The ready(-to-)wear collection is ready to wear for anyone.

This file has quick references and explanations.

For notes taken from lecture, try out our autumn-winter 2022 haute couture collection.

Generics

Generics are any interface, class, or method whose type is determined by a parameter.

Below is example code for a generic class `Stack` and its usage in a main method

```
// Generics.java
import java.util.ArrayList;

class Stack <E> {
    ArrayList<E> stack;
    public Stack() {
        stack = new ArrayList<E>();
    }
    public void push (E element) {
        stack.add(0, element);
    }
    public E pop() {
        return stack.remove(0);
    }
    public boolean isEmpty() {
        return (stack.size() == 0);
    }
    public E top() {
        return stack.get(0);
    }
} // end of Stack


public class Generics {
    public static void main(String[] args) {
        Stack<Integer> iStack = new Stack<Integer>();
        Stack<String> sStack = new Stack<String>();
    } // end of main
}
```

Why Make Generics?

Generics, like everything in oop, are made to reduce writing of duplicate code.

We've used clever workarounds involving type casting and pointing objects to variables before. These workarounds come at the cost of:

- readability
- debuggability
- flexibility
- robustness

The aforementioned workarounds still have their place however we're able to gain the previously stated to be lost things when using generics.

Generic Creation & Usage

When creating a generic type we supply a type parameter which is a single uppercase letter.

By convention, the name of these type parameters depend on their use.

name	use
E	element
K	key for map
V	value for map
N	number
T	"Type, for "1st" generic type parameter"
S	""2nd""
U	""3rd""
V	""4th""

Generic Classes and Instance Variables

To create a generic class, enclose a single uppercase letter within angled brackets after the class' name.

```
class Stack <E> {
...
```

If there are multiple generic types to be used then they should be separated by commas and spaces within the angled brackets

```
class Stack <T, S, U, V> {
...
```

Generic instance variables will simply replace where the original data type once was with the generic type supplied.

```
int[] array;
// turns into
E[] array;
```

When instantiating a generic class the client will put the data type that they wish to use within the angled brackets.

```
Stack<Integer> iStack = new Stack<Integer>(); // using Integer
Stack<String> sStack = new Stack<String>();   // using String
```

Generic methods

Like classes, generic methods can also have multiple parameters.

There are different types of generic methods:

- generic methods inside of generic classes that:
  - depend on the generic class' generic type (1)
  - don't depend on the generic class' generic type (2)
  - use both it's own generic types and the class' generic type (3)
- generic methods that use their own generic type inside a normal class (4)

Making for a total of 4 different types of generic methods you'll encounter.

Here are some examples:

```
class aGenericClass <E> {

  // (1) a generic method that depends on the generic class' generic type
  public void printClass(E element) {
    System.out.println(element.getClass());
  }
  // it only uses E
```

```
  // (2) a generic method that doesn't depend on the generic class' generic type
  public T getMiddle(T[] array) {
    return array[array.length/2];
  }
  // it doesn't use E and instead uses T
}

class genericClassWithMultipleParams <K, V> {
  K key;
  V value;

  // (3) a generic method that uses both the generic class' type(s) and its own
type(s)
  public <S, T> void ParamterizedMethod (K k, V v, S s, T t) {
    S var1;
    T var2;
    K var3;
    V var4;
  }
  // K and V can be said to belong to the class,
  // their scope being the whole class,
  // while S & T can be said to belong to the method only,
  // their scope being the whole method
  // not the class
}


class nonGenericClass {
  // (4) generic methods that use their own generic type inside a normal class
  public E genericMethod(E e1, E e2){
    return e1.compareTo(e2);
  }
}
```

When using a generic method one calls them as normal. The generic method will be fed an object as an argument, get the type of the fed object, and replace their type with that type.

Generic Methods, Autoboxing, and Wrapper Classes

Generics types can only be replaced by primitive types but when we create and pass primitives we usually don't create an object of them.

This is handled by a process called autoboxing where Java will just create an object for us automatically.

It does this using wrapper classes which are methods belonging to the primitive type's class.

type	wrapper class
boolean	Boolean.valueOf()
byte	Byte.valueOf()
char	Character.valueOf()
...	...
and so on for all the primitives	

Wrapper classes are actually just static factory methods, they just get a special name because they're for primitives.

Generic Inheritance

```
// Storing a String inside an Object variable
String name = "John";
Object object = name;
```

The above code is fine as you can sotre a subtype in a variable of supertype.

```
//Storing a String[] inside an Object[] variable
String[] names = {"John", "Jane"};
Object[] objects = names;
```

The above code works fine because arrays are said to be `covariant`. We can store an array of a subtype inside of an array of supertype variable, in this case we stored an array of type `String` inside an array of type `Object` variable.

However we cannot do the same thing with a generic like `ArrayList<>`.

```
List<Object> objectList = stringList;
String s = objectList.get(0);
```

The above code does not work. `String` is a subclass of `Object` but `ArrayList<String>` is not a subclass of `ArrayList<Object>`.

In this sense there is not an inheritance relation.

Arrays of Generics

When handling of generics one should be careful to note the nature of generics as a data type in memory.

At compile time, Java doesn't know how much space to reserve for a generic object as it's up in the air what it will become until runtime. Java will then throw a compiler error.

```
class genericArrayMaker <E> {
  E[] array;  // this is just creating a variable so we're fine
  final int ARRAY_SIZE = 10;
  public defaultArray() {
    array = (E[]) new Object[ARRAY_SIZE];
  }
  public copyArray(E[] input) {
    array = (E[]) new Object[input.length];
    for (int i = 0; i < input.length; i++)
      array[i] = input[i];
  }
}
```

We get around this by making an array of `Object` and casting it to be our generic type after the fact so that java knows how much memory to reserve when compiling.