CSSE2002/7023

Programming in the large

Week 7.1: Intro to Java Generics

In this hour

- Creating generic types
- Using generic types
- Generic methods
- Generics and inheritance
- Type erasure
- Restrictions

We follow the content in the Oracle tutorial closely, see here for more info

`https://docs.oracle.com/javase/tutorial/java/generics/index.html`

A non-generic class

Consider this example class:

```
public class Holder5 {
 private Object [] theHeldVariables;

 public Holder5() {
  theHeldVariables = new Object[5];
 }

 public void setHeldVariable(int i, Object newHeldVariable) {
  theHeldVariables[i] = newHeldVariable;
 }
 public Object getHeldVariable(int i) {
  return theHeldVariables[i];
 }
}
```

If we use this class:

```
Holder5 holder5 = new Holder5(); // new holder for Integers
holder5.setHeldVariable(2, new Integer(5));
Integer two = (Integer) holder5.getHeldVariable(2);
```

For different types:

```
Holder5 holder5 = new Holder5(); // new holder for Floats
holder5.setHeldVariable(2, new Float(10.0));
Float two = (Float) holder5.getHeldVariable(2);

Holder5 holder5 = new Holder5(); // new holder for Strings
holder5.setHeldVariable(2, new String("hello"));
String two = (String) holder5.getHeldVariable(2);
```

I accidentally added an Integer to my strings ... oops.

```
Holder5 holder5 = new Holder5(); // new holder for Strings
holder5.setHeldVariable(2, new String("hello"));
holder5.setHeldVariable(3, new Integer(7));
String two = (String) holder5.getHeldVariable(2);
String three = (String) holder5.getHeldVariable(3); // exception
```

What are generic types?

- A class or interface that uses other classes as parameters at *compile time*.

```
public class X<T> {
 private T myFirstVariable;

 public T getMyFirstVariable() {
  return myFirstVariable;
 }
}
```

A generic class

```
public class Holder5<T> {
 private T [] theHeldVariables;

 public Holder5() {
  theHeldVariables = new T[5];
 }

 public void setHeldVariable(int i, T newHeldVariable) {
  theHeldVariables[i] = newHeldVariable;
 }
 public T getHeldVariable(int i) {
  return theHeldVariables[i];
 }
}
```

If we use this class:

```
// new holder for Integers
Holder5<Integer> holder5 = new Holder5<>();
holder5.setHeldVariable(2, new Integer(5));
Integer two = holder5.getHeldVariable(2);
```

Why use generic types?

They enable programmers to write algorithms that work accross different types with:

- Type checking at compile time to prevent runtime exceptions.
- Casting no longer required for conversion.

Generics example

Generics.java
GenericsExample.java

Generics

- Defined using the following format:

```
class Name <T1, T2, ..., TN> {
    /* contents */
}
```

- Convention is types are designated by a single letter (e.g., T)
  Java libraries use:
  - E - Element (used extensively by the Java Collections Framework)
  - K - Key
  - N - Number
  - T - Type
  - V - Value
  - S,U,V etc. - 2nd, 3rd, 4th types

  See

  `https://docs.oracle.com/javase/tutorial/java/generics/types.html`

Using generics

- As per Java collections:

```
List<String> myList = new ArrayList<String>();
```

or

```
List<String> myList = new ArrayList<>();
```

- Custom classes:

```
Holder5<String> holder5 = new Holder5<>();
```

Generic methods

- Generics can be applied at a method level:

```
public class Counter {
public static <T>
int count(T [] array, T value) {
    int count = 0;
    for (T elem : array) {
        if (elem.equals(value)) {
            count++;
        }
    }
    return count;
}
}
```

- Call with:

```
Integer [] array = {1, 2, 3, 4, 4, 5, 6, 6};
Counter.<Integer>count(array, 4);
```

Bounded type parameters

- Sometimes we want to place certain restrictions on generic parameters.
- We can used *bounded* type parameters.

```
public class NumberHolder <T extends Number> {
 private T number;

 public NumberHolder(T number) { this.number = number; }

 public T getNumber() { return number; }

 public double getAsDouble() {
  return number.doubleValue();
 }
}

NumberHolder<Integer> h = new NumberHolder<>(5);
System.out.println("Relation to 4: " + h.getNumber().compareTo(4));
System.out.println("Double value: " + h.getAsDouble());
```

Generics and inheritance

- There are multiple ways of inheriting from a generic class:

Extend by passing type parameters:

```
class X <T> extends class Y <T> {

}
```

Extend by passing concrete type:

```
class Z <T> extends class Y <String> {

}
```

`GenericsInheritanceExample.java`

Generics and inheritance

Using a subclass as a generic parameter does not imply any relationship between the generic classes.

i.e.

```
class B extends A { ... }
```

does not imply e.g., that

```
ArrayList<B> extends ArrayList<A>
```

The following will not work:

```
// will not compile "incompatible types"
ArrayList<A> listOfA = new ArrayList<B>();
```

Type erasure

The Java compiler handles generics at compile time by:

- Replacing generic types with `Object`.
- Replacing bounded generic types with the bound.
- Adding casts where reqquired.
- Adding existing bridging methods when required.

Type erasure

The Java compiler handles generics at compile time by:

- Replacing generic types with `Object`.
- Replacing bounded generic types with the bound.
- Adding casts where reqquired.
- Adding existing bridging methods when required.

```
public class Holder <T> {
  private T variable;
  public T getVariable() { return variable; }
}
Holder<String> holder = new Holder<>();
String string = holder.getVariable();
```

becomes:

```
public class Holder {
  private Object variable;
  public Object getVariable() { return variable; }
}
Holder holder = new Holder();
String string = (String) holder.getVariable();
```

Restrictions on generics

- No primitive types
- No instantiating generic type parameters

  e.g., `T elem = new T(); // compile error`

  static variable means...
- No static fields can be of a type parameter type

  e.g., `public static T myVariable; // compile error`
- No arrays of parameterised types

  e.g.,

  //compile error

```
ArrayList<String> [] array = new ArrayList<>[]();
```

- No generic exceptions
- Restrictions on overloaded functions

  e.g.,

```
 class X { public int method1(List<String> list);
public int method1(List<Float> list); }
```