Standard Classes

2020 Spring: AP Computer Science A

January 29th, 2020

Today

- Wrapper classes
- Math class
- StringBuffer class
- Generics
- Arrays (2D)
- Lists and ArrayLists
- Iterators

Wrapper Classes

- **Wrapper classes provide a *wrapper for primitive types***
  - Sometimes we have to use the primitive types as objects

Primitive Type	Wrapper Class Name
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`char`	`Character`
`float`	`Float`
`double`	`Double`
`boolean`	`Boolean`

Wrapper Classes

- **Constructors for wrapper classes (don't do this)**

```
Integer i = new Integer(1);
Long l = new Long(10);
Double d = new Double(3.14);
Character c = new Character('p');
Boolean b = new Boolean(true);
```

- **Getting the primitive values inside the wrapped object**

```
int iv = i.intValue();
long lv = l.longValue();
double dv = d.doubleValue();
char cv = c.charValue();
boolean bv = b.booleanValue();
```

Auto Boxing / Unboxing

- ***Boxing***: Converting a primitive value to a wrapper class object
- ***Unboxing***: Converting a wrapper class object to a primitive value

- Auto boxing and unboxing are supported in Java

```
Integer x = 2147;          // boxing
int sum = x + 27;          // unboxing
System.out.println(sum);
```

- Constructors for wrapper classes aren't used very often

Wrapper Class – Static Methods

- **Instead of constructors, Java also has static methods**

```
Integer i = Integer.valueOf(50);
Double d = Double.valueOf(3.1415);
Boolean b = Boolean.valueOf(false);
```

- **Converting String to primitive types**

```
int i = Integer.parseInt("2147");
double d = Double.parseDouble("3.145");
boolean b = Boolean.parseBoolean("true");
```

Math Class

- **Java provides methods for mathematical calculations**
- **All methods and fields are static**
  - No need to instantiate an object

- https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/Math.html

Math Class

```
print(Math.random()); // 0 <= x < 1
print((int) (Math.random() * 6 + 1));

Random r = new Random();
print(r.nextInt(100)); // 0 <= x < 100

print(Math.abs(-10));
print(Math.floor(3.1));
print(Math.ceil(3.1));
print(Math.round(3.13)); // round to integer

print(Math.max(1, 2));
print(Math.min(1, 2));

print(Math.pow(5, 2));
print(Math.sqrt(3));

print(Math.sin(Math.toRadians(30)));
print(Math.cos(Math.PI / 3));
print(Math.tan(Math.PI / 6));
```

`StringBuffer` Class

- Strings are immutable!
- You can change string values with `StringBuffer`
- `StringBuffer` is much faster than `String`

- https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/StringBuffer.html

StringBuffer Class

```
StringBuffer sb = new StringBuffer(); // empty string
print(sb.toString()); // ""

sb.append("asdf");
print(sb.toString()); // "asdf"

sb.insert(1, "gh");
print(sb.toString()); // "aghsdf"

sb.delete(1, 2);
print(sb.toString());// "ahsdf"

sb.deleteCharAt(1);
print(sb.toString());// "asdf"

sb.reverse();
print(sb.toString()); // "fdsa"
```

Generic Types

- **Generic types enable you to *generalize* the type you use in a class**
- **The type is checked at compile time**

- **Benefits**
  - Enables type safety
  - Reduced cost for type checking
  - *Won't have to define same structure for different types*
    - `Stack` (Interface) can be used to implement
      - `IntStack, DoubleStack, StringStack` (bad – 3 classes)
    - Generic `Stack` class
      - The type of its element will be set at compile time

Generic Types

- **Declare type parameter as <T> in the class name**
- **Can use T as type**
  - Can be used as parameters
  - Can be used as return types
- **T can also be constrained to a group of types (seen later)**

```
class MyType<T> {
    T val;

    public MyType() {
        val = null;
    }

    public MyType(T val) {
        this.val = val;
    }

    public void set(T val) {
        this.val = val;
    }

    public T get() {
        return val;
    }

    public String toString() {
        return val.toString();
    }
}
```

Generic Types

- **For instantiation, specify the type inside <T>**
  - T must be an object, *not a primitive type*
    - Necessity of wrapper classes

```
MyType<Integer> t = new MyType<Integer>();
t.set(35);
System.out.println(t.get());
System.out.println(t);

MyType<String> s = new MyType<String>("example");
System.out.println(s);
```

Generic Types

- **Suppose we change the class parameter T to the following**

```
class MyType<T extends Number> {
    ...
}
```

  - `MyType<String>` doesn't work anymore
    - `String` does not extend `Number`

- **You can restrict the types by using `extends` in the type parameter**

2D Arrays

- **Similar to 1D arrays**

```
type[][] arr = new type[row][column];
```

- **Example**
  - Access and assignment are also similar

```
int[][] arr = new int[10][10]; // 10 x 10 int array

arr[1][3] = -1;
print(arr[5][7]);
```

2D Arrays

- **Initializer list syntax is also possible**

```
int[][] arr2 = { { 1, 0, 1 }, { 0, 1, 0 }, { 0, 0, 1 } };
```

- **Usage in return types and parameters**

```
public static int[][] foo(int[][] arr) {
    return arr;
}
```

- **Can have dynamic length for each row**

```
int[][] arr3 = new int[3][];
arr[0] = new int[1];
arr[1] = new int[3];
arr[2] = new int[2];
```

Limitations of Arrays (Revisited)

- **Cannot resize**
  - Need to reallocate another array with new keyword
- **Cannot insert, or delete elements in the middle**
  - Insertion at index i
    - Need to push all the elements after i
  - Deletion at index i
    - Need to pull all the elements after i
  - Complexity
- **We need something better**

List<E> Interface

- **Contains useful methods for manipulating a list of objects**
- https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/List.html

Definition	Description
`boolean add(E e)`	Add element at the end of the list
`void add(int idx, E e)`	Add element at specified index
`void clear()`	Clear the list
`boolean contains(Object o)`	Check if the list contains the specified object
`boolean isEmpty()`	Check if the list is empty
`E remove(int idx)`	Remove and return the item at the specified index
`int size()`	Return the size of the list
`E set(int idx, E e)`	Set the element at the specified index to the specified object
`E get(int idx)`	Returns the element at the specified index
`Iterator<E> iterator()`	Returns an iterator over the elements in the list

ArrayList<E>

- **Similar to arrays, but *resizing is done automatically***
  - But note that *resizing is slow*

- **Used as arrays in generic classes**
  - Generic arrays cannot be instantiated

ArrayList<E>

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

arr.add(40); // [40]
arr.add(20); // [40, 20]
arr.add(1, 10); // [40, 10, 20]
arr.add(30); // [40, 10, 20, 30]

print(arr);
print(arr.get(1)); // 10

for(int e : arr) {   // you can use for-each
    print(e);
}

Collections.sort(arr); // [10, 20, 30, 40]

arr.set(1, 50); // [10, 50, 30, 40]

arr.size(); // 4

arr.isEmpty(); // false
```

Exercises

- **Implement a generic `Stack`, `Queue` class**
  - No size limit

- **Implement a double-ended queue (`Deq`)**
  - `pushBack, pushFront, popBack, popFront`

Iterator<E>

- **Java `Collections`**
  - Java provides predefined data structures for collections of objects

- ***iterator*: An object used to iterate a collection**

Definition	Description
`boolean hasNext()`	Returns true if there is at least one more element to be examined, false otherwise
`E next`	Returns the next element in iteration
`void remove()`	Deletes the last element that was returned

Iterator<E>

- **Usage**
  - Print and delete each element in an `ArrayList`

```
ArrayList<String> arr = new ArrayList<String>();

arr.add("element 1");
arr.add("Hi");
arr.add("Hello");
arr.add("안녕");

Iterator<String> it = arr.iterator();

while (it.hasNext()) {
    String str = it.next();
    System.out.println(str);
    it.remove();
}

System.out.println(arr.isEmpty());   // true
```

Matrix

- **Implement a Matrix class!**
  - Should be compatible with Vector class