Lecture - Generics

Introduction to generics and generic classes

Beginning with version 5.0, Java allows class and method definitions that include *parameters for types*.

Such definitions are called *generics*.

Generic programming with a type parameter enables code to be written that applies to any class.

Consider two classes:

We want to write generic code that can cater to multiple things together.

```
class SampleInteger {
    private Integer data;

    public void setData (Integer newData) {
        data = newData;
    }

    public Integer getData () {
        return data;
    }
}
```

We can write generic code for similar things, but not for everything
→ because some two different item may have different data type.

```
class SampleDouble {
    private Double data;

    public void setData (Double newData) {
        data = newData;
    }

    public Double getData () {
        return data;
    }
}
```

Imagine we wanted to do the same thing again with Byte, Short, Long etc.. Then imagine that we want to modify a method. We would have to make changes in many parts of the code. We might even forget some, leading to inconsistent behaviour between different classes.

We can avoid this by writing the code once, and letting the compiler make multiple versions for us. That is what generics do. Here is a generic class that replaces the two above classes.

```
class Sample<T> {
    private T data;
```

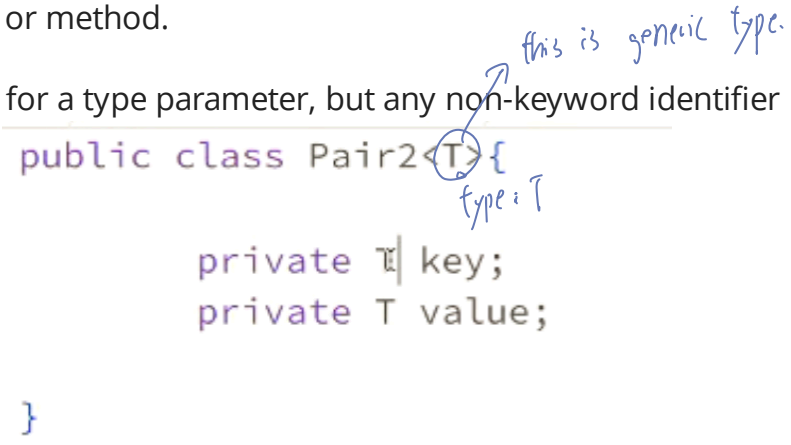
```
    public void setData (T newData) {
        data = newData;
    }

    public T getData () {
        return data;
    }
}
```

Here, `T` is a parameter for a type, and `<T>` tells the compiler that this is a generic method, parameterized by type `T`.

Generic classes and methods have a type parameter (a.k.a type variable). A type parameter can have any reference type (i.e., any class type) plugged in for the type parameter. When a specific type is plugged in, this produces a specific class type or method.

Traditionally, a single uppercase letter is used for a type parameter, but any non-keyword identifier may be used.



Conventional choices are

- E: Element (e.g., ArrayList)
- K: Key (e.g., HashMap<K, V>)
- V: Value
- N: Number
- T: Type
- S, U, V, and so on: Second, third, and fourth types

A class that is defined with a parameter for a type is called a *generic class* or a *parameterized class*.

The type parameter is included in angle brackets after the class name in the class definition heading.

The type parameter can be used like other types used in the definition of a class (e.g., instance variable declarations, method parameters).

Instantiation

A class definition with a type parameter is stored in a file and compiled just like any other class.

Once a parameterized class is compiled, it can be used like any other class. However, the class type plugged in for the type parameter *must be specified before it can be used in a program*. Doing this is said to *instantiate* the generic class.

```
Sample<Double> object = new Sample<Double>();
```

Tip: There are many pitfalls that can be encountered when using type parameters.

Compiling with the -Xlint option will provide more informative diagnostics of any problems or

potential problems in the code, including warnings

```
javac -Xlint Sample.java
```

you can provide a type at a run time when you're creating the object

GenericPairDem... Pair.java Pair2.java

```
public class Pair2<T>{

    private T key;
    private T value;

    public Pair2(T key, T value){
        this.key = key;
        this.value = value;
    }

    public T getKey(){
        return this.key;
    }

    public T getValue(){
        return this.value;
    }
```

random type (provide as T)

T could be any type, when I create an object

```
public class PairDemo(){

    public static void main(){
        Pair2<Integer, Integer> pairInteger = new Pair2(3, 4);
        Pair2<Double, Double> pairDouble = new Pair2(3.0, 4.0);
        System.out.println(pairInteger.getValue());
        System.out.println(pairDouble.getKey());


    }

    public static Pair2<T> returnComplexValue(){
        return pairInteger;
    }
}
```

like here is Integer

this is incorrect, should one have one type, because we only create one in Pair 2

Another example:

public class Person {
public int id;
public String name;

GenericPairD... Pair.java Pair2.java PairDemo.. Person.java

```
5
6   public Person(int id, String name){
7       this.id = id;
8       this.name = name;
9   }
10
11  public int getId(){
12      return this.id;
13  }
14
15  public String getName(){
16      return this.name;
17  }
```

```
public static void main(String[] args){
    Pair2<Integer> pairInteger = new Pair2(3, 4);
    Pair2<Double> pairDouble = new Pair2(3.0, 4.0);
    System.out.println(pairInteger.getValue());
    System.out.println(pairDouble.getKey());

    Pair2<Person> personPair = new Pair2(new Person(1, "Me"), new Person(2, "You") )
    System.out.println(personPair.getKey().getName());
```

Me

this called anonymous object."

Generic pair class

Fill in each of the BLANKs in the below with one of the following

T

<T>

Pair<T>

(Pair<T>)

Pair<String>

String

or leave it blank.

> Note that method *definitions* including **constructors *do not*** include type parameter in angle brackets. However, when *calling* constructors, the type parameter is needed.

Modify the `GenericPairDemo` class to use Integer object instead of String object.

Next, modify it to use int objects. Having trouble? Don't worry. It is impossible!

The type plugged in for a type parameter must always be a *reference type*, which includes arrays.

It cannot be a primitive type such as `int`, `double`, or `char`. However, now that Java has automatic boxing, this is not a big restriction. This ability to be used in generics is one of the big reasons for the classes `Integer`, `Double`, `Character` and so on.

Identify where in your code you used automatic boxing:

Multiple type parameters

A generic class definition can have any number of type parameters.

Multiple type parameters are listed in angle brackets just as in the single type parameter case, but are separated by commas. In this case, the rule of single-letter parameters is frequently broken; different types represented by the same letter are distinguished by a digit after the letter.

```
public class TwoTypePair<T1, T2> {
    private T1 first;
    private T2 second;

    public TwoTypePair() {
        first = null;
        second = null;
    }

    public TwoTypePair(T1 firstItem, T2 secondItem) {
        first = firstItem;
        second = secondItem;
    }

    public void setFirst(T1 newFirst) {
        first = newFirst;
    }

    public void setSecond(T2 newSecond) {
        second = newSecond;
    }

    public T1 getFirst () {
        return first;
    }

    public T2 getSecond () {
        return second;
    }

    public String toString () {
        return ("first: " + first.toString() + "\n"
              + "second: " + second.toString());
    }

    public boolean equals (Object otherObject) {
        if (otherObject == null
                || getClass() != otherObject.getClass()) {
            return false;
        } else {
            TwoTypePair<t1, T2> otherPair =
```

```
                (TwoTypePair<T1, T2>)otherObject;
                    // The first  equals is the equals of T1.
                    // The second equals is the equals of T2.
            return (first.equals(otherPair.first)
                && second.equals(otherPair.second));
        }
    }
}
```

```
public class Person{

    public Person(int id, String name){
        this.id = id;
        this.name = name;
    }

    public int getId(){
        return this.id;
    }

    public String getName(){
        return this.name;
    }
}
```

```
public class KeyValuePair<K, V> {

    private K key;
    private V value;

    public KeyValuePair(K key, V value){
        this.key = key;
        this.value = value;
    }

    public K getKey(){
        return this.key ;
    }
}
```

public V getValue(){
    return this.value;
}

PairDemo.java

```
Person p = new Person();
Pair2<Integer> pairInteger = new Pair2<>(3, 4);
KeyValuePair<Integer, Person> personKVP = new KeyValuePair<>();
```

it will automatic understand (not necessary to write)

In general, you need to provide datatype.

```
Person p = new Person();

KeyValuePair<Integer, Person> personKVP = new KeyValuePair<>(1, new Person(1, "Me"));
KeyValuePair<Integer, Person> person2KVP = new KeyValuePair<>(2, new Person(2, "You"));
System.out.println(person2KVP.getKey());=>2
System.out.println(person2KVP.getValue().getName());=> You
```

```
public class Car{ extends Entity

    //private String name;
    private String manufacturer;

    public Car(String name, String manu){
        this.name = name;
        this.manufacturer = manu;

    }

    public String getName(){
        return this.name;
    }

    public String getManufacturer(){
        return this.manufacturer;
    }
}
```

```
public class Calculator{

    public int sum(int a, int b){
        return a+b;
    }
}
```

generics:

```
public class Descriptor<T> {

    private T value;

    public Descriptor(T val){
        this.value = val;
    }

    public String describe(){
        return this.val.getName();
    }
}
```

modify

```
public class Descriptor<T extends Entity> {

    private T value;

    public Descriptor(T val){
        this.value = val;
    }

    public String describe(){
        return this.value.getName();
    }

}
```

T only extend from entity class

```
Descriptor<Car> descCar = new Descriptor(new Car("AMG", "Mercedes"));
Descriptor<Person> descPerson = new Descriptor(new Person(1,"Trina"));
Descriptor<Calculator> descCalc = new Descriptor(new Calculator());
```

It could be any valid class type.

This will be incorrect after modify

we can not describe calculator, we need to restrict the class type here.

add Entity here, the class should extend from Entity

```
public class Entity{

    protected String name;

    public String getName(){
        return this.name;
    }
}
```

Bounds for type parameters

Sometimes it makes sense to restrict the possible types that can be plugged in for a type parameter `T`.

For instance, to ensure that only classes that implement the `Comparable` interface are plugged in for `T`, define a class as follows:

```
public class RClass<T extends Comparable>
```

"`extends Comparable`" serves as a *bound* on the type parameter T

Any attempt to plug in a type for `T` which does not implement the `Comparable` interface will result in a compiler error message.

A bound on a type may be a class name (rather than an interface name)

Then only descendent classes of the bounding class may be plugged in for the type parameters

```
public class ExClass<T extends Class1>
```

A bounds expression may contain **multiple interfaces and up to one class** (just the same as a class can implement multiple interfaces and extend up to one class).

If there is more than one type parameter, the syntax is as follows:

```
public class Two<T1 extends Class1, T2 extends Class2 & Comparable>
```

bound the class type.

```
public class Pair<T extends Comparable> {
    private T first;
    private T second;

    public T max() {
        if (first.compareTo(second) <= 0) {
            return first;
        else
            return second;
        }
    }
}
```

implement

could be interface or inheritant.

Tip: Generic interfaces

An interface can have one or more type parameters.

The details and notation are the same as they are for classes with type parameters.

Generic methods (advanced) not reifiable.

When a generic class is defined, the type parameter can be used in the definitions of the methods for that generic class.

In addition, a generic method can be defined that has its own type parameter that is not the type parameter of any class.

A generic method can be a member of an ordinary class or a member of a generic class that has some other type parameter. The type parameter of a generic method is ***local*** to that method, not to the class.

The type parameter must be placed (in angle brackets) after all the modifiers, and before the returned type

local to the method

```
public static <T> T genMethod(T[] a)
```

When one of these generic methods is invoked, the method name is prefaced with the type to be plugged in, enclosed in angle brackets.

```
String s = NonG.<String>genMethod(c);
```

```
public class Utility {
    public static <T> T getMidPoint(T[] a) {
        return a[a.length/2];
    }

    public static <T> T getFirst(T[] a) {
        return a[0];
    }

    public static <T1, T2> boolean isSameClass(T1 a, T2 b) {
        return (a.getClass() == b.getClass());
    }
}
```

local to this one

Inheritance with generic classes

A generic class can be defined as a derived class of an ordinary class or of another generic class.

As in ordinary classes, an object of the subclass type would also be of the superclass type.

Inheritance with generic classes.

A generic class can be defined as a derived class of an ordinary class or of another generic class. The syntax is

```
class child<T> extends parent<T>
```

As in ordinary classes, an object of the subclass type would also be of the superclass type.

Pitfall: Relationships between type parameters are not preserved

Given two classes `A` and `B`, and a generic class `G`, there is no relationship between `G<A>` and `G<B>`.

This is true even if there is a relationship between `A` and `B`, such as `B` being a subclass of `A`.

For example, if class `HourlyEmployee` is derived from class `Employee`, then there is no relationship between `G<HourlyEmployee>` and `G<Employee>`.

Note however that relationships between generics are preserved, as stated on the second line of this slide. If generic class `H<T>` extends `G<T>`, then

- `H<A>` is a subclass of `G<A>`
- `H<B>` is a subclass of `G<B>`

Modify `UnorderedPairDemo` to use `instanceof` to confirm that the type of `p1` is a descendant of `Pair<String>`.

Pitfalls

A Type Parameter Cannot Be Used Everywhere a Type Name Can Be Used

Within the definition of a parameterized class definition, there are places where an ordinary class name would be allowed, but a type parameter is not allowed.

In particular, **the type parameter cannot be used in simple expressions using `new` to create a new object**.

For instance, the type parameter cannot be used as a constructor name or like a constructor:

```
T object = new T();
T[] a = new T[10];
```

See https://www.baeldung.com/java-generic-array for an explanation of why the array case is not allowed, and https://www.softwaretestinghelp.com/java-generic-array/ for a (rather complicated) work-around.

An Instantiation of a Generic Class Cannot be an Array Base Type

Arrays such as the following are illegal:

```
Pair<String>[] a = new Pair<String>[10];
```

Although this is a reasonable thing to want to do, it is not allowed because of the way that Java implements generic classes.

However, we can store these types in a generic Object array. When retrieving them, we must cast them back to the desired type:

```
Object[] a = new Object [10];
a[0] = new Pair<String>(10, 20);
Pair<String> c = (Pair<String>) a[0];
```

A Generic Class Cannot Be an Exception Class

It is not permitted to create a generic class with `Exception`, `Error`, `Throwable`, or any descendent class of `Throwable`.

A generic class cannot be created whose objects are throwable:

```
public class GEx<T> extends Exception
```

The above example will generate a compiler error message.

Growing array container class (Advanced)

Build a generic class `GrowArray<T>`, inheriting the Container class, that stores data in an array.

It should allow the following methods:

```
T get(int idx);
```

Returns the item at index `idx` in the array. Returns `null` if `idx` is greater than the last index in the array.

```
boolean set(int idx, T item);
```

Inserts `item` at index `idx`. If `idx` is greater than the last index, then the array is extended, by at least a factor of 2, and `true` is returned. Otherwise, `false` is returned.

```
int size ();
```

Returns the location of the highest index that has been `set`.

Advanced: Implement the Iterator interface.

Exercise: What are the benefits of this over an ArrayList?

Optional

```
int capacity();
```

Returns the length of the underlying array, which is one more than the largest index that can be used without resizing.

```
int trim (int size);
```

Reduce the size of the underlying array to `size`. If it is already smaller than `size`, do not change it.

```
boolean set (int idx, GrowArray<T>values);
```

Equivalent to

```
for (int i = 0; i < values.size(); i++)
    set(idx + i, values.get(i));
```

Returns true if the underlying array grew. Use values.size() to ensure the array only grows once.