Generics in Java

EECS2030 B & E: Advanced Object Oriented Programming
Fall 2021

CHEN-WEI WANG

YORK UNIVERSITÉ UNIVERSITY

Learning Outcomes

This module is designed to help you learn about:

1. A ***general*** collection `Object[]`: storage vs. retrieval
2. A ***generic*** collection `E[]`: storage vs. retrieval
3. Reinforce: Polymorphism, ***Type Casting***, `instanceof` checks

Motivating Example: A Book of Objects

```
1 public class Book {
2   private String[] names;
3   private Object[] records;
4   /* add a name-record pair to the book */
5   public void add (String name, Object record) { ... }
6   /* return the record associated with a given name */
7   public Object get (String name) { ... } }
```

Question: Which line has a type error?

```
1 Date birthday; String phoneNumber;
2 Book b; boolean isWednesday;
3 b = new Book();
4 phoneNumber = "416-67-1010";
5 b.add ("Suyeon", phoneNumber);
6 birthday = new Date(1975, 4, 10);
7 b.add ("Yuna", birthday);
8 isWednesday = b.get("Yuna").getDay() == 4;
```


Motivating Example: Observations (1)

- In the `Book` class:
  - By declaring the attribute

    ```
    Object[] records
    ```

    We meant that each book instance may store any object whose *static type* is a *descendant class* of `Object`.
  - Accordingly, from the return type of the `get` method, we only know that the returned record is an `Object`, but not certain about its *dynamic type* (e.g., `Date`, `String`, *etc.*).
    ∴ a record retrieved from the book, e.g., `b.get("Yuna")`, may only be called upon methods in its *static type* (i.e,. `Object`).
- In the tester code of the `Book` class:
  - In **Line 1**, the *static types* of variables `birthday` (i.e., `Date`) and `phoneNumber` (i.e., `String`) are *descendant classes* of `Object`.
  - So, **Line 5** and **Line 7** compile.

Motivating Example: Observations (2)

In a ***polymorphic collection***, ***dynamic types*** of stored objects (e.g., `phoneNumber` and `birthday`) need not be the same.

- Methods expected on the ***dynamic types*** (e.g., method `getDay` of class `Date`) may be new methods **not** inherited from `Object`.
- This is why **Line 8** would fail to compile, and may be fixed using an explicit *cast* :

```
isWednesday = ((Date) b.get("Yuna")).getDay() == 4;
```

- But what if the ***dynamic type*** of the returned object is not a `Date`?

```
isWednesday = ((Date) b.get("Suyeon")).getDay() == 4;
```

- To avoid such a `ClassCastException` at runtime, we need to check its ***dynamic type*** before performing a cast:

```
if (b.get("Suyeon") instanceof Date) {
  isWednesday = ((Date) b.get("Suyeon")).getDay() == 4;
}
```


Motivating Example: Observations (2.1)

- It seems: Combining ***`instanceof`*** checks & ***type casts*** works.
- Can you see any potential problem(s) w.r.t. the ***Single-Choice*** design principle?
- **Hints:** What happens when you have a large number of records of distinct ***dynamic types*** stored in the book (e.g., `Date`, `String`, `Person`, `Account`, *etc.*)?

Motivating Example: Observations (2.2)

Imagine that the tester code (or an application) stores 100 different record objects into the book.

- All of these records are of *static type* `Object`, but of distinct *dynamic types*.

```
Object rec1 = new C1(); b.add(..., rec1);
Object rec2 = new C2(); b.add(..., rec2);
...
Object rec100 = new C100(); b.add(..., rec100);
```

where classes ***C1*** to ***C100*** are *descendant classes* of `Object`.

- **Every time** you retrieve a record from the book, you need to check "exhaustively" on its *dynamic type* before calling some method(s).

```
Object rec = b.get("Jim");
if (rec instanceof C1) { ((C1) rec).m1; }
...
else if (rec instanceof C100) { ((C100) rec).m100; }
```

- Writing out this list multiple times is tedious and error-prone!

Motivating Example: Observations (3)

We need a solution that:

- Saves us from explicit `instanceof` checks and type casts
- Eliminates the occurrences of `ClassCastException`

As a sketch, this is how the solution looks like:

- When the user declares a `Book` object `b`, they must *commit to the kind of record that `b` stores at runtime* .
  e.g., `b` stores either `Date` objects only or `String` objects only, but ***not a mix***.
- When attempting to **store** a new record object `rec` into `b`, what if `rec`'s *static type* is not a **descendant class** of the type of book that the user previously commits to?
  ⇒ A *compilation error*
- When attempting to **retrieve** a record object from `b`, there is *no longer a need to check and cast* .
  ∵ ***Static types*** of all records in `b` are guaranteed to be the same.

Parameters

- In mathematics:
  - The same *function* is applied with different *argument values*.
    e.g., `2 + 3`, `1 + 1`, `10 + 101`, *etc.*
  - We *generalize* these instance applications into a definition.
    e.g., $+ : (\mathbb{Z} \times \mathbb{Z}) \to \mathbb{Z}$ is a function that takes two integer *parameters* and returns an integer.
- In Java programming:
  - We want to call a *method*, with different *argument values*, to achieve a similar goal.
    e.g., `acc.deposit(100)`, `acc.deposit(23)`, *etc.*
  - We generalize these possible method calls into a definition.
    e.g., In class `Account`, a method `void deposit(int amount)` takes one integer *parameter*.
- When you design a mathematical function or a Java method, always consider the list of *parameters*, each of which representing a set of possible *argument values*.

Java Generics: Design of a Generic Book

```
class Book <E> {
  private String[] names;
  private E[] records;
  /* add a name-record pair to the book */
  public void add (String name, E record) { ... }
  /* return the record associated with a given name */
  public E get (String name) { ... } }
```

Question: Which line has a type error?

```
1 Date birthday; String phoneNumber;
2 Book<Date> b; boolean isWednesday;
3 b = new Book<Date>();
4 phoneNumber = "416-67-1010";
5 b.add ("Suyeon", phoneNumber);
6 birthday = new Date(1975, 4, 10);
7 b.add ("Yuna", birthday);
8 isWednesday = b.get("Yuna").getDay() == 4;
```


Java Generics: Observations

- In class `Book`:
  - At the class level, we *parameterize the type of records* that an instance of book may store: `class Book<E>`
    where `E` is the name of a type parameter, which should be ***instantiated*** when the user declares an instance of `Book`.
  - Every occurrence of `Object` (the most general type of records) is replaced by `E`.
  - As soon as `E` at the class level is committed to some known type (e.g., `Date`, `String`, *etc.*), every occurrence of `E` will be replaced by that type.
- In the tester code of `Book`:
  - In **Line 2**, we commit that the book `b` will store `Date` objects only.
  - **Line 5** now fails to compile. [`String` is not `Date`'s descendant]
  - **Line 7** still compiles.
  - **Line 8** does *not need* any instance check and type cast, and does *not cause* any `ClassCastException`.
    ∵ Only `Date` objects were allowed to be stored.

Example Generic Classes: `ArrayList`

An ArrayList acts like a "resizable" array (indices start at 0).
Extra tutorial here.

`int`	**`size()`** Returns the number of elements in this list.
`boolean`	**`add(E e)`** Appends the specified element to the end of this list.
`void`	**`add(int index, E element)`** Inserts the specified element at the specified position in this list.
`boolean`	**`contains(Object o)`** Returns `true` if this list contains the specified element.
`E`	**`remove(int index)`** Removes the element at the specified position in this list.
`boolean`	**`remove(Object o)`** Removes the first occurrence of the specified element from this list, if it is present.
`int`	**`indexOf(Object o)`** Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
`E`	**`get(int index)`** Returns the element at the specified position in this list.

Using Generic Classes: `ArrayList`

```
1  import java.util.ArrayList;
2  public class ArrayListTester {
3    public static void main(String[] args) {
4      ArrayList<String> list = new ArrayList<String>();
5      println(list.size());
6      println(list.contains("A"));
7      println(list.indexOf("A"));
8      list.add("A");
9      list.add("B");
10     println(list.contains("A")); println(list.contains("B")); println(list.contains("C"));
11     println(list.indexOf("A")); println(list.indexOf("B")); println(list.indexOf("C"));
12     list.add(1, "C");
13     println(list.contains("A")); println(list.contains("B")); println(list.contains("C"));
14     println(list.indexOf("A")); println(list.indexOf("B")); println(list.indexOf("C"));
15     list.remove("C");
16     println(list.contains("A")); println(list.contains("B")); println(list.contains("C"));
17     println(list.indexOf("A")); println(list.indexOf("B")); println(list.indexOf("C"));
18
19     for(int i = 0; i < list.size(); i ++) {
20       println(list.get(i));
21     }
22   }
23 }
```


Using Generic Classes: `HashTable`

```
1  import java.util.Hashtable;
2  public class HashTableTester {
3    public static void main(String[] args) {
4      Hashtable<String, String> grades = new Hashtable<String, String>();
5      System.out.println("Size of table: " + grades.size());
6      System.out.println("Key Alan exists: " + grades.containsKey("Alan"));
7      System.out.println("Value B+ exists: " + grades.containsValue("B+"));
8      grades.put("Alan", "A");
9      grades.put("Mark", "B+");
10     grades.put("Tom", "C");
11     System.out.println("Size of table: " + grades.size());
12     System.out.println("Key Alan exists: " + grades.containsKey("Alan"));
13     System.out.println("Key Mark exists: " + grades.containsKey("Mark"));
14     System.out.println("Key Tom exists: " + grades.containsKey("Tom"));
15     System.out.println("Key Simon exists: " + grades.containsKey("Simon"));
16     System.out.println("Value A exists: " + grades.containsValue("A"));
17     System.out.println("Value B+ exists: " + grades.containsValue("B+"));
18     System.out.println("Value C exists: " + grades.containsValue("C"));
19     System.out.println("Value A+ exists: " + grades.containsValue("A+"));
20     System.out.println("Value of existing key Alan: " + grades.get("Alan"));
21     System.out.println("Value of existing key Mark: " + grades.get("Mark"));
22     System.out.println("Value of existing key Tom: " + grades.get("Tom"));
23     System.out.println("Value of non-existing key Simon: " + grades.get("Simon"));
24     grades.put("Mark", "F");
25     System.out.println("Value of existing key Mark: " + grades.get("Mark"));
26     grades.remove("Alan");
27     System.out.println("Key Alan exists: " + grades.containsKey("Alan"));
28     System.out.println("Value of non-existing key Alan: " + grades.get("Alan"));
29     }
30 }
```


Example Generic Classes: `HashTable`

A HashTable acts like a two-column table of (searchable) keys and values. ***Extra tutorial here.***

`int`	**`size()`** Returns the number of keys in this hashtable.
`boolean`	**`containsKey(Object key)`** Tests if the specified object is a key in this hashtable.
`boolean`	**`containsValue(Object value)`** Returns true if this hashtable maps one or more keys to this value.
`V`	**`get(Object key)`** Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
`V`	**`put(K key, V value)`** Maps the specified key to the specified `value` in this hashtable.
`V`	**`remove(Object key)`** Removes the key (and its corresponding value) from this hashtable.

Bad Example of using Generics

Has the following client made an appropriate choice?

```
Book<Object> book
```

NO!!!!!!!!!!!!!!!!!!!!!!!!

- It allows **all** kinds of objects to be stored.
  ∴ All classes are descendants of ***Object***.
- We can expect **very little** from an object retrieved from this book.
  ∴ The ***static type*** of `book`'s items are ***Object***, root of the class hierarchy, has the *minimum* amount of methods available for use.
  ∴ Exhaustive list of casts are unavoidable.

[***bad*** for extensibility and maintainability]

Beyond this lecture ...

- Study `https://docs.oracle.com/javase/tutorial/java/generics/index.html` for further details on Java generics.
- Play with the source code `ExampleBooks`.
- Review the basic `ArrayList` and `HashTable` methods:
  - `ArrayList`: `https://www.youtube.com/watch?v=Gg_RRaGN7o8&list=PL5dxAmCmjv_4uhxBzBt-CnSGw6kZ9C-xe&index=5`
  - `Hashtable`: `https://www.youtube.com/watch?v=vM_JTnvDn1g&list=PL5dxAmCmjv_4uhxBzBt-CnSGw6kZ9C-xe&index=7`

Index (1)

Learning Outcomes
Motivating Example: A Book of Objects
Motivating Example: Observations (1)
Motivating Example: Observations (2)
Motivating Example: Observations (2.1)
Motivating Example: Observations (2.2)
Motivating Example: Observations (3)
Parameters
Java Generics: Design of a Generic Book
Java Generics: Observations
Example Generic Classes: `ArrayList`

Index (2)

Using Generic Classes: `ArrayList`
Example Generic Classes: `HashTable`
Using Generic Classes: `HashTable`
Bad Example of using Generics
Beyond this lecture ...