# Core Java

## Agenda

- Exception Handling
- Generic programming
- Generic class
- Generic method

## Exception Handling

### Exception chaining

- Sometimes an exception is generated due to another exception.
- For example, database SQLException may be caused due to network problem SocketException.
- To represent this an exception can be chained/nested into another exception.
- If method's throws clause doesn't allow throwing exception of certain type, it can be nested into another (allowed) type and thrown.

### User defined exception class

- If pre-defined exception class are not suitable to represent application specific problem, then user-defined exception class should be created.
- User defined exception class may contain fields to store additional information about problem and methods to operate on them.
- Typically exception class's constructor call super class constructor to set fields like message and cause.
- If class is inherited from RuntimeException, it is used as unchecked exception. If it is inherited from Exception, it is used as checked exception.

## 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
  - using java.lang.Object class -- Non typesafe
  - using Generics -- Typesafe

### Generic Programming Using java.lang.Object

```
```Java
class Box {
    private Object obj;
    public void set(Object obj) {
        this.obj = obj;
```
```

```
    }
    public Object get() {
        return this.obj;
    }
}
```
```Java
Box b1 = new Box();
b1.set("Nilesh");
String obj1 = (String)b1.get();
System.out.println("obj1 : " + obj1);

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

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

## Generic Programming Using Generics

- Added in Java 5.0.
- Similar to templates in C++.
- We can implement
    - Generic classes
    - Generic methods
    - 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;
    public void set(TYPE obj) {
        this.obj = obj;
    }
    public TYPE get() {
        return this.obj;
```

```
    }
}
```

```
Box<String> b1 = new Box<String>();
b1.set("Nilesh");
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 -- type of object is
inferred/guessed looking at reference declaration

Box<> b3 = new Box<>(); // compiler error -- type must be given while
declaring reference

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

Box b5 = new Box(); // okay -- compiler warning "raw types" -- internally
considered as Object type (for T).
    // not recommended -- doesn't do compiler time type-checking

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

```
// T can be any type so that T is Number or its sub-class.
class Box<T extends Number> {
    private T obj;
    public T get() {
        return this.obj;
    }
    public void set(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.

```
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<Object> ob = new Box<Object>(new Object());
printBox(ob); // error
```

**Lower bounded generic types**

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

```
public static void printBox(Box<? super 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); // error
Box<Object> fb = new Box<Object>(new Object());
printBox(fb); // okay
Box<Number> nb = new Box<Number>(null);
printBox(nb); // okay
```

# Generic Programming

## Generic Methods

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

```
// non type-safe
void printArray(Object[] arr) {
    for(Object ele : arr)
        System.out.println(ele);
    System.out.println("Number of elements printed: " + arr.length);
}
```

```
// type-safe
<T> void printArray(T[] arr) {
    for(T ele : arr)
        System.out.println(ele);
    System.out.println("Number of elements printed: " + arr.length);
}
```

```
String[] arr1 = { "John", "Dagny", "Alex" };
printArray(arr1); // printArray<String> -- String type is inferred

Integer[] arr2 = { 10, 20, 30 };
printArray(arr2); // printArray<Integer> -- Integer type is inferred
```

## 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) {  // compiler error
    newobj = (T)obj;    // compiler error
}
```

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
    // ...
}
```

## Type erasure

- The generic type information is erased (not maintained) at runtime (in JVM). `Box<Integer>` and `Box<Double>` both are internally (JVM level) treated as Box objects. The field "T obj" in Box class, is treated as "Object obj".
- Because of this method overloading with genric type difference is not allowed.

```
void printBox(Box<Integer> b) { ... }
    // void printBox(Box b) { ... } <-- In JVM
void printBox(Box<Double> b) { ... } //compiler error
    // void printBox(Box b) { ... } <-- In JVM
```