CS 228: Introduction to Data Structures
Lecture 16

Raw Types

Generic types were introduced in Java 6. Prior to that, a container class such as an `ArrayList` just stored objects of type `Object`. Thus, you could write code like this:

```
ArrayList arr = new ArrayList();
arr.add(new Bee(1,”yellow”,“hill”));
arr.add(new Locust(1,”brown”));

for (int i = 0; i < arr.size(); ++i)
{
  Bee b = (Bee) arr.get(i); //need downcast
  b.makeHoney();
}
```

This code is unsafe, because it compiles, but yields a type error (`ClassCastException`) at runtime. Generic types were introduced to help prevent this kind of problem, by

enhancing type checking. In particular, `ArrayList` is now a generic class:

```
public class ArrayList<E> extends
AbstractList<E>
```

Using generics we can re-write the previous code so that the compiler will ensure that anything we `add()` to the list is compatible with the type `Bee`:

```
ArrayList<Bee> arr = new ArrayList<Bee>();
arr.add(new Bee(1,"yellow","hill"));
arr.add(new Locust(1,"brown")); // error is
      // now caught at compile time!

for (int i = 0; i < arr.size(); ++i)
{
  Bee b = arr.get(i); //no downcast
  b.makeHoney();
}
```

You can still use `ArrayList` the old way — it is called a "raw type" — but you will get a compiler warning.

Erasure

When introducing generics, the designers of Java wanted to maintain backward compatibility with raw types, while ensuring some sort of type checking. The approach they settled on uses ***erasure***; we explain this idea next.

Warning. Erasure leads to features and behaviors that may seem strange, confusing, or unintuitive; you can find plenty of discussion on this topic online. Here, we will limit ourselves only to the aspects of erasure that are essential for this course.

The Java compiler uses the type arguments to check for type errors. It then ***erases*** the type arguments when it produces the class file itself. The class file only keeps the ***upper bound*** of the type parameter, which defaults to `Object`, unless specified otherwise. This phenomenon is called ***erasure***[1]. Erasure ensures backward compatibility by, in a sense, reverting to raw types. The following code shows erasure at work.

[1] The technical term for this is that generic types in Java are “non-reified”

```
public class GenTest
{
  public static void main(String[] args)
  {
    ArrayList<String> arr1
       = new ArrayList<String>();
    ArrayList<Insect> arr2
       = new ArrayList<Insect>();
    arr1.add("Hello, world!");
    arr2.add(new Mantis(1, “green”));

    String s = arr1.get(0);
    Insect p = arr2.get(0);

    System.out.println(s);
    System.out.println(p.getColor());
    System.out.println(arr1.getClass()
       == arr2.getClass());
  }
}
```

Nothing about this is surprising until you get to the last line. There, because of erasure, `arr1` has the same runtime class as `arr2`. One way to see this is to run a decompiler like `Cavaj` on the class file for `GenTest`. Doing so would produce something like this:

```
import java.io.PrintStream;
import java.util.ArrayList;
public class GenTest
{
    public GenTest()
    {}

    public static void main(String args[])
    {
        ArrayList arr1 = new ArrayList();
        ArrayList arr2 = new ArrayList();
        arr1.add("Hello, world!");
        arr2.add(new Mantis(1, “green”));

        String s = (String)arr1.get(0);
        Insect p = (Insect)arr2.get(0);

        System.out.println(s);
        System.out.println(p.getColor());
        System.out.println(arr1.getClass()
            == arr2.getClass());
    }
}
```

Notice how all the type parameters have been erased.
Note also how the compiler inserts casts to make it

appear, for example, that `arr2.get(0)` returns an `Insect`.

Since the actual type information is not present in the compiled class files, you generally cannot do anything with a type parameter that would require the type to be known at runtime. For example, statements such as `new T()`, `new T[]`, `x instanceof T`, will not compile.

You ***can*** cast to a generic type, but since it accomplishes nothing, and you get a warning. For instance, after erasure

```
E foo = (E) bar;  // unchecked warning
```

is essentially equivalent to the code

```
Object foo = (Object) bar;
```

Creating a Generic Array

There is one case where we need an unsafe cast: to create an array of a generic type. This is done as follows.

```
E[] arr = (E[]) new Object[10];
        // Works, but triggers type warning
```

We will use this technique in the next part of this course, which deals with collections.

This is the only place in Com S 228 where you are allowed to have a type warning!

Collections

We often need a way to store and access a ***collection*** of elements of the same type. One option is to use an array, but this is only good for particular cases. Java and other programming languages offer many more options. Which option to choose depends on the properties we want our collection to have. Here are some factors to consider.

- Is the collection bounded in size?
- Does it allow duplicates?
- Does it allow null elements?
- Is the collection ***linearly ordered***? That is, does each element has a unique successor? Thus, it makes sense to ask about the 17th element, or given one element, to ask for the next one.
- Do elements have ***multiple successors***, like a directory tree?
- Is there ***no ordering*** at all, as in a mathematical set?
- Is there ***random access*** to all elements? E.g., can you go just as easily to item 23 as to item 2,323,232?
- Is the access sequence restricted somehow? Common restrictions are
    - **First-in first-out (FIFO):** We can only access the oldest element. Such a data structure is called a ***queue***.

- **Last-in first-out (LIFO):** We can only access the newest element. Such a data structure is called a ***stack***.

- **By priority:** We can only access the element of highest priority. Such a data structure is called a ***priority queue***.

In the weeks ahead, we will study the implementation and behavior of different types of collections. Depending on what properties are important in a given application, one type of collection might be superior to another. For example, if you need fast lookup, but don't need linear ordering, then a *hash set* could be more efficient than an `ArrayList`. (We'll study hash sets in the last part of this course.)

The Java Collections Framework

The Java Collections Framework — first added in JDK 1.2 — is a flexible and well-designed hierarchy of classes and interfaces. The basic interfaces from which everything else is built are `Collection<E>`[2] and `Iterator<E>`[3]. `Collection` is the least common denominator of all collections — it accommodates all the possibilities enumerated in the previous section (e.g., order/no order, random access/sequential access). For instance, `ArrayList<E>` is an implementation of `Collection<E>` in which there is a linear order and random access.

Here are the key methods of `Collection<E>`:

```
boolean add(E item)
int size()
boolean contains(Object obj)
boolean remove(Object obj)
Iterator<E> iterator()
```

The `add()`, `size()`, `contains()`, and `remove()` methods are self-explanatory (except, perhaps, for the fact

[2] http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

[3] http://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

that `contains()`, and `remove()` take as arguments an `Object` rather than an item of type `E` — we will address this later).

The `iterator()` method returns an ***iterator***, an object that enables us to traverse, access, and modify the elements in a collection. Java iterators are specified by the `Iterator<E>` interface, which contains essentially three methods[4]:

```
boolean hasNext()
E next()
void remove()
```

We can access the elements by repeatedly calling `next()` until `hasNext()` returns false. In this way, we are guaranteed to get all elements exactly once. If we call `next()` when `hasNext()` is false, we get a `NoSuchElementException`. In general, there are no guarantees about ordering — if we iterate over the collection again, we could get the elements in a different order.[5]

---

[4] Java 8 adds a `default` method `forEachRemaining()` to this list — see http://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html. We will not cover this method in Com S 228.

[5] The semantics of `remove()` are more complex; we'll explain them next time.

Example. The code below puts some strings in a `Collection<String>` and iterates over it. It uses the fact that `ArrayList` implements `Collection`[6].

```
Collection<String> c =
                    new ArrayList<String>();
c.add("Huey");
c.add("Louie");
c.add("Dewey");

Iterator<String> iter = c.iterator();
while (iter.hasNext())
{
   String s = iter.next();
   System.out.println(s);
}
```

Note. You may recall that an `ArrayList` has a `get(i)` method, which enables you to access the `i`-th element. There is no such method in `Collection`, because `Collection` does not assume linear order.

[6] http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

Foreach Loops

The pattern of iterating through the elements of a collection is so common that Java has an idiom for it. For instance, suppose c is of type `Collection<String>`. Then, you can write the code

```
Iterator<String> iter = c.iterator();
while (iter.hasNext())
{
   String s = iter.next();
   System.out.println(s);
}
```

more concisely as

```
for (String s : c)
{
   System.out.println(s);
}
```

This is called a ***foreach loop***. When you use a foreach loop, the Java compiler instantiates the iterator for you. Of course, this assumes that `iterator()` has been implemented.