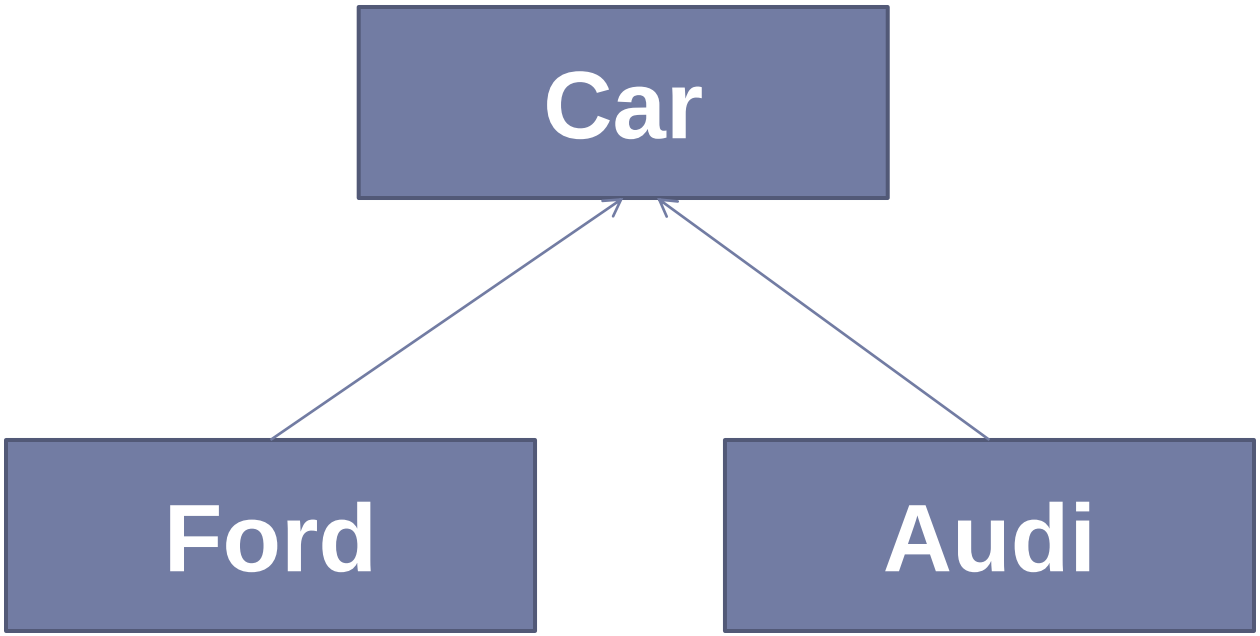
OOP TA Session 11

Generics

Inheritance as a tool for Generality

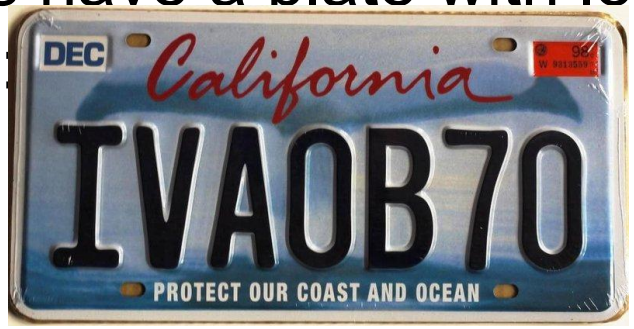
- A Simple Car hierarchy:



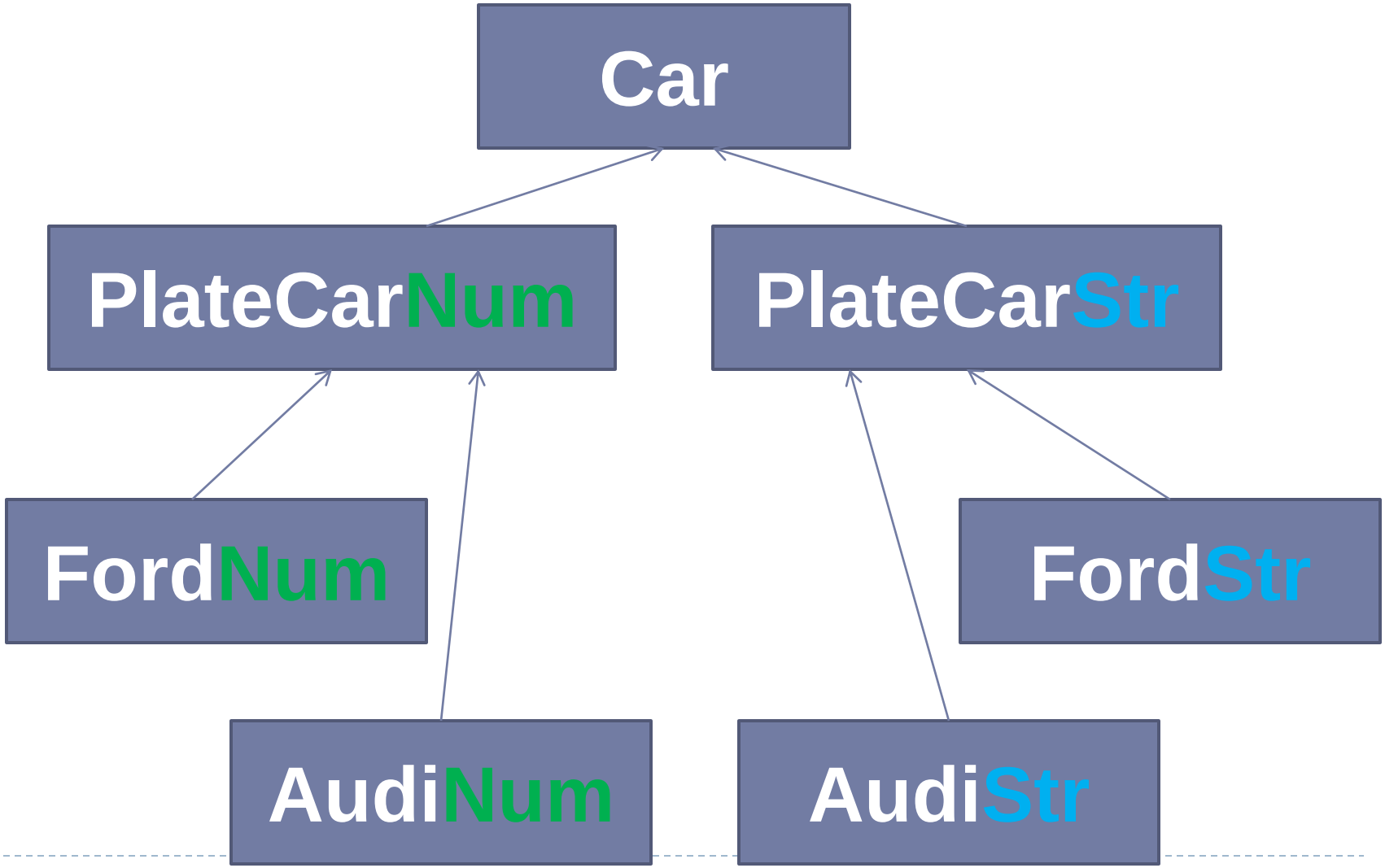
Inheritance as a tool for Generality

- Some cars have plate with digits (we'll treat as a number):

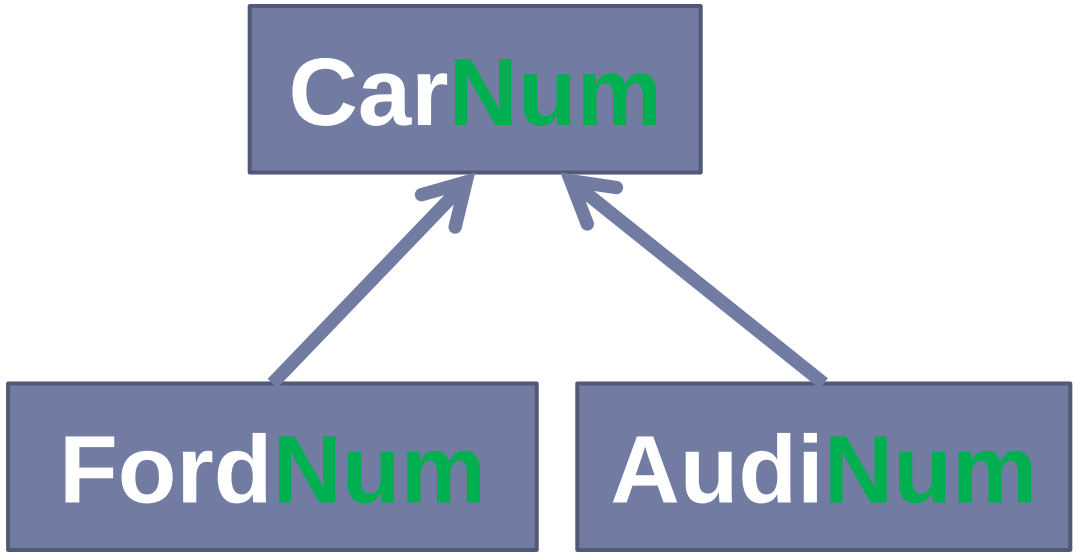
- And some cars have a plate with letters in it (we'll treat as string):

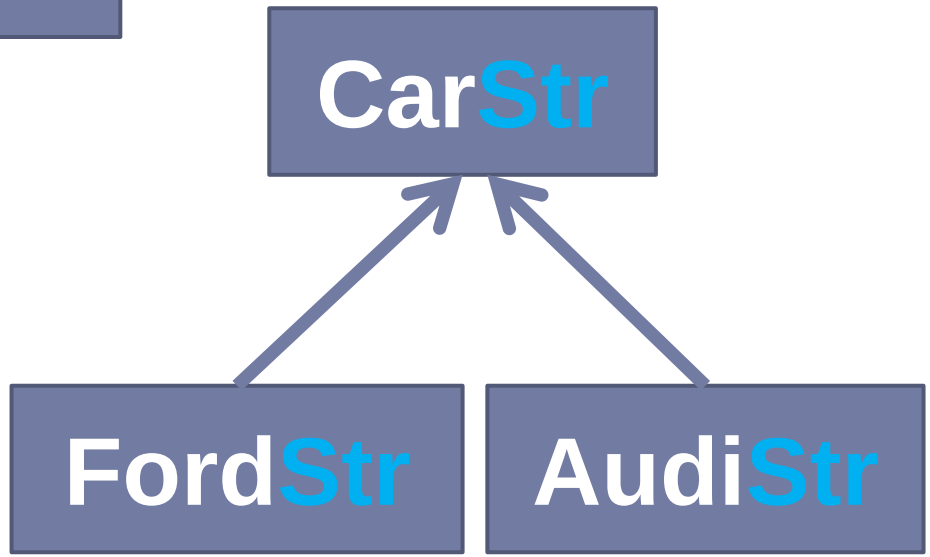


Inheritance as a tool for Generality

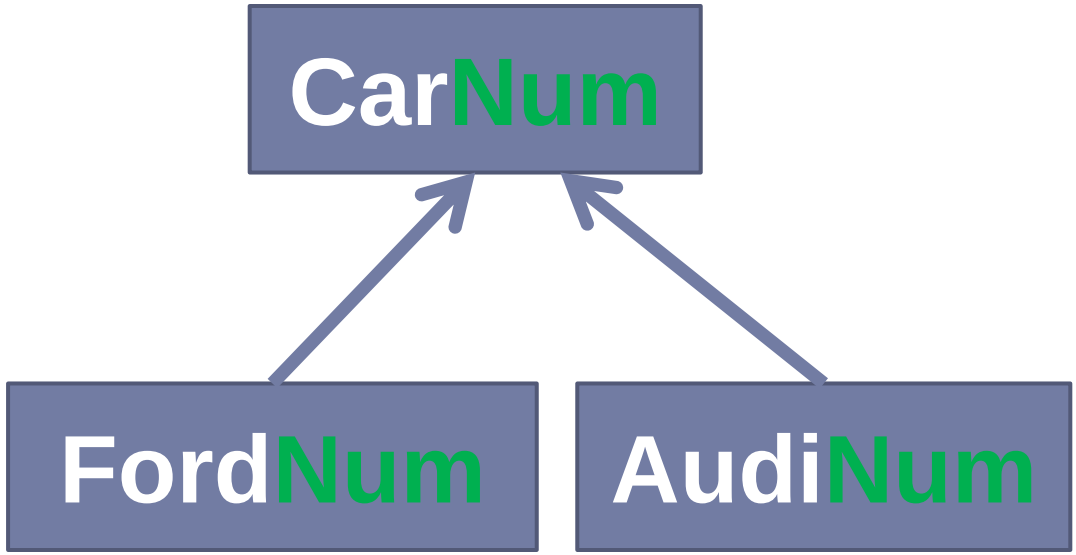


Inheritance as a tool for Generality

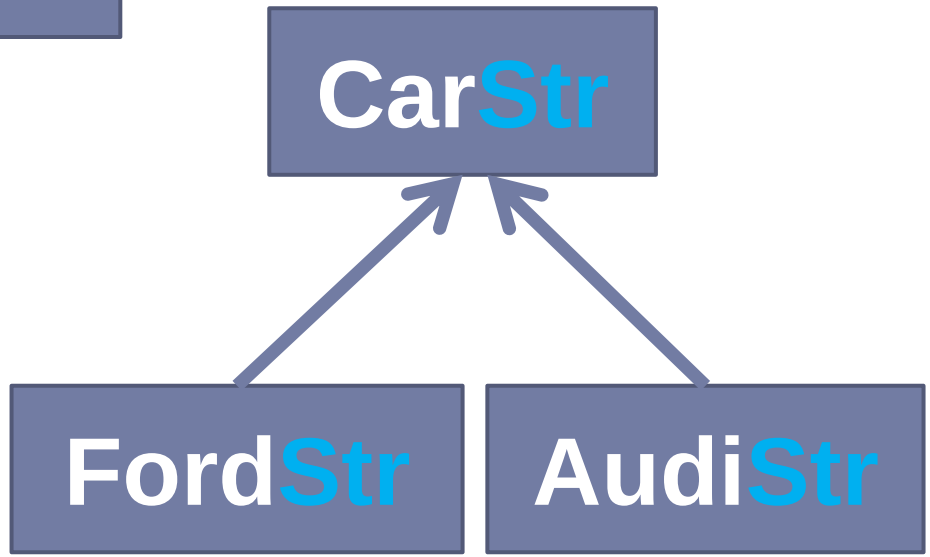




Inheritance as a tool for Generality

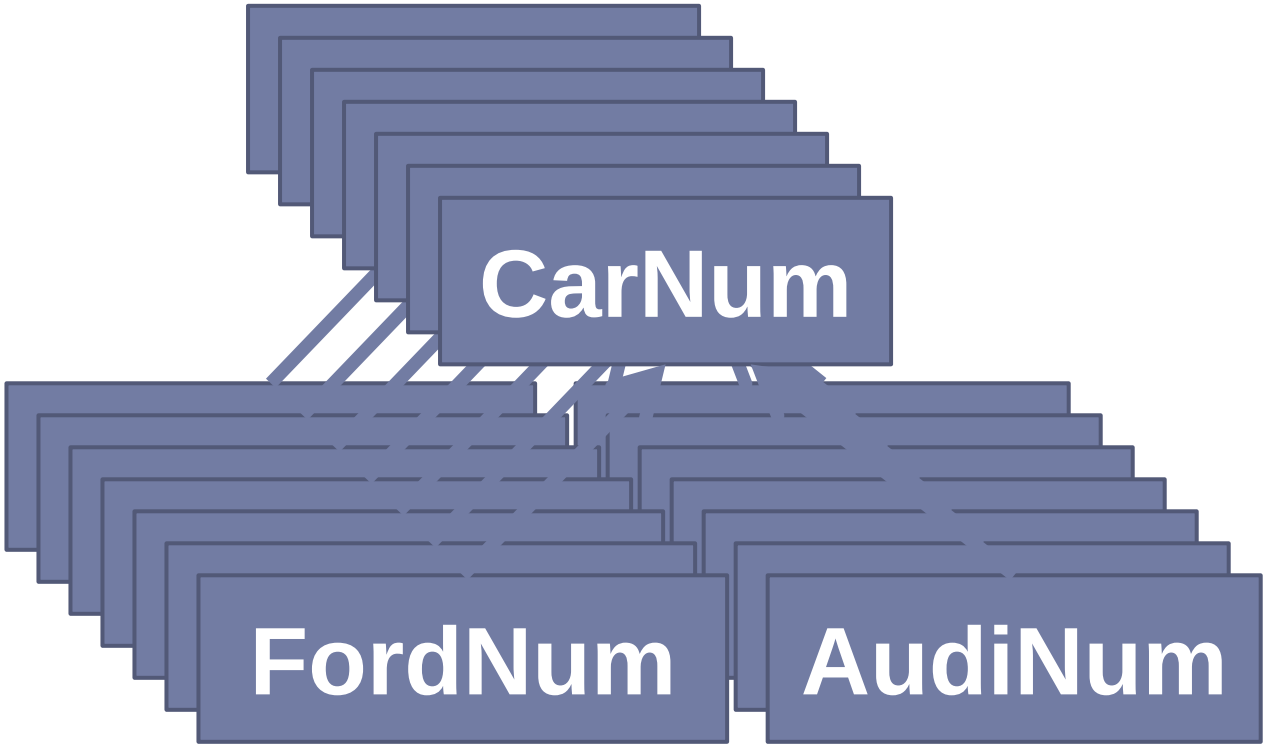


A bit better.



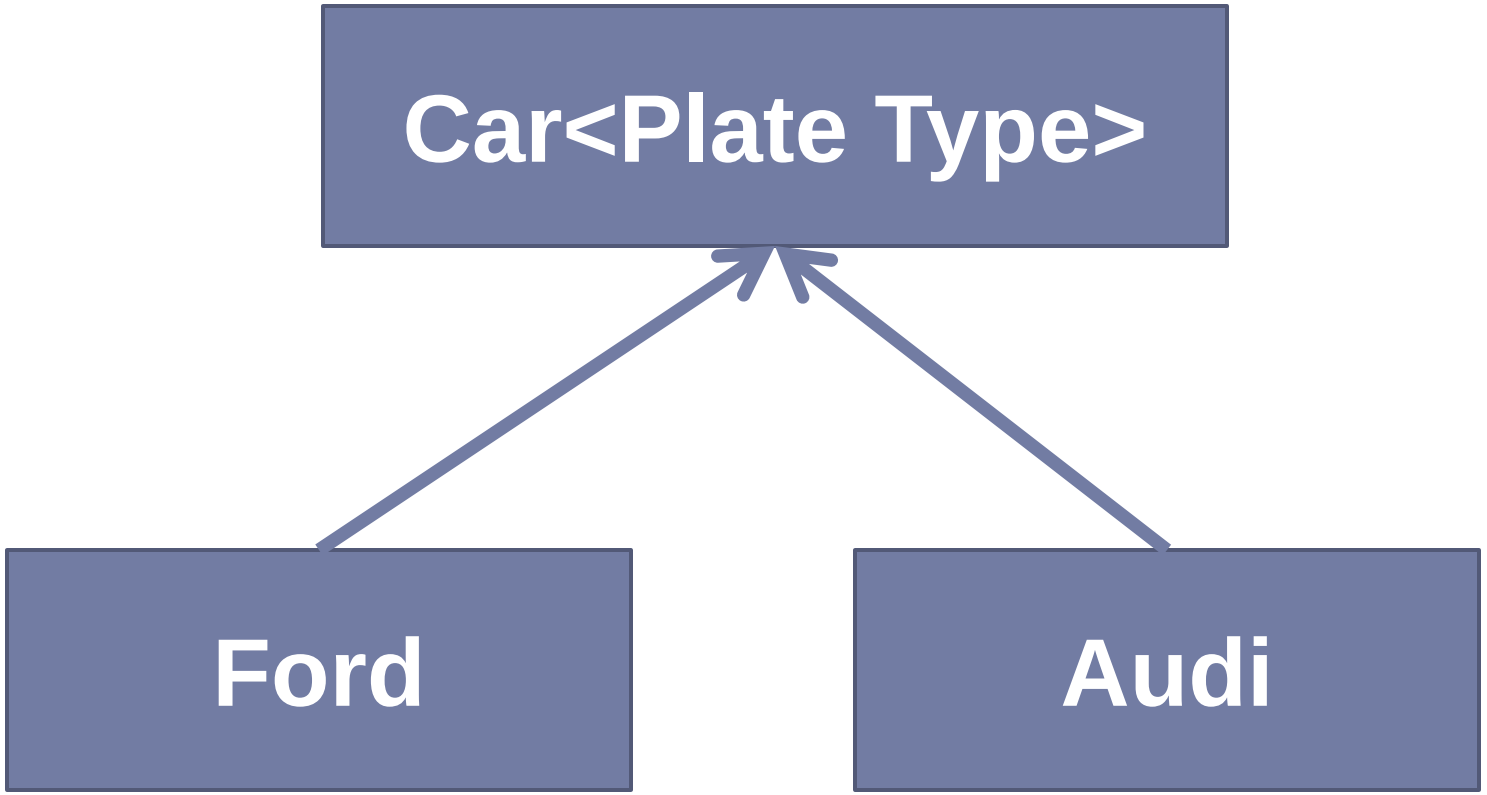
Inheritance as a tool for Generality

- What if the plate could be a lot of different types ?



Using Generics to **Parameterize** Data Types

- Using generics is equivalent to writing a “family” of classes that differ in internal type (In C++ generics are called class Templates).

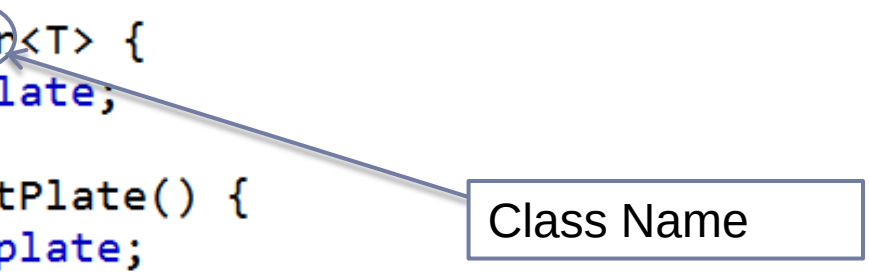


Using Generics to create a “template” of a class

```
public class Car<T> {
    private T plate;

    public T getPlate() {
        return plate;
    }

    public void setPlate(T newPlate) {
        plate = newPlate;
    }
}
```

Using Generics to create a “template” of a class

```
public class Car<T> {
    private T plate;

    public T getPlate() {
        return plate;
    }

    public void setPlate(T newPlate) {
        plate = newPlate;
    }
}
```



Using Generics to create a “template” of a class

```
public class Car<T> {
    private T plate;

    public T getPlate() {
        return plate;
    }

    public void setPlate(T newPlate) {
        plate = newPlate;
    }
}
```

Type Parameter

Using Generics to create a “template” of a class

```
public class Car<T> {
    private T plate;

    public T getPlate() {
        return plate;
    }

    public void setPlate(T newPlate) {
        plate = newPlate;
    }
}
```

So this is the template, but where do we define the specific classes?

Using Generics to create a “template” of a class

```
public class Car<T> {
    private T plate;

    public T getPlate() {
        return plate;
    }

    public void setPlate(T newPlate) {
        plate = newPlate;
    }
}
```

So this is the template, but where do we define the specific classes? **On instantiation.**

Using Generics to create a “template” of a class

```
public class Car<T> {
    private T plate;

    public T getPlate() {
        return plate;
    }

    public void setPlate(T newPlate) {
        plate = newPlate;
    }
}
```

```
Car<Integer> c = new Car<Integer>();

c.setPlate(12345);
System.out.println(c.getPlate() + 1);
```

Prints: 12346

Using Generics to create a “template” of a class

```
public class Car<T> {
    private T plate;

    public T getPlate() {
        return plate;
    }

    public void setPlate(T newPlate) {
        plate = newPlate;
    }
}
```

```
Car<Integer> c = new Car<Integer>();

c.setPlate(12345);
System.out.println(c.getPlate() + 1);
```

Prints: 12346

Using Generics to create a “template” of a class

```
public class Car<T> {
    private T plate;

    public T getPlate() {
        return plate;
    }

    public void setPlate(T newPlate) {
        plate = newPlate;
    }
}
```

```
Car<Integer> c = new Car<>();

c.setPlate(12345);
System.out.println(c.getPlate() + 1);
```

Prints: 12346

Using Generics to create a “template” of a class

```
public class Car<T> {
    private T plate;

    public T getPlate() {
        return plate;
    }

    public void setPlate(T newPlate) {
        plate = newPlate;
    }
}
```

```
Car<String> c = new Car<String>();

c.setPlate("12345");
System.out.println(c.getPlate() + 1);
```

Prints: 123451

Generics with Collections

- Naturally, Collection library use Generics to allow collections for different types:

```
List<Integer> l = new ArrayList<Integer>();

Set<String> s = new HashSet<String>();

Set<Car<String>> cars = new HashSet<Car<String>>();
```

Using of raw types

```
public class MyArray<T> {
    T[] internalArray;

    public MyArray() {}

    public MyArray(T[] arr) {
        internalArray = arr;
    }

    public T getCellAt(int i) {
        return internalArray[i];
    }
}
```

- What if we instantiate MyArray without type parameter?

```
MyArray myArr = new MyArray();
```

Using of raw types

```
public class MyArray<T> {
    T[] internalArray;

    public MyArray() {}

    public MyArray(T[] arr) {
        internalArray = arr;
    }

    public T getCellAt(int i) {
        return internalArray[i];
    }
}
```

- What if we instantiate MyArray without type parameter?

```
MyArray myArr = new MyArray();
```

T is being set to Object +
We will get a warning

Motivation – Type Safe

- A type safe program is a program in which type errors are compile-time errors and not run-time errors
  - String a = **new** Integer();

Compile-time type error

  - **Object** i = **new** Integer(5);

run-time type error

  - String b = (String)i;

Motivation – Type Safe

- Therefore:

```
List<String> a = new ArrayList<>();
a.add("elem_0");
Integer b = a.get(0);
```

Compile-time type error

```
List a = new ArrayList();
a.add("elem_0");
Integer b = (Integer)a.get(0);
```

run-time type error

Few Restrictions

- Cannot instantiate Generic Types with Primitive Types

```
List<int> l = new ArrayList<int>();
```

- Cannot create instances of Type parameters.

```
T t = new T();
```

- Cannot create arrays of generic class.

  Can break the type-safety `List<Integer>[] l = new LinkList<>[2];`

Bounded Type Parameters

- What functions the parameter type E has?

Bounded Type Parameters

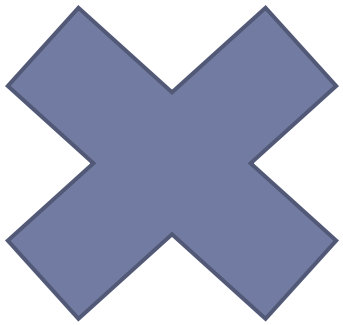
- What functions the parameter type E has?

```
public class Car<E> {
    private E engine;
    //..
    public void drive() {
        //..
        engine.ignite();
        //..
    }
}
```

Bounded Type Parameters

- What functions the parameter type E has?

```
public class Car<E> {
    private E engine;
    //..
    public void drive() {
        //..
        engine.ignite();
        //..
    }
}
```

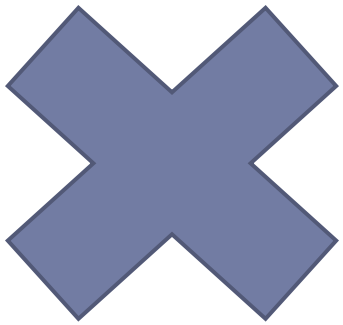


Compilation Error

Bounded Type Parameters

- What functions the parameter type E has?

```
public class Car<E> {
    private E engine;
    //..
    public void drive() {
        //..
        engine.ignite();
        //..
    }
}
```



Compilation Error

Object functions only

Bounded Type Parameters

- So what if we want to use a ***specific*** member/method of a parameterized type ?

```
public class Car<E> {
    private E engine;
    //..
    public void drive() {
        //..
        engine.ignite();
        //..
    }
}
```

Bounded Type Parameters

- So what if we want to use a ***specific*** member/method of a parameterized type ?

```
public interface Ignitable {
    public void ignite();
    //..
}
```

```
public class Car<E extends Ignitable> {
    private E engine;

    //..

    public void drive() {
        //..
        engine.ignite();
        //..
    }
}
```

Generic Static Methods

- Static variables and methods are shared among all the instances of a given class

```
class SomeType<T> {
    public static void doSomthing(T args) {
        System.out.println(args);
    }

}
```

- Therefore, It is illegal to refer to the type parameters of a type declaration in a static method, or in the declaration of a static variable

Generic Methods

- However, static and non-static method can also be generic:

```
public class SomeType {

    public <T> void doSomething(T arg) {
        System.out.println(arg);
    }

}
```

T is the method's generic parameter

```
SomeType s = new SomeType();

s.doSomething(new Car());
s.doSomething(3);
s.doSomething("Shalom");
```

Generic Recursive Definitions

- *java.util.Comparable* is a java interface used when classes want to compare their objects to other objects

- We would like that this line:

```
new Car("Audi").compareTo(new String("Mush"));
```

will result a compile error.

Generic Recursive Definitions

- *java.util.Comparable* is a java interface used when classes want to compare their objects to other objects

- We would like that this line:

```
new Car("Audi").compareTo(new String("Mush"));
```

will result a compile error.

Many methods/classes want objects to be comparable to other objects **of the same class**

Generic Recursive Definitions

- The Comparable Interface source code:

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

- Which type will we choose for the following?

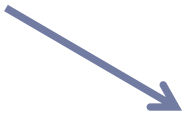
```
public class Car implements Comparable<        >
```

Generic Recursive Definitions

- The Comparable Interface source code:

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

- Which type will we choose for the following?

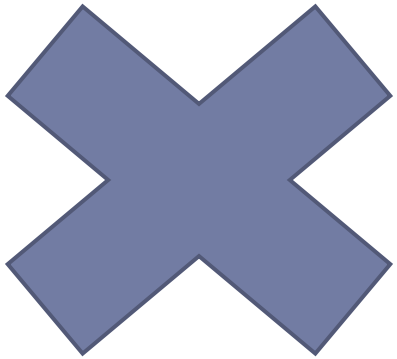


```
public class Car implements Comparable<Car>
```

Generics and Inner Classes

- Remember, the type parameter (<T>,<E> ..), get's its value when an instance is created.

```
public class Person<T> {
    private static class Brain {
        private T ineternalMember;
    }
    //..
}
```



Compile Error: Static class is not necessary bounded to an instance

Generics and Inner Classes

- However,

```
public class Person<T> {
    private static class Brain<E> {
        private E ineternalMember;
    }
    //..
}
```

```
public class Person<T> {
    private class Brain {
        private T ineternalMember;
    }
    //..
}
```