Java

Generics & Collections

Generics

Generics

- Many algorithms are logically the same no matter what type of data they are being applied to (Stack of Integer, String or Thread)
- Generics (introduced by JDK 5) allows to create classes, interfaces, and methods that will work in a type-safe manner with various kinds of data
- Generics allows to define an algorithm once, independently of any specific type of data
  - The expressive power generics added to the language fundamentally changed the way that Java code is written

Generics

- The term generics means **parameterized types**
- It enables to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter
- Using generics, it is possible to create a single class, for example, that automatically works with different types of data

Generics

- Java has always given you the ability to create generalized classes, interfaces, and methods by operating through references of type **Object**
- In pre-generics code, generalized classes, interfaces, and methods used **Object** references to operate on various types of objects
- **The problem is that they could not do so with type safety**

Generics

- **Generics added the type safety that was lacking**
- They also streamlined the process
  - it is no longer necessary to explicitly employ casts to translate between Object and the type of data that is actually being operated upon
- With generics, all casts are automatic and implicit

Generic Class

public class MyGenerics<T>

- Here, T is the name of a type parameter. This name is used as a placeholder for the actual type that will be passed to MyGenerics when an object is created

  MyGenerics<Integer> myGenerics = new MyGenerics<>()

- MyGenerics uses a type parameter, MyGenerics is a generic class
- Type parameters can be bounded
- ***Example**: MyGenerics(1-3).java*

Generics Only with Reference Types

- When declaring an instance of a generic type, the type argument passed to the type parameter must be a reference type
- You cannot use a primitive type, such as int or char
- The following declaration is illegal:

```
MyGenerics<int> intOb = new MyGenerics<int>();
// Error, can't use primitive type
```

Generic Method

- Methods inside a generic class can make use of a class' type parameter
- However, it is possible to declare a generic method that uses one or more type parameters of its own
- Furthermore, it is possible to create a generic method that is enclosed within a non-generic class
- It is possible for constructors to be generic, even if their class is not
- ***Example**: MyGenerics4.java*

Generic Interface

- In addition to generic classes and methods, you can also have generic interfaces
- Generic interfaces are specified like generic classes
- The generic interface offers two benefits
  - It can be implemented for different types of data
  - It allows to put constraints (that is, bounds) on the types of data for which the interface can be implemented
- ***Example****: MyGenerics5.java*

Wildcard and Bounded Wildcard

- The wildcard argument is specified by the ?, and it represents an unknown type
  - MyClass<?> matches any MyClass object
- Wildcard arguments can be bounded in much the same way that a type parameter can be bounded
  - A bounded wildcard is important when you are creating a generic type that will operate on a class hierarchy
- ***Example**: MyGenerics(6-7).java*

Collections

Collections

- The java.util package contains one of the Java's most powerful framework - **Collections**
- Collections is significantly affected by generics
- This framework defines several classes, such as lists and maps, that manage massive number of objects
- The collection classes have always been able to work with any type of object
- With generics the collection classes can now be used with complete type safety

Collection Interface

- It is the foundation upon which the Collection framework is built (***interface Collection<E>***)
- It must be implemented by any class that defines a collection
- Some functions

boolean add(E obj)
void clear()
boolean isEmpty()
boolean remove(Object obj)
boolean addAll(Collection c)
boolean contains(Object obj)
int size()
boolean removeAll(Collection c)

List Interface

- ***interface List<E>***
- Some functions

void add(int index, E obj)

boolean addAll(int index, Collection c)

E get(int index)

int indexOf(Object obj)

int lastIndexOf(Object obj)

E remove(int index)

Deque Interface

- ***interface Deque<E>***
- Some functions

void addFirst(E obj)	*void addLast(E obj)*
E getFirst()	*E getLast()*
E peekFirst()	*E peekLast()*
E pollFirst()	*E pollLast()*
E pop()	*void push(E obj)*
E removeFirst()	*E removeLast()*

ArrayList

- It extends the ***AbstractList*** class and implements the ***List*** Interface.
- It is a variable length array of object references that can dynamically increase or decrease in size (dynamic array)
- ArrayList is better for storing and accessing data
- ArrayList is non-synchronized
- ***Example****: ArrayListDemo(1-3).java*

LinkedList

- It extends the ***AbstractSequentialList*** class and implements the ***List, Deque*** and ***Queue*** Interface
- It provides a linked-list data structure
- LinkedList internally uses a doubly linked list to store the elements
- LinkedList is better for manipulating data
- LinkedList is non-synchronized
- ***Example****: LinkedListDemo.java*

Arrays

- The ***Arrays*** class provides various methods that are useful when working with arrays
- Some methods such as binarySearch, copyOf, copyOfRange, fill, sort are there
- ***Example****: ArraysDemo.java*

Vector

- It extends the ***AbstractList*** class and implements the ***List*** Interface
- It implements a dynamic array same as ArrayList
- Vector is synchronized
- ArrayList increments 50% of the current array size if the number of elements exceeds its capacity
- Vector increments 100% essentially doubling the current array size
- ***Example****: VectorDemo.java*

HashTable

- It stores key-value pairs
- Neither keys nor values can be null
- When using HashTable, you specify an object that is used as a key and the value you want linked to that key
- The key is then hashed and the resulting hash code is used as the index at which the value is stored within the table
- ***Example****: HashTableDemo.java*

HashMap

- It also stores key-value pairs like ***HashTable***
- Differences:

	HashMap	HashTable
Synchronized	No	Yes
Thread-Safe	No	Yes
Keys and values	One null key, any null values	Not permit null keys and values
Performance	Fast	Slow in comparison
Superclass	AbstractMap	Dictionary

- Use ***ConcurrentHashMap*** for multi-threading
- ***Example****: HashMapDemo.java*

Custom Comparator

- Required to sort a collection/array of custom objects
- Must implement the ***Comparable*** interface
- Must implement the following method

  public int compareTo(Object o) {

  }
- ***Example****: ComparatorDemo.java*