Generics in Java

EECS2030 B: Advanced
Object Oriented Programming
Fall 2018

CHEN-WEI WANG

Motivating Example: A Book of Objects

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

Due to *polymorphism*, the *dynamic types* of stored objects (e.g., `phoneNumber` and `birthday`) need not be the same.

- Methods supported in 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` *check* and *type cast* works.
- Can you see any potential problem(s)?
- **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> {
  String[] names;
  E [] records;
  /* add a name-record pair to the book */
  void add (String name, E record) { ... }
  /* return the record associated with a given name */
  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 a `Date`]
  - **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.

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 features available for use.
  ∵ Exhaustive list of casts are unavoidable.

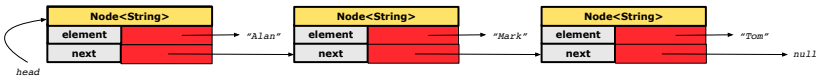
[***bad*** for extensibility and maintainability]

Generic Classes: Singly-Linked List (1)

```
public class Node< E > {
  private E element;
  private Node< E > next;
  public Node( E e, Node< E > n) { element = e; next = n; }
  public E getElement() { return element; }
  public Node< E > getNext() { return next; }
  public void setNext(Node< E > n) { next = n; }
  public void setElement( E e) { element = e; }
}
```

```
public class SinglyLinkedList< E > {
  private Node< E > head;
  private Node< E > tail;
  private int size = null;
  public void addFirst( E e) { ... }
  Node< E > getNodeAt (int i) { ... }
  ...
}
```

Generic Classes: Singly-Linked List (2)



Approach 1

```
Node<String> tom = new Node<>("Tom", null);
Node<String> mark = new Node<>("Mark", tom);
Node<String> alan = new Node<>("Alan", mark);
```

Approach 2

```
Node<String> alan = new Node<>("Alan", null);
Node<String> mark = new Node<>("Mark", null);
Node<String> tom = new Node<>("Tom", null);
alan.setNext(mark);
mark.setNext(tom);
```


Generic Classes: Singly-Linked List (3)

Assume we are in the context of class `SinglyLinkedList`.

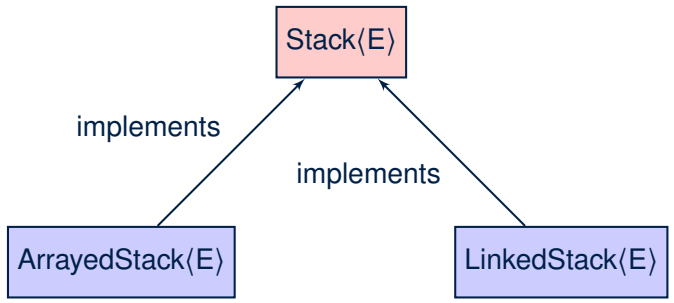
```
void addFirst (E e) {
  head = new Node<E>(e, head);
  if (size == 0) { tail = head; }
  size ++;
}
```

```
Node<E> getNodeAt (int i) {
  if (i < 0 || i >= size) {
    throw new IllegalArgumentException("Invalid Index"); }
  else {
    int index = 0;
    Node<E> current = head;
    while (index < i) {
      index ++; current = current.getNext();
    }
    return current;
  }
}
```


Generic Stack: Interface

```
public interface Stack<E> {
  public int size();
  public boolean isEmpty();
  public E top();
  public void push(E e);
  public E pop();
}
```


Generic Stack: Architecture



Generic Stack: Array Implementation

```
public class ArrayedStack< E > implements Stack< E > {
  private static final int MAX_CAPACITY = 1000;
  private E [] data;
  private int t; /* top index */
  public ArrayedStack() {
    data = (E []) new Object[MAX_CAPACITY];
    t = -1; }
  public int size() { return (t + 1); }
  public boolean isEmpty() { return (t == -1); }
  public E top() {
    if (isEmpty()) { /* Error: Empty Stack. */ }
    else { return data[t]; } }
  public void push(E e) {
    if (size() == MAX_CAPACITY) { /* Error: Stack Full. */ }
    else { t ++; data[t] = e; } }
  public E pop() {
    E result;
    if (isEmpty()) { /* Error: Empty Stack */ }
    else { result = data[t]; data[t] = null; t --; }
    return result; }
}
```


Generic Stack: SLL Implementation

```
public class LinkedStack< E > implements Stack< E > {
  private SinglyLinkedList< E > data;
  public LinkedStack() {
    data = new SinglyLinkedList< E >();
  }
  public int size() { return data.size(); }
  public boolean isEmpty() { return size() == 0; }
  public E top() {
    if (isEmpty()) { /* Error: Empty Stack. */ }
    else { return data.getFirst(); } }
  public void push(E e) {
    data.addFirst(e); }
  public E pop() {
    E result;
    if (isEmpty()) { /* Error: Empty Stack */ }
    else { result = top(); data.removeFirst(); }
    return result; }
}
```


Generic Stack: Testing Both Implementations

```
@Test
public void testPolymorphicStacks() {
  Stack<String> s = new ArrayedStack<>();
  s.push("Alan"); /* dynamic binding */
  s.push("Mark"); /* dynamic binding */
  s.push("Tom"); /* dynamic binding */
  assertTrue(s.size() == 3 && !s.isEmpty());
  assertEquals("Tom", s.top());

  s = new LinkedStack<>();
  s.push("Alan"); /* dynamic binding */
  s.push("Mark"); /* dynamic binding */
  s.push("Tom"); /* dynamic binding */
  assertTrue(s.size() == 3 && !s.isEmpty());
  assertEquals("Tom", s.top());
}
```


Beyond this lecture . . .

- Study `https://docs.oracle.com/javase/tutorial/java/generics/index.html` for further details on Java generics.

Index (1)

Index (2)