Agenda

- Strings
    - String
    - StringBuffer
    - StringBuilder
- String Tokenizer
- Garbage Collector
- Generics
- Generic class

Strings

- java.lang.Character is wrapper class that represents char.
- In Java, each char is 2 bytes because it follows unicode encoding.
- String is sequence of characters.
    - 1. java.lang.String: "Immutable" character sequence
    - 2. java.lang.StringBuffer: Mutable character sequence (Thread-safe)
    - 3. java.lang.StringBuilder: Mutable character sequence (Not Thread-safe)
- String helpers
    - 1. java.util.StringTokenizer: Helper class to split strings

String Class Object

- java.lang.String is class and strings in java are objects.
- String constants/literals are stored in string pool.
- String objects created using "new" operator are allocated on heap.
- In java, String is immutable. If try to modify, it creates a new String object on heap.

```
String name = "sunbeam"; // goes in string pool

String name2 = new String("Sunbeam"); // goes on heap
```

- Since strings are immutable, string constants are not allocated multiple times.
- String constants/literals are stored in string pool. Multiple references may refer the same object in the pool.
- String pool is also called as String literal pool or String constant pool.
- From Java 7, String pool is in the heap space (of JVM).
- The string literal objects are created during class loading.

StringBuffer and StringBuilder

- StringBuffer and StringBuilder are final classes declared in java.lang package.
- It is used create to mutable string instance.
- equals() and hashCode() method is not overridden inside it.
- Can create instances of these classes using new operator only. Objects are created on heap.

- StringBuffer implementation is thread safe while StringBuilder is not thread-safe.
- StringBuilder is introduced in Java 5.0 for better performance in single threaded applications.
- The default capactiy is 16.
- If the capacity is full it increases i.e new capacity is allocated

```
newCapacity = (oldCapacity*2) + 2
```

String Tokenizer

- Used to break a string into multiple tokens - like split() method.
- Methods of java.util.StringTokenizer
  - boolean hasMoreTokens()
  - String nextToken()
  - String nextToken(String delim)

Garbage Collector

- Garbage collection is automatic memory management by JVM.
- If a Java object is unreachable (i.e. not accessible through any reference), then it is automatically released by the garbage collector.
- An object become eligible for GC in one of the following cases:

```
// 1. Nullify the reference.
MyClass obj = new MyClass();
obj = null;

//2. Reassign the reference.
MyClass obj = new MyClass();
obj = new MyClass();

//3. Object created locally in method.
void method() {
MyClass obj = new MyClass();
// ...
}
```

- GC is a background thread in JVM that runs periodically and reclaim memory of unreferenced objects.
- Before object is destroyed, its finalize() method is invoked (if present).
- One should override this method if object holds any resource to be released explicitly e.g. file close, database connection, etc.

```
class Test {
    Scanner sc = new Scanner(System.in);

    @Override
    protected void finalize() throws Throwable {
        sc.close();
    }
}

public class Program{

    public static void main(String[] args) {
        Test t1 = new Test();
        t1 = null;
        System.gc();// request GC
    }
}
```

- GC can be requested (not forced) by one of the following.
  1. System.gc();
  2. Runtime.getRuntime().gc();
- GC is of two types i.e. Minor and Major.
  1. Minor GC: Unreferenced objects from young generation are reclaimed. Objects not reclaimed here are moved to old/permanent generation.
  2. Major GC: Unreferenced objects from all generations are reclaimed. This is unefficient (slower process).
- JVM GC internally use Mark and Compact algorithm.
- GC Internals: https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

Generic Programming

- Code is said to be generic if same code can be used for various (practically all) types.
- Best example:
  - Data structure e.g. Stack, Queue, Linked List, ...
  - Algorithms e.g. Sorting, Searching, ...
- Two ways to do Generic Programming in Java
  1. using java.lang.Object class -- Non typesafe
  2. using Generics -- Typesafe

2.Generics using Generics

- Added in Java 5.0.
- Similar to templates in C++.
- We can implement
  1. Generic classes
  2. Generic methods
  3. Generic interfaces
- Advantages of Generics

  - Stronger type checking at compile time i.e. type-safe coding.
  - Explicit type casting is not required.
  - Generic data structure and algorithm implementation.

Generic classes

- Implementing a generic class

```
class Box<TYPE> {
  private TYPE obj;
  s
  public void set(TYPE obj) {
    this.obj = obj;
  }
  public TYPE get() {
    return this.obj;
  }
}
```

```
Box<String> b1 = new Box<String>();
b1.set("Sunbeam");
String obj1 = b1.get();
System.out.println("obj1 : " + obj1);

Box<Date> b2 = new Box<Date>();
b2.set(new Date());
Date obj2 = b2.get();
System.out.println("obj2 : " + obj2);

Box<Integer> b3 = new Box<Integer>();
b3.set(new Integer(11));
String obj3 = b3.get(); // Compiler Error
System.out.println("obj3 : " + obj3);
```

- Instantiating generic class

```
Box<String> b1 = new Box<String>(); // okay

Box<String> b2 = new Box<>(); // okay -- type inference

Box<> b3 = new Box<>(); // error -- type must be given while creating generic
class reference, as reference cannot be auto-detected

Box<Object> b4 = new Box<String>(); // error

Box b5 = new Box(); // okay -- internally considered Object type -- compiler
warning "raw types"
```

```
Box<Object> b6 = new Box<Object>(); // okay -- Not usually required/used
```

Generic types naming convention

1. T : Type
2. N : Number
3. E : Element
4. K : Key
5. V : Value
6. S,U,R : Additional type param

Bounded Generic types

- Bounded generic parameter restricts data type that can be used as type argument.
- Decided by the developer of the generic class.

```
class Box<T extends Number>{
    private T obj;

    public T getObj() {
        return obj;
    }

    public void setObj(T obj) {
        this.obj = obj;
    }
}
```

- The Box<> can now be used only for the classes inherited from the Number class.

```
Box<Number> b1 = new Box<>(); // okay
Box<Boolean> b2 = new Box<>(); // error
Box<Character> b3 = new Box<>(); // error
Box<String> b4 = new Box<>(); // error
Box<Integer> b5 = new Box<>(); // okay
Box<Double> b6 = new Box<>(); // okay
Box<Date> b7 = new Box<>(); // error
Box<Object> b8 = new Box<>(); // error
```

Unbounded Generic Types

- Unbounded generic type is indicated with wild-card "?".
- Can be given while declaring `generic class reference`.
- Remember unbounded work for class references and not for class types.

```
class Box<T> {
  private T obj;

  public Box(T obj) {
    this.obj = obj;
  }

  public T get() {
    return this.obj;
  }

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

public static void printBox(Box<?> b) {
  Object obj = b.get();
  System.out.println("Box contains: " + obj);
}

Box<String> sb = new Box<String>("DAC");
printBox(sb); // okay
Box<Integer> ib = new Box<Integer>(100);
printBox(ib); // okay
Box<Date> db = new Box<Date>(new Date());
printBox(db); // okay
Box<Float> fb = new Box<Float>(200.5f);
printBox(fb); // okay
```

Upper bounded generic types

- Generic param type can be the given class or its sub-class.

```
public static void printBox(Box<? extends Number> b) {
Object obj = b.get();
System.out.println("Box contains: " + obj);
}

Box<String> sb = new Box<String>("DAC");
printBox(sb); // error
Box<Integer> ib = new Box<Integer>(100);
printBox(ib); // okay
Box<Date> db = new Box<Date>(new Date());
printBox(db); // error
Box<Float> fb = new Box<Float>(200.5);
printBox(fb); // okay
```

- Here the upper bound is set (to Number) that means all the classes that inherits Number are allowed

Lower bounded generic types

- Generic param type can be the given class or its super-class.

```
public static void printBox(Box<? super Integer> b) {
Object obj = b.get();
System.out.println("Box contains: " + obj);
}

Box<String> sb = new Box<String>("DAC");
printBox(sb); // error
Box<Integer> ib = new Box<Integer>(100);
printBox(ib); // okay
Box<Date> db = new Box<Date>(new Date());
printBox(db); // error
Box<Float> fb = new Box<Float>(200.5f);
printBox(fb); // error
Box<Number> nb = new Box<Number>(null);
printBox(nb); // okay
```

- Here the lower bound is set (to Integer) that means all the classes that are super classes of that lower bound class are allowed.

Generic Methods

- Generic methods are used to implement generic algorithms.
- Example

```
    // Not Type-safe
//  public static void printArray(Object[] arr) {
//      for (Object element : arr) {
//          System.out.println(element);
//      }
//  }

    // Type-safe
    public static <Type> void printArray(Type[] arr) {
        for (Type element : arr) {
            System.out.println(element);
        }
    }

    public static void main(String[] args) {
        String[] arr = { "Rohan", "Nilesh", "Amit" };
        printArray(arr);

        Integer[] arr2 = { 10, 20, 30, 40 };
        Program01.<Integer>printArray(arr2);

        Double[] arr3 = { 10.11, 20.12, 30.13 };
        // printArray(arr3); // type is inferred
        // Program01.<Integer>printArray(arr3); // compiler error
        Program01.<Double>printArray(arr3);// OK
}
```

Generics Limitations

1. Cannot instantiate generic types with primitive Types. Only reference types are allowed.

```
ArrayList<Integer> list = new ArrayList<Integer>(); // okay
ArrayList<int> list = new ArrayList<int>(); // compiler error
```

2. Cannot create instances of Type parameters.

```
Integer i = new Integer(11); // okay
T obj = new T(); // error
```

3. Cannot declare static fields with generic type parameters.

```
   class Box<T> {
   private T obj; // okay
   private static T object; // compiler error
   // ...
   }
```

4. Cannot Use casts or instanceof with generic Type params.

```
if(obj instanceof T) {
newobj = (T)obj;
}
```

5. Cannot Create arrays of generic parameterized Types

```
T[] arr = new T[5]; // compiler error
```

6. Cannot create, catch, or throw Objects of Parameterized Types

```
throw new T(); // compiler error
try {
// ...
} catch(T ex) { // compiler error
// ...
}
```

7. Cannot overload a method just by changing generic type. Because after erasing/removing the type param, if params of two methods are same, then it is not allowed.

```
public void printBox(Box<Integer> b) {
// ...
}
public void printBox(Box<String> b) { // compiler error
// ...
}
```