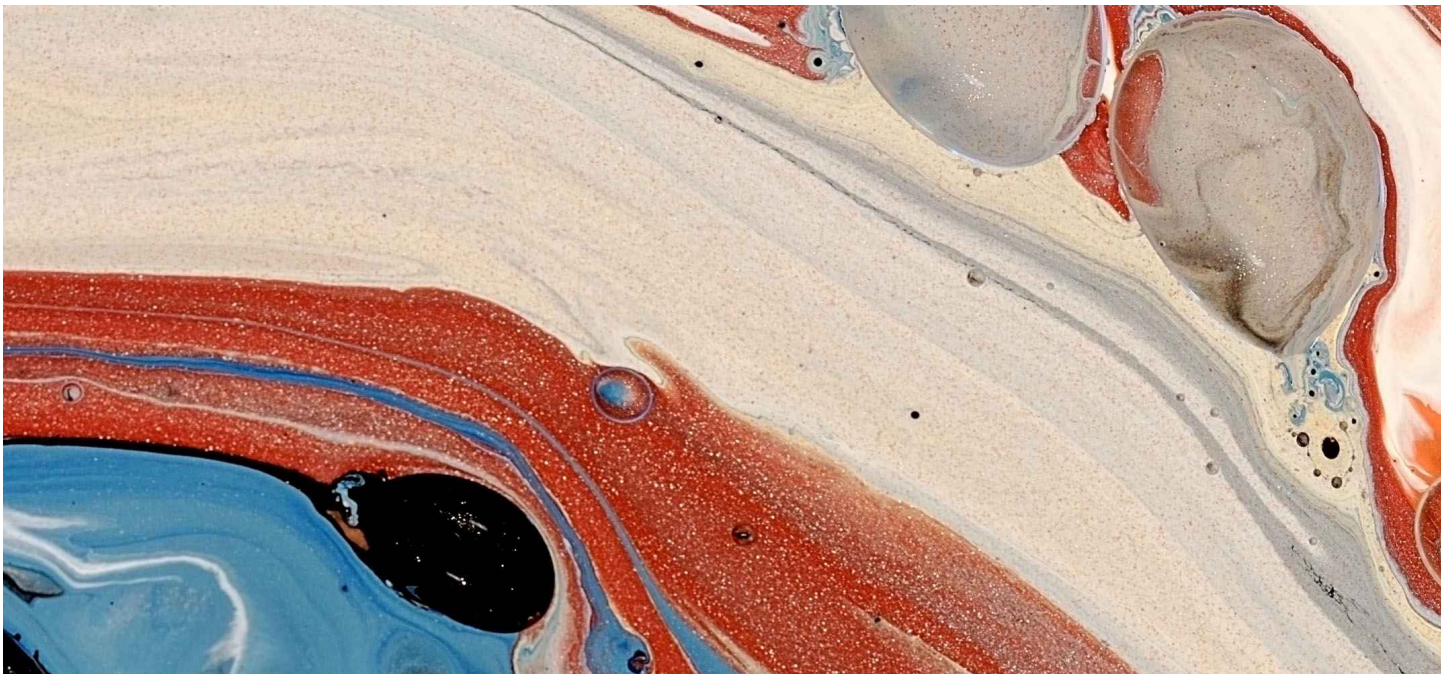


# 25-Java 5: Generics

*Author: Vincent Lau*



## Learning Objectives

- Explain the purpose of using Generic types
- Create Java classes that use Generics
- Create and use Generic methods
- Use upper bounds and wildcards with Generics
- Understand the concept of type erasure

## Overview

### Why Generic? - Before Java 5 (1.5)

- **Prior to Java 1.5, the Collections API supported only raw types** - there was no way for type arguments to be parameterized when constructing a collection.

```
// Before Java 1.5, no type arguments  to be parameterized
// Forget this syntax, just show you the history
// Without generics
List customers = new ArrayList();
customers.add(new Object());
customers.add("customers");
customers.add(new Integer(99));
```

- This allowed any type to be added and **led to potential casting exceptions at runtime.**

## After Java 5 (1.5)

In Java 1.5, `Generics` were introduced with the aim of **reducing bugs** and adding an extra layer of abstraction over types. It **allowed us to parameterize the type arguments for classes,** including those in the Collections API - when declaring and constructing objects.

## Benefits of Generics

- In a nutshell, generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods. Type parameters provide a way for you to re-use the same code with different inputs.

```
public interface List<E> extends Collection<E> {
                // interface methods
}
```

- Code that uses generics has many benefits over non-generic code:

### Stronger Type Checks at Compile Time

  - A Java compiler applies strong type checking to generic code and raises errors if the code violates type safety. **Fixing compile-time errors is much easier than fixing runtime errors, which can be difficult to find.**

### Elimination of casts

```
// Without generics
List list = new ArrayList();
```

```
list.add("abc");
// For the get method, it doesn't know what type will be returned
// So we have to cast String
String s = (String) list.get(0); // possible to have runtime error

// With generics, after Java 1.5
// List<String> list = new ArrayList<String>(); // before Java 1.7
List<String> list = new ArrayList<>(); // after Java 1.7
list.add("hello");
String s = list.get(0);   // no cast is required

```

- **Enabling programmers to implement generic algorithms**
  - By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type-safe and easier to read.

# Diamond Operator

---

## Before Java 7 (1.7)

```
// specify the parameterized type in the ArrayList constructor
List<String> customers = new ArrayList<String>();
```

- Before the introduction of Diamond Operator in Java 1.7, we have to specify the parameterized type in the *ArrayList* constructor, which can be somewhat duplicated.

```
// Just demonstrate how duplicated and unnecessary
// In the real world ... should not have such complicated structure
Map<String, List<Map<String, Map<String, Integer>>>> customers
 = new HashMap<String, List<Map<String, Map<String, Integer>>>>();
```

- The reason for this approach is that **raw types still exist for the sake of backward compatbility**, so the compiler needs to differentiate between the use of raw types and generics:

```
// Backward Compatbility, we can still use the belows:
```

```
2 List<String> generics = new ArrayList<String>(); // Before Java 1.7,  specify
  the parameterized type in the ArrayList constructor
3 List<String> raws = new ArrayList<>(); // After Java 1.7
```

- Even though the compiler still allows us to use raw types in the constructor, it will prompt us with a **warning message**:

```
1 ArrayList is a raw type. References to generic type ArrayList<E> should be
  parameterized
```

## After Java 7 (1.7)

- The diamond operator - introduced in Java 1.7, **adds type inference and reduces the verbosity in the assignments - when using generics.**
- The Java 1.7 compiler's type inference feature **determines the most suitable constructor declaration that matches the invocation**.
- In small programs, this might seem like a trivial addition. But in larger programs, this can add significant robustness and makes the program easier to read.

```
1 List<String> strings = new ArrayList<>();
```

# Generic Types

- A *generic type* is a generic class or interface that is parameterized over types.
- A type variable can be any **non-primitive** type: any class type, any interface type, any array type, or even another type variable.
- The same technique can be applied to create generic interfaces.

```
1 public class Box<T> {
2     // T stands for "Type", naming convention only.
3     // You can use other Character, such as S.
4     private T t;
5
6     public void set(T t) {
7         this.t = t;
8     }
9
```

```
10      public T get() {
11          return t;
12      }
13
14      public static void main(String[] args) {
15
16          Box<Integer> integerBox = new Box<>();
17          integerBox.set(1);
18          System.out.println(integerBox.get());  // prints 1
19
20          Box<String> stringBox = new Box<>();
21          stringBox.set("hello");
22          System.out.println(stringBox.get());  // prints "hello"
23
24          Box<int[]> intArrayBox = new Box<>();
25          intArrayBox.set(new int[]{1, 2, 3});
26          System.out.println(Arrays.toString(intArrayBox.get()));  // prints
    [1,2,3]
27
28          Box<int> intBox = new Box<>();  // compilation error
29      }
30  }
```

## Type Parameter Naming Conventions

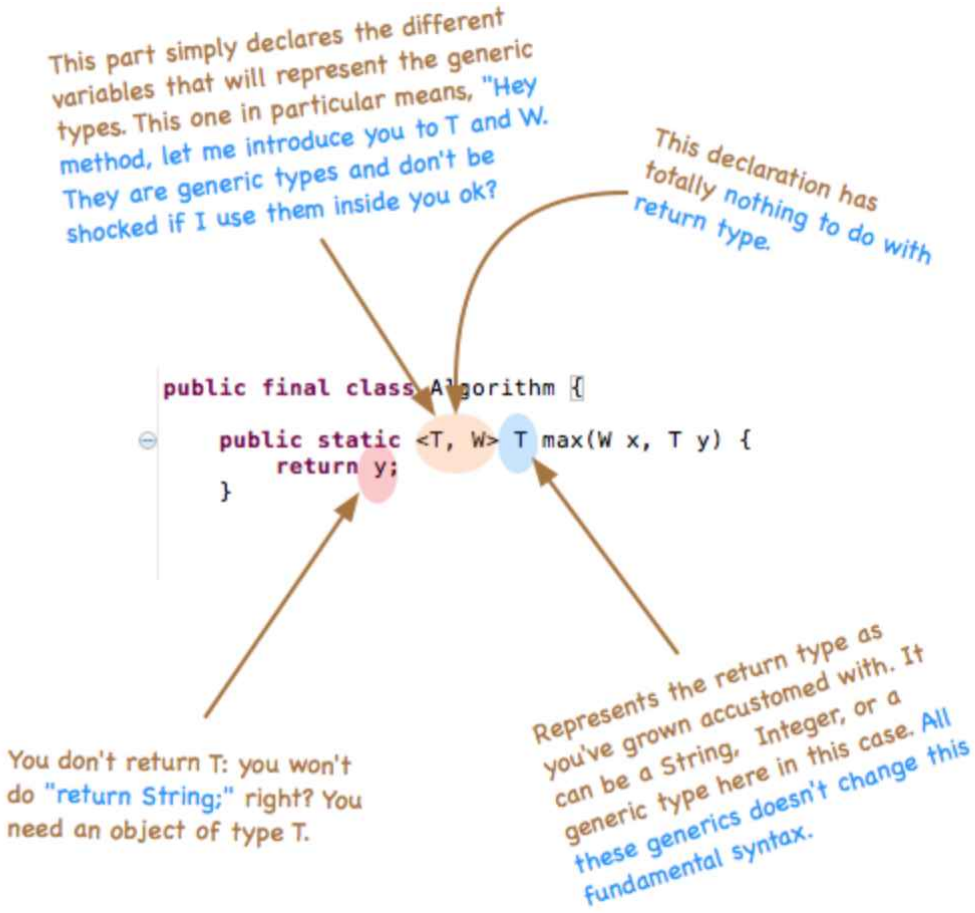
- By convention, type parameter names are **single, uppercase letters**. The most commonly used type parameter names are:
  - **T - Type**
  - **S, U, V etc. - 2nd, 3rd, 4th types**
  - **K - Key**
  - **V - Value**
  - E - Element (used extensively by the **Java Collections Framework**)

# Generic Methods

---

- *Generic methods* are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is **limited to the method** where it is declared.
- Static and non-static generic methods are allowed, as well as generic class constructors.

- The syntax for a generic method includes a list of type parameters, **inside angle brackets**, which **appears before the method's return type**. For **static** generic methods, the type parameter section **must appear before the method's return type**.




```java
// method return type: Box<T>
// list of parameters of this method: <T>, and <T> define the input parameter T
// parameter T
public static <T> Box<T> createBox(T value) {
    Box<T> box = new Box<>();
    box.set(value);
    return box;
}

public static void main(String[] args) {
    Box<String> nameBox = createBox("John Doe");
    System.out.println(nameBox.get());  // prints "John Doe"

    Box<Integer> ageBox = createBox(25);
    System.out.println(ageBox.get());  // prints 25
}
```

## Bounded Type Parameters

## Generics and Polymorphism

- Suppose we have an *Animal* class, and a *Dog* class that extends it.

```
class Animal {
    private String name;

    public Animal(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public void makeSound() {
        System.out.println("I am " + name);
    }
}

class Dog extends Animal {
    private boolean hasBone;

    public Dog(String name, boolean hasBone) {
        super(name);
        this.hasBone = hasBone;
    }

    public boolean isHasBone() {
        return hasBone;
    }

    @Override
    public void makeSound() {
        System.out.println("Woof woof my name is " + getName());
    }
}
```

- **Polymorphism does not apply to type parameters:**

```
public static void makeSound(Box<Animal> animalBox) {
    animalBox.get().makeSound();
```

```
}

// Alternative - same method signature
public static <T extends Animal> void makeSound(Box<T> animalBox) {
    animalBox.get().makeSound();
}

// solution for line 20
// public static void makeSound(Box<Dog> dogBox) {
//     dogBox.get().makeSound();
//}

public static <T> Box<T> createBox(T value) {
    Box<T> box = new Box<>();
    box.set(value);
    return box;
}

public static void main(String[] args) {
    Box<Animal> animalBox = createBox(new Animal("unknown animal"));
    makeSound(animalBox);  // OK
    Box<Dog> dogBox = createBox(new Dog("dog", true));
    makeSound(dogBox);  // Not OK -- Compilation error at line 1
}
```

- This is because `Box<Dog>` cannot be passed into the *makeSound()* method, which takes a parameter of type `Box<Animal>`.

```
public static void makeSound(Box<Animal> animalBox) {
    animalBox.get().makeSound();
}

makeSound(new Box<Animal>());  // OK

makeSound(new Box<Dog>());  // Not OK
// coz the method signature is makeSound(Box<Animal> animalBox)
```

## Generic Methods and Bounded Type Parameters

- We can update our method with `Bounded Type Parameters <T extends Animal>`:

```
public static <T extends Animal> void makeSound(Box<T> b) {
    T animal = b.get();
```

```
    animal.makeSound();
}
```

- Or with `Upper Bounded Wildcard <? extends Animal>`:

```
// Same as the above
public static void makeSound(Box<? extends Animal> animalBox) {
    T animal = animalBox.get();
    animal.makeSound();
}
```

- Whichever approach to use depends on the situation - whether the code depends on the `Bounded Type Parameter T`.
- Since the *makeSound()* method does not really depend on `T`. We prefer using the `Upper Bounded Wildcard`.

### Multiple Bounds

- A type can also have multiple upper bounds:
- Since *Number* is a class, we have to put it before the *Comparable* interface in the list of bounds. Otherwise, it will cause a compile-time error.

```
<T extends Number & Comparable<T>>
```

## Using Wildcards with Generics

- The previous section shows that an `upper bounded wildcard` restricts the unknown type to be a specific type or a subtype of that type, and is represented using the *extends* keyword.
- In a similar way, a `Lower Bounded Wildcard` restricts the unknown type to be a specific type or a *super type* of that type.
- A lower bounded wildcard is expressed using the wildcard character ('?'), followed by the *super* keyword, followed by its *lower bound*: `<? super Dog>`.
- **Note:** You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

### Lower Bound Wildcard

```java
class Animal {}

class Cat extends Animal {}
class Dog extends Animal {}

class RedCat extends Cat {}

public class LowerBoundWildcard {

   public static void addCat(List<? super Cat> catList) {
      catList.add(new RedCat());
      System.out.println("Cat Added");
   }

   public static void main(String[] args) {
      List<Animal> animalList= new ArrayList<>();
      List<Cat> catList= new ArrayList<>();
      List<RedCat> redCatList= new ArrayList<>();
      List<Dog> dogList= new ArrayList<>();

      //add list of super class Animal of Cat class
      addCat(animalList);

      //add list of Cat class
      addCat(catList); //

      //compile time error
      //cannot add list of subclass RedCat of Cat class
      addCat(redCatList);

      //compile time error
      //Dog is not super class of Cat
      addCat(dogList);
   }
}

```

## Type Erasure

- Generics were added to Java to ensure type safety. And to ensure that generics won't cause overhead at runtime, **the compiler applies a process called *type erasure* on generics at compile time.**

- Type erasure removes all type parameters and replaces them with their bounds OR with *Object* if the type parameter is unbounded.
- This way, the bytecode after compilation contains only normal classes, interfaces and methods, ensuring that no new types are produced.
- Proper casting is applied as well to the *Object* type at compile time.

## Example

- If the type is unbounded, List<T>:

```java
// java source code (.java)
public <T> List<T> genericMethod(List<T> list) {
    return list.stream().collect(Collectors.toList());
}
```

- With type erasure, the unbounded type *T* is replaced with *Object*:

```java
// After compilation (.class)
public List<Object> withErasure(List<Object> list) {
    return list.stream().collect(Collectors.toList());
}
```

- If the type is bounded, the type will be replaced by the bound (i.e. *Animal* in this case) at compile time:

```java
// java source code (.java)
public <T extends Animal> void genericMethod(T t) {
    // do something
    int a = 2;
    if (a > 1)
        t.drink();
}
```

- and would change after compilation:

```java
// after compilation (.class)
public void genericMethod(Animal t) {
    // do something
}
```

## Questions

- What is the purpose of using Generic types?
- Create Java classes that use Generics
- Create and use Generic methods
- What is the purpose of using bounded type parameters or wildcards?
- What is type erasure?