to `E[]` though -- this should only be used when creating a new array with `new`. Otherwise, other code might put items in your array that you are not expecting.

Workaround #2: The Array of a Parameterized Type

It's not just `E[]` that forbids array creation: we can't create an array where the elements have any parameterized type:

```
class C1<E> {
  E myEfield;
  ...
}

class C2 {
   // Doesn't work: C1<String>[] x = new C1<String>[100];
   C1<String>[] x = new C1[100]; // works fine
   ...
}

class C3<E> {
   C1<E>[] x = new C1<E>[100]; // not allowed
}
```

One natural reaction is to try our first workaround:

```
C1<E>[] x = (C1<E>[]) new Object[100]; // won't work either
```

This "natural reaction" will compile, but it will generate a `ClassCastException` when executed. To understand why, let's digress briefly to "plain old non-generic arrays" and Java's problematic treatment of array subtyping. Assume `B` is a subtype of `A`, such as with:

```
class A { ... }
class B extends A { ... }
```

Then this works:

```
B[] b_array = new B[100];
A[] a_array = b_array;
b_array = (B[])a_array;
```

Since an array of `B` objects contains all `A` objects (thanks to subtyping every `B` "is also" an `A`), this may seem fine. And it is allowed, provided two things:

1. You never assign into the array an `A` that is not a `B`. If you do this, at run-time you will get an `ArrayStoreException`. This is crucial so that any use of `b_array` can never see an array element that is not a `B`. But the compiler doesn't catch this.
2. When you do a cast like `(B[])a_array`, the code checks that `a_array` actually refers to an array that hold elements of type `B`. If it doesn't, then you get a `ClassCastException`.

The point is: Arrays "know their element type" and this is used to check that arrays are used properly and throw exceptions for misuse.

So now back to generics: While arrays "know their element type", they only know the "raw" type -- the type that forgets all about generics. But that's still enough for our "natural reaction" not to work: You cannot cast an array that holds elements of type `Object` to an array that holds elements of "raw type" `C1`. Such an array could have elements that are not of type `C1`, so this would not be safe and you get a `ClassCastException`.

So here's the workaround you actually need:

```
C1<E>[] x = (C1<E>[]) new C1[100]; // WORK-AROUND #2
```

When we created the array we said it would always hold all `C1` elements. `C1` here is a "raw type" -- we haven't said everything about the type of elements (like `C1<String>` or `C1<Integer>`), but we said that much. That's enough for the cast to succeed. This is just as safe/dangerous as our first work-around: you should only do this for newly created arrays or you can get strange errors where arrays don't hold elements of the type you think they do.

Work-around #3: Arrays of inner classes inside parameterized types

The last situation we'll walk through is actually very similar to work-around #2 once you understand what inner classes "really are". Consider:

```
class C<E> {
    class D { // inner class
      ...
    }
    D[] array = new D[100]; // doesn't work
}
```

Now this really seems annoying: `D` doesn't "look generic" so why can't we create an array of type `D[]`? Well, inner classes are convenient and great for indicating "this class `D` is used only inside class `C`", but the Java language does not treat them particularly specially. The class `D` is actually the class `C<E>.D` here: the class `D` defined inside the generic class `C<E>`. And so it is like you wrote this harder to read version:

```
C<E>.D[] array = new C<E>.D[100]; // same thing: doesn't work
```

Now it turns out Java doesn't let you write types like `C<E>.D` -- it will give a parse error. But that's what you "are really saying" when you write `D` inside class `C<E>`. So we'll use `C<E>.D` to *explain* what's going on, even though you can't write it.

The reason this doesn't work is `C<E>.D` mentions a type parameter and we know from work-around #2 that you can't create such arrays. Fortunately, the same work-around applies: use the raw type in the new expression and then cast it to the generic type:

```
C<E>.D[] array = (C<E>.D[]) new C.D[100]; // wordy work-around, not legal Java
```

Now you can't write this, but that's okay: just remember that `D` means `C<E>.D` and you get something that is easier to read anyway:

```
D[] array = (D[]) new C.D[100]; // WORK-AROUND #3
```

So while this 'cleaner' version looks different than work-around #2, the wordier version shows it's really the same concept extended to inner classes. The whole point is that `C.D` is a "raw type" but `D`, which means `C<E>.D`, is not.

Digression: Don't Shadow Type Parameters in Inner Classes

Though not related to arrays, another mistake related to inner classes is to write:

```
class C<E> {
   class D<E> { // inner class
       ...
   }
   ...
}
```

This works but is probably not what you meant: You are making `D` a generic class too -- the type parameter inside `D` is *different* than the enclosing type parameter for `C`. It's entirely analogous to the error of having a local variable in a method shadow a field name:

```
class F {
   int x;
   void m() {
      int x = 17;
      ++x; // why is this not referring to the field x?
   }
}
```

But in the case of generics, the mistake will lead to type errors. For example, this won't type-check:

```
class C<E> {
   E x;
   class D<E> {
       E y = x; // not the same type!
       ...
   }
}
```

But this works fine:

```
class C<E> {
   E x;
   class D {
       E y = x;
       ...
   }
}
```