ADVANCED JAVA

Generics Primer

Vivek Shah bonii@di.ku.dk

August 20, 2018

DIKU, University of Copenhagen

MOTIVATION - DYNAMIC CHECKING

- Consider a *pair* class that can hold a pair of objects:

```
class DynamicCheckedPair {
  Object first;
  Object second;
}
```

MOTIVATION - DYNAMIC CHECKING

- Consider a *pair* class that can hold a pair of objects:

```
class DynamicCheckedPair {
  Object first;
  Object second;
}
```

- Given a pair `p`, we can put *any* value into `first` and `second`:
  `p.first = "hello";`
- We cannot get anything but an `Object` out of the pair.
- We have to *cast*:
  `String x = (String)p.first;`

MOTIVATION - DYNAMIC CHECKING

- Consider a *pair* class that can hold a pair of objects:

```
class DynamicCheckedPair {
  Object first;
  Object second;
}
```

- Given a pair `p`, we can put *any* value into `first` and `second`:
  `p.first = "hello";`
- We cannot get anything but an `Object` out of the pair.
- We have to *cast*:
  `String x = (String)p.first;`
- Casting can cause runtime exception. ☹
  - Compiles and runs fine until this line.

Motivation - static checking

- To get static checking, we cannot use casts. Instead make a class for every pair needed:

```
class PairStringInteger {
  String first;
  Integer second;
}
```

MOTIVATION - STATIC CHECKING

- To get static checking, we cannot use casts. Instead make a class for every pair needed:

```
class PairStringInteger {
  String first;
  Integer second;
}
```

- Like before we can put a string into `first`:
  `p.first = "hello";` ☺
- … but not an integer:
  `p.first = 42;` ☹
- No need to cast:
  `String x = p.first;` ☺

MOTIVATION - STATIC CHECKING

- To get static checking, we cannot use casts. Instead make a class for every pair needed:

```
class PairStringInteger {
  String first;
  Integer second;
}
```

- Like before we can put a string into `first`:
  `p.first = "hello";` ☺
- … but not an integer:
  `p.first = 42;` ☹
- No need to cast:
  `String x = p.first;` ☺
- Many different pairs needed → lots of boilerplate. ☹☹☹

GENERIC CLASSES

- No more boilerplate with *generics*.

```
class Pair<A,B> {
  A first;
  B second;
}
```

GENERIC CLASSES

- No more boilerplate with *generics*.

```
class Pair<A,B> {
  A first;
  B second;
}
```

- Initial extension by Philip Wadler in 1998.
- Introduced in Java 1.5 (2004).
- *Parametric polymorphism* in Java.

GENERIC CLASSES

- No more boilerplate with *generics*.

```
class Pair<A,B> {
  A first;
  B second;
}
```

- Initial extension by Philip Wadler in 1998.
- Introduced in Java 1.5 (2004).
- *Parametric polymorphism* in Java.
- `A` and `B` are *type parameters*.

GENERIC CLASSES

- No more boilerplate with *generics*.

```
class Pair<A,B> {
  A first;
  B second;
}
```

- Initial extension by Philip Wadler in 1998.
- Introduced in Java 1.5 (2004).
- *Parametric polymorphism* in Java.
- `A` and `B` are *type parameters*.
- `Pair` is a type indexed family of classes.
  - `Pair<-,->` : $\mathit{Type} \times \mathit{Type} \rightarrow \mathit{Class}$.

GENERIC CLASSES

- No more boilerplate with *generics*.

```
class Pair<A,B> {
  A first;
  B second;
}
```

- Initial extension by Philip Wadler in 1998.
- Introduced in Java 1.5 (2004).
- *Parametric polymorphism* in Java.
- `A` and `B` are *type parameters*.
- `Pair` is a type indexed family of classes.
  - `Pair<-,->` : $\mathit{Type} \times \mathit{Type} \to \mathit{Class}$.
  - `Pair<String, Double>` is a class.
    - `String` and `Double` are called *type arguments*.

GENERIC CLASSES

- Type parameters can also be used in methods and constructors:

```
class Pair<A,B> {
  private A first;
  private B second;

  Pair(A x, B y) {
    first = x; second = y;
  }

  A getFirst() {
    return first;
  }

  void setFirst(A x) {
    first = x;
  }
}
```

GENERIC CLASSES

- Type parameters can also be used as type arguments for other generic classes.

```
class Triple<A,B,C> {
  private Pair<A,B> firstAndSecond;
  private C third;

  Triple(A x, B y, C z) {
    firstAndSecond = new Pair(x, y); third = z;
  }

  A getFirst() {
    return firstAndSecond.getFirst();
  }

  C getThird() {
    return z;
  }
}
```

- Type parameters can also be used as type arguments for other generic classes.

```
class Triple<A,B,C> {
  private Pair<A,B> firstAndSecond;
  private C third;
```

Question:

Say we want pairs to have a method `swap` that return a new pair with `first` and `second` swapped.

1. What would the method signature be?
2. What would the implementation look like?

```
    return firstAndSecond.getFirst();
  }

  C getThird() {
    return z;
  }
}
```

- Type parameters can also be used as type arguments for other generic classes.

```
class Triple<A,B,C> {
  private Pair<A,B> firstAndSecond;
  private C z;

  public Triple(A a, B b, C c) {
    firstAndSecond = new Pair<A,B>(a,b);
    z = c;
  }

  public A getFirst() {
    return firstAndSecond.first;
  }

  public B getSecond() {
    return firstAndSecond.second;
  }

  public C getThird() {
    return z;
  }
}
```

Answers:

```
public Pair<B, A> swap() {
    Pair<B, A> p = new Pair<B, A>();
    p.first = this.second;
    p.second = this.first;
    return p;
}
```

GENERIC INTERFACES

- It is also possible to declare generic interfaces.

```
interface Mutation<A> {
  void mutate(A x);
}

class LowerCaseName implements Mutation<Employee> {
  void mutate(Employee e) { ... }
}

class IdentityMutation<A> implements Mutation<A> {
    void mutate(A x) { ; }
}
```

GENERIC METHODS

- Individual methods can also be generic:

```
public <A> A printReturn(A x) {
    System.out.println(x);
    return x;
}
```

GENERIC METHODS

- Individual methods can also be generic:

```
public <A> A printReturn(A x) {
    System.out.println(x);
    return x;
}
```

- Here, type parameter `A` is not associated with the class instance. The caller must supply type argument:

```
        obj.<String>printReturn("hello");
        obj.<Integer>printReturn(42);
```

GENERIC METHODS

- Individual methods can also be generic:

```
public <A> A printReturn(A x) {
    System.out.println(x);
    return x;
}
```

- Here, type parameter `A` is not associated with the class instance. The caller must supply type argument:

```
obj.<String>printReturn("hello");
obj.<Integer>printReturn(42);
```

- Type argument can usually be inferred:

```
obj.printReturn("hello");
```

GENERIC CLASSES - INSTANCES

- Constructors take type arguments as well:

```
Pair<String, Double> p1 = new Pair<String, Integer>();
Pair<Double, Double> p2 = new Pair<Double, Double>
                                        (45.3, 0.1);

p1.setFirst("Hello");
Double x = p2.getFirst() + p2.getSecond();
```

Generic classes - instances

- Constructors take type arguments as well:

```
Pair<String, Double> p1 = new Pair<String, Integer>();
Pair<Double, Double> p2 = new Pair<Double, Double>
                                        (45.3, 0.1);

p1.setFirst("Hello");
Double x = p2.getFirst() + p2.getSecond();
```

- Since Java 7, you can use the "diamond operator" <> to infer type arguments:

```
Pair<String, Double> p1 = new Pair<>();
Pair<Double, Double> p2 = new Pair<>(45.3, 0.1);
```

Generics - other examples

GENERICS - OTHER EXAMPLES

- `interface Collection<E>`
  - `boolean add(E e)`
  - `int size()`

GENERICS - OTHER EXAMPLES

- `interface Collection<E>`
  - `boolean add(E e)`
  - `int size()`
- `interface List<E> extends Collection<E>`
  - `E get(int index)`

GENERICS - OTHER EXAMPLES

- `interface Collection<E>`
  - `boolean add(E e)`
  - `int size()`
- `interface List<E> extends Collection<E>`
  - `E get(int index)`
- `Map<K,V>`
  - `V put(K key, V value)`
  - `V get(Object o)`

GENERICS - OTHER EXAMPLES

- `interface Collection<E>`
  - `boolean add(E e)`
  - `int size()`
- `interface List<E> extends Collection<E>`
  - `E get(int index)`
- `Map<K,V>`
  - `V put(K key, V value)`
  - `V get(Object o)`
- `Comparator<T>`
  - `int compare(T o1, T o2)`

GENERICS - OTHER EXAMPLES

- `interface Collection<E>`
  - `boolean add(E e)`
  - `int size()`
- `interface List<E> extends Collection<E>`
  - `E get(int index)`
- `Map<K,V>`
  - `V put(K key, V value)`
  - `V get(Object o)`
- `Comparator<T>`
  - `int compare(T o1, T o2)`
- `Iterator<E>`
  - `E next()`

GENERICS - OTHER EXAMPLES

- `interface Collection<E>`
  - `boolean add(E e)`
  - `int size()`
- `interface List<E> extends Collection<E>`
  - `E get(int index)`
- `Map<K,V>`
  - `V put(K key, V value)`
  - `V get(Object o)`
- `Comparator<T>`
  - `int compare(T o1, T o2)`
- `Iterator<E>`
  - `E next()`
- `Class<T>`
  - `T newInstance()`

Generic classes - instances

- Omitting type arguments gives *raw* types

```
Pair p1 = new Pair();
```

Generic classes - instances

- Omitting type arguments gives *raw* types

```
Pair p1 = new Pair();
```

  - Type parameters are then *erased* to most general type (`Object`).
  - Equivalent to `DynamicCheckedPair`.
  - Not equivalent to `Pair<Object, Object>`.

Generic classes - instances

- Omitting type arguments gives *raw* types

```
Pair p1 = new Pair();
```

  - Type parameters are then *erased* to most general type (`Object`).
  - Equivalent to `DynamicCheckedPair`.
  - Not equivalent to `Pair<Object, Object>`.
    - Subtle difference related to subtyping.
    - This is where generics gets messy.
  - Never use raw types.

Generic classes - runtime representation

- Bytecode does not support parametric polymorphism.
  - Generics retrofitted into Java.

Generic classes - runtime representation

- Bytecode does not support parametric polymorphism.
  - Generics retrofitted into Java.
- Actual implementation of generics is sort of hacky.
- For all type arguments, a generic class is represented as the raw class.
  - `Pair<String, Double>` represented as `Pair`.
  - `Pair<Foo, Bar>` represented as `Pair`.
  - Called *Type erasure*.

Generic classes - runtime representation

- Bytecode does not support parametric polymorphism.
  - Generics retrofitted into Java.
- Actual implementation of generics is sort of hacky.
- For all type arguments, a generic class is represented as the raw class.
  - `Pair<String, Double>` represented as `Pair`.
  - `Pair<Foo, Bar>` represented as `Pair`.
  - Called *Type erasure*.
- Type indexed family of classes is only an illusion.

Generic classes - runtime representation

- Bytecode does not support parametric polymorphism.
  - Generics retrofitted into Java.
- Actual implementation of generics is sort of hacky.
- For all type arguments, a generic class is represented as the raw class.
  - `Pair<String, Double>` represented as `Pair`.
  - `Pair<Foo, Bar>` represented as `Pair`.
  - Called *Type erasure*.
- Type indexed family of classes is only an illusion.
  - ... but it is a type-checked illusion:

```
Pair<Double, Double> p = new Pair<Double, Double>();
p.first = "Helloo"; //Compile-time error, would run.
Double x = p.first; //Insert (Double) cast.
String y = p.first; //Compile-time error.
```

Generic classes - runtime representation

- Bytecode does not support parametric polymorphism.

Question:

1. What would be the result of the following?

```
Pair<String, Integer> p1 = ...;
Pair<Double, Double> p2 = ...;
return p1.getClass()== p2.getClass();
```

2. Is `Pair<A,B>` a complete replacement for `DynamicCheckedPair`?
3. Is `Pair<String, Integer>` a complete replacement for `PairStringInteger`?
4. Is the following legal?

```
Pair<String, Integer> p = ...;
Pair<Object, Object> p2 = p;
```

GENERIC CLASSES - RUNTIME REPRESENTATION

- Bytecode does not support parametric polymorphism.

Answers:

GENERIC CLASSES - RUNTIME REPRESENTATION

- Bytecode does not support parametric polymorphism.

Answers:

1. true, both classes are the raw pair.

GENERIC CLASSES - RUNTIME REPRESENTATION

- Bytecode does not support parametric polymorphism.

Answers:

1. true, both classes are the raw pair.
2. Yes, even though a single `DynamicCheckedPair` instance can be reused with different types, an instance of `Pair` (the raw type) can be used in exactly the same way.

GENERIC CLASSES - RUNTIME REPRESENTATION

- Bytecode does not support parametric polymorphism.

Answers:

1. true, both classes are the raw pair.
2. Yes, even though a single `DynamicCheckedPair` instance can be reused with different types, an instance of `Pair` (the raw type) can be used in exactly the same way.
3. Not quite. Since we only have the raw types at runtime, we cannot do stuff like
   `p instanceof Pair<String, Integer>`, but
   `p instanceof PairStringInteger` is perfectly fine.

Generic classes - runtime representation

- Bytecode does not support parametric polymorphism.

Answers:

1. true, both classes are the raw pair.
2. Yes, even though a single `DynamicCheckedPair` instance can be reused with different types, an instance of `Pair` (the raw type) can be used in exactly the same way.
3. Not quite. Since we only have the raw types at runtime, we cannot do stuff like
   `p instanceof Pair<String, Integer>`, but
   `p instanceof PairStringInteger` is perfectly fine.
4. No, `p2.first = 42;` is legal but that violates the type of `p`.