Object Oriented Programming with Java

Generics

Generics

- A facility of generic programming. Added in 2004 (version 5)
- Generics is trying to solve two different problems...

Problem 1: inefficient overloading

We need a function that can calculate the average of a vector of any type

```
public double average(Integer[] vector) {
	double avg=0.0;
	for (Integer v : vector)
		avg += v
	return avg/vector.length;
}
```

```
public double average(Float[] vector) {
	double avg=0.0;
	for (Float v : vector)
		avg += v
	return avg/vector.length;
}
```

```
public double average(Double[] vector) {
	double avg=0.0;
	for (Double v : vector)
		avg += v
	return avg/vector.length;
}
```

```
public double average(Byte[] vector) {
	double avg=0.0;
	for (Byte v : vector)
		avg += v
	return avg/vector.length;
}
```

Problem 1: inefficient overloading

The only variation among these overloaded methods is the **type** of the vector

```
public double average(Integer[] vector) {
        double avg=0.0;
        for (Integer v : vector)
                avg += v
        return avg/vector.length;
}
```

```
public double average(Float[] vector) {
        double avg=0.0;
        for (Float v : vector)
                avg += v
        return avg/vector.length;
}
```

```
public double average(Double[] vector) {
        double avg=0.0;
        for (Double v : vector)
                avg += v
        return avg/vector.length;
}
```

```
public double average(Byte[] vector) {
        double avg=0.0;
        for (Byte v : vector)
                avg += v
        return avg/vector.length;
}
```

Problem 1: inefficient overloading

Ideally, we would like to create just **one** method that can dynamically accept any type **T**

```
public double average(T[] vector) {
	double avg=0.0;
	for (T v : vector)
		avg += v
	return avg/vector.length;
}
```

don’t try this code, it won’t compile as is

```
public double average(Float[] vector) {
	double avg=0.0;
	for (Float v : vector)
		avg += v
	return avg/vector.length;
}
```

```
public double average(Double[] vector) {
	double avg=0.0;
	for (Double v : vector)
		avg += v
	return avg/vector.length;
}
```

```
public double average(Byte[] vector) {
	double avg=0.0;
	for (Byte v : vector)
		avg += v
	return avg/vector.length;
}
```

Problem 2: "lost" types

Consider the following code:

```
ArrayList alist = new ArrayList();
Person gmu = new Person("Mason", 89);
alist.add(gmu);
```

And then we can't directly get a Person back out like this:

```
Person p = alist.get(0);
```

→ compile-time error: found Object, needed Person

Instead, we must **downcast** everything that comes out of the ArrayList:

```
Person p = (Person) alist.get(0);
```

Problem 2: "lost" types

This issue would always arise when we retrieve things from the ArrayList

It's even more annoying with the for-each loop:

```
ArrayList personList = new ArrayList();
// add many Person objects

//NOT ALLOWED:
for (Person p : personList)
        p.whatever();
```

Instead, we must use **downcasting** after we retrieve with **Object type**, like this:

```
// allowed, but annoying, harder to read, and error-prone
for (Object p : personList)
{
        ((Person) p).whatever();
}
```

Generics: Establish & Remember Types

- Generics allow us to define **type parameters** – we can parameterize blocks of code with types!

- Where can we add type parameters?
  - **at class declarations** → available for entire class definition
  - **at method signatures** → available throughout just this method

- instead of **just** having the regular parameter list where we supply values, we can **also** give a listing of type parameters, which can then show up as the types of our formal parameters

- Think of it as an extra level of overloading but more powerful and dynamic

Declaring Generic Classes

We can add a generic type to a class definition:

```
public class Foo <T>
{
	// T can be anywhere: like field types.
	public T someField;

	public Foo(T t)// T used as parameter type
	{
		this.someField = t;
	}

	// T used as return type and param type
	public T doStuff(T t, int x) { … }
}
```

Simply add the ***<T>*** or ***<E>*** or whatever name you like right after the class name, and then use ***T*** instead of a specific type in your code

Declaring Generic Classes with more types

```
public class Pair <R,S>
{
  public R v1;
  public S v2;

  public Pair(R r, S s)
  {
    v1 = r;
    v2 = s;
  }

  public String toString()
  {
    return ("("+v1+","+v2+")");
  }
}
```

Declaring Generic Methods

- In a generic method the **<T>** notation must go **before the return type**
- It may then be used as a type anywhere in the method: parameter types, local definitions' types... even the return type!
- All we know about `u1` or `u2` is that it is a value of the **U** type. That's not much info! But we can still write useful, highly re-usable code this way

```
public <T> void echo (T t1, T t2)
{
    System.out.print(t1 + t2);
}

public <U> U choose (U u1, U u2, boolean b)
{
    return (b ? u1 : u2);
}
```

Declaring both Generic Class and Generic Method

We can declare new generic types that are only visible with one method, like **<U>**, and have a different generic type for the class, like **<T>**

```
public class Foo <T>
{
    …
    public <U> void choose (U u1, U u2, boolean b, T var)
    {
        return (b ? u1 : u2);
    }
}
```

Creating Generic Objects

```
public class Pair <R,S> {
    public R v1;
    public S v2;

    public Pair(R r, S s) {
        v1 = r;
        v2 = s;
    }

    public String toString() {
        return ("("+v1+","+v2+")");
    }
}

public class RunMe {
    public static void main (String[] args) {
        Pair<Integer, Double> a = new Pair<Integer, Double>(1, 2.0);
        Pair<Integer, Double> b = new Pair<>(3, 4.0);   // since Java 7
    }
}
```

Calling Generic Methods

Given a generic method (which happens to be static in this case):

```
public class Foo {
    public static <U> U choose (U u1, U u2, boolean b) {
        return (b ? u1 : u2);
    }
}
```

We instantiate the parameters and can call it like this:

```
    String s = Foo.<String>choose("yes","no",true);
    String t = Foo.choose("yes","no",true);
```

If it were non-static, we'd need an object to call it:

```
    Foo f = new Foo();
    String s = f.<String>choose("yes","no",true);
    String t = f.choose("yes","no",true);
```

Since Java 7 we only need to specify the generic type when it's not clear from the parameters

Generics with Java Collections

- You will see many collections that use Java Generics, usually to represent a grouping of elements of some contained type.
  → e.g., a list of something, a set of something, a map of something

- Instantiate the given type parameter(s) to what you want contained by that collection. Example:

```
ArrayList<Integer> xs = new ArrayList<>();
```

- Then you can use the methods of the collection, and Java will know that this `ArrayList` holds only Integer values, and not just any old Object values, **without you needing to cast everywhere**.

Example: Using ArrayList Generically

Let's look at how we actually get to use generics with ArrayList:
→ we need to **instantiate** the class's type parameter:

```
//instantiate the type parameter with <>'s:
ArrayList<String> slist = new ArrayList<String>(); // before Java 7
ArrayList<String> slist = new ArrayList<>();       // since Java 7

//now use all methods without having to specify again
slist.add("hello");
slist.add("goodbye");
String elt = slist.get(0);
System.out.println("some element: " + elt);
System.out.println("the list: " + slist);
```

Wildcard

When we need to declare a generic type parameter but we're not going to use it anywhere in the code (but it still needs to be present for syntactic reasons), we can use the wildcard **?** to indicate this don't-care situation. Examples:

```
public static int countItems(ArrayList<?> c)
{
    return c.size();
}
```

```
public void processElements(ArrayList<?> elements)
{
    for(Object o : elements)
        System.out.println(o);
}
```

Practice Problems

Write a non-generic class that:

1. Has a generic method that returns a subarray of the first three items of a generic array

```
public <U> U[] subArray(U[] arr)
```

2. Has a generic method that returns the max value of a generic array

```
public <T> T maxValue(T[] arr)
```

Issues with Generics

```
public class IssuesWithGenerics
{
  public <U> U[] subArray(U[] arr)
  {
    U[] subArray = new U[3];   // ERROR
    for(int i=0; i<subArray.length; i++)
        subArray[i] = arr[i];
    return subArray;
  }

  public <T> T maxValue(T[] arr)
  {
    T max = arr[0];
    for (int i = 1; i < arr.length; i++)
    {
      if(arr[i] > max)    // ERROR
      {
        max = arr[i];
      }
    }
    return max;
  }
}
```

Solving the Issues with Generics – instantiation

```java
public <U> U[] subArray(U[] arr)
{
  U[] subarray = new U[3];  // this instantiation is not allowed
  ...
}
```

```java
@SuppressWarnings("unchecked")  // to avoid compiler warnings
public <U> U[] subArray(U[] arr)
{
  U[] subarray = (U[]) new Object[3];    // downcasting
  // note that the cast type is U[] not just U
  ...
}
```

Solving the Issues with Generics – comparison

```
public <T> T maxValue(T[] arr){
  T max = arr[0];
  for (int i = 1; i < arr.length; i++){
    if(arr[i] > max){   // this comparison is not allowed
      max = arr[i];
    }
  }
  return max;
}
```

```
public <T extends Comparable<T>> T maxval(T[] arr){
  T max = arr[0];
  for (int i = 1; i < arr.length; i++){
    if(arr[i].compareTo(max)>0){
      max = arr[i];
    }
  }
  return max;
}
```

Bounded Type Parameters – Upper bound

- Sometimes you want to restrict the types that can be used as type arguments
- For example, a method that operates on numbers might only want to accept instances of `Number` or its subclasses
- To declare an **upper bound** we use the **`extends`** keyword: `<T extends SomeType>`
- It means that T is a sub-class of SomeType or implements the SomeType interface
- We can even add multiple extensions: `<T extends A & B & C>`
- The same way that classes/interfaces gave us subtypes, we're now saying "any class that's a subtype of SomeType". But we can make multiple claims at once this way
- In multiple extensions, if one of the bounds is a class, it must be specified first (i.e. before the interfaces)

Bounded Type Parameters – Lower bound

- We can also use generics with a lower bound, indicating that it's acceptable for a type parameter to be any type that is a supertype of something particular. Example:

```
<? super PickupTruck>
```

  In this case, we can use any type that can accept `PickupTruck` values like `PickupTruck`, `Truck`, and `Vehicle`

- Look for instance at `Collections.sort`

```
public static <T> void sort(List<T> list, Comparator<? super T> c)
```

  We could have limited ourselves to `Comparator<T>` that would only allow `Comparator<PickupTruck>` values to sort a list of `PickupTruck` objects. But if we also had a `Comparator<Truck>`, or a `Comparator<Vehicle>` it would make perfect sense to use them as another way to sort trucks or vehicles, which certainly includes PickupTrucks.

  By using the **super** keyword we can set a **lower bound** and accept all these different comparators when sorting a list of PickupTrucks.

Upper and Lower bounds combined

- We can also mix upper and lower bounds in the same declaration. Example:

```
public static <T extends Comparable<? super T>> sort(List<T> list)
```

It means that the items in the list must be descendants of `Comparable` (otherwise it's impossible to compare them and then sort them), but the implementation of the Comparable interface (i.e. the implementation of the `compareTo` method) doesn't necessarily have to come from T directly, it can be inherited from its ancestor(s).

- How did we come up with the above declaration. Three attempts:

```
public static <T> sort(List<T> list) // error
```

```
public static <T extends Comparable<T>> sort(List<T> list) // ok but limited
```

```
public static <T extends Comparable<? super T>> sort(List<T> list) // best
```

Revisiting *is-a* relationship with Generics

```
public void someMethod(Number n) { /* ... */ }
someMethod(new Integer(10));
someMethod(new Double(10.1));
```

is this valid?

```
ArrayList<Number> box = new ArrayList<>();
box.add(new Integer(10));
box.add(new Double(10.1));
```

is this valid?

```
public void boxTest(Box<Number> n) { /* ... */ }
boxTest(new Box<Integer>());
```

is this valid?

`Box<Integer>` is not a subtype of `Box<Number>`. Given two concrete types **A** and **B** (e.g, **`Number`** and **`Integer`**), `MyClass<A>` has no relationship to `MyClass<B>`, regardless of whether or not A and B are related.

Nested Generic Types

Let's create 2D array of Integers with ArrayList (3 rows, variable-size columns)

```
public class ArrayListMatrix {
    public static void main(String[] args) {
        ArrayList<ArrayList<Integer> > matrix = new ArrayList<ArrayList<Integer> >(3);

                              ?

        for (int i = 0; i < matrix.size(); i++) {
            for (int j = 0; j < matrix.get(i).size(); j++) {
                System.out.print(matrix.get(i).get(j) + " ");
            }
            System.out.println();
        }
    }
}
```

Nested Generic Types

Let's create 2D array of Integers with ArrayList (3 rows, variable-size columns)

```
public class ArrayListMatrix {
    public static void main(String[] args) {
        ArrayList<ArrayList<Integer> > matrix = new ArrayList<ArrayList<Integer> >(3);

        ArrayList<Integer> row1 = new ArrayList<Integer>();
        row1.add(1);
        row1.add(2);
        matrix.add(row1);

        ArrayList<Integer> row2 = new ArrayList<Integer>();
        row2.add(5);
        matrix.add(row2);

        ArrayList<Integer> row3 = new ArrayList<Integer>();
        row3.add(10);
        row3.add(20);
        row3.add(30);
        matrix.add(row3);

        for (int i = 0; i < matrix.size(); i++) {
            for (int j = 0; j < matrix.get(i).size(); j++) {
                System.out.print(matrix.get(i).get(j) + " ");
            }
            System.out.println();
        }
    }
}
```

Diamond operator

```java
// Before Java 7: the generic type must be on
// both sides of the expression
List<String> students = new ArrayList<String>();
```

```java
// From Java 7: the generic type can be
// inferred on the right side of expression
List<String> students = new ArrayList<>();
```

Before Java 9, the diamond operator won’t work with inner anonymous classes. Compiler will give the following error for this code on the right

`error: cannot infer type arguments for Sum`

```java
abstract class Sum<T> {
    abstract T add(T num1, T num2);
}

public class CS112 {
    public static void main(String[] args)
    {
        Sum<Integer> obj = new Sum<>() {
            Integer add(Integer n1, Integer n2)
            {
                return (n1 + n2);
            }
        };
        Integer result = obj.add(10, 20);
        System.out.println("sum is " + result);
    }
}
```

Type Erasure

To implement generics, the Java compiler applies type erasure to:

- Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
- Insert type casts if necessary to preserve type safety.
- Generate bridge methods to preserve polymorphism in extended generic types.

Restrictions with Generics

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

- Cannot instantiate generic types with primitive types
- Cannot create instances of type parameters
- Cannot declare static fields whose types are type parameters
- Cannot use casts or instanceof with parameterized types
- Cannot create arrays of parameterized types
- Cannot create, catch, or throw objects of parameterized types
- Cannot define overloaded methods that will have the same signature after type erasure