



Lesson Objectives

After completing this lesson, participants will be able to

- Understand concept of Generics
- Implement generic based collections



This lesson discusses about generics feature in Java.

Lesson outline:

- 15.1: Generics
- 15.2: Writing Generic Classes
- 15.3: Using Generics with Collections
- 15.4: Best Practices

15.1: Introduction to Generics



Generics

Generics is a mechanism by which a single piece of code can manipulate many different data types without explicitly having a separate entity for each data type.

What and Why of Generics:

JDK 1.5 introduces several extensions to the Java programming language. One of these is generics. Generics allow you to abstract over types. The most common examples are container types, such as those in the Collection hierarchy. Here is a typical usage of that sort:

```
List myIntegerList = new LinkedList(); // 1
myIntegerList.add(new Integer(0)); // 2
Integer intObj = (Integer) myIntegerList.iterator().next(); // 3
```

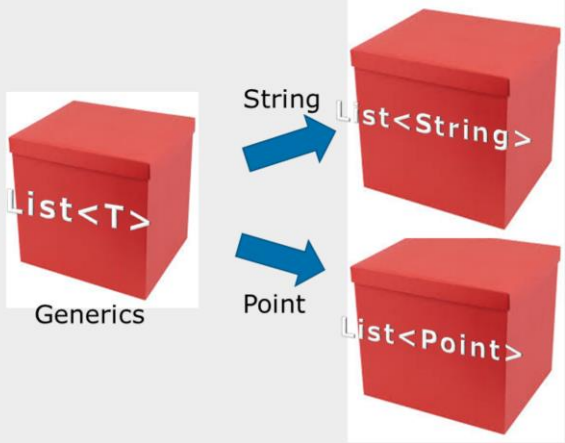
The cast on line 3 is slightly annoying. Typically, the programmer knows what kind of data has been placed into a particular list. However, the cast is essential. The compiler can only guarantee that an Object will be returned by the iterator. To ensure the assignment to a variable of type Integer is type safe, the cast is required. Of course, the cast not only introduces clutter, it also introduces the possibility of a run time error, since the programmer might be mistaken. What if programmers could actually express their intent, and mark a list as being restricted to contain a particular data type? This is the core idea behind generics.

15.1: Introduction to Generics

Generics

Generics allows programmer to create parameterized types

Instances of such types can be created by passing reference types



The diagram shows a generic box labeled `List<T>` on the left. Two blue arrows point from it to two specific boxes on the right. The top arrow is labeled `String` and points to a box labeled `List<String>`. The bottom arrow is labeled `Point` and points to a box labeled `List<Point>`.

What is Generics?:

Generics allows to write Parameterized type like `List<T>` and allows us to pass references to create instance of that type. Like when we pass the reference of `String` to `List<T>` as a reference type it creates List of strings.

Why generics?

Use of generics enables stricter compiler check. It means when List is created of type string, compiler only allows to add string elements to the list.

No need to perform casting. In case of generics enabled collections, no need to perform explicit casting.

15.2: Writing Generic Classes

Generics Fundamentals

Consider the class given to send the message of type String
Can we reuse the same class to send message of type Employee?



```
public class Sender{  
    private String message;  
    public setMessage(String message) {  
        this.message = message;  
    }  
    public sendMessage() {  
        //logic to send message  
    }  
}
```

Generics Fundamentals:

Consider the example given in the slide, if you want to reuse the same class to send employee object as a message, then we need to create one more additional class as shown below:

```
public class Sender{  
    private Employee message;  
    public setMessage(Employee message){  
        this.message = message;  
    }  
    public sendMessage(){  
        //logic to send message  
    }  
}
```

Is it possible to create a class with generic type of message? Yes. Generics enables us to create a parameterized class and later we can instantiate it as per our required reference type.

15.2: Writing Generic Classes



Writing Generic Types

How to create a sender class to send generic type of message?

```
public class Sender<T>{  
    private T message;  
    public setMessage(T message) {  
        this.message = message;  
    }  
    public sendMessage() {  
        //logic to send message  
    }  
}
```

```
Sender<String> stringSender = new Sender<String>();  
Sender<Employee> empSender = new Sender<Employee>();
```

Generics Fundamentals:

As shown in the slide example, The sender class is declared as parameterized generic type of one parameter as "T". The "T" in diamond operator refers as generic type of message.

In case of String message sender, the class instance would be initialized as:

```
Sender<String> stringSender = new Sender<String>() ;
```

In the above instance creation, the type T is replaced by String reference type. The same generic sender can be used to send message of type employee as shown below:

```
Sender<Employee> stringSender = new Sender<Employee>() ;
```

15.2: Writing Generic Classes

Generics Terminology

Below listed are different conventions used in generics

Syntax	Meaning
<T>	T denotes instance of any reference type
<?>	? denotes object of any type
<? super T>	? denotes lower bound object of type T
<? extends T>	? denotes upper bound object of type T (class)
<K, V>	K and V denotes instance of any type (same as T)

Generics Terminology:

In generics, different conventions are used to indicate the applicable reference type. For example, class `Sender<T>` indicates, the allowed reference type to create instance of `Sender` are:

Any reference type T

Subclass of T

The wildcard `?` is used to indicate any type. There are two variations in using wildcard.

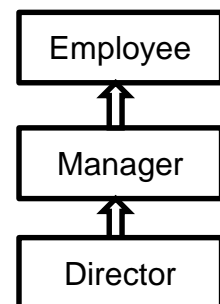
`? super T`: indicates lower bound meaning, any reference types which are superclass of T are allowed.

`? extends T`: indicated upper bound meaning, any reference types which are subclass of T are allowed.

Consider the given example for inheritance relationship.

`List<? super Manager>` means, list can be created of `Manager`, `Employee` etc. That is all superclass's of `Manager`.

`List<? extends Manager>` means, list can be created of `Manager`, `Director` etc. That is all subclasses of `Manager`.



15.3: Using Generics With Collections

Using Generics with Collections

▪ Before Generics:

```
List myIntegerList = new LinkedList(); // 1
myIntegerList.add(new Integer(0)); // 2
Integer intObj = (Integer) myIntegerList.iterator().next(); // 3
```



Note: Line no 3 if not properly typecasted will throw runtime exception

▪ After Generics:

```
List<Integer> myIntegerList = new LinkedList<Integer>(); // 1
myIntegerList.add(new Integer(0)); //2
Integer intObj = myIntegerList.iterator().next(); // 3
```

What and Why of Generics:

Observe the program fragment given above (in box #2) using generics:

Notice the type declaration for the variable `myIntegerList`. It specifies that this is not just an arbitrary `List`, but a `List of Integer`, written as `List<Integer>`. We say that `List` is a generic interface that takes a type parameter - in this case, `Integer`. We also specify a type parameter while creating the list object. Notice that the cast is gone from line 3. It may seem that all that's accomplished is just moving the clutter around. Instead of a cast to `Integer` on line 3, we have `Integer` as a type parameter on line 1.

However, there is a very big difference here. The compiler can now check the type correctness of the program at compile-time. When we say that `myIntegerList` is declared with type `List<Integer>`, this tells us something about the variable `myIntegerList`, which holds true wherever and whenever it is used, and the compiler will guarantee it. In contrast, the cast tells us something the programmer thinks is true at a single point in the code. The net effect, especially in large programs, is improved readability and robustness.

15.3: Using Generics with Collections



What problems does Generics solve?

Problem: Collection element types:

- Compiler is unable to verify types.
- Assignment must have type casting.
- ClassCastException can occur during runtime.

Solution: Generics

- Tell the compiler type of the collection.
- Let the compiler fill in the cast.
 - **Example:** Compiler will check if you are adding Integer type entry to a String type collection (compile time detection of type mismatch).

9

What and Why of Generics:

What problems does Generics solve?

Type cast not being checked at compile-time leads to a major problem occurring at application's runtime.

Biggest achievement of the Generics is to avoid the runtime exceptions.

15.3: Using Generics with Collections



Using Generic Classes: 1

You can instantiate a generic class to create type specific object.

In J2SE 5.0, all collection classes are rewritten to be generic classes.

- Example:

```
Vector<String> vector = new Vector<String>();  
vector.add(new Integer(5)); // Compile error!  
vector.add(new String("hello"));  
String string = vector.get(0); // No casting needed
```

10

Usage of Generics:

Usage of Generic classes is pertaining to the type that is required.

Once the Generic class is used for specific type, compile-time and runtime errors can be avoided.

15.3: Using Generics with Collections

Using Generic Classes: 2

Generic class can have multiple type parameters.

Type argument can be a custom type.

- Example:

```
HashMap<String, Mammal> map =  
    new HashMap<String, Mammal>();  
map.put("wombat", new Mammal("wombat"));  
Mammal mammal = map.get("wombat");
```

11

Usage of Generics:

Generic classes in use can have multiple arguments. Arguments can be standard as well as custom types.

15.3: Using Generics with Collections



Generics

Using generics, you can do this:

```
Object object = new Integer(5);
```

You can even do this:

```
Object[] objArr = new Integer[5];
```

So you would expect to be able to do this: `ArrayList<Object> arraylist = new ArrayList<Integer>();`

But you can't do it!!

- This is counter-intuitive at the first glance.

12

Note: Generic classes cannot be assigned according to the super or subclass hierarchy of them.

15.3: Using Generics with Collections



Generics

Why does this compile error occur?

- It is because if it is allowed, `ClassCastException` can occur during runtime – this is not type-safe.

```
ArrayList<Integer> ai = new ArrayList<Integer>();  
ArrayList<Object> ao = ai; // If it is allowed at compile time,  
ao.add(new Object()); Integer i = ao.get(0); // will result in runtime  
ClassCastException
```

There is no inheritance relationship between type arguments of a generic class.

13

Note: Generic classes do not support inheritance relationship between type arguments.

15.3: Using Generics with Collections

Generics



The following code works:

```
ArrayList<Integer> ai = new ArrayList<Integer>();  
List<Integer> li = new ArrayList<Integer>();  
Collection<Integer> ci = new ArrayList<Integer>();  
Collection<String> cs = new Vector<String>(4);
```

Inheritance relationship between Generic classes themselves still exists.

14

Note: Although the inheritance relationship between the type arguments of the generic classes does not exist, Inheritance relationship between Generic classes themselves still exist.

15.3: Using Generics with Collections

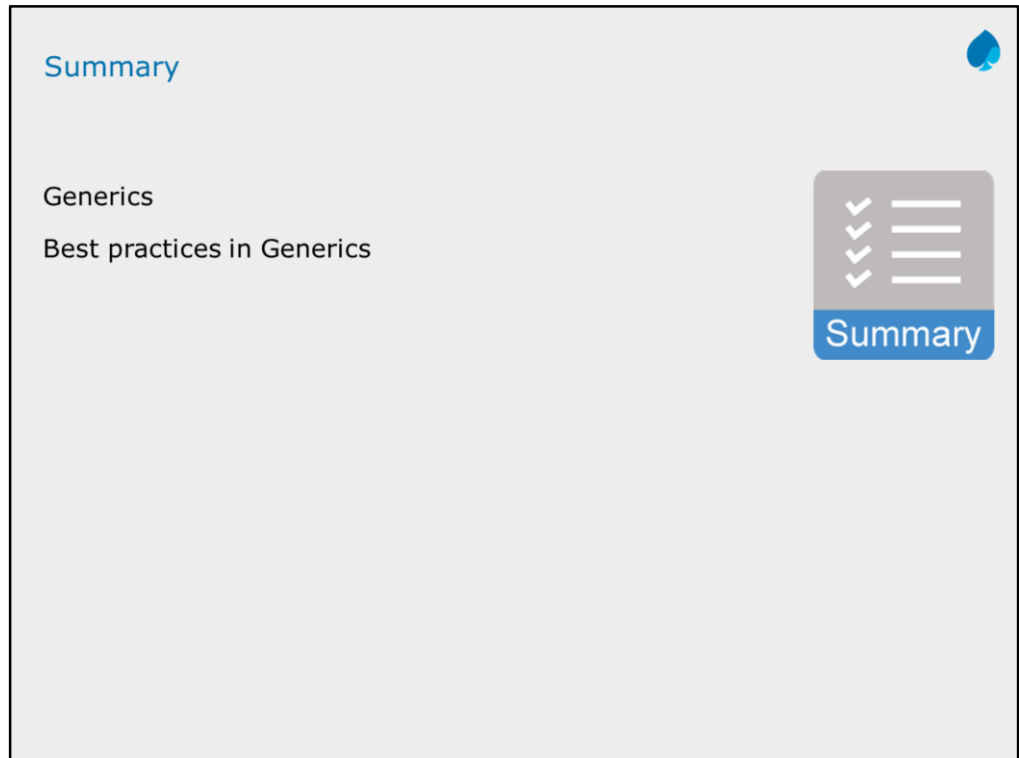


Generics

The following code works:

```
ArrayList<Number> an = new ArrayList<Number>();  
an.add(new Integer(5));  
an.add(new Long(1000L));  
an.add(new String("hello")); // compile error
```

The entries maintain inheritance relationship.



Review Questions

Question 1: If a method created to accept argument of `List<Object>`, then which of the following are valid options to pass? Ex: `void printList(List<Object> list)`

- Option1: `List<Object>`
- Option2: `List<Integer>`
- Option3: `List<Float>`
- Option 4: All of the above

