Java Basic-Jan28

15. Generics

which is not efficient to write different codes for all kinds of data type

```
class GTNode1 {
  String key;
  Integer value;
  public GTNode1 (String key, Integer value) {
    this.key = key;
    this.value = value;
  }
}

class GTNode2 {
  Integer key;
  Float value;
  public GTNode2 (Integer key, Float value) {
    this.key = key;
    this.value = value;
  }
}
...
```

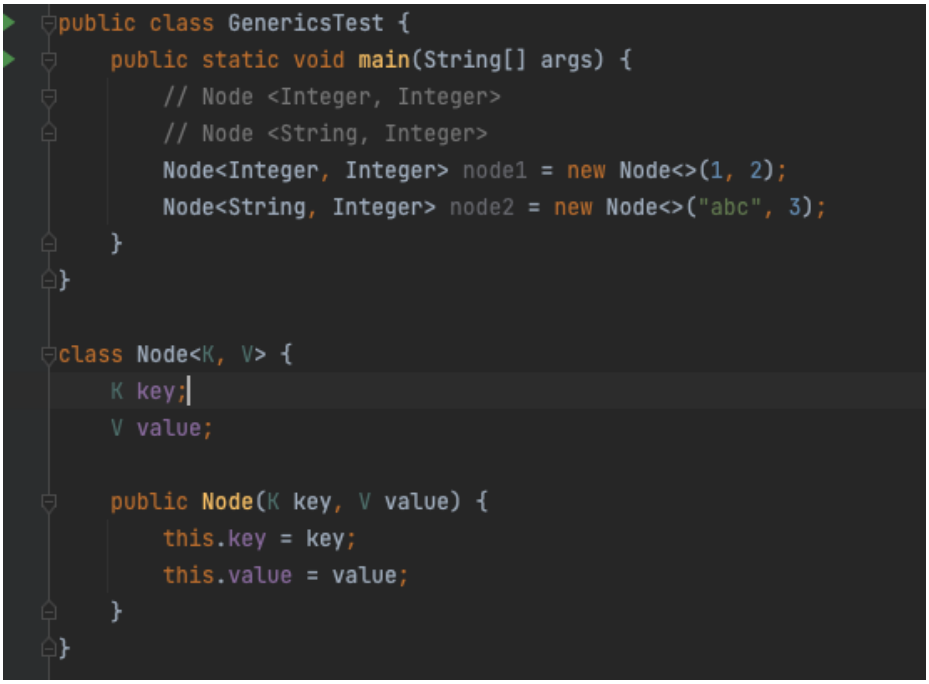
generics basic

advantage of generics

- easier and less error- prone
- enforce type correctness at compile time
  - return type must be same with input type - guarantee the data consistency(一致性)
- without causing any extra overhead to your application
  - we don’t need to create multiple class/method to map different data types

for Node<K, V>, any letter is ok. K and V represent different data types.

Upper Case 大写字母



```java
public class GenericsTest {
    public static void main(String[] args) {
        // Node <Integer, Integer>
        // Node <String, Integer>
        Node<Integer, Integer> node1 = new Node<>(1, 2);
        Node<String, Integer> node2 = new Node<>("abc", 3);
    }
}

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

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

Use the way to define a generic method

for example: to define a method accept the input could be any types of array, like String[], Integer[] and so on.

```
public static <E> E getFirstElements(E[] arr) { // E can be any letter
  return arr[0];
}
// for this kind of method. we can not input a primitive type, like int[]
// we should use Integer[], which is same with Map<Integer, Integer>


public static <E, U> E method(E[] arr, U[] arr2) {
  return arr[0];
}
```

Why primitive type can not used in Generic? - trigger compile

type erasure - HW

16. I/O Stream

Basic

- A stream is just a continuous flow of data.
- The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination
- The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java.

ByteStream

- Byte stream performs input and output of 8-bit bytes.
- All byte stream classes are descended from InputStream and OutputStream.

CharacterStream

- Character stream is 2 bytes stream used for character transfer.
- All character stream classes are descended from Reader and Writer.

```
* Stream:
*  based on bit units: ByteStream, CharacterStream (16 bits)
*  based on direction: InputStream, OutputStream
*
* Abstract Level      ByteStream            CharacterStream
* Input               InputStream           Reader
* Output              OutputStream          Writer
*
* Abstract Level      FileStream             Processing Stream (e.g. Buffered Stream)
* Input Stream        FileInputStream        BufferedInputStream
* OutputStream        FileOutputStream       BufferedOutputStream
* Reader              FileReader             BufferedReader
* Writer              FileWriter             BufferedWriter
*
* There are overall more than 40 types of IO streams in Java.
*
* type                    ByteStream(in)          ByteStream(out)          CharacterStream(in)     CharacterStream(in)
* abstract super class    InputStream             OutputStream             Reader                  Writer
* file access             FileInputStream         FileOutputStream         FileReader              FileWriter
* array access            ByteArrayInputStream    ByteArrayOutputStream    CharArrayReader         CharArrayWriter
* pipe access             PipedInputStream        PipedOutputStream        PipedReader             PipedWriter
* string access           n/a                     n/a                      StringReader            StringWriter
* buffered stream         BufferedInputStream     BufferedOutputStream     BufferedReader          BufferedWriter
* transfer stream         n/a                     n/a                      InputStreamReader       OutputStreamWriter
* object stream           ObjectInputStream       ObjectOutputStream       n/a                     n/a
*                         FilterInputStream       FilterOutputStream       FilterReader            FilterWriter
* print stream            n/a                     PrintStream              n/a                     PrintWriter
* push back input         PushbackInputStream     n/a                      PushbackReader          n/a
* Special Stream          DataInputStream         DataOutputStream         n/a                     n/a
```

```java
public class JavaIOStream {
    public static void main(String[] args) throws IOException {
        InputStream in = null;
        OutputStream out = null;

        try {
            in = new FileInputStream( name: "/Users/shaohua/Desktop/JavaMaterial/input.txt");
            out = new FileOutputStream( name: "/Users/shaohua/Desktop/JavaMaterial/output.txt");
            int c;
            while ((c = in.read()) != -1) {
                System.out.print((char)c);
                out.write(c);
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (in != null) in.close();
            if (out != null) out.close();
        }
    }
}
```

deme for writer and reader

FileReader and FileWriter are CharacterStream, so they read/write 16 bits(2 bytes) each time

```java
FileReader reader = null;
FileWriter writer = null;
try {
    reader = new FileReader("in.txt");
    writer = new FileWriter("out.txt");
    int a = 0;
        while ((a = reader.read()) != -1) {
            writer.write(a);
        }
    } catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        reader.close();
        writer.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

File

- the file class is part of java.io
- give you access to underlying(基础的) file system

```
String dirName = "/Users/shaohua/Desktop/JavaMaterial";
File d = new File(dirName);
String[] paths = d.list();
for (String path: paths) {
   System.out.println(path);
}
```

scenario: log, CSV file...

BufferedInput/OutputStream for **read line**

a buffer for pre-store the data

underlying is read by byte or other ?

Extra:

read data input byte array: Byte[] buffer = new Byte[100];

pass the buffer into InputStream, InputStream will read the data input the buffer, and then next time if we want to get the data, just read the data from the buffer.

Block Stream: when execute read/write, java will be waiting for the end of the IO operation, and then go on.

non-blocking IO: when do reading and writing the Java will not suspend

17. Serialization and Deserialization

To serialize an object means to convert its state to a byte stream so that the byte stream can be reverted back into a copy of the object

```
public classs Person implement Serializable {
  private static final long serialVersionUID = 123123124L;
  // all the class has unique UID, same class should have same ID
}
//Attention: java.io.Serializable just is a marker interface.
// Nothing is defined inside it.

//serialVersionUID is the identifier of the class.
//The same class should have the same uid.
```

after converting to byte stream, it can be sent to other application

other application can deserialize the byte stream and convert to the copy of object

- Object inside another serializable object must be serializable too.
- If parent class is serializable, then child class is automatically serializable

Serialization is not safe

- Serialization allows class refactoring
    - When a password in an object is serialized, it can be deserialized.

use “transient” keyword to prevent variable from being serialized, which is used to protect some sensitive data, like SSN

“static” field won’t be serialized too

```
//Entity
import java.io.Serializable;

// need to implement Serializable Interface
public class Employee implements Serializable {
   public String name;
   public int age;
   public transient int SSN; // make SSN not be serialized

   public String getName() {
       return name;
   }

   public int getAge() {
       return age;
   }

   public int getSSN() {
       return SSN;
   }

   public void setName(String name) {
       this.name = name;
   }

   public void setAge(int age) {
       this.age = age;
   }

   public void setSSN(int SSN) {
       this.SSN = SSN;
   }
}


//Serialization
import java.io.*;

public class JavaSer{
   public static void main(String[] args) {
       Employee emp = new Employee();
       emp.setAge(123);
       emp.setName("Kaidong");
       emp.setSSN(123456);

       try {
           OutputStream fileout = new FileOutputStream("/Users/shaohua/Desktop/JavaMaterial/employee.ser");
           ObjectOutputStream out = new ObjectOutputStream(fileout);
           out.writeObject(emp);
           out.close(); // first open,
           fileout.close();// last close
       } catch (FileNotFoundException e) {
           e.printStackTrace();
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

//deserialization
public class JavaDes {
    public static void main(String[] args) {
        Employee e = null;
        try {
            InputStream fileIn = new FileInputStream("/Users/shaohua/Desktop/JavaMaterial/employee.ser");
            ObjectInputStream in = new ObjectInputStream(fileIn);
            e = (Employee) in.readObject(); // readObject() return Object, wich need to cast to Employee
            in.close();
            fileIn.close();
        } catch (FileNotFoundException ex) {
            ex.printStackTrace();
        } catch (IOException ex) {
            ex.printStackTrace();
        } catch (ClassNotFoundException ex) {
            ex.printStackTrace();
        }


        System.out.println(e.getAge());
        System.out.println(e.getName());
        System.out.println(e.getSSN());
    }
}
```

18. Java 8 features

Lambda

- functional programming
- less code

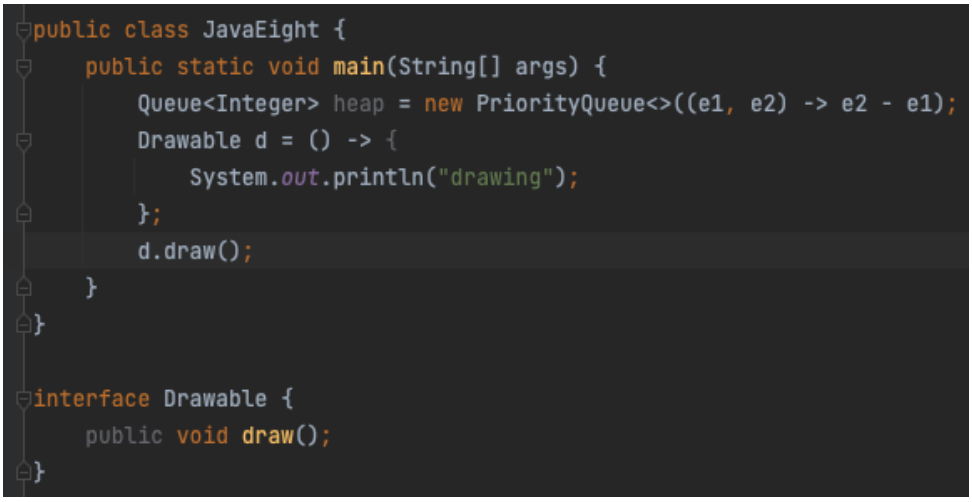
(arguments) -> {body}

contains two parts:

in the **parenthesis, we pass the argument list into it**

and then return the result in **the curly brackets** (the body)

we can pass the lambda into Drawable interface



```java
public class JavaEight {
    public static void main(String[] args) {
        Queue<Integer> heap = new PriorityQueue<>((e1, e2) -> e2 - e1);
        Drawable d = () -> {
            System.out.println("drawing");
        };
        d.draw();
    }
}

interface Drawable {
    public void draw();
}
```

Functional Interface

(use @FunctionalInterface to mark an interface is a functional interface)

- the annotation is optional, we don't have to write

every functional interface should have **ONLY** one abstract method

and any number of (concrete)default and static methods

```java
public class JavaEight {
    public static void main(String[] args) {
        SayBye sb = () -> {
            System.out.println("Bye");
        };
        sb.saybBye();
        sb.sayHello();
        sb.sayGM();
    }
}
@FunctionalInterface  // optional
interface SayBye {
    void saybBye();

    default public void sayHello() {
        System.out.println("Hello");
    }

    default public void sayGM() {
        System.out.println("Good morning");
    }
}
```

serializable is **mandatory** to be implemented

@FunctionalInterface is **optional**

Pre-define Interface in Java (There are lots of Interfaces, 4 popular in following)

work with lambda expression

Think Deeply:

why there is only one abstract class in Functional interface?

- cuz lambda exp will pass(match) the function in the abstract class automatically
-

```java
<R> Stream<R> map(Function<? super T, ? extends R> mapper);
```

1. Predicate
    - accept one argument and return boolean
    - public Boolean test(T t);

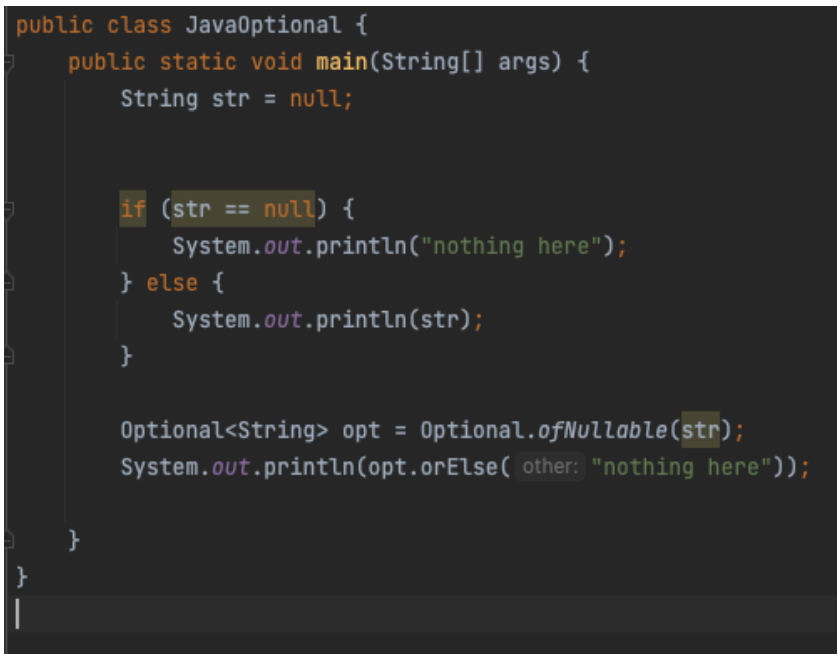
2. Function
    - public R apply(T t); // R is generic

3. Consumer
    - public void accept(T t);

4. Supplier
    - public R get(); // R is generic

```
Supplier<Double> generateRandomNumber = () -> Math.random();
System.out.println(generateRandomNumber.get());
```

Optional



use if else to do a null check to make sure the String is not null

```
Optional<String> opt = Optional.ofNullable(str); //try to get something inside str
opt.orElse("nothing here"); // if str is null, put the words
// same with the if else block below
//replace do the if else check every time
//orElse also can Throw exception by orElseThrow();
```

Stream API

- intermediate operation: return **a stream as result**
  - map, flatmap, filter, ….
- terminal operation: return **non-stream**,
  - forEach , collect…

Homework 4.2

- explore other stream API (at least 10)
  - map vs. flatmap, distinct, limit
- Method reference

JVM Q:

1. What's the main components of JVM?
2. What's different between stack and heap?
3. How many different types of GC are there?
4. Should know different generations, young, old, and permanent
   a. don't need to explain GC process, but the concepts like what is GC root

evaluation:

at least concept, coding experience will be better

if not good for basic concepts, will have coding question

1. **map:** The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.
    a. One-To-One mapping

```
List number = Arrays.asList(2,3,4,5);
List square = number.stream().map(x->x*x).collect(Collectors.toList());
```

2. flatmap: Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
    a. One-To-Many mapping

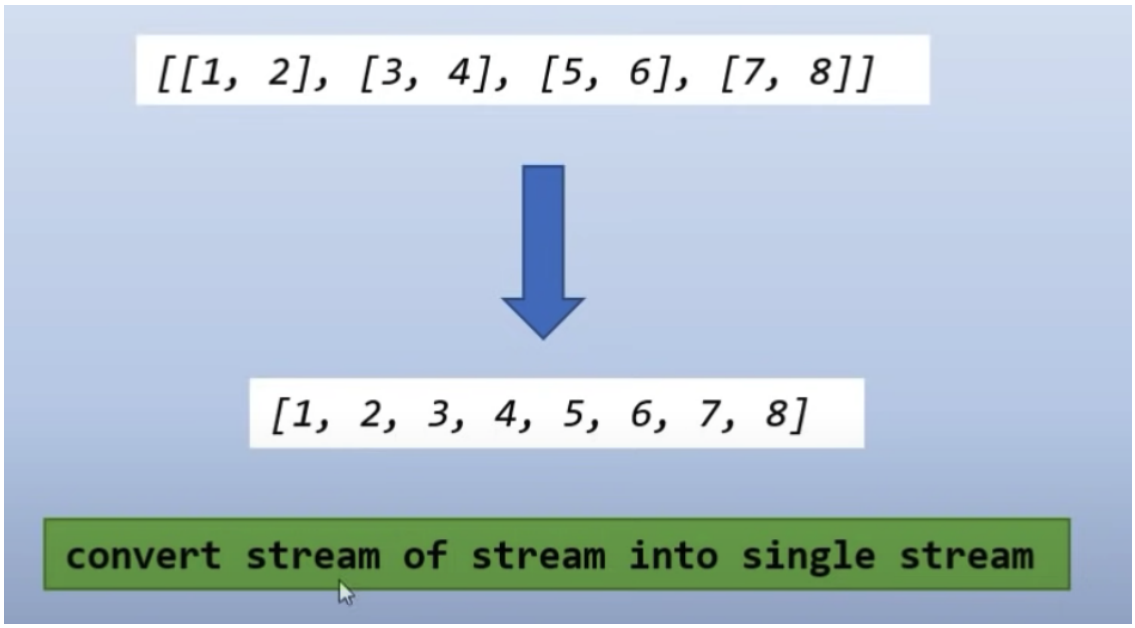
map() vs. flatmap

The difference is that the map operation produces one output value for each input value, whereas the flatmap operation produces an arbitrary number (zero or more) values for each input value.

Differences between Java 8 Map() Vs flatMap() :

map()	flatMap()
It processes stream of values.	It processes stream of stream of values.
It does only mapping.	It performs mapping as well as flattening.
It's mapper function produces single value for each input value.	It's mapper function produces multiple values for each input value.
It is a One-To-One mapping.	It is a One-To-Many mapping.
Data Transformation : From Stream to Stream	Data Transformation : From Stream<Stream to Stream
Use this method when the mapper function is producing a single value for each input value.	Use this method when the mapper function is producing multiple values for each input value.

data Flattering



Example of flatmap()

```
// a person has several phone numbers
// List<String> phoneNum = person.getPhoneNumber();
List<String> phoneNum = persons.stream().flatmap(
                person -> person.getPhoneNUmber().stream()
                )
              .collect(Collectors.toList());)
```

3. distinct
   - This method uses *hashCode()* and *equals()* methods to get distinct elements.
   - It's useful in removing duplicate elements from the collection.

4. limit

   restrict the number of stream elements, which take first N elements by limit(N)

   ```
   list.stream().limit(N).collect(Collectors.toList());
   ```

5. skip

   an intermediate operation that discards the first *n* elements of a stream.

```
Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    .filter(i -> i % 2 == 0)
    .skip(2)
    .forEach(i -> System.out.print(i + " "));
//result: 6 8 10
```

6. findFirst

   returns an *Optional* for the first entry in the stream; the *Optional* can, of course, be empty.

7. toArray - terminal operation

   If we need to get an array out of the stream, we can simply use *toArray().*

```
Employee[] employees = empList.stream().toArray(Employee[]::new);
```

8. min and max

   return the minimum and maximum element in the stream respectively, based on a comparator.

```
Employee firstEmp = empList.stream()
      .min((e1, e2) -> e1.getId() - e2.getId())
      .orElseThrow(NoSuchElementException::new);
```

9. peek

Perform multiple operations on each element of the stream before any terminal operation is applied.

```
empList.stream()
      .peek(e -> e.salaryIncrement(10.0))
      .peek(System.out::println)
      .collect(Collectors.toList());
```

10. sorted

this sorts the stream elements based on the comparator passed we pass into it.

```
List<Employee> employees = empList.stream()
      .sorted((e1, e2) -> e1.getName().compareTo(e2.getName()))
      .collect(Collectors.toList());
```

Method reference

Method references are a special type of lambda expressions. They're often used to create simple lambda expressions by referencing existing methods.

There are four kinds of method references:

- Static methods
- Instance methods of particular objects
- Instance methods of an arbitrary object of a particular type
- Constructor

Syntax

- *ContainingClass*::*MethodName*

```
List<Integer> numbers = Arrays.asList(5, 3, 50, 24, 40, 2, 9, 18);
// normal
numbers.stream()
  .sorted((a, b) -> a.compareTo(b));
// method reference
numbers.stream()
  .sorted(Integer::compareTo);
```