CSD-4464 Java EE

Class 3: Generics

Lambton College

Generics – keeping duplicate code minimal

- Java **Generic** methods and classes enable programmers to create a very general (generic) solution and avoid writing the same logic multiple times
- Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.
- Example – imagine writing a single sort method that could sort the elements in an Integer array, a String array, or an array of any type

Note* generics don't have to be the letter(s) T, you can use any letter you like

Generics

- Generics are declared with angle brackets (< Type Parameter >)
- Each type parameter section contains one or more type parameters separated by commas.
- A type parameter, also known as a type variable
- Bounded vs Unbounded types
- You can have both generic classes and generic methods
- Example – SomeClass<T> { // T is a type declaration which which you can pass in types to, like SomeClass <String> (reads Optional String)

Generic Declarations

- Generic Classes have the Generic Types declared at the top of the class

Example public class SomeClass <T> { //tells the compiler that a variable type T will be used in the code

- Generic Methods have the Generic Types declared before the Return Type declaration

Example – public <T> T myMethod(T param1) //Method reads “I have a variable type T, and my return is that same Type and my argument is that same Type”

Generic Declarations - Continued

Lets take a look at generic method that takes in two generic parameters

```
Public <T, U> String makeString(T p1, U p2) {
   return p1.toString() + p2.toString()
}
```

If we were to call it with Integer i, String s

```
String newString = makeString(i,s);
```

The compiler will see

```
Public String makeString(Integer p1, String p2) {
```

Bounded Types

- There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter, and this can be done with bounded types
- Bounded types are generic types that extends a parent class
- Example – a method that sorts an array may want to limit the types to types that extend Comparable, or a method that performs math operations may want types that extend the Number class

  e.x Optional<T extends Number>

Wildcard Types

- Declared with a '?' in the type parameter, does not need to be predeclared
- Signifies that we don't know what the type will be, but we don't care
- Compiler is smart enough to figure everything out at runtime, However your code has to be smart enough to not mix types
- Example List<?> myList // List with an Unknown type
  myList.size() //doesn't mix types, legal code

  myList.add(1) //mixing type captures, illegal code

Wildcard types - continued

- If you do need to perform typed operations (example from before myList.add(1)) you can cast the wildcard into a type with ((Type)variableName).methodCall()

Example

((*List*<Integer>)myList).add(1);

- Guard yourself by checking types with instanceof before casting – you can't convert incompatible types with casting

Example

If (myList instanceOf List<Intenger>)

((*List*<Integer>)myList).add("bob"); //illegal