



Generics

Generics

- **Generic Type Examples**
 - List<Car>
 - ArrayList<Fruit>
- ..
 - Generics give Java code
 - ..
 - Code is written once, but handles different types. Selection is done at compile-time.
- **It's different than Runtime Polymorphism**
 - .. gives runtime polymorphism
 - Code is written once, but handles different types. Selection is done at **run-time**.

Generics and Different Types

- **Generics handle any object type**
 - Code written with a generic can handle **any** type of object, not just ones related via inheritance.
 - The same **ArrayList** code can make:
 - an **ArrayList** of **Cars**, or
 - an **ArrayList** of **Fruit**,
 - ... etc.
- Once created, an object of type **ArrayList<Car>** cannot handle **Fruit**:
 - Compiler knows an **ArrayList<Car>** object holds **Cars**

```
ArrayList<Car> myCars = new ArrayList<>();  
Car firstCar = myCars.get(0);
```

Generic Method

- **Generic Method**
 - A method which has a..
 - It can use this type parameter as a regular type
- **Can call a generic method with any type of object**
 - Compiler ensures that it preserves the type

T is the
type parameter

```
public <T> List<T> makeIntoList(T obj1, T obj2) {  
    List<T> myList = new ArrayList<>();  
    myList.add(obj1);  
    myList.add(obj2);  
    return myList;  
}
```

Generic Method Example

```
public class GenericMethod {  
  
    public static <T> List<T> makeIntoList(T obj1, T obj2){  
        List<T> myList = new ArrayList<>();  
        myList.add(obj1);  
        myList.add(obj2);  
        return myList;  
    }  
  
    public static void main(String[] args) {  
  
        List<String> myStrings = makeIntoList("Hello", "World");  
        List<Integer> myIntegers = makeIntoList(5, 10);  
  
        Car car1 = new Car("Forester", 2050);  
        Car car2 = new Car("Model T", 1920);  
        List<Car> myCars = makeIntoList(car1, car2);  
    }  
}
```

Generic Class

- Generic Classes

have a **type parameter** for the whole class

```
public class ShippingCrate<T> {
    private T item;

    public ShippingCrate(T item) {
        this.item = item;
    }

    public T getItem() {
        return item;
    }

    public void printLabel() {
        System.out.println("One shipping crate containing: ");
        System.out.println("    " + item.toString());
    }
}
```

Generic Interfaces

- **Generic Interfaces**

- Like a class, has a type parameter for the whole interface.
- Very useful to make flexible code

- Can use

..
for client code to provide an implementation which fills in a part of our algorithm.

- Our object is then typed to the type the client needs.

```
// Create an object that, given an item,  
// provides the description you want.  
public interface Describer<T> {  
    String getDescription(T item);  
}
```

Arrays and Generics

(Covariant vs Invariant Types)
(Reified vs Type Erasure)

Arrays and Generics Don't Play Well

• ..

```
// Cannot create an array of anything generic  
new List<String>[2];  
new List<E>[2];
```

Generic array creation not allowed

Bad

```
// You *CAN* create a generic of arrays:  
List<String[]> myStrings = new ArrayList<String[]>();  
List<E[]> myThings = new ArrayList<E[]>();
```

OK

• Why?

- Java's type system guarantees that when using generics Java will not throw a runtime cast exceptions.
- If you have a collection of **TeddyBears**, **get()** must always gives you a teddy bear! ... *but it's complicated...*

```
List<TeddyBear> bears = new ArrayList<>();  
TeddyBear myBear = bears.get(0);
```

Understanding Covariant

- Let **A** be a subtype of **B**.
- **Variance** refers to if a language allows an array (or list) of **A** to be used instead of an array (or list) of **B**

```
Object[] data = new Long[10];
```

Variance:
Some languages allow this; others do not.

- Java arrays are **covariant**:

– ..

- You can write code to use **B[]**, and instead pass it **A[]**!

```
public class ArrayCovariant {
    static int countLength(Object[] data) {
        return data.length;
    }

    public static void main(String[] args) {
        String[] strings = {"Broken", "Type"};
        Object[] objects = strings;
        System.out.println(countLength(strings));
        System.out.println(countLength(objects));
    }
}
```

Understanding Covariant (cont)

- With **covariant** types, must..
 - Since it will accept a different type, it's possible to write code that violates the type system.

```
public class ArrayCovariant {  
  
    public static void main(String[] args) {  
        String[] strings = {"Broken", "Type"};  
        Object[] objects = strings;    // OK; Covariant!  
  
        // Generates runtime exception: ArrayStoreException  
        objects[0] = 5;  
    }  
}
```

- **objects** is of type **Object[]**, which is a subtype of **String[]**
- **objects[]** is of type **Object[]**, so compiler lets us put in an **Integer**
- But, the array only stores **String**, so must runtime check!

Understanding Invariant

- Generics (such as lists) are **invariant**:
 - ..
 - If you need a **List**, you cannot use **List<A>** instead.
- Compile time checking of types!

```
// Demonstrate invariant type  
// **Will not compile**  
ArrayList<Object> data = new ArrayList<String>();
```

Required type: ArrayList <Object>

Provided: ArrayList <String>

Covariant vs Invariant Types

- Covariant seems more flexible; is it better?
 - ..
 - With **covariant** types, we need to do **runtime type** checking.
 - With **invariant** types, the **compile** does all our checking!

```
// With covariant arrays, this is a _runtime_ error  
Object[] objectArray = new Long[1];  
objectArray[0] = "I don't fit in!";    // Runtime error  
  
// With invariant generics, this is a _compile time_ error  
List<Object> objectList = new ArrayList<Long>(); // Compile time error!  
objectList.add("I don't fit in!");
```

- Heuristic:
 - Advice given in Effective Java (3rd ed) by Joshua Bloch

Type Erasure

- Java **generics** only know about their type parameters at **compile time**.
 - ..
 - Since the type system is strong for generics, the compile is able to enforce all type constraints so that we don't need to check at runtime.
- **Reification**
 - ..
arrays know and enforce their element types at runtime.
 - ..
generics do not enforce their element types at runtime.

Summary

- **Inheritance**
 - Provides **run-time polymorphism**
- **Generic**
 - Provides **compile-time polymorphism**
 - **Generic methods**
Written once, work on any (specific) type of object
 - **Generic class**
Handle any (specific) type of object
 - **Generic interface**
Provides flexible ability to the strategy pattern
- **Arrays** are **covariant** types (**reified**);
Generics are **invariant** types (**type-erasure**).