=> If we do type casting by specifying the wrong class name then there is possibility of getting ClassCastException

```
Object obj=ctx.getBean("wdf"); // actually gives Object ref pointing to WeekDayFinder class obj ref
        //type casting
LocalDate finder=(LocalDate)obj; // So type casting with LocalDate class will throw ClassCastException
```

=> To solve these problems and to make the code as type safe code take the support of Generics
=> Generics makes the code to flexible return types , parameter types and etc...

=> Making the code as type safe code is nothing but avoid type casting related ClassCastException problem

upto spring 2.4

```
public Object getBean(String beanId) ;    //with out generic support

eg:  Object obj = ctx.getBean("wdf");
     WeekDayFinder finder=(WeekDayFinder) obj;   //Since we are doing type casting here
                                                   we can say the code is not type safe code
```

From spring 2.5 (With Generics support , there is no possibility of getting type casting problem)
===========

```
public <T>  T  getBean(String beanid ,Class<T> clazz)
       Type  return                      class name in the form of
       decl  type                        java.lang.Class obj based which
                                         the return of the method is
                                         decided

  eg1:: WeekDayFinder finder=ctx.getBean("wdf",WeekDayFinder.class);
                  (or)
    Class<WeekDayFinder> class=WeekDayFinder.class;
        WeekDayFinder finder=ctx.getBean("wdf",class);
  eg2::  LocalDate  date=ctx.getBean("ldate",LocalDate.class);       Here the code
                                                                       is type safe code
  eg3::  LocalTime time=ctx.getBean("ltime",LocalTime.class);          becoz of generics
                                       bean    passing LocalTime class
                                       id       in the form of java.lang.Class obj
                                                this makes the return type of
                                                ctx.getBean(..) as the LocalTime
                                               since method returning some LocalTime
                                               class obj there is no need of going for
                                               any typecasting
```

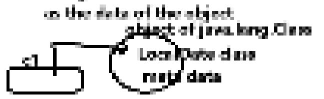
=> java.lang.Class is a pre-defined java class
=> The object of java.lang.class represents/holds class or Interface or Enum or annotation or data type in a running java app

=> How numeric data type(int,long,float,double,...) variable hold numeric values, String variables hold text data Similarly the object of java.lang.Class can hold class or interface or Enum or data type or annotation

=> The easiest way to create the object of java.lang.Class is using static "class" property of every class given by java compiler dynamically

```
Class c1=LocalDate.class;
```

=> Gives the object of java.lang.Class having LocalDate and its metadata as the data of the object



=> "class" , "length" are built-in properties in any java app

```
int a[]=new int[]{10,20,30};
int size=a.length;  //gives the size of the array

Class c2=WeekDayFinder.class;
```

obj of java.lang.Class
c2
(WeekDayFinder
class metadata)

java.lang.Class
===========
=> The object of this class holds given class/interface/Enum/annotation/data type metadata i.e basic information + additional information

=> There are multiple ways to create this object except using new operator
=> If java class is having private constructor , we can not create object for it using new operator outside of the class Definition

a) using getClass() method of java.lang.Object class

```
Date d=new Date();
Class clazz=d.getClass();
```

java.lang.Class obj
.. date class & its meta data
clazz

Date class super class
Date class implemented interfaces
Date class member variables
Date class method details
Date class constructors
and etc.

b) by using Class.forName(-) method

```
Class clazz1=Class.forName("java.util.Date");
```

Loads the given "Date" class and returns the obj of java.lang.Class having "Date" class metadata as the data of the object

java.lang.Class obj
Date class metadata
clazz1

c) using "class" property [Best]

=> This is compiler generated static property of every java class of type java.lang.Class hold current class metadata information

```
Class clazz3=LocalDate.class;
Class clazz5=WeekFinder.class;
```

obj of java.lang.Class
clazz3
LocalDate
class metadata

=> if we do type casting by specifying the wrong class name then there is possibility of getting ClassCastException

Object obj=ctx.getBean("wdf"); // actually gives Object ref pointing to WeekDayFinder class obj ref //type casting

LocalDate finder=(LocalDate)obj; // So type casting with LocalDate class will throw ClassCastException

=> To solve these problems and to make the code as type safe code take the support of Generics

=> Generics makes the code to flexible return types, parameteter types and etc...

=> Making the code as type safe code is nothing but avoid type casting related ClassCastException problem

upto spring 2.4

public Object getBean(String beanid);

//with out generic support

eg:: Object obj = ctx.getBean("wdf");

WeekDayFinder finder=(WeekDayFinder) obj; //Since we are doing type casting here

we can say the code is not type safe code

From spring 2.5 (With Generics support, there is no possibility of getting type casting problem)

========

public <T> I getBean(String beanid, Class<↓› clazz)

Type return

decl type

class name in the form of java.lang.Class obj based which the return of the method is decided

eg1:: WeekDayFinder finder=ctx.getBean("wdf",WeekDayFinder.class);

(or)

Class<WeekDayFinder> clazz=WeekDayFinder.class;

WeekDayFinder finder-ctx.getBean("wdf",clazz);

eg2:: LocalDate date=ctx.getBean("ldate", LocalDate.class); eg3:: LocalTime time=ctx.getBean("ltime", LocalTime.class);

java.lang.Class

================

bean

id

Here the code is type safe code becoz of generics passing LocalTime class in the form of java.lang.Class obj this makes the return type of ctx.getBean(-,-) as the LocalTime since method returning same LocalTime class obj there is no need of going for any typecasting

=>The object of this class holds given class/interface/Enum/annotation/data type metadata i.e basic information + additional information

=> There are multiple ways to create this object expect using new operator

=> if java class is having private constructor, we can not create object for it using new operator outside of the class Definition

(a) using getClass() method of java.lang.Object class

Date d=new Date();

Class clazz=d.getClass();

java.lang.Class obj

=>java.lang.Class is a pre-defined java class

=>The object of java.lang.class represents/holds class or interface or Enum or annotation or data type in a running java app

=> How numeric data type(int, long, float, double,...) variable hold numeric values, String variables hold text data Similarly the object of java.lang.Class can hold class or interface or Enum or data type or annotation

=>The easiest way to create the object of java.lang.Class is using static "class" property of every class given by java compiler dynamically Class c1 = LocalDate.class;

=>Gives the object of java.lang.Class having LocalDate and its metadata as the data of the object object of java.lang.Class

LocalDate class meta data

=> "class","length" are built-in properties in any java app

int a[] = new int[]{10,20,30};

int size=a.length; //gives the size of the array

Class c2=WeekDayFinder.class:

c2

obj of java lang.Class

WeekDayFinder class metadata)

clazz

..

date class & its meta data

Date class super class

Date class implemented interfaces

Date class member variables

Date class method details

Date class constructors and etc.

b) by using Class.forName(-) method

Class clazz1=Class.forName("java.util.Date");

Loads the given "Date" class and returns the obj of java.lang.Class having "Date" class metadata as the data of the object

java.lang.Class obj

Date class metadata

clazz1

c) using "class" property (Best)

=>This is compiler generated static property of every java class of type java.lang.Class

hold current class metadata information

Class clazz2=LocalDate.class;

Class clazz3-WeekFinder.class;

clazz3

w.r.t spring programming

obj of java.lang.Class

LocalDate class metadata

AnnotationConfigApplicationContext ctx=new

AnnotationConfigApplicationContext(AppConfig.class);

Here we are not passing AppConfig class name, we are actually pssing the object of java.lang.Class having AppCpnfig class meta data

Sample code giving method of certain using the methods invoked on the object of java.lang.Class

===

```
System.out.println(":
Class clazz1=WeekDayFinder.class;
System.out.println("class name::"+clazz1.getName());
=");
System.out.println("Suepr class name::"+clazz1.getSuperclass());
System.out.println("Implemented interfaces ::"+Arrays.toString(clazz1.getInterfaces()));
System.out.println("methods info ::"+Arrays.toString(clazz1.getDeclaredMethods()));
```

OUTput

======

```
class name::com.nt.sbeans.WeekDayFinder
Suepr class name::class java.lang.Object
Implemented interfaces :: [interface java.io.Serializable]
methods info::[public java.lang.String com.nt.sbeans.WeekDayFinder.showMessage(java.lang.String), public void
com.nt.sbeans.WeekDayFinder.setTime(java.time.LocalTime), public void com.nt.sbeans.WeekDayFinder.assign
Time(java.time.LocalTime), public void com.nt.sbeans.WeekDayFinder.setDate(java.time.LocalDate), public void
com.nt.sbeans.WeekDayFinder.putDate(java.time.LocalDate)]
```

=> Built-in threads in java app are :: main, gc (garbage collector)

=> Built-in streams in java app are :: System.in, System.out, System.err

=> Built-in properties in java app are :: class, length

=> Built-in reference variables/objs in java app are :: this, super

In java

======

=>"class" is a keyword

=>"class" is built-in property

=>"java.lang.Class" is pre-defined class

=>"class" is oops terminology