Unit 25: Unchecked Warnings

After this unit, students should:

- be aware of how to use generics with an array
- be aware of unchecked warnings that compilers can give when we are using generics
- be able to make arguments why a piece of code is type-safe for simple cases
- know how to suppress warnings from compilers
- be aware of the ethics when using the @SuppressWarnings("unchecked") annotation
- know what is a raw type
- be aware that raw types should never never be used in modern Java

Creating Arrays with Type Parameters

We have seen how arrays and generics do not mix well. One way to get around this is to use Java Collections, a library of data structures provided by Java, instead of arrays, to store our items. The `ArrayList` class provides similar functionality to an array, with some performance overhead.

```
1  ArrayList<Pair<String,Integer>> pairList;
2  pairList = new ArrayList<Pair<String,Integer>>(); // ok
3
4  pairList.add(0, new Pair<Double,Boolean>(3.14, true));  // error
5
6  ArrayList<Object> objList = pairList;  // error
```

`ArrayList` itself is a generic class, and when parameterized, it ensures type-safety by checking for appropriate types during compile time. We can't add a `Pair<Double,Boolean>` object to a list of `Pair<String,Integer>`. Furthermore, unlike Java array, which is covariant, generics are invariant. There is no subtyping relationship between `ArrayList<Object>` and `ArrayList<Pair<String,Integer>>` so we can't alias one with another, preventing the possibility of heap pollution.

Using `ArrayList` instead of arrays only *gets around* the problem of mixing arrays and generics, as a user. `ArrayList` is implemented with an array internally after all. As computing students, especially computer science students, it is important to know how to

implement your own data structures instead of using ones provided by Java or other libraries.

Let's try to build one:

```
1  // version 0.1
2  class Array<T> {
3    private T[] array;
4
5    Array(int size) {
6      this.array = (T[]) new Object[size];
7    }
8
9    public void set(int index, T item) {
10     this.array[index] = item;
11   }
12
13   public T get(int index) {
14     return this.array[index];
15   }
16
17   public T[] getArray() {
18     return this.array;
19   }
20 }
```

This generic class is a wrapper around an array of type `T`. Recall that we can't `new T[]` directly. On Line 6, to get around this restriction, we `new` an `Object` array instead, and cast it to an array of `T[]` instead.

The code now compiles, but we receive the following message:

```
1  $ javac Array.java
2  Note: Array.java uses unchecked or unsafe operations.
3  Note: Recompile with -Xlint:unchecked for details.
```

Let's do what the compiler tells us, and compile with the `-Xlint:unchecked" flags.

```
1  $ javac -Xlint:unchecked Array.java
2  Array.java:6: warning: [unchecked] unchecked cast
3      array = (T[]) new Object[size];
4                    ^
5    required: T[]
6    found:    Object[]
7    where T is a type-variable:
8      T extends Object declared in class Array
9  1 warning
```

We get a warning that our Line 6 is doing an unchecked cast.

Unchecked Warnings

An unchecked warning is basically a message from the compiler that it has done what it can, and because of type erasures, there could be a run-time error that it cannot prevent. Recall that type erasure generates the following code:

```
(String) array.get(0);
```

Since `array` is an array of `Object` instances and Java array is covariant, the compiler can't guarantee that the code it generated is safe anymore.

Consider the following:

```
Array<String> array = new Array<String>(4);
Object[] objArray = array.getArray();
objArray[0] = 4;
array.get(0);  // ClassCastException
```

The last line would generate a `ClassCastException`, exactly a scenario that the compiler has warned us.

It is now up to us humans to change our code so that the code is safe. Suppose we remove the `getArray` method from the `Array` class,

```
// version 0.2
class Array<T> {
  private T[] array;

  Array(int size) {
    this.array = (T[]) new Object[size];
  }

  public void set(int index, T item) {
    this.array[index] = item;
  }

  public T get(int index) {
    return this.array[index];
  }
}
```

Can we prove that our code is type-safe? In this case, yes. Since `array` is declared as `private`, the only way someone can put something into the `array` is through the `Array::set` method[1]. `Array::set` only put items of type `T` into `array`. So the only type of objects we can get out of `array` must be of type `T`. So we, as humans, can see that casting `Object[]` to `T[]` is type-safe.

If we are sure (and only if we are sure) that the line

```
    array = (T[]) new Object[size];
```

is safe, we can thank the compiler for its warning and assure the compiler that everything is going to be fine. We can do so with the `@SuppressWarning("unchecked")` annotation.

```
// version 0.3
class Array<T> {
  private T[] array;

  Array(int size) {
    // The only way we can put an object into array is through
    // the method set() and we only put object of type T inside.
    // So it is safe to cast `Object[]` to `T[]`.
    @SuppressWarnings("unchecked")
    T[] a = (T[]) new Object[size];
    this.array = a;
  }

  public void set(int index, T item) {
    this.array[index] = item;
  }

  public T get(int index) {
    return this.array[index];
  }
}
```

`@SuppressWarning` is a powerful annotation that suppresses warning messages from compilers. Like everything that is powerful, we have the responsibility to use it properly:

- `@SuppressWarning` can apply to declaration at a different scope: a local variable, a method, a type, etc. We must always use `@SuppressWarning` to the *most limited* scope to avoid unintentionally suppressing warnings that are valid concerns from the compiler.
- We must suppress a warning *only if* we are sure that it will not cause a type error later.
- We must always add a note (as a comment) to fellow programmers explaining why a warning can be safely suppressed.

Note that since `@SuppressWarnings` cannot apply to an assignment but only to declaration, we declare a local variable `a` in the example above before assigning `this.array` to `a`.

Raw Types

Another common scenario where we can get an unchecked warning is the use of *raw types*. A raw type is a generic type used without type arguments. Suppose we do:

```
Array a = new Array(4);
```

The code would compile perfectly. We are using the generic `Array<T>` as a raw type `Array`. Java allows this code to compile for backward compatibility. This is anyway what the code looks like after type erasure and how we would write the code in Java before version 5. Without a type argument, the compiler can't do any type checking at all. We are back to the uncertainty that our code could bomb with `ClassCastException` after it ships.

Mixing raw types with paramterized types can also lead to errors. Consider:

```
Array<String> a = new Array<String>(4);
populateArray(a);
String s = a.get(0);
```

where the method `populateArray` uses raw types:

```
void populateArray(Array a) {
    a.set(0, 1234);
}
```

Since we use raw types in this method, the compiler can't help us. It will warn us:

```
Array.java:24: warning: [unchecked] unchecked call to set(int,T) as a
member of the raw type Array
    a.set(0, 1234);
         ^
  where T is a type-variable:
    T extends Object declared in class Array
1 warning
```

If we ignore this warning or worse, suppress this warning, we will get a run-time error when we execute `a.get(0)`.

Raw types must not be used in your code, ever. The only exception to this rule is using it as an operand of the `instanceof` operator. Since `instanceof` checks for run-time type and type arguments have been erased, we can only use the `instanceof` operator on raw types.

---

1. Another win for information hiding! ↵