CSSE2002/7023

Programming in the Large

Week 10.1: Generics

In this Session[1]

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

[1] https://docs.oracle.com/javase/tutorial/java/generics/index.html

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 floats5 = new Holder5(); // new holder for Floats
floats5.setHeldVariable(2, new Float(10.0));
Float two = (Float) floats5.getHeldVariable(2);
```

```
Holder5 string5 = new Holder5(); // new holder for Strings
string5.setHeldVariable(2, new String("hello"));
String two = (String) string5.getHeldVariable(2);
```

Accidentally adding an Integer to my Strings ... oops.

```
string5.setHeldVariable(3, new Integer(7));
String two = (String) string5.getHeldVariable(2);
String three = (String) string5.getHeldVariable(3);
                                        // exception
```

What are Generic Types?

Class or interface that uses other classes or interfaces as parameters at *compile time*.

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

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

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

Generic Class

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

  public Holder5() {
    theHeldVariables = (T[]) new Object[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);
```

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<>();
  ```

Why Use Generic Types?

Allows 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

SimpleGenerics.java
GenericsExample.java

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 item : array) {
            if (item.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 restrict the possible types of generic parameters.
- We can use *bounded* type parameters:
  `public class NumberHolder<T extends Number> {...}`
- We know that any objects of type `T` in our code will be able to access any methods of `Number` — for example, we could now add the following code to the class `NumberHolder`:

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

because we know that `number` has access to `Number.doubleValue()`.

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> {

}
```

See: `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>();
```

Wildcards (?)

Wildcards in Java (indicated by a ?) represent an unknown type. They are useful when generic types are required, but do not necessarily need to be named, for example:

```
public void printList(List<?> toPrint) {
    for (Object item : toPrint) {
        System.out.println(item);
    }
}
```

Why not just take a `List<Object>` as the parameter?

- That would *only* take a `List<Object>` as an argument (remember how inheritance works from the previous slide)
- `List<?>` allows *any type of* `List` as an argument (e.g. `List<String>`, `List<Integer>`, ...)

Bounded Wildcards

`?`		– allows any type
`?`	`extends Type`	– allows `Type` or any subclass of `Type`
`?`	`super Type`	– allows `Type` or any superclass of `Type`

More on bounded wildcards in the prac on generics.

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 required.
- Adding existing bridging methods when required.

Type Erasure Example

```
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 item = new T(); // compile error`
- 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`
  `Pair<String, Integer>[] array = new Pair<>[5];`
- No generic exceptions
- Restrictions on overloaded functions
  e.g.
  ```
  public int method1(List<String> list);
  public int method1(List<Float> list);
  ```

See

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