CSE 1325 - Object-Oriented Programming

Generic Programming

Alex Dillhoff

University of Texas at Arlington

Generic Programming

Generic Programming involves using generic classes and generic methods.

What makes a method or class *generic* is the inclusion of a type parameter.

Generic Programming

A type parameter enables a class or method to work with any generic type.

This allows for general solutions that can adapt to many different environments.

Generic Programming

We have already used generic programming via `ArrayList<>`.

Note that it has a type parameter specified by the arrows `<>`.

Generic Programming

How did Java handle `ArrayList` prior to generic programming?

Example: `ArrayListExample`

Generic Programming

One major problem with the previous example is that it requires the developer to be careful about which data is stored in the `ArrayList`.

Since the internal array takes on any `Object`, additional work must be done to check the types of objects that are retrieved.

Generic Programming

With generic programming, we can create code that is less error-prone.

For example, if we attempt to add an object of a different type to `ArrayList<String>`, the code will simply not compile.

Generic Classes

Most of what constitutes *generic programming* will involve using generic classes and methods in your own programming.

Most of the useful features that take advantage of generic types are already implemented and provided as part of the Java API.

Generic Classes

It is possible that you may need to write own generic class.

In the next example, we'll see how to do exactly that.

Example: `PairTest.java`

Generic Classes

In the previous example, we saw how to create a generic class as well as a generic method.

One problematic implementation is that of the generic method `ArrayAlg.getMiddle(T...)`.

Generic Classes

Java will attempt to resolve all elements to a common class using a process called **autoboxing**.

For example, all primitive types can be **autoboxed** to a corresponding class type (e.g. `int` to `Integer`).

Generic Classes

If the generic types cannot be resolved to a common type, an error occurs.

The solution is to make sure the input elements are all of the same type.

Generic Classes

Writing generic methods and classes can be difficult because your code needs to be ready for ANY possibility.

It is difficult to predict how other developers will use your generic implementations.

Generic Methods

It is also possible to write standalone generic methods.

Consider the task of writing a method that determines if some object exists in a collection, without assuming the types of either the object or the collection.

Generic Methods

Example: `ObjectExistsExample`

Generic Methods

Java's type parameters provide the tools to create robust and useful methods while enforcing relationship and type restrictions.

Generic Methods

As seen in the previous example, a developer cannot mix the types of input without triggering a compiler error.

This is a much better outcome then ending up with a runtime error.

Generic Methods

In the previous example we implemented the `Comparable` interface using the syntax.

```
T extends Comparable<T>
```

We can ensure that classes extend to multiple interfaces by chaining them with `&`.

```
T extends Comparable<T> & Serializable
```

Generics and JVM

The Java VM does not have access to generic types, only ordinary classes.

If a generic type is defined, the VM provides a corresponding **raw** type.

Generics and JVM

The VM replaces the generic type with whatever type they are *bounded* by.

This *bounded* type is the first interface that the generic adheres to.

If there is no bounded type, the `Object` type is used.

Generics and JVM

The act of replacing a generic type with an ordinary one is called **type erasure**.

Consider the `Pair` example from earlier. There is no bounded type, so the raw type is `Object`.

Generics and JVM

The VM would replace the generic implementation with the following:

```
public class Pair {
    private Object first;
    private Object second;
    ...
}
```

Translating Back

Type erasure is, of course, reversible. Consider the following code:

```
Pair<User> users = ...;
User u = users.getFirst();
```

Translating Back

With **type erasure**, the type of `getFirst()` is `Object`.

The compiler will translate the `Object` to `User` by:

1. Calling the raw method `public Object getFirst()`.
2. Casting the returned `Object` to `User`.

Translating Bounded Types

What about **bounded types**?

The raw type of the object will be that of the first bound (e.g. `Comparable` in our previous example).

For types with multiple bounds, it is the first type used.

Type Erasure and Subclasses

CASE STUDY: Let's review an example from "Core Java Volume I – Fundamentals (11th Ed.)" by Cay S. Horstmann.

Type Erasure and Subclasses

Consider the following subclass:

```
class DateInterval extends Pair<LocalDate> {
  public void setSecond(LocalDate second) {
    if (second.compareTo(getFirst()) >= 0) {
      super.setSecond(second);
    }
  }
}
```

Type Erasure and Subclasses

After type erasure, this turns into:

```
class DateInterval extends Pair {
      public void setSecond(LocalDate second) {
    ...
  }
}
```

Type Erasure and Subclasses

`DateInterval` is a subclass that overrides `setSecond`.

There is another `setSecond` method from the parent class.

Type Erasure and Subclasses

After type erasure of the original method, the signature is:

```
public void setSecond(Object second);
```

The two preceding methods look like they would be separate in the VM, but they shouldn't be.

Type Erasure and Subclasses

Consider the following code.

```
var interval = new DateInterval(...);
// assignment to superclass
Pair<LocalDate> pair = interval;
pair.setSecond(someDate);
```

Which method do we expect to be called?

Type Erasure and Subclasses

We may expect `DateInterval.setSecond()` to be called since the polymorphic object is originally of type `DateInterval`.

However, type erasure interferes with polymorphism!

Type Erasure and Subclasses

To remedy this, the compiler will generate a **bridge method**.

In this case, it generates

```
public void setSecond(Object second) {
        setSecond((LocalDate) second);
}
```

Type Erasure and Subclasses

How does this bridge method resolve the issue?

The object `pair` is of type `Pair<LocalDate>`, which has a single method called `setSecond`.

The VM calls the method that uses a `DateInterval` input, so the bridge method is called.

Type Erasure and Subclasses

Summary of Translating Generics:

1. There are no generics in the VM.
2. Type parameters are replaces by their *first* bound.
3. The compiler uses bridge methods to preserve polymorphism.
4. Type casting is inserted to preserve type safety.

Generic Type Compatibility

Assume, for example, we have a `User` class and a subclass `Admin`.

Is `Pair<Admin>` also a subclass of `Pair<User>`?

Generic Type Compatibility

Assume, for example, we have a `User` class and a subclass `Admin`.

Is `Pair<Admin>` also a subclass of `Pair<User>`?

NO

Generic Type Compatibility

If this were the case, it would introduce new type safety issues.

We could use polymorphism to assign a regular `User` to one of the objects in `Pair<Admin>`, leading to unintended consequences.

Wildcard Types

Wildcard types allows the generic type parameter itself to vary.

```
Pair<? extends User>
```

The `?` above is a wildcard type. The code above puts a further restriction on the `Pair` class.

Wildcard Types

```
Pair<? extends User>
```

Instead of supporting any type, the `Pair` class will now only support `User` or its subclasses.

Wildcard Types

The types in generic methods are also updated using the wildcard:

```
? extends User getFirst();
void setFirst(? extends User u) { ... }
```

Wildcard Types

This wildcard makes it impossible to call `setFirst`... WHY?

Wildcard Types

This wildcard makes it impossible to call `setFirst`... WHY?

The compiler knows that the wildcard type needs to be a subset of `User`, but does not know the specific type.

It is safer to not make an assumption about the type to return and instead report an error.

Wildcard Types

A call to `getFirst()` is fine since the return value can be safely cast to `User`.

Wildcard Types

We can reverse the above wildcard behavior...

```
void setFirst(? super User u) { ... }
? super User getFirst();
```

Wildcard Types

```
void setFirst(? super User u) { ... }
```

For the set method, we can safely pass a `User` or `Admin` object.

However, it could not safely pass a more general `Object` object.

Wildcard Types

```
? super User getFirst();
```

The getter can only safely return an `Object` object.

The wildcard cannot guarantee type safety for any other type.