Examples of Java bean and Json string

Fastjson is a Java library that can be used to convert Java Objects into their JSON representation (known as serialization). It can also be used to convert a JSON string to an equivalent Java object (known as deserialization). Due to the large scale of FastJson, we restricted our focus to the deserialization part. In order to facilitate the display of identified categories and corresponding choices, we first introduce a Java bean and a Json string.

Assume that there is a *Person* object, which stores a human's name, ID, property, family members, height, salary and marital status, etc (as shown in Figure 1(b)). Accordingly, there is a JsonStr (as shown in Figure 1(a)), which includes the keys (the field names of *Person* object) and values (the values of fields in *Person* object).

```
String JsonStr = "{\"height\":1.83," +
                 "\"property\":230000.10," +
                 "\"salary\":13000," +
                 "\"age\":32," +
                 "\"maxExpense\":25000," +
                 "\"id\":123456789," +
                 "\"isMarried\":true," +
                 "\"sex\":\"M\"," +
                 "\"birthday\":793987200000," +
                 "\"name\":\"Jack\"," +
                 "\"vehicle\":\"Car\"," +
                 "\"family\":{\"father\":\"Kim\",\"mather\":\"Mary\"}," +
                 "\"favoriteSports\":[\"PingPong\",\"hiking\"]," +
                 "\"favoriteFoods\":[\"apple\",\"banana\"]}";
```

(a) A Json string

```
@Data
public class Person {
    private float height;
    private double property;
    private short salary;
    private byte age;
    private int maxExpense;
    private long id;
    private boolean isMarried;
    private char sex;
    private Date birthday ;
    private String name ;
    private enum Vehicle{Car, Motorcycle, Bike, E_Bike}
    private Vehicle vehicle ;
    private Map<String,String> family ;
    private List<String> favoriteSports ;
    private Set<String> favoriteFoods ;
}
```

(b) A Java bean

Figure 1 Java bean and corresponding Json String

Categories and Choices

The deserialization function of FastJson is to assign the values stored in the text to a Java bean. The realization of this function mainly considers the deserialization method of different data types, therefore, the test data of Json string contains 14 commonly used data types (float, double, short, byte, int, long, boolean, char, date, string, enum, Map, List, and Set). Furthermore, we consider different usage scenarios for different types of data (Listed in Table 1).

Table 1 different usage scenarios for different types of data

#	type	options
1	Float	1a: A normal float type data, which has up to 7 significant digits
		1b: A overflowing float type data, which has at least 8 significant digits
2	Double	2a: A normal double type data, which has up to 16 significant digits
		2b: A overflowing double type data, which has at least 17 significant digits
3	Short	3a: A short type of data, and the value of that belongs to [-2^15, 2^15 -1]
4	Byte	4a: A byte type of data, and the value of that belongs to [-2^7, 2^7 -1]
5	Int	5a: A int type of data, and the value of that belongs to [-2^31, 2^31 -1]
6	Long	6a: A long type of data, and the value of that belongs to [-2^63, 2^63 -1]
7	Boolean	7a: A boolean type data, and the value of that may be true or false
8	Char	8a: A char type data, the value of that may be any character (for instance ‘a’, ‘b’, and ‘c’)
9	Date	9a: The format of the date is ISO-8601
		9b: The format of the date is yyyy-MM-dd
		9c: The format of the date is yyyy-MM-dd HH:mm:ss
		9d: The format of the date is yyyy-MM-dd HH:mm:ss.SSS
		9e: The format of the date is millisecond number
		9f: The format of the date is Millisecond number that is surrounded by double quotes
		9j: The format of the date is .NET Json
10	String	10a: A string type data
11	Enum	11a: An enum type data
12	Map	12a: a map type data
13	List	13a: a list type data
14	Set	14a: a set type data

When performing the deserialization process, the developer can call the ***parseObject*** interface, which contains 3 parameters: (1) *JsonString*, which contains the fields’ information of a Java bean. Furthermore, the string can be converted into array and binary data; (2) *Class*, which is a Java bean object; (3) *Feature*, the type of that is an enum. *Feature* can control the process of FastJson parsing a Json String. For ***parseObject*** interface, *JsonString* is a necessary parameter, while *Class* and *Feature* are not necessary parameters. The identified categories and choices are listed in Table 2.

Table 2 Definition of categories and choices for FastJson

#	Category	Choices
1	JsonString	JS-1: A Json String that is composed of 1a, 2a, 3a, 4a, 5a, 6a, 7a, 8a, 9a, 10a, 11a, 12a, 13a, and 14a (for instance \“{\“height\”:1.83}, \“property\”:230000.01,...,\“birthday\”:1985-08-20T08:26:21Z,...}\”).
		JS-2: A Json String that is composed of 1a, 2a, 3a, 4a, 5a, 6a, 7a, 8a, 9b, 10a, 11a, 12a, 13a, and 14a (for instance \“{\“height\”:1.83}, \“property\”:230000.01,...,\“birthday\”:1985-03-01,...}\”).
		JS-3: A Json String that is composed of 1a, 2a, 3a, 4a, 5a, 6a, 7a, 8a, 9c, 10a, 11a, 12a, 13a, and 14a (for instance \“{\“height\”:1.83}, \“property\”:230000.01,...,\“birthday\”:1985-03-01 19:03:01,...}\”).
		JS-4: A Json String that is composed of 1a, 2a, 3a, 4a, 5a, 6a, 7a, 8a, 9d, 10a, 11a, 12a, 13a, and 14a (for instance \“{\“height\”:1.83}, \“property\”:230000.01,...,\“birthday\”:1985-03-01 19:03:01.098,...}\”).
		JS-5: A Json String that is composed of 1a, 2a, 3a, 4a, 5a, 6a, 7a, 8a, 9e, 10a, 11a, 12a, 13a, and 14a (for instance \“{\“height\”:1.83}, \“property\”:230000.01,...,\“birthday\”:1574163958480,...}\”).
		JS-6: A Json String that is composed of 1a, 2a, 3a, 4a, 5a, 6a, 7a, 8a, 9f, 10a, 11a, 12a, 13a, and 14a (for instance \“{\“height\”:1.83}, \“property\”:230000.01,...,\“birthday\”:\“1574163958480\”,...}\”).
		JS-7: A Json String that is composed of 1a, 2a, 3a, 4a, 5a, 6a, 7a, 8a, 9j, 10a, 11a, 12a, 13a, and 14a (for instance \“{\“height\”:1.83}, \“property\”:230000.01,...,\“birthday\”:\“Wed Mar 01 00:00:00 CST 1995\”,...}\”).
		JS-8: A Json String that is composed of 1a, 2b, 3a, 4a, 5a, 6a, 7a, 8a, 9a, 10a, 11a, 12a, 13a, and 14a (for instance \“{\“height\”:1.83}, \“property\”:230000000000000000.01,...,\“birthday\”:1985-08-20T08:26:21Z,...}\”).
		JS-9: A Json String that is composed of 1a, 2b, 3a, 4a, 5a, 6a, 7a, 8a, 9b, 10a, 11a, 12a, 13a, and 14a (for instance \“{\“height\”:1.83}, \“property\”:230000000000000000.01,...,\“birthday\”:1985-03-01,...}\”).
		JS-10: A Json String that is composed of 1a, 2b, 3a, 4a, 5a, 6a, 7a, 8a, 9c, 10a, 11a, 12a, 13a, and 14a (for instance \“{\“height\”:1.83}, \“property\”:230000000000000000.01,...,\“birthday\”:1985-03-01

		09:03:01,...}\").
		JS-11: A Json String that is composed of 1a, 2b, 3a, 4a, 5a, 6a, 7a, 8a, 9d, 10a, 11a, 12a, 13a, and 14a (for instance \"{\"height\":1.83}, \"property\":230000000000000000.01,...,\"birthday\":1985-03-01 09:03:01.098,...}\").
		JS-12: A Json String that is composed of 1a, 2b, 3a, 4a, 5a, 6a, 7a, 8a, 9e, 10a, 11a, 12a, 13a, and 14a (for instance \"{\"height\":1.83}, \"property\":230000000000000000.01,...,\"birthday\": 1574163958480,...}\").
		JS-13: A Json String that is composed of 1a, 2b, 3a, 4a, 5a, 6a, 7a, 8a, 9f, 10a, 11a, 12a, 13a, and 14a (for instance \"{\"height\":1.83}, \"property\":230000000000000000.01,...,\"birthday\": \"1574163958480\",...}\").
		JS-14: A Json String that is composed of 1a, 2b, 3a, 4a, 5a, 6a, 7a, 8a, 9j, 10a, 11a, 12a, 13a, and 14a (for instance \"{\"height\":1.83}, \"property\":230000000000000000.01,...,\"birthday\": \"Wed Mar 01 00:00:00 CST 1995\",...}\").
		JS-15: A Json String that is composed of 1b, 2a, 3a, 4a, 5a, 6a, 7a, 8a, 9a, 10a, 11a, 12a, 13a, and 14a (for instance \"{\"height\":1.837645678}, \"property\":230000.01,...,\"birthday\": 1985-08-20T08:26:21Z,...}\").
		JS-16: A Json String that is composed of 1b, 2a, 3a, 4a, 5a, 6a, 7a, 8a, 9b, 10a, 11a, 12a, 13a, and 14a (for instance \"{\"height\":1.83}, \"property\":230000.01,...,\"birthday\": 1985-03-01,...}\").
		JS-17: A Json String that is composed of 1b, 2a, 3a, 4a, 5a, 6a, 7a, 8a, 9c, 10a, 11a, 12a, 13a, and 14a (for instance \"{\"height\":1.837645678}, \"property\":230000.01,...,\"birthday\": 1985-03-01 19:03:01,...}\").
		JS-18: A Json String that is composed of 1b, 2a, 3a, 4a, 5a, 6a, 7a, 8a, 9d, 10a, 11a, 12a, 13a, and 14a (for instance \"{\"height\":1.837645678}, \"property\":230000.01,...,\"birthday\":1985-03-01 09:03:01.098,...}\").
		JS-19: A Json String that is composed of 1b, 2a, 3a, 4a, 5a, 6a, 7a, 8a, 9e, 10a, 11a, 12a, 13a, and 14a (for instance \"{\"height\":1.837645678}, \"property\":230000.01,...,\"birthday\": 1574163958480,...}\").

		JS-20: A Json String that is composed of 1b, 2a, 3a, 4a, 5a, 6a, 7a, 8a, 9f, 10a, 11a, 12a, 13a, and 14a (for instance \"{\"height\":1.837645678}, \"property\":230000.01,...,\"birthday\": \"1574163958480\",...}\").
		JS-21: A Json String that is composed of 1b, 2a, 3a, 4a, 5a, 6a, 7a, 8a, 9j, 10a, 11a, 12a, 13a, and 14a (for instance \"{\"height\":1.837645678}, \"property\":230000.01,...,\"birthday\": \"Wed Mar 01 00:00:00 CST 1995\",...}\").
		JS-22: A Json String that is composed of 1b, 2b, 3a, 4a, 5a, 6a, 7a, 8a, 9a, 10a, 11a, 12a, 13a, and 14a (for instance \"{\"height\":1.837645678}, \"property\":230000000000000000.01,...,\"birthday\": 1985-08-20T08:26:21Z,...}\").
		JS-23: A Json String that is composed of 1b, 2b, 3a, 4a, 5a, 6a, 7a, 8a, 9b, 10a, 11a, 12a, 13a, and 14a (for instance \"{\"height\":1.83}, \"property\":230000000000000000.01,...,\"birthday\": 1985-03-01,...}\").
		JS-24: A Json String that is composed of 1b, 2b, 3a, 4a, 5a, 6a, 7a, 8a, 9c, 10a, 11a, 12a, 13a, and 14a (for instance \"{\"height\":1.837645678}, \"property\":230000000000000000.01,...,\"birthday\": 1985-03-01 19:03:01,...}\").
		JS-25: A Json String that is composed of 1b, 2b, 3a, 4a, 5a, 6a, 7a, 8a, 9d, 10a, 11a, 12a, 13a, and 14a (for instance \"{\"height\":1.837645678}, \"property\":230000000000000000.01,...,\"birthday\":1985-03-01 09:03:01.098,...}\").
		JS-26: A Json String that is composed of 1b, 2b, 3a, 4a, 5a, 6a, 7a, 8a, 9e, 10a, 11a, 12a, 13a, and 14a (for instance \"{\"height\":1.837645678}, \"property\":230000000000000000.01,...,\"birthday\": 1574163958480,...}\").
		JS-27: A Json String that is composed of 1b, 2b, 3a, 4a, 5a, 6a, 7a, 8a, 9f, 10a, 11a, 12a, 13a, and 14a (for instance \"{\"height\":1.837645678}, \"property\":230000.01,...,\"birthday\": \"1574163958480\",...}\").
		JS-28: A Json String that is composed of 1b, 2b, 3a, 4a, 5a, 6a, 7a, 8a, 9j, 10a, 11a, 12a, 13a, and 14a (for instance \"{\"height\":1.837645678}, \"property\":230000000000000000.01,...,\"birthday\": \"Wed

		Mar 01 00:00:00 CST 1995\",...}\").
2	JsonStirngFormat	JF-1: one single normal Json string (for instance "{\"name\":\"jack\",\"ID\":123456,...}")
		JF-2: the Json string array (for instance "[{\"name\":\"jack\",\"ID\":123456,...}",{\"name\":\"Mary\",\"ID\":123789,...}"])
		JF-3: the binary Json string
3	classType	CT-1: A specified Java bean object (for instance Person.class)
		CT-2: a TypeReference object (for instance new TypeReference<Person>(){})
4	Quotation	DoubleQuo: the field names are wrapped in double quotes (for instance "{\"name\":\"jack\",*this is a comment*\ \"ID\":123456,...}")
		SingleQuo: the field names are wrapped in single quotes (for instance "{'name':'jack','ID':123456,...}")
		NoQuo: the field names are exposed (for instance "{name:\"jack\",ID:123456,...}")
5	Comment	HaveCo: there are comments in the Json string (for instance "{\"name\":\"jack\",*this is a comment*\ \"ID\":123456,...}")
		NoCO: there is no comment in the Json String (for instance "{\"name\":\"jack\",\"ID\":123456,...}")
6	BigDecimal	TreatBD: treat numbers as BigDecimal objects
		TreatDou: treat numbers as double data
7	FieldContent	HaveContent: the Json string includes the values of all fields (for instance "{\"name\":\"jack\", \"ID\":123456,...}")
		NoContent: the Json string does not contain all the field values (for instance "{\"ID\":123456,...}")
8	Match	INMatch: ignore not match (for instance "{\"name1\":\"jack\", \"ID\":123456,...}")
		CMatch: Check whether the keys in Json string matches the fields of the Java object
9	OrderedFields	OField: the fields keep original order
		RField: the fields have random order
10	CheckSpecialKey	CSKey: check whether there are special characters in the Json string (There are too many special keys in the Java, we only test the commonly used special keys, for example "\", "$", "\r", "\b", "\n", "\t", "、", "@", "$", and "¥")
		ISKey: treat special characters as general characters
11	UseObjectArray	AArray: allow object array (allows one or more object arrays fields in a Java bean)
		NAArray: Object arrays are not allowed (one or more object arrays fields are not allowed in a Java bean)

Partition

To partition the input domain of FastJson, we first divided the categories into two groups – the independent and the dependent categories. The independent group includes all categories that contain elements which can form a valid test case on their own. Dependent categories need the presence of elements of other categories to form valid test cases. Categories 1 and 2 are classified as independent categories whilst the rest are classified as dependent. The dependent and independent categories are listed in Table 3.

Table 3 Independent and Dependent Categories for FastJson

Independent Category	**Dependent Category**
JsonString	classType
JsonStringFormat	Quotation
	Comment
	BigDecimal
	FieldContent
	Match
	OrderedFields
	CheckSpecialKey
	UseObjectArray

For partitioning, We choose independent categories and ignore dependent ones. As a consequence, we can construct 84 partitions (JsonString has 28 choices and JsonStringFormat has 3 choices).