Enumerations:

* In Java, an enumeration defines a class type. An enumeration can have constructors, methods and instance variables.

* An enumeration is created using the enum keyword.

(e.g.)
```
enum fruits
{
        apple, orange, grapes, papaya
}

// Here apple, orange and so on are called enumeration constants
```

* The enumeration constants are implicitly declared as public static final member. Their type is the type of the enumeration in which they are declared.

* A variable can be created for an enumeration.

(e.g.) fruits f;

// Here f is of type fruits, the only values that it can be assigned
// are those defined by the enumeration.

f = fruits.apple;

* Two enumeration constants can be compared for equality by using == relational operator.

if (f == fruits.orange)

* An enumeration value can also be used to control a switch statement. All of the case statements must use constants from the same enum.

```
switch(f)     // implicitly specifies the enum type of case constants
{
        case apple:       …….
        case orange:      ……..
}
```

* An enumeration constant is displayed by
System.out.println(fruits.apple);

The values() and valueOf() methods:

* The values() method returns an array that contains a list of the enumeration constants.

public static enum-type[] values()

* The valueOf() method returns the enumeration constant whose value corresponds to the string passed in str.

public static enum-type valueOf(String str)

Java Enumerations are class type:

* Each enumeration constant is an object of its enumeration type.

* When a constructor is defined for an enum, the constructor is called when each enumeration constant is created.

* Each enumeration constant has its own copy of any instance variables defined by the enumeration.

* An enumeration can’t inherit another class.

* An enum cannot be a superclass. It can’t be extended.

* All enumerations automatically inherit from java.lang.Enum.

* An enumeration constant’s position in the list of constants can be obtained by using the ordinal() method.

```
final int ordinal()          // returns the ordinal value of the
                             // invoking constant.
```

* The ordinal value of two constants of the same enumeration can be compared by using the compareTo() method.

```
final int compareTo(enum-type e)
```

* If the two ordinal values are the same, then zero is returned.

* If the invoking constant has an ordinal value less than e's, then compareTo() returns a negative value.

* If the invoking constant has an ordinal value greater than e's, then a positive value is returned.

(e.g.) if (tp.compareTo(tp1) < 0)

……..

* Equality of an enumeration constant can be checked by using equals() method. Two enumeration references can be compared for equality by using ==.

Type Wrappers:

* Primitive types can be converted to object types by using the wrapper classes, which are in java.lang package.

Primitive types	Wrapper Class
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long

Converting primitive types to Object types

```
Integer ival=new Integer(i);
Float fval=new Float(f);
Double dval=new Double(d);
```

Extracting the value contained in an object

```
int i=ival.intValue();      //Object to primitive type
float f=fval.floatValue();
double d=dval.doubleValue();
```

Converting String objects to Objects

```
ival=Integer.valueOf(str);          // String to integer object
dval=Double.valueOf(str);
```

Converting primitive types to Strings

```
str=Integer.toString(i);            // primitive integer to string
str=Long.toString(l);
```

Autoboxing and auto-unboxing:

* Auto-boxing is the process by which a primitive type is automatically encapsulated into its equivalent type wrapper whenever an object of that type is needed.

* Auto-unboxing is the process by which the value of a boxed object is automatically extracted from a type wrapper when its value is needed. There is no need to call a method intValue() or doubleValue().

(e.g.) Integer iob=100; // autobox an int

int i=iob; //auto-unbox

Autoboxing and Methods:

* Autoboxing/unboxing occur when an argument is passed to a method or when a value is returned by a method.

```
class autobox
{
        static int m(Integer v)
        {
                return v;           // auto-unbox to int
        }

        public static void main(String[] args)
        {
                Integer iob=m(100);
                System.out.println(iob);
        }
}
```

Autoboxing/Unboxing occurs in expressions:

```
class autobox
{       public static void main(String[] args)
        {
                Integer iob,iob2;
                int i;

                iob=100;
                System.out.println("Original value of iob:"+iob);

                ++iob;  //unboxes iob, increments and then reboxes back
                 System.out.println("Original value of iob:"+iob);

                iob2=iob+(iob/3)  // unboxed, evaluated and reboxed
                 System.out.println("Original value of iob:"+iob);

                i=iob+(iob/3)  // unboxed, evaluated and not reboxed
                 System.out.println("Original value of iob:"+iob);
}       }
```

* Auto-unboxing allows to mix different types of numeric objects in an expression.

(e.g.) Integer iob=100;
Double dob=98.6;

dob=dob+iob;
System.out.println(“dob after expression:”+dob);

* Integer numeric objects can be used to control a switch statement.

(E.g.) Integer iob=2;
switch(iob)
{
case 1:……
case 2:……
}

Autoboxing/unboxing boolean and character values:

(e.g.)
```
Boolean b=true;
if(b)      System.out.println(“b is true”);
while(b) {……..}

Character ch=‘x’;
char ch2=ch;

System.out.println(ch2);
```

Generics:

* Parameterized types

* A class, interface or method that operates on a parameterized type is called generics.

Advantages of Generics:

1) **Type Safety**: Hold only single type of objects in generics.

2) **Type casting is not required**: No need to typecast the object.

```
(e.g.) List list=new ArrayList();        // Before generics
       list.add("hi");
       String s = (String)list.get(0);    // typecasting

       List<String> list=new ArrayList<String>();       // Before generics
       list.add("hi");
       String s = list.get(0);    // typecasting
```

3) Compile-Time Checking: Checked at compile time rather than run time

Generics work only with objects:

* Cannot pass a primitive type to a type parameter.

(e.g) Gen<int> strob=new Gen<int>(53) // Error

Generic types differ based on their type arguments:

* Reference of one specific version of a generic type is not type compatible with another version of the same generic type.

iob=strob // wrong

Syntax for Generic class:

class class-name <type-param-list>{//……..}

creating a reference:

class-name <type-arg-list> var-name = new class-name<type-arg-list>
(cons-arg-list)

* Type Parameters can’t be instantiated

* No Static member can use a type parameter. No static method can access object of type T.

* A generic class cannot extend Throwable. Cannot create exception classes.

Generic Method

- You can write a single generic method declaration that can be called with arguments of different types.
- Based on the type of the arguments passed to the generic method, compiler handles each method call appropriately.
- All generic method declaration have a type parameter section (“< >”) that precedes the method return type.
- Each parameter section contains one or more type parameter separated by commas.

- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Example

```
public class genMeth{
public static void main(String ag[]){
 Integer[] intArray={1,2,3,4};
Character[] charArray= {'h','e','l','l','o'};
print(intArray);
print(charArray);
}
public static void print(Integer [] ele)
{
for(Integer i: ele)
{System.out.println(i);}
}
}
```

```
error: method print in class genMeth cannot be
applied to given types;
print(charArray);
^
  required: Integer[]
  found: Character[]
  reason: actual argument Character[] cannot be
converted to Integer[] by method invocation
conversion
1 error
```

Example

```
public class genMeth{
public static void main(String ag[]){
 Integer[] intArray={1,2,3,4};
Character[] charArray= {'h','e','l','l','o'};
print(intArray);
print(charArray);
}
public static void print(Integer [] ele)
{
for(Integer i: ele)
{System.out.println(i);}
}
public static void print(Character [] ele)
{
for(Character i: ele)
{System.out.println(i);}
} }
```

Output

```
1
2
3
4
h
e
l
l
o
```

Example of java generic method to print array elements

```
public class genMeth1{
public static<T> void print(T [] ele)
{
for(T i: ele)
{System.out.println(i);}
}
public static void main(String ag[]){
 Integer[] intArray={1,2,3,4};
Character[] charArray= {'h','e','l','l','o'};
print(intArray);
print(charArray);
} }
```

we can create generic method that can accept any type of argument.

Output

```
1
2
3
4
h
e
l
l
o
```

Generic Class

- A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.
- As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

Example

```
public class GenericsTypeOld {
        private Object t;
        public Object get() {
                return t;
        }
        public void set(Object t) {
                this.t = t;
        }

    public static void main(String args[]){
                GenericsTypeOld type = new GenericsTypeOld();
                type.set("Pankaj");
                String str = type.get();
System.out.println(str);
        }
}
```

```
Oyutput
error: incompatible types
                String str = type.get();
                                     ^
  required: String
  found:    Object
1 error
```

Example

```
public class GenericsTypeOld {
        private Object t;
        public Object get() {
                return t;
        }
        public void set(Object t) {
                this.t = t;
        }

    public static void main(String args[]){
                GenericsTypeOld type = new GenericsTypeOld();
                type.set("Pankaj");
                String str = (String) type.get(); // Type casting required
System.out.println(str);
        }
}
```

Output
Pankaj

Generic class example

```
public class GenericsType<T> {
        T t;
        public T get(){
                return this.t;
        }
        public void set(T t1){
                this.t=t1;
        }
        public static void main(String args[]){
                GenericsType<String> type = new GenericsType<>();
                type.set("Pankaj"); //valid
                System.out.println(type.get());
                GenericsType type1 = new GenericsType(); //raw type
                type1.set("Java"); //valid
System.out.println(type1.get());
                type1.set(10); //valid and autoboxing support
System.out.println(type1.get());
        }
}
```

Notice the use of GenericsType class in the main method. We don't need to do type-casting and we can remove ClassCastException at runtime.

If we don't provide the type at the time of creation, compiler will produce a warning that "GenericsType is a raw type. References to generic type GenericsType<T> should be parameterized".

When we don't provide type, the type becomes Object and hence it's allowing both String and Integer objects but we should always try to avoid this because we will have to use typecasting while working on raw type that can produce runtime errors.

Output
Pankaj
Java
10

Generic Constructors

```
class Test<T>
{
        // An object of type T is declared
        T obj;
        Test(T obj) { this.obj = obj; } // constructor
        public T getObject() { return this.obj; }
}
class Main
{
        public static void main (String[] args)
        {
                // instance of Integer type
                Test <Integer> iObj = new Test<Integer>(15);
                System.out.println(iObj.getObject());

                // instance of String type
                Test <String> sObj = new Test<String>("Java");
                System.out.println(sObj.getObject());
        }
}
```

Output
15
Java

Generic Restrictions

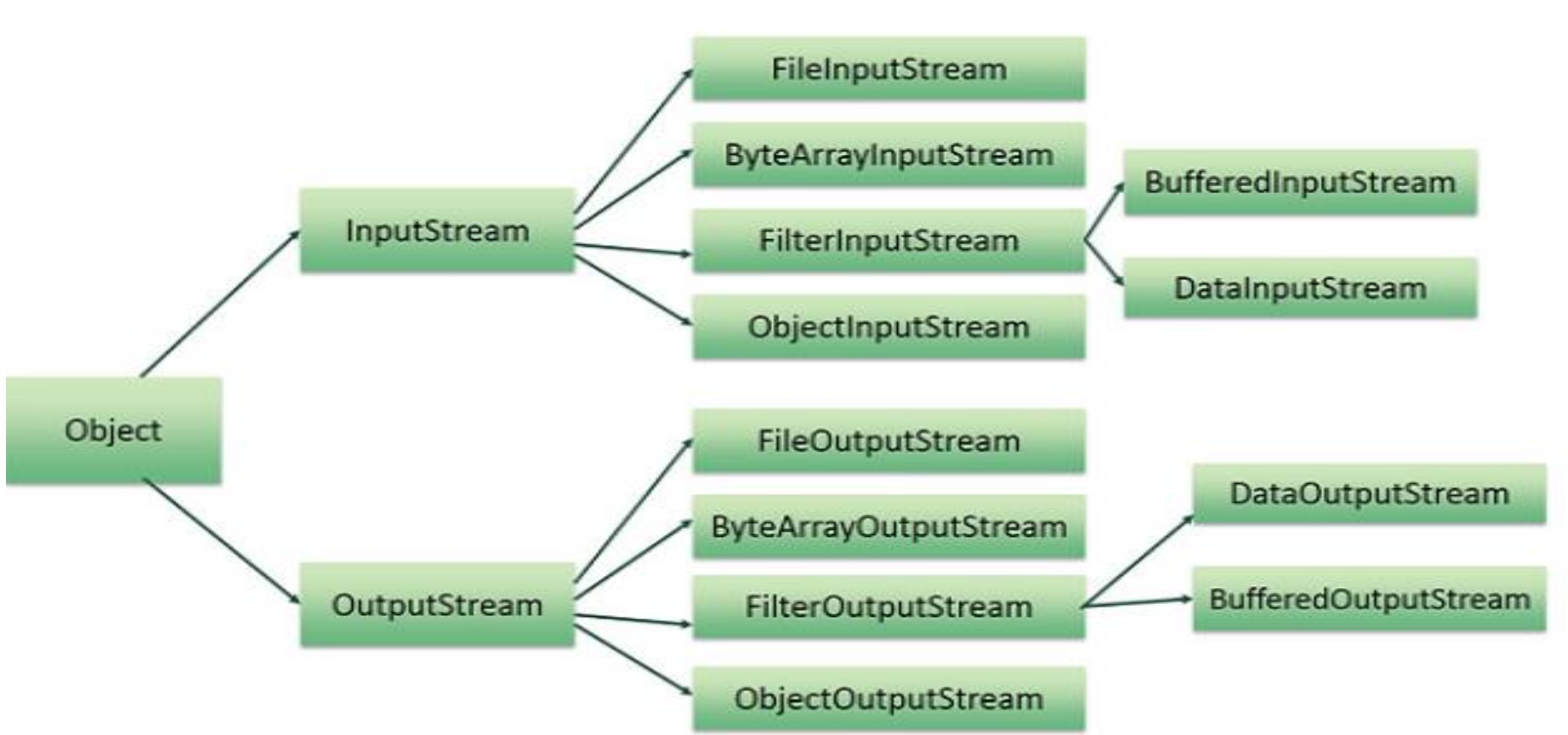
To use Java generics effectively, you must consider the following restrictions:

- Cannot Instantiate Generic Types with Primitive Types
- Cannot Create Instances of Type Parameters
- Cannot Declare Static Fields Whose Types are Type Parameters
- Cannot Use Casts or instanceof With Parameterized Types
- Cannot Create Arrays of Parameterized Types
- Cannot Create, Catch, or Throw Objects of Parameterized Types
- Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

Refer: https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html

I/O Streams

- Stream- A sequence of data.
- I/O Stream means an input source or output destination representing different types of sources e.g. disk files.
- The java.io package provides classes that allow you to convert between Unicode character streams and byte streams of non-Unicode text.


Object
InputStream
FileInputStream
ByteArrayInputStream
FilterInputStream
ObjectInputStream
BufferedInputStream
DataInputStream
OutputStream
FileOutputStream
ByteArrayOutputStream
FilterOutputStream
ObjectOutputStream
DataOutputStream
BufferedOutputStream

Byte Stream

- Java byte streams are used to perform input and output of 8-bit bytes.
- Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**.

Some important Byte Stream Classes

Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.
BufferedOutputStream	Used for Buffered Output Stream.
DataInputStream	Contains method for reading java standard datatype
DataOutputStream	An output stream that contain method for writing java standard data type
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that write to a file.
InputStream	Abstract class that describe stream input.
OutputStream	Abstract class that describe stream output.
PrintStream	Output Stream that contain print() and println() method

Character Stream

- Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

Some important Character stream classes

Stream class	Description
BufferedReader	Handles buffered input stream.
BufferedWriter	Handles buffered output stream.
FileReader	Input stream that reads from file.
FileWriter	Output stream that writes to file.
InputStreamReader	Input stream that translate byte to character
OutputStreamReader	Output stream that translate character to byte.
PrintWriter	Output Stream that contain print() and println() method.
Reader	Abstract class that define character stream input
Writer	Abstract class that define character stream output