Reflection *and* Generics

Presented by Amir Hossein Ashrafian
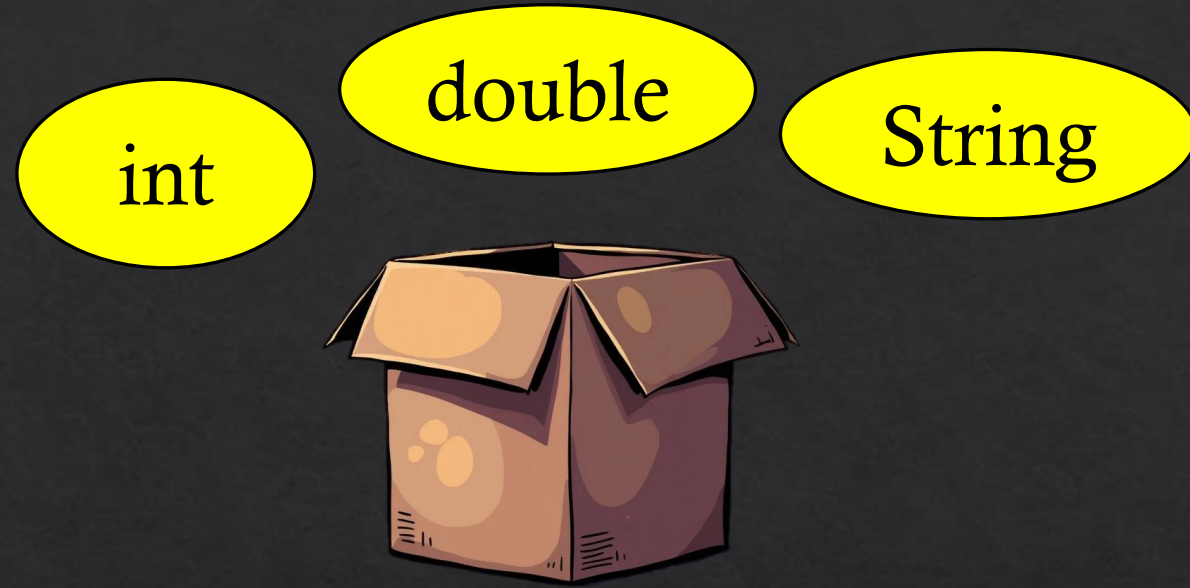
AP Fall 1403 - Dr. Mojtaba Vahidi Asl

GENERIC

int double String

Imagine you want to create a class that can store any type of data in a box.

Without generics, you'd have to write separate classes for each data type

int double String
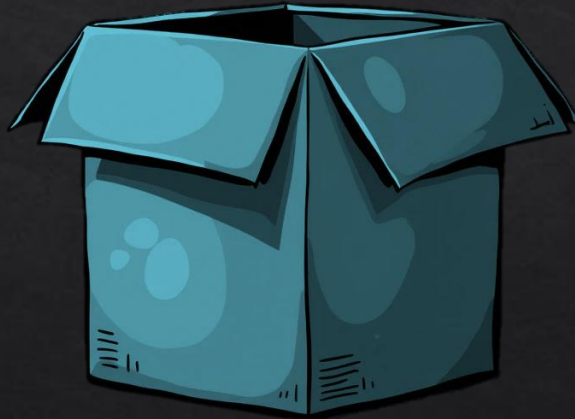
```
1  class IntegerBox {
2      private Integer value;
3
4      public void setValue(Integer value) {
5          this.value = value;
6      }
7
8      public Integer getValue() {
9          return value;
10     }
11 }
```

```
1  class StringBox {
2      private String value;
3
4      public void setValue(String value) {
5          this.value = value;
6      }
7
8      public String getValue() {
9          return value;
10     }
11 }
```

```
1  class DoubleBox {
2      private Double value;
3
4      public void setValue(Double value) {
5          this.value = value;
6      }
7
8      public Double getValue() {
9          return value;
10     }
11 }
```

which would make your code repetitive and complex.

GENERIC

Generics solve this problem by allowing you to write a single class that can work with any data type.

```
1    class Box<T> {
2        private T value;
3
4        public void setValue(T value) {
5            this.value = value;
6        }
7
8        public T getValue() {
9            return value;
10       }
11   }
```
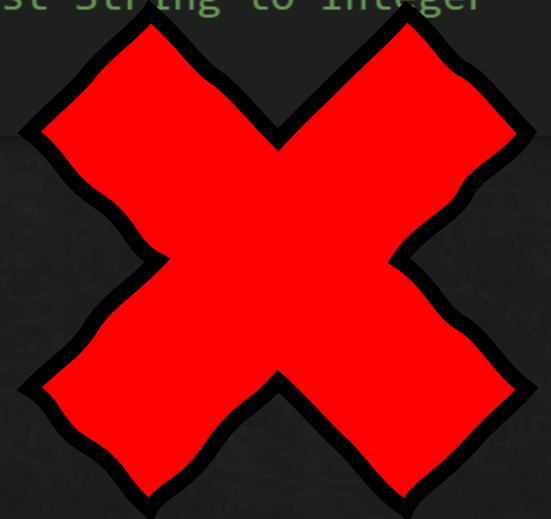
Using a sticker

Type Safety is a key advantage of using generics in Java.

```
1   ArrayList list = new ArrayList();
2   list.add(10); // Integer
3   list.add("Hello"); // String (No compile-time error)
4
5   for (Object obj : list) {
6       Integer number = (Integer) obj; // Runtime error: trying to cast String to Integer
7   }
```

Runtime Error

Type Safety is a key advantage of using generics in Java.

```java
ArrayList<Integer> list = new ArrayList<>();
list.add(10); // Integer
// list.add("Hello"); // Compile-time error: incompatible types

for (Integer number : list) {
    System.out.println(number); // Safe: No need to cast
}
```

Compile error

Therefore

Advantages of Generic :

Type Safety **Reusability**

Implementation

Generic **class** ,
method and **interface**

```java
// A generic interface with a type parameter T
interface Container<T> {
    void set(T item);
    T get();
}

// A class that implements the generic interface
class Box<T> implements Container<T> {
    private T value;

    @Override
    public void set(T value) {
        this.value = value;
    }

    @Override
    public T get() {
        return value;
    }
}
```

Limitations

Using Primitive Types in Generics:

```
1   Box<int> intBox = new Box<>();
```

Error: Cannot use primitive types

Solution: use wrapper types such as Integer , Double and Character

```
1   Box<Integer> integerBox = new Box<>();
```

Correct way

Limitations

Using Generics in Static Contexts:

```
1  public class Box<T> {
2      private T value;
3
4      public static T create() {
5          return new T();
6      }
7  }
```

because the generic type belongs to the instance, not the class.

Solution: You need to declare the static method itself as generic:

Correct way

```
1  public class Box<T> {
2      public static <U> U create(U value) {
3          return value;
4      }
5  }
```

Limitations

Bounded Types:

```java
class NumBox<T extends Number> {
    private T value;


    public void set(T value) {
        this.value = value;
    }

    public T get() {
        return value;
    }

    public static void main(String[] args) {
        NumBox<String> myBox = new NumBox<>() ;
    }
}
```

Error: The type String is not a valid substitute for the bounded parameter

Solution: Integer and Double are Number

```java
NumBox<Integer> myBox = new NumBox<>();
NumBox<Double> myBox = new NumBox<>();
```
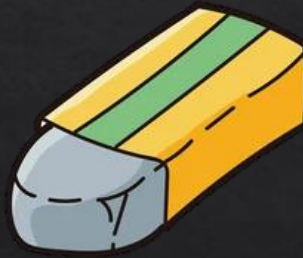
Correct way

Limitations

Type Erasure

Generic types are erased to their bounds or Object

If a type parameter doesn't have a bound, it's replaced with Object during type erasure.

```
1   class Box<T> {
2       private T value;
3
4
5       public void set(T value) {
6           this.value = value;
7       }
8
9       public T get() {
10          return value;
11      }
12
13  }
```
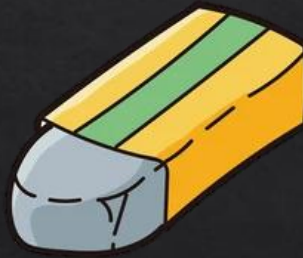
```
1   class Box {
2       private Object value;
3
4
5       public void set(Object value) {
6           this.value = value;
7       }
8
9       public Object get() {
10          return value;
11      }
12
13  }
```

Type Erasure

Bounded types are replaced with their bounds:

```java
class Box<T extends Number> {
    private T value;


    public void set(T value) {
        this.value = value;
    }

    public T get() {
        return value;
    }


}
```

```java
class Box {
    private Number value;


    public void set(Number value) {
        this.value = value;
    }

    public Number get() {
        return value;
    }


}
```

Limitations

Cannot Instantiate Generic Types

You cannot create instances of a generic type directly due to Java's **type erasure** mechanism.

```java
public class Box<T> {
    private T value;
    public Box() {
        value = new T();
    }
}
```

Error: Cannot instantiate type T

Limitations

Creating Arrays of Generic Types

In Java, you cannot directly create arrays of a generic type.

```
1   T[] array = new T[10];
```

Error: Generic array creation

Limitations

Overloading with Generic Types

You cannot overload methods based solely on their generic types, as the type information is erased at runtime, causing method signatures to look identical.

```java
1  public class Box<T> {
2      // This will not compile
3      public void doSomething(List<String> list) {}
4      public void doSomething(List<Integer> list) {}
5  }
```

Error: Method is already defined

Limitations

Generics and Exception Handling

You cannot use generic types as exceptions because Java doesn't allow a generic class to extend throwable

```
1  public class MyException<T> extends Exception {
2  }
```

Error: Cannot extend Throwable with T

Wildcard ?

➢ The wildcard is used to support **unspecified types**.

➢ It is applied when the exact generic type is **unknown at compile time.**

➢ It is also used when there is **no need to specify** the exact type.

1 Unbounded Wildcard

```
List<?> list = new ArrayList<>();
```

2 Upper Bounded Wildcard

```
List<? extends Number> list = new ArrayList<>();
```

3 Lower Bounded Wildcard

```
List<? super Number> list = new ArrayList<>();
```

are there any questions<Useful> ?

NO

YES

Scenario:

Let's assume you have a basic class called Zodiac

```java
public class Zodiac {
    private String name;
    private int age;

    public Zodiac(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void kill() {
        System.out.println("my name is " + name + ", I killed someone" );
    }
}
```

Normally, if you wanted to create an instance of Zodiac and call its method, you would write something like this:

```
1   Zodiac zodiac = new Zodiac("amirhossein", 20);
2   zodiac.kill();
```

With Reflection:

Now, using reflection, you can inspect and manipulate the class at runtime without knowing its structure during compile time.

```java
1  Class<?> zodiacClass = Zodiac.class ;
2  Constructor<?> zodiacConstructor = zodiacClass.getConstructor(String.class,int.class);
3  Object zodiac = zodiacConstructor.newInstance("amirhossein",19);
4  Method method = zodiacClass.getMethod("kill");
5  method.invoke(zodiac);
```

Dynamic loading

➢ **Dynamic loading** means loading classes or resources **during runtime**, not at compile time.

➢ It loads components only **when needed**, **saving memory** and improving performance.

➢ It adds flexibility but **can cause errors** if the component isn't found at runtime.

Now, when the time comes to load the
class into memory, what gets stored?

Class Object

Class Object

✓ **Represents runtime class information** for any Java class or interface.

✓ Allows **inspection and manipulation** of fields, methods, and constructors.

✓ **Singleton per class:** Only one class object exists for each class, shared by all instances.

Obtain the Class Object

ClassName.class

instance.getClass()

Class.forName(full Name)

Through field

Through method

Through static methoc

1

2

3

Implementation

```java
Class<?> zodiacClass1 = Zodiac.class;
Class<?> zodiacClass2 = new Zodiac("amirhossein",21).getClass();
Class<?> zodiacClass3 = Class.forName("Zodiac");
```

1 Field Class

2 Method Class

3 Constructor Class

4 Annotation Class

```java
public class Circle {
    public double radius;
    public static double PI = 3.14;
    public Circle(double radius){
        this.radius = radius ;
    }
    public double Area(){
        return PI * radius * radius ;
    }
    public double perimeter(){
        return 2*PI*radius;
    }
    @Deprecated
    public static void information(){
        System.out.println("PI = "+PI);
    }
}
```

Field Class

ClassObject.getFields();

```java
Circle circle = new Circle(2.5);
Class<?> circleClass = Class.forName("Circle");
Field[] fields = circleClass.getFields();
for(Field x : fields){
    if (x.getName().equals("radius")){
        System.out.println(x.get(circle));
    }
}
```

ClassObject.getField(fieldName);

```java
Circle circle = new Circle(2.8);
Class<?> circleClass = circle.getClass();
Field field = circleClass.getField("radius");
System.out.println(field.get(circle));
```

Method Class

ClassObject.getMethods();

```java
1  Circle circle = new Circle(2.8);
2  Class<?> circleClass = circle.getClass();
3  Method[] methods = circleClass.getMethods();
4  for(Method x : methods){
5      if (x.getName().equals("Area")){
6          x.invoke(circle);
7      }
8  }
```

ClassObject.getMethod(MethodName, args);

```java
1  Circle circle = new Circle(2.8);
2  Class<?> circleClass = circle.getClass();
3  Method method = circleClass.getMethod("Area");
4  method.invoke(circle);
```

Constructor Class

ClassObject.getConstructors();

```
1   Class circleClass = Circle.class;
2   Constructor[] constructors = circleClass.getConstructors();
3   Object circle = constructors[0].newInstance(2.5);
```

ClassObject.getConstructor(args);

```
1   Class circleClass = Circle.class;
2   Constructor constructor = circleClass.getConstructor(double.class);
3   Object circle = constructor.newInstance(2.5);
```

Annotation Class

ClassObject.getAnnotations();

```
1  Class circleClass = Circle.class;
2  Annotation[] annotations = circleClass.getAnnotations();
3  for(Annotation x : annotations){
4      System.out.println(x.annotationType());
5  }
```

MethodObject.getAnnotations();

```
1  Class circleClass = Circle.class;
2  Method method = circleClass.getMethod("information");
3  Annotation[] annotations = method.getAnnotations();
4  for(Annotation x : annotations){
5      System.out.println(x.annotationType());
6  }
```

getDeclaredMethods()

getDeclaredFields()

getMethods()

getFields()

1 Public and private

2 The inherited methods/fields are not returned.

1 Just Public

2 The public inherited method/fields are returned

Static method and field

nonstatic

Static

`methodObject.invoke(objectName,args)`

`methodObject.invoke(null,args)`

`fieldObject.get(objectName)`

`fieldObject.get(null)`

Change of access level

`methodObject` / `fieldObject` `.setAccessible(true)`

private → **public**

The scope of this access change is limited to the object that returns the specified method or field.
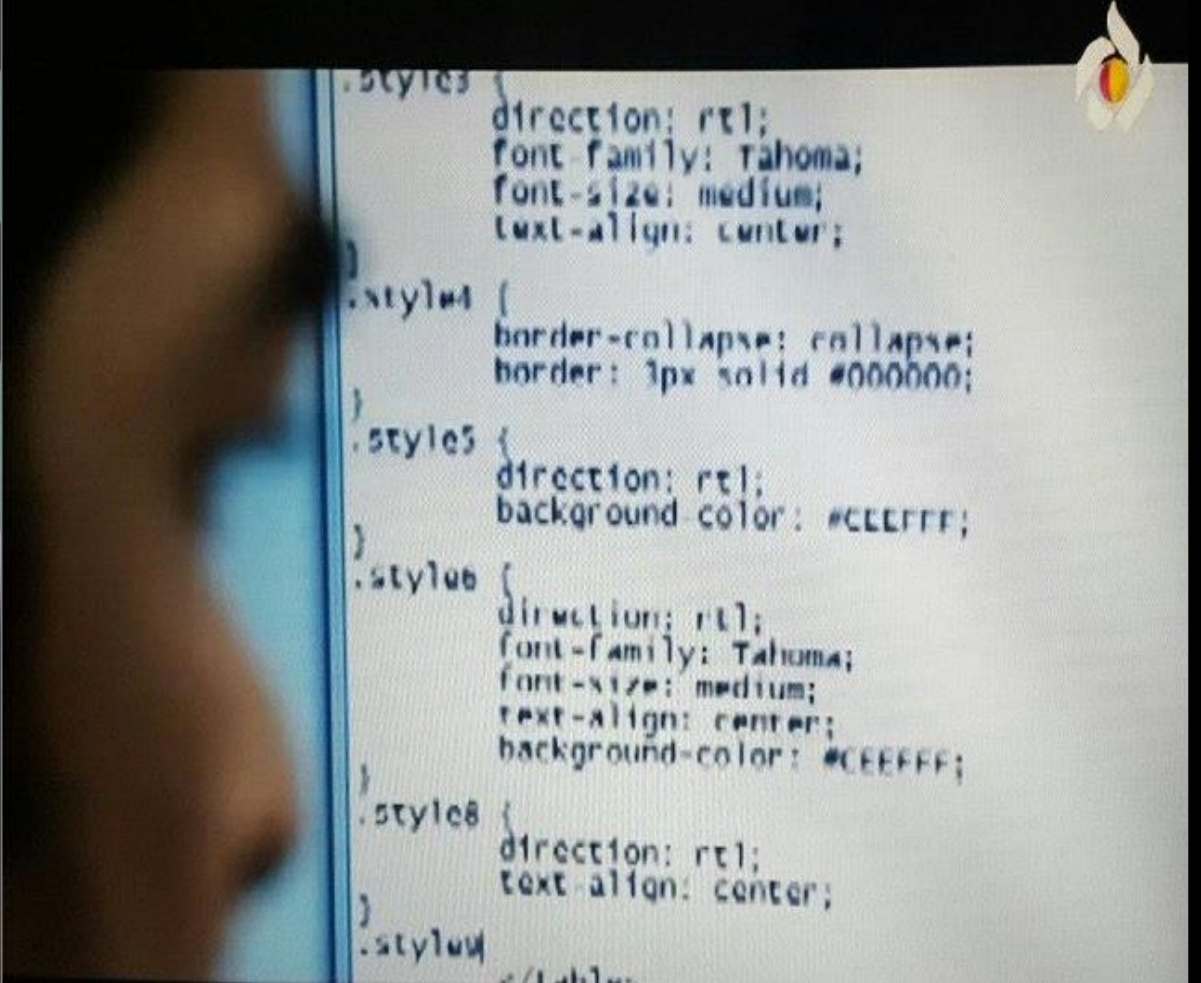
Implementation

```java
public class MeliBank {

    private static String bankName = "meli";

    private static void SendAllFundToYourAccont(String accountNumber){
        System.out.println("All funds have been sent to <"+ accountNumber +"> .");
    }
}
```

```java
Method method = MeliBank.class.getDeclaredMethod("SendAllFundToYourAccont", String.class);
method.setAccessible(true);
method.invoke(null, "6037-9981-2945-3493");
```

```java
Field field = MeliBank.class.getDeclaredField(bankName);
field.setAccessible(true);
field.set(null, "amirhosseinBank");
```

Resources

Generics:

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

https://www.baeldung.com/java-generics

https://www.geeksforgeeks.org/generics-in-java

reflection:

https://docs.oracle.com/javase/tutorial/reflect

https://www.baeldung.com/java-reflection

https://www.geeksforgeeks.org/reflection-in-java/

are there any questions ?

NO

YES

Thanks for your Attention.