

How the Java code is run?

- Lets say you have created one class **A** (the file name is **A.java** of course)
- So, when you compile it, one file **A.class** will be generated. Its just a file that is being stored in the disc.
- After the compilation done, you run that **A.class** file using **java A**
 - ⌘ Now, 2 things will be generated; one object of type **Class** and one **metaspace**
 - ⌘ One variable will be created which will be of type **Class** ; its nothing but **A.class**
 - ⌘ And one metaspace, which will contain all the metadata about the class **A** like *variables, methods, etc etc*.
 - ⌘ That **A.class** is the reference of that **metaspace**.
 - ⌘ In code, we generally perform *reflection* using **A.class** so it feels like **A.class** contains everything; but it is just a reference to that **metaspace**.
- So, there will be nothing like **class A** during the run time; JVM creates **A.class** which is of type **Class** and it keeps the reference of **metaspace** which contains all the metadata about the *class A*.
- **NOTE:** **class** keyword is used to create one class; **Class** is a actual class.

```
class A {  
  
}
```

- ⌘ **class** (lowercase c) is used to define one class

```
public final class Class<T> implements java.io.Serializable,  
    GenericDeclaration,  
    Type,  
    AnnotatedElement,  
    TypeDescriptor.OfField<Class<?>>,  
    Constable {  
  
    private static final int ANNOTATION = 0x00002000;
```

- ⌘ It is an actual class which name is **Class**.
 - ⌘ This is why it is being said **“Class is itself a class, and every class in Java is an object of type Class”**
- In case of generic type, it stores the generic information in **metaspace**.

```
class A<T> { 1usage new *  
}  
class B extends A<String>{  
}
```

- ⌘ You need to pass raw type of type just like the above, I have passed **String** while inheriting **A** from **B**, otherwise how can it know which type is being passed.
- ⌘ Because, after the compilation, the **generics of objects are gone**; the only way to preserve this is **inheriting the generic class passing the raw** (in my case *String*) **type**.
- ⌘ **ParameterizedTypeReference** (abstract class) works in this way; *we* create an object of a anonymous subclass which inherit **ParameterizedTypeReference** to keep the generics preserved.
 - ⌘ It provides some useful methods by default; so instead of creating our own class to preserve the generics, we use this abstract class.

```
System.out.println(B.class);
System.out.println(B.class.getSuperclass());
System.out.println(B.class.getGenericSuperclass());
```

```
class com.alok.projects.prod_ready_features.B
class com.alok.projects.prod_ready_features.A
com.alok.projects.prod_ready_features.A<java.lang.String>
```

- ⌘ output
- ⌘ **getGenericSuperClass()** gives you the super class with generic type; it is of type **ParameterizedType**
- ⌘ **getSuperClass()** gives you the super class only without generics; it is of type **Class**.

```
Type (interface)
├─ Class
├─ ParameterizedType ← YOU ARE HERE
├─ TypeVariable
├─ WildcardType
└─ GenericArrayType
```

- ⌘ Both **Class**, and **ParameterizedType** implements **Type** (which is an interface).
- ⌘ So, if you think like the below:

```
Type c = B.class.getSuperclass();
ParameterizedType ptype = (ParameterizedType) c;
```

- ⌘ It'll not work; **why?** Read the next topic about upcasting & downcasting.

```
System.out.println(B.class.getSuperclass().instanceof Class); // true
System.out.println(B.class.getGenericSuperclass().instanceof ParameterizedType); // true
```

- ⌘ Both will be **true**.

- To see the **metaspace** of **B**, execute the command “**javap -v B**”

```
_features/B;  
d_ready_features/A<Ljava/lang/String>;
```

- You'll find something like this in the output.

Downcasting during runtime

- Lets say one interface is there named as **X** and 2 classes **A** and **B** are implementing that interface.

```
interface X { 2 usages 2 implementations new *
    void m1(); no usages 2 implementations new *
}
class A implements X{ no usages new *
    public void m1() { no usages new *
        System.out.println("m1 method in A");
    }
    public void m2() { no usages new *
        System.out.println("m2 method in A");
    }
}
class B implements X{ no usages new *
    public void m1() { no usages new *
        System.out.println("m1 method in B");
    }
    public void m3() { no usages new *
        System.out.println("m3 method in B");
    }
}
```

- **A** and **B** are having one extra functions each **m2** and **m3** respectively.
- Now its obvious, if we create a variable of type **X** and store the object of type **A** then we'll not be able to call the method **m2**.

```
X ob = new A();
ob.m2();
```

Cannot resolve method 'm2' in 'X'

Cast qualifier to 'com.alok.project'

- So there are 2 things, **variable type** and **object type**.
 - Lets say one **interface** or **class** is there (in our example **X**)
 - So, the child class may be containing **same** or **more** number of methods.
 - So, when we store an **object** of type child in a **variable** of type parent, then the variable can ignore the methods that is not in the parent class/interface.

```
// variable X
// object A
X obX = new A();
```

- ⌘ But vice versa is not true; if we have a **object of type parent** and **variable of type child**, and if we try to store that object to this variable it'll not be possible;
 - ⌘ Because, that object (of type parent) might be containing less number of fields and methods than the child.
 - ⌘ We know, to convert **parent type to child type**, we have to do downcast; but only if the **object type is same as the child**.
- Consider the following examples to get a clear picture:

```
class X { 1 usage 2 inheritors new *
    void m1() { no usages 2 overrides new *
        System.out.println("m1 in X");
    }
}
class A extends X{ no usages new *
    public void m1() { no usages new *
        System.out.println("m1 method in A");
    }
    public void m2() { no usages new *
        System.out.println("m2 method in A");
    }
}
```

```
class X { 1 usage 2
}
class A extends X{
}
```

- ⌘ This is my classes (parent is X, child is A)

```
X obX = new A();
```

- ⌘ It is completely fine; implicitly **upcasting** is happening here.
- ⌘ Object type: A, variable type: X
- ⌘ A contains all the fields and methods that is present in X.
- ⌘ So, we can access all the things from **obX**; so there is no error.

```
X obX = new A();
A obA = (A) obX;
```

- ⌘ It is also same; as we know **downcasting** has to be mentioned explicitly.
- ⌘ Object type: A, variable type: A

- ⌘ So, **obA** can access all the variables of **A**; so no error.

```
X obX = new X();  
A obA = (A) obX;
```

⌘

- ⌘ It'll give **run-time** error; not compile-time.
- ⌘ Object type: **X**, Variable type: **A**
- ⌘ **A** might be containing more number of **methods/fields** than **X**.
- ⌘ So, it'll give run-time error.

➤ So, there might be a thinking:

- ⌘ Lets say **A** and **B** are inheriting **X**.
- ⌘ We'll create one **object** of type **A**, and store that in a **variable** of type **X** (upcasting).
- ⌘ Then we'll downcast that to **B**, so now we can convert the **A** type to **B** type; it is **not possible**;

```
X obX = new A();  
B obB = (B) obX; XXXXX
```

⌘

- ⌘ Because, even if you change the **variable type**, but the actual **object** is same only;

➤ So, upcasting and downcasting is only possible when the object contains all the fields and methods that is being present in the **type of which variable is being created**.