Generics

Generics

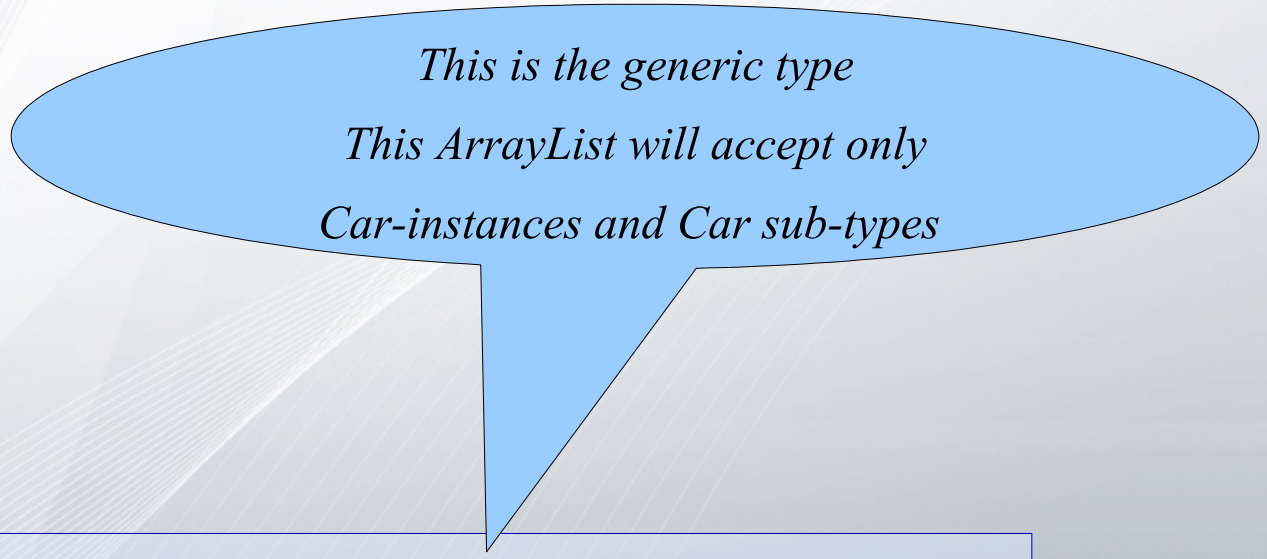
- What is a generic
- Generic types
- Generic methods
- Type erasure

Problem

- Create a demo class and a list in its main method
- Add a car instance, person instance, an array of Integers and another list
- Now try to retrieve them – problem...
- The problem grows much bigger – another developer adds instances of new types and a third developer tries to use all these objects

Solution of the problem

- For any collection (List, Set, Map, Queue, etc), we can specify the type of objects using a generic type




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

Introduction to Generics

- Generics exist in Java since Java 1.5
- Generics are similar to generics in C# or templates in C/C++
- Generics provide generality to classes/methods
- Generics work on compile time
- Generics can help avoiding runtime time errors

Problem

- Create a class VehicleStore that has a list of vehicles- either of cars or trucks or tractors
- Solution:

  Each store type may extend the VehicleStore and specify the type by itself - i.e CarVehicleStore with an array of cars and TruckVehicleStore with an array of trucks **BUT**

  - we have a lot of repeating of code
  - hard maintainability
  - class explosion (imagine we have tractors, ships, yachts, planes, bicycles, rafts and so on)

Solution

- We can generalize the Vehicle store by introducing **generic types**:

Generic type that VehicleStore uses

Declaring a list, that stores <T> objects

```
public class VehicleStore<T> {

    private List<T> vehicles;

}
```

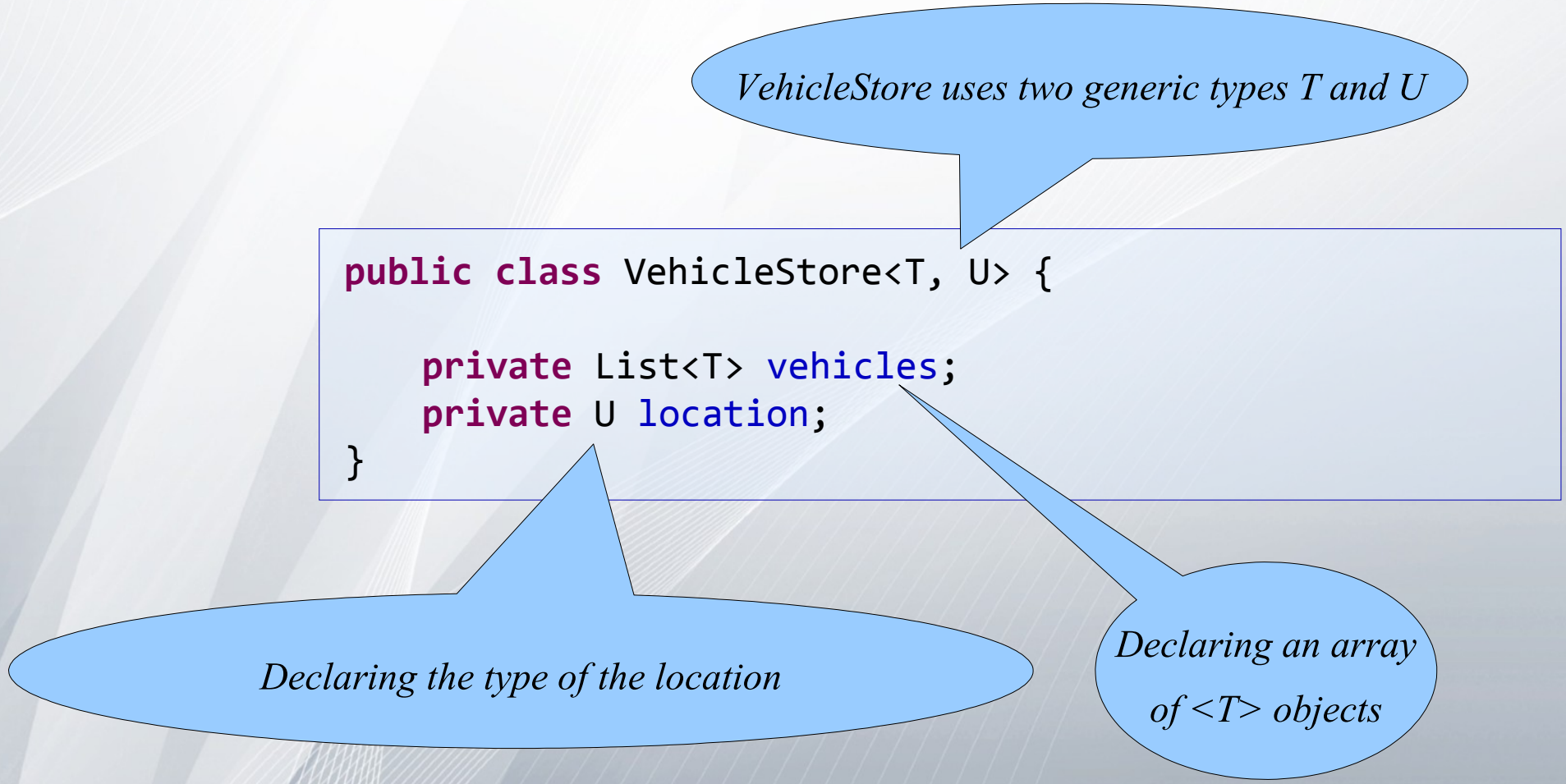
```
VehicleStore<Car> store= new VehicleStore<Car>();

VehicleStore<Car> store= new VehicleStore();
```

These brackets are called diamonds

Decalring a VehicleStore specified to use Car

Generics

- Defining more than one generic type is allowed



Generics for methods and constructors

- Generics can be used instead of a regular reference type
- Generics can be:
  - Passed as method or constructor parameters
  - Return types
  - Creating objects
- Simply: use the generic type instead of the reference type

Generics for methods and constructors

```
public VehicleStore(List<T> vehicles) {
    this.vehicles = vehicles;
}

public List<T> getVehicles() {
    return Collections.unmodifiableList(vehicles);
}

public void add(T t){
    vehicles.add(t);
}

public T getFirst(){
    if(vehicles.size()>0) {
        return vehicles.get(0);
    }
    else {
        return null;
    }
}
```

Generics for the passed type in constructor

Generic return type

Generic type parameter

Generic return type

Method calls

```
VehicleStore<Car> store = new VehicleStore<Car>();
Car honda = new Car();
Car bmw = new Car();
store.<Car>add(honda);
store.add(bmw);
Truck ford = new Truck();
store.add(ford);
```

Calling the same generic Method – shortened and most used syntax

Compile time error!

Calling add() with generic for Car with the full syntaxis

Exercise

- Create two classes: mammal and bird
- Create a generic class Cage and add an animal (either mammal or bird) to it
- Add method add() to the cage
- Create a demo class Zoo with cages. Add a lion, eagle, monkey, elephant and pinguin to them

  Hint: We should use two instances of Cage – one for birds and one for mammals

Polymorphism and generics

Vehicle is an interface

Car and Truck implement it

```
VehicleStore<Vehicle> store= new  VehicleStore<Vehicle>();
Car car = new Car();
Car bmw = new Car();
store.add(bmw);
Truck tr = new Truck();
store.add(tr);
```

This will go for both car and truck

they can be added to the vehicles

Generics can be specified for a parent class/interface and the children classes can use it

Exercise

- Extend the above functionality:
  - Create an interface Animal and make Bird and Mammal implement it
  - Specify the Cage to work for Animals

Deeper into subtyping

```
VehicleStore<Vehicle> store= new VehicleStore<Vehicle>();
VehicleStore<Car> cars = new VehicleStore<Car>();
cars = store;
store = cars;
```

This will fail into compilation error

This will fail into compilation error

Generic instance of a parent type is not a generic instance of the children type and vice versa

Problem

- And now create a cage and try to specify it for cars

  problem - the cages are not for cars

- We can restrict the generic to a type and all its subtypes

  For example the interface Vehicle

```
public class VehicleStore<T extends Vehicle> {

}
```

T is of type Vehicle or its subtypes.

ALWAYS use extends disregarding Whether Vehicle is a class or interface

Bounded types

- The restriction in generics can be applied on classes and methods
- Such types are called bounded types

```
public <U extends Vehicle> boolean isContained(U u)
{
    return vehicles.contains(u);
}
```

The check can be made against Vehicles only

```
store.contains(tv);
store.contains(car);
```

Tv is an istance of TVSet – class
Nothing to do with vehicles
This results to compilation error

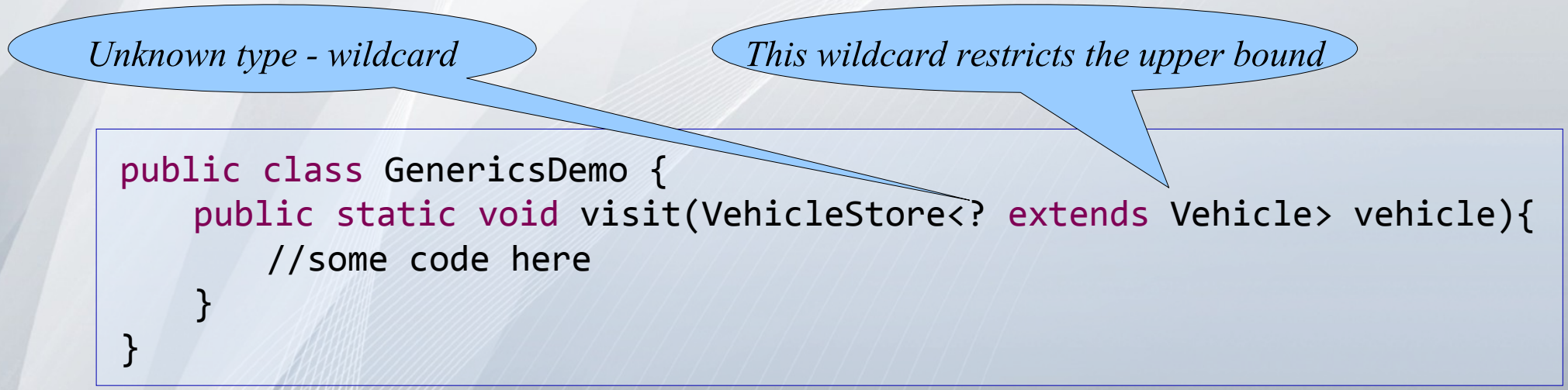
Car is a vehicle and this call is legal

Exercise

- Specify the Cage to be for Animals only
- And add a method to check whether an animal is contained in the cage

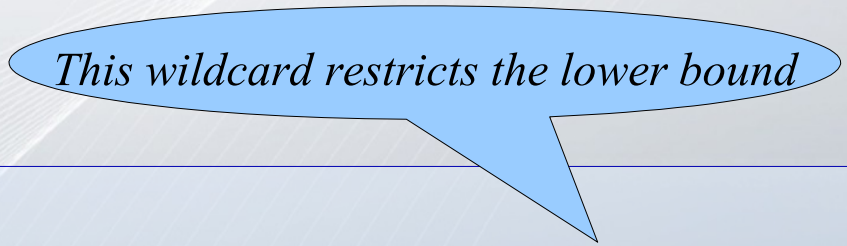
Unbounded types - Wildcards

- Bounded types are known types – these known types are in the generic class
- Although we may want to use generics outside of the generic class
- For the unknown types there are wildcards



Unbounded types - Wildcards

- *Extends* provides the upper bound of a wildcard
- *Super* provides the lower bound




```
public class GenericsDemo {
    public static void visit(VehicleStore<? super Vehicle> vehicle){
        //some code here
    }
}
```

Exercise

- Create a method cleanCage() in the class Zoo and apply it for Animals

Type erasure

- Try to create a new instance of the generic type
- Try to create an array of the generic type

Impossible

- Generics work at compile time – when the code is being compiled all information about the generic type is removed:

  List<Car> list = new ArrayList<Car>() is translated to

  List list = new ArrayList()

Type erasure

```
T t= new T();
t instanceof T;
E obj = (E)new Object();
E array = new E[10];
```

This code will never compile.

Neither of these statements will

The reason is type erasure –

at compile time all „T“ are removed

Summary