As we know that, *Generics*was added in *Java 5* to provide compile-time type checking and removing the risk of *ClassCastException* that was common while working with collection classes.

The whole **collection framework** was re-written to use generics for type-safety. Let’s see in this post how generics help us using collection classes safely.

Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces, and methods. Much like the more familiar formal parameters used in method declarations, **type parameters provide a way for you to re-use the same code with different inputs**. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

**Benefits of Generics**

**1. Stronger type checks at compile time**

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

**Example:** In this example, **List** holds only a **String** type of objects in generics. It doesn’t allow to store other objects

List<String> list = new ArrayList<String>();

list.add("abc");

**2. Elimination of casts**

The following code snippet without generics requires casting:

List list = new ArrayList();

list.add("hello");

String s = (String) list.get(0);

When re-written to use generics, the code does not require casting:

List<String> list = new ArrayList<String>();

list.add("hello");

String s = list.get(0); // no cast

**3. 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.

**Generic Types**

A generic type is a class or interface that is parameterized over types. We use angle brackets (<>) to specify the type parameter. We can define our own classes with generics type.

**Simple *Box* Class**

Begin by examining a non-generic *Box* class that operates on objects of any type. It needs only to provide two methods: *set*, which adds an object to the *box*, and *get*, which retrieves it:

public class Box {

private Object object;

public void set(Object object) {

this.object = object;

}

public Object get() {

return object;

}

public static void main(String[] args) {

Box type = new Box();

type.set("String");

System.out.println(type.get());

Box type1 = new Box();

type1.set(100);

System.out.println(type1.get());

Integer integer = (Integer) type.get();

System.out.println(integer);

}

}

Since its methods accept or return an *Object*, you are free to pass in whatever you want, provided that it is not one of the primitive types.

There is no way to verify, at compile time, how the class is used. One part of the code may place an Integer in the box and expect to get Integers out of it, while another part of the code may mistakenly pass in a **String**, resulting in a runtime error.

For Example, If we set **String** value to the first object like:

Box type = new Box();

type.set("String");

Now, we don't know what type it is so let's try to get an Integer value from it like:

Integer integer = (Integer) type.get();

System.out.println(integer);

In above, compiler forces to make a cast and this code throw an exception like:

Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer

at com.javaguides.generics.classes.Box.main(Box.java:21)

**A Generic Version of the Box Class**

A generic class is defined with the following format:

class name<T1, T2, ..., Tn> { /\* ... \*/ }

The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the type parameters (also called type variables) T1, T2, ..., and Tn.

Let's re-write the *Box*class using Generics.

public class Box<T> {

private T t;

public void set(T t) {

this.t = t;

}

public T get() {

return t;

}

public static void main(String[] args) {

Box<String> type = new Box<>();

type.set("String");

Box<Integer> type1 = new Box<>();

type1.set(100);

/\*Integer integer = (Integer) type.get(); // compiler error

System.out.println(integer);\*/

}

}

Note that from the above program, a cast is not required there and the compiler gives an error:

Cannot cast from String to Integer

**Real World Examples for Generic Classes**

**Collection Framework** classes and interfaces are examples for Generics.

For Example: **ArrayList** class declaration from java.util package.

public class ArrayList<E> extends AbstractList<E>

implements List<E>, RandomAccess, Cloneable, java.io.Serializable

{

.......

}

For Example,  **HashSet** class declaration from java.util package.

public class HashSet<E>

extends AbstractSet<E>

implements Set<E>, Cloneable, java.io.Serializable

{

.....

}

For Example,  **HashMap** class declaration from *java.util* package.

public class HashMap<K,V> extends AbstractMap<K,V>

implements Map<K,V>, Cloneable, Serializable {

....

}

**Real World Examples for Generic Interfaces**

An *Iterable*interface from JDK 8 - *java.lang* package is an example for a Generic interface.

public interface Iterable<T> {

Iterator<T> iterator();

default void forEach(Consumer<? super T> action) {

Objects.requireNonNull(action);

for (T t : this) {

action.accept(t);

}

}

default Spliterator<T> spliterator() {

return Spliterators.spliteratorUnknownSize(iterator(), 0);

}

}

One more example for Generic interface is **Comparable** interface.

public interface Comparable<T> {

public int compareTo(T o);

}

**Type Parameter Naming Conventions**

By convention, type parameter names are single, uppercase letters. The type parameters naming conventions are important to learn generics thoroughly.

The most commonly used type parameter names are:

* E - Element (used extensively by the Java Collections Framework)
* K - Key
* N - Number
* T - Type
* V - Value
* S, U, V etc. - 2nd, 3rd, 4th types

**Invoking and Instantiating a Generic Type**

Example to invoke and instantiate a Generic Type

Box<String> type = new Box<>();

type.set("String");

Box<Integer> type1 = new Box<>();

type1.set(100);

Place or between the class name and the parenthesis.

**The Diamond**

In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments *(<>)* as long as the compiler can determine, or infer, the type arguments from the context. This pair of angle brackets, <>, is informally called the diamond.

For example, you can create an instance of *Box*with the following statement:

Box<Integer> integerBox = new Box<>();

**Multiple Type Parameters**

A generic class can have multiple type parameters.

**Example**: The generic **OrderedPair** class, which implements the generic *Pair* interface:

public interface Pair<K, V> {

public K getKey();

public V getValue();

}

public class OrderedPair<K, V> implements Pair<K, V> {

private K key;

private V value;

public OrderedPair(K key, V value) {

this.key = key;

this.value = value;

}

public K getKey() { return key; }

public V getValue() { return value; }

}

The following statements create two instantiations of the *OrderedPair*class:

Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);

Pair<String, String> p2 = new OrderedPair<String, String>("hello", "world");

The code, *new OrderedPair<String, Integer>*, instantiates *K* as a String and *V* as an Integer. Therefore, the parameter types of *OrderedPair's*constructor are **String** and Integer, respectively. Due to **autoboxing**, it is valid to pass a **String** and an int to the class.

As mentioned in The Diamond, because a Java compiler can infer the K and V types from the declaration *OrderedPair<String, Integer>,* these statements can be shortened using the diamond notation:

OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);

OrderedPair<String, String> p2 = new OrderedPair<>("hello", "world");

To create a generic interface, follow the same conventions as for creating a generic class.

**Real world Example for Multiple Type Parameters**

**HashMap** class is a good example of Multiple Type Parameters.

public class HashMap<K,V> extends AbstractMap<K,V>

implements Map<K,V>, Cloneable, Serializable {

...

}

The interface type **Map** interface is a good example for Multiple Type Parameters.

public interface Map<K,V> {

...

}

Refer below Java Generics examples for hands-on experience