Programming Fundamentals 2

Pierre Talbot

18 May 2021

University of Luxembourg

Chapter IX. Parametric Polymorphism

Introduction

- **Context**: In lab 2, you implemented `DynamicArray`.
- **Problem**: It can only store integer values.
- **Today**: How can we design an array for any kind of values?

```
public class DynamicArray {
  private ?? data;

  public DynamicArray() { ?? }
  public int size() { ?? }
  public boolean add(?? e) { ?? }
  public ?? get(int index) { ?? }
}
```

Solution 1: with `Object`

We can use an array of `Object`, since, remember, every class inherits from `Object`.

```
public class ArrayList {
  static final int DEFAULT_CAPACITY = 10;
  private Object[] data;
  private int size = 0;

  public ArrayList() { data = new Object[DEFAULT_CAPACITY]; }
  public int size() { return size; }
  public void add(Object e) {
    ensureCapacity();
    data[size] = e;
    ++size;
  }
  public Object get(int i) {
    if(i < 0 || i >= size) { throw OutOfBoundException(); }
    return data[i];
  }
  private void ensureCapacity() { /* ... */ }
}
```

Problems. . .

- Make a list of String.
- Add and retrieve a string with this list.
- Is it easy and readable?

Problems. . .

- Make a list of String.
- Add and retrieve a string with this list.
- Is it easy and readable?

The *downcast* `(String)e;` is not very readable and secure, why?

```
ArrayList personNames = new ArrayList();
personNames.add(new String("Gertrude"));
personNames.add(new String("Johnny")));
Object e = personNames.get(1);
String name = (String)e;
```

Problems. . .

- Make a list of String.
- Add and retrieve a string with this list.
- Is it easy and readable?

The *downcast* `(String)e;` is not very readable and secure, why?

```
ArrayList personNames = new ArrayList();
personNames.add(new String("Gertrude"));
personNames.add(new String("Johnny")));
Object e = personNames.get(1);
String name = (String)e;
```

Exception `ClassCastException` for:

```
Integer i = (Integer)e;
```

And so?

- Until Java 5.0, it was the only solution.
- In Java 5.0, the concept of *generics* enables parametric polymorphism.

What are the problems of an array of `Object`?

- *Casts* are required.
- No compile-time check if the *cast* is invalid.
- For instance: `House h = (House)e`, in the previous example, compiles, but an exception is thrown at runtime.

Parametric polymorphism: don't repeat yourself!

- To avoid casts, we could create a `ArrayList` class for each types, *e.g.*, `ArrayListInteger` or `ArrayListPokemonCard`.
- But the implementation of the methods would be **redundant**.
- Actually, **we don't even need to know the underlying type** to implement these methods!
- **Solution**: Use generics!

Advantages

- The code is safer and more readable.
- Decrease runtime casts.
- Allows us to write generic classes and algorithms more easily.

Solution 2: Generics (first try)

```
public class ArrayList<T> {
  static final int DEFAULT_CAPACITY = 10;
  private T[] data;
  private int size = 0;

  public ArrayList() { data = new T[DEFAULT_CAPACITY]; }
  public int size() { return size; }
  public void add(T e) { /* as in solution 1 */ }
  public T get(int i) { /* as in solution 1 */ }
  private void ensureCapacity() { /* */ }
}
```

- `ArrayList<T>` in now parametric in a type `T`.
- `ArrayList<T>` remains a class, that can be used as a "normal class".

A subtlety (second try)

```
public class ArrayList<T> {
  static final int DEFAULT_CAPACITY = 10;
  private T[] data;
  private int size = 0;

  public ArrayList() { data = (T[]) new Object[DEFAULT_CAPACITY];}
  public int size() { return size; }
  public void add(T e) { /* idem */ }
  public T get(int i) { /* idem */ }
  private void ensureCapacity() { /* */ }
}
```

Java does not support creating array of generic elements. Therefore, we create an array of objects that we cast immediately to the generic type.

Generics are transformed at compile-type

Backward compatible extension

- When generics were introduced, a lot of code already exists, so this existing code should not break with new Java version.
- **Solution**: Generics are *erased* at compile-time, and transformed into `Object`.
- Hence, generics are actually transformed to the code we had in solution 1, but we have additional safety guarantees.

Compiling generics

Two techniques

1. *Code expansion* (such as in C++), a new class is automatically created for each class instantiation:
   - `ArrayList<Double>` $\longrightarrow$ `ArrayListDouble`
   - `ArrayList<String>` $\longrightarrow$ `ArrayListString`
   - The parametric type `T` is replaced by the real one.
2. *Type erasure* (as in Java)
   - The parametric type `T` is replaced by a super type (`Object`).
   - Type conversions are added by the compiler automatically.
   - Generated code is the same as for solution 2.

Two usages of generic classes, a single code

- In Java, the generic type is replaced by `Object`.
- Which means that we can actually use `ArrayList` as a generic class **or not**.
- For instance, we can write `ArrayList` without generic parameter, and we will have a class with array of objects.

Two usages of generic classes, a single code

Non-generic usage

```
ArrayList x = new ArrayList();
x.add(new String("M. George"));
x.add(new Integer(0));
String name = (String)x.get(0);
```

Two usages of generic classes, a single code

Non-generic usage

```
ArrayList x = new ArrayList();
x.add(new String("M. George"));
x.add(new Integer(0));
String name = (String)x.get(0);
```

```
Line 2: warning: [unchecked] unchecked call to add(T) as a member
    of the raw type ArrayList
 x.add(new String("M. George"));
Line 3: warning: [unchecked] unchecked call to add(T) as a member
    of the raw type ArrayList
 x.add(new Integer(0));
```

Two usages of generic classes, a single code

Non-generic usage

```
ArrayList x = new ArrayList();
x.add(new String("M. George"));
x.add(new Integer(0));
String name = (String)x.get(0);
```

```
Line 2: warning: [unchecked] unchecked call to add(T) as a member
    of the raw type ArrayList
 x.add(new String("M. George"));
Line 3: warning: [unchecked] unchecked call to add(T) as a member
    of the raw type ArrayList
 x.add(new Integer(0));
```

- Compiler will output *warnings*.
- Heterogeneous array (several types) are generally a bad idea, it is better to use inheritance or enumeration instead.
- Always the risk to generate an exception if we mess up the cast.

Two usages of generic classes, a single code

Generic usage

```
ArrayList<String> x = new ArrayList<String>();
x.add(new String("M. George"));
x.add(new Integer(0));
String name = x.get(0);
```

Two usages of generic classes, a single code

Generic usage

```
ArrayList<String> x = new ArrayList<String>();
x.add(new String("M. George"));
x.add(new Integer(0));
String name = x.get(0);
```

```
Line 3: error: incompatible types: Integer cannot be converted
                to String
    x.add(new Integer(0));
```

Two usages of generic classes, a single code

Generic usage

```
ArrayList<String> x = new ArrayList<String>();
x.add(new String("M. George"));
x.add(new Integer(0));
String name = x.get(0);
```

```
Line 3: error: incompatible types: Integer cannot be converted
                to String
    x.add(new Integer(0));
```

- The compiler generates an *error*.
- It guarantees we can only put in the list what is specified in the angle brackets (`ArrayList<MyType>`).
- No need to *cast* when we use `get`, we give the compiler enough information so it can safely add the cast itself.

Advanced concepts of generics

Multiple generics parameters

- Some classes need several generics type.
- For instance in the associative array data structure.

Associative array

- Associate a key to a value. For instance, the name of someone to its address.
- `HashMap<String, Address> directory = new HashMap<String, Address>();`

```
public class SimpleMap<K,V> { // Key and Value
  private ArrayList<Pair<K,V>> data;
  private static class Pair<K,V> {
    public K key;
    public V value;
  }
  // ...
}
```

Type inference

- Type inference allows us to ask the compiler to *guess* (or *infer*) the type of an expression.
- It is not very powerful in Java but still useful for clarity.

```
HashMap<String, Address> directory = new HashMap<>();
```

Generic methods I

Challenge

Create a static method `head` which takes an `ArrayList` and returns the first element.

Generic methods I

Challenge

Create a static method `head` which takes an `ArrayList` and returns the first element.

Non-generic

```
public class ArrayListTools {
  public static Object head(ArrayList data) {
    return data.get(0);
  }}
```

Generic methods II

Is it working if we write the following?

```
ArrayList<String> names = new ArrayList<String>();
ArrayListTools.head(names);
```

Generic methods II

Is it working if we write the following?

```
ArrayList<String> names = new ArrayList<String>();
ArrayListTools.head(names);
```

- No! `ArrayList<String>` does not inherit from `ArrayList<Object>`.
- *Invariant types*: Inheritance is not propagated to type parameters, *i.e.*, `X<T>` and `X<U>` are never subtypes of each other.
- *Covariant types*: This is not the case with array, *i.e.*, `String[]` is a subtype of `Object[]`.

Generic methods II

Is it working if we write the following?

```
ArrayList<String> names = new ArrayList<String>();
ArrayListTools.head(names);
```

- No! `ArrayList<String>` does not inherit from `ArrayList<Object>`.
- *Invariant types*: Inheritance is not propagated to type parameters, *i.e.*, `X<T>` and `X<U>` are never subtypes of each other.
- *Covariant types*: This is not the case with array, *i.e.*, `String[]` is a subtype of `Object[]`.

We should use a generic method:

Generic method

```
public class ArrayListTools {
  public static <T> T head(ArrayList<T> data) {
    return data.get(0);
  }}
```

Bounded type parameters

When a class is instantiated with a generic type `T`, it has no information on `T`, thus cannot call any method on this object.

We can bound the type.

```
class SortedArrayList<T extends Comparable> {
  private T[] data;
  // ...
    data[i].compareTo(data[i+1]); // ok, T implements Comparable.
}
```

- Subtlety: We use `extends` even if `Comparable` is an interface.
- We can also give several type bounds: `<T extends Comparable & Cloneable>`.

More on generics

- Lower and upper type bounds.
- Wildcard (<?>).
- ...

More on the topic:

- *Effective Java, Chapter 5.*
- `http://en.wikipedia.org/wiki/Generics_in_Java`
- `http://en.wikipedia.org/wiki/Wildcard_%28Java%29`
- On a more general topic: `http://en.wikipedia.org/wiki/Covariance_and_contravariance_%28computer_science%29`
- Another book: *Java Generics and Collections, Maurice Naftalin and Philip Wadler, O'reilly, 2006*