# Introduction to Generics

- Generics is a feature that allow classes, interfaces, and methods to operate on objects of various types, providing the flexibility to handle different data types without losing type safety.
- Java's Generics are introduced in java 5 with the syntax <T> to allow parameters, return types, and class types to be replaced by a specific data type at compile time.
- **Benefits of Generics:**
  - Type Safety: Ensures that objects are used in a type-safe manner, reducing runtime errors.
  - Code Reusability: You can write code that works with different types without having to duplicate logic.
  - Compile-time Checking: The compiler can enforce constraints, catching errors at compile time rather than runtime.

# Generic Class

- A **generic class** in Java is a class that can work with any type specified by the programmer at runtime.
- This is achieved by using **type parameters** in the class definition.
- A generic class provides type safety, eliminates the need for casting, and allows for code reuse.

class ClassName<T>

{ // Class body where T can be used as a type }

T is the type parameter (a placeholder for the type the class will operate on).

You can use multiple type parameters (e.g., <T, U>).

# Simple Generics Class

• Syntax:

```
class ClassName<T> {
   // T is a placeholder for a type
}
```

Example 1: Generic Class

```
class Box<T> {
   private T item;
   public void setItem(T item) {
       this.item = item;
   }
   public T getItem() {
```

# Type Parameters

- Type parameters in Java are the placeholders used in generics to define the types that a class, interface, or method can work with.
- A type parameter is defined using angle brackets (<>) and is typically represented by a single uppercase letter, though more descriptive names can also be used.
- Commonly Used Type Parameter Conventions:
  - T: Type (generic data type).
  - E: Element (commonly used in collections like List, Set, etc.).
  - K`` and V`: Key and Value (used in maps).
  - N: Number (numeric data type).
  - R: Return type.
- Note: A class, interface, or method can use multiple type parameters separated by comma.

# Generics Types

- Generic types allow defining classes, interfaces, or methods with a type parameter.

  **Defining Generics in Classes and Interfaces**

  **Generic Methods**

  **Bounded Type Parameters**

  **Wildcard Generics**

  **Type Erasure**

# Defining Generics in Classes

• Syntax:

• Defining the class

```
class ClassName<T1, T2, ...> {
   // Code here
  // T1, T2 can be used to as types for attributes and methods
}
```

Creating the instance:

```
ClassName<Type> obj=new ClassName<Type>();
```

Example Program:

# Defining Generics in Interfaces

• Syntax:

• Defining the class

```
interface InterfaceName<T1, T2, ...> {
   // Code here
   // T1, T2 can be used to as types for attributes and methods
}
```

Creating the Subclass:

```
class ClassName<T1,T2,…>  implements InterfaceName<T1,T2,….>
{  //code of the class }
```

Example Program:

# Defining Generics Methods

- A generic method defines a type parameter that applies only within that method.
- **Syntax:**

```
public <T> void methodName(T parameter) {
   // Code here
}
```

- Example:

# Generics Methods Example

```
public class GenericMethodExample {
   public static <T> void printArray(T[] array) {
       for (T element : array) {
           System.out.print(element + " ");         }
       System.out.println();      }
   public static void main(String[] args) {
       Integer[] intArray = {1, 2, 3};
       String[] strArray = {"A", "B", "C"};
       printArray(intArray);
       printArray(strArray);
   } }
```

# Bounded Type Parameters

- In Java, bounded type parameters are used in generics to restrict the types that can be substituted for a type parameter.
- This is done using the extends keyword.

Syntax:

<T extends BoundType >

**Description:**

- <T> is a type parameter.
- extends BoundType restricts T to be either:
- A subclass of BoundType.
- BoundType itself.
- BoundType can be a class or an interface.

# Bounded Type Parameters

- **Single Bound:**
- You can use a single class or interface as the upper bound.
- Syntax: <T extends ClassName> or <T extends InterfaceName>.
- **Multiple Bounds:**
- You can specify multiple bounds, combining a class and one or more interfaces.
- Syntax: <T extends ClassName & Interface1 & Interface2>.
- Note: If a class is specified, it must come first.
- **Type Bounds Are Inclusive:**
- The specified bound is included in the allowed types.

# Single Bound Example

```
class SingleBoundExample<T extends Number> {
   private T number;
   public SingleBoundExample(T number) {
      this.number = number;     }
   public double square() {
      return number.doubleValue() * number.doubleValue();      }}
public class Main {
   public static void main(String[] args) {
      SingleBoundExample<Integer> intBox = new SingleBoundExample<>(5);
      System.out.println(intBox.square());
      SingleBoundExample<Double> doubleBox = new SingleBoundExample<>
```

# Multi Bound example

```
class MultiBoundExample<T extends Number & Comparable<T>> {
private T value;
public MultiBoundExample(T value) {        this.value = value;    }
public boolean isGreater(T other) {
return value.compareTo(other) > 0;
}}
```

# Wild cards

In Java, wildcards (?) are special placeholders used in generics to represent an unknown type.

They make generic code more flexible by allowing methods and classes to work with a range of types without being tied to a specific one.

Wildcards are commonly used in method parameters, return types, and field declarations.

**Types of Wildcards:**

- Unbounded Wildcard: ?
- Upper-Bounded Wildcard: ? extends Type
- Lower-Bounded Wildcard: ? super Type

# Unbounded Wildcard (?)

Represents an unknown type. It is used when you don't care about the type but want to ensure type safety.

Syntax: List<?> list = ...;

Example: import java.util.List;

```
public class UNBWildcardExample {
   public static void printList(List<?> list) {
      for (Object obj : list) {    System.out.println(obj);        }     }
   public static void main(String[] args) {
      List<String> stringList = List.of("Apple", "Banana", "Cherry");
      List<Integer> intList = List.of(1, 2, 3);
      printList(stringList); // Output: Apple, Banana, Cherry
```

# Upper-Bounded Wildcard (? extends Type)

Restricts the unknown type to be a specific type or its subclasses.

Syntax: List<? extends Type> list = ...;

**Example:** import java.util.List;

```
public class UBWildcardExample {
   public static double sumList(List<? extends Number> list) {
      double sum = 0.0;
      for (Number num :list) {
         sum += num.doubleValue();   }      return sum;  }
   public static void main(String[] args) {
      List<Integer> intList = List.of(1, 2, 3);
      List<Double> doubleList = List.of(1.1, 2.2, 3.3);
```

# Lower-Bounded Wildcard (? super Type)

Restricts the unknown type to be a specific type or its super classes.

Syntax: List<? super Type> list = ...;

**Example:** import java.util.List;

```
public class WildcardExample {
   public static void addNumbers(List<? super Integer> list) {
       list.add(10);
       list.add(20);
       // list.add(3.14); // Compile-time error    }
   public static void main(String[] args) {
       List<Object> objList = List.of(new Object(), "Hello");
       List<Number> numberList = List.of(1.1, 2.2);
```

# Wildcard Limitations

- Cannot instantiate generic types with wildcards:

```
List<?> list = new ArrayList<?>(); // Compile-time error
```

- Cannot use wildcards in generic class declarations:

```
class Box<?> { } // Compile-time error
```

- Cannot call methods that depend on the specific type of the wildcard.

# Wildcard Differences

## Key Differences Between Wildcards

| Wildcard Type | Accepts | Read | Write |
|---|---|---|---|
| `?` (Unbounded) | Any type ( `List<String>` , `List<Integer>` ) | Yes | No |
| `? extends Type` | Type or its subclasses ( `List<Integer>` ) | Yes | No |
| `? super Type` | Type or its superclasses ( `List<Object>` ) | No | Yes |

# Inheritance & Sub Types

Generic types do not support polymorphism in the usual way.

For example, List<Object> is not a supertype of List<String>.

- Example:
  - List<Object> objectList;
  - List<String> stringList;
  - // objectList = stringList; // Compile-time error why this

List<Object> objectList = stringList;

objectList.add(42); // Adding an Integer to the list

String s = stringList.get(0); // This would fail at runtime!

Java prevents this situation at compile-time by treating List<Object> and List<String> as incompatible.

# Inheritance & Sub Types

However, you can use wildcards to achieve polymorphism.

**Example with Wildcards:**

```
public static void addStrings(List<? super String> list) {
   list.add("New String"); }
```

- List<? super String> means a List that can contain String or any of its supertypes (like List<Object> or List<String>).
- This makes the method flexible: it can accept a List<Object> or a List<String>, allowing polymorphism.

# Covariance with Wildcards

- Covariance in Java wildcards refers to the ability to allow a method or variable to accept a type and any of its subtypes.
- This is achieved using the <? extends Type> wildcard syntax in Java generics.
- It is commonly used when working with collections and generic methods to provide more flexibility while maintaining type safety.
- List<? extends Parent> list = new ArrayList<Child>();
  - Parent p = list.get(0); // Allowed
  - list.add(new Parent()); // Compilation Error
  - list.add(new Child()); // Compilation Error

# Covariance Example

# Contravariance with Wildcards

- Contravariance with wildcards in Java allows you to define a generic structure that can accept a type and any of its supertypes.
- It is achieved using the <? super Type> wildcard syntax.
- Contravariance is useful when you need to perform operations that add elements to a collection or modify it while ensuring type safety.
- List<? super Child> list = new ArrayList<Parent>();
  - list.add(new Child());  // Allowed
  - list.add(new GrandChild()); // Allowed if GrandChild extends Child
  - Parent p = list.get(0); // Compilation Error
  - Object obj = list.get(0); // Allowed

# Generic Superclass

- A generic superclass is a class that declares a generic type parameter.
- **Example for a Generic Superclass**

```
class GenericSuperclass<T> {
   T data;
   public GenericSuperclass(T data) {
       this.data = data;
   }
   public T getData() {
       return data;
   }
}
```

# Generic Subclass

- A generic subclass is a class which extends a generic superclass.
- This can be done in two ways.
- Case 1: Subclass Retains Generic Type
- Case 2: Subclass Specifies Type Parameter

We hava 3rd one also

- Case 3: Mixed Generic Subclass

# 1. Subclass Retains Generic Type

The subclass does not specify the type parameter and remains generic.

**Example:**

```
class GenericSubclass<T> extends GenericSuperclass<T> {
   public GenericSubclass(T data) {
       super(data);
   }
   public void display() {
       System.out.println("Data: " + getData());
   }
}
```

# 2.Subclass Specifies Type Parameter

The subclass fixes the type parameter to a specific type.

**Example:**

```
class SpecificSubclass extends GenericSuperclass<String> {
   public SpecificSubclass(String data) {
       super(data);
   }
   public void display() {
       System.out.println("Data: " + getData());
   }
}
```

# Example on Generic Super class and Subclsses

```
public class Main {
   public static void main(String[] args) {
      GenericSubclass<String> obj = new GenericSubclass<>("Hello");
      obj.display(); // Output: Data: Hello
      SpecificSubclass obj = new SpecificSubclass("Hello");
      obj.display(); // Output: Data: Hello
   }
}
```

# 3. Mixed Generic Subclass

A subclass can also add its own generic type parameters while extending a generic superclass.

Syntax:

```
class MixedSubclass<T, U> extends GenericSuperclass<T> {
   U additionalData;
   public MixedSubclass(T data, U additionalData) {
       super(data);
       this.additionalData = additionalData;
   }
   public void display() {
       System.out.println("Data: " + getData());
```

# 3. Mixed Generic Subclass

```
public class Main {
   public static void main(String[] args) {
      MixedSubclass<String, Integer> obj = new MixedSubclass<>("Hello", 100);
      obj.display();
      // Output:
      // Data: Hello
      // Additional Data: 100
   }
}
```

# 4.Raw Types with Generic Superclass

If a subclass don’t provide a type parameter for a generic superclass, it becomes a raw type.

```
class Subclass extends GenericSuperclass {
   public Subclass(Object data) {
       super(data); // Using raw type
   }}
public class Main {
   public static void main(String[] args) {
       Subclass obj = new Subclass("Raw Type Example");
       System.out.println(obj.getData()); // Allowed, but not type-safe
   }}
```

# Type Inference

- Type inference is a feature in Java where the compiler infers the type of a variable automatically based on the context in which it is used, rather than requiring the programmer to explicitly specify the type.
- This feature reduces verbosity and enhances code readability.

# Type Inference

*Type inference* is a Java compiler's ability to look at each method invocation and corresponding declaration to determine the type argument (or arguments) that make the invocation applicable.

The inference algorithm determines the types of the arguments and, if available, the type that the result is being assigned, or returned.

Finally, the inference algorithm tries to find the *most specific* type that works with all of the arguments.

static <T> T pickup (T a1, T a2) { return a2; }

Number n= pick(10,20.5);

# Type Inference & Generic Methods

Generic Methods introduced you to type inference, which enables you to invoke a generic method as you would an ordinary method, without specifying a type between angle brackets.

# Type Inference and Instantiation of Generic Classes

You can replace the type arguments required to invoke the constructor of a generic class with an empty set of type parameters (<>) as long as the compiler can infer the type arguments from the context.

This pair of angle brackets is informally called the diamond.

For example:

Map<String, List<String>> myMap = new HashMap<String, List<String>>();

You can substitute the parameterized type of the constructor with an empty set of type parameters (<>) from java 1.7

Map<String, List<String>> myMap = new HashMap<>();

# Type Inference and Generic Constructors of Generic and Non-Generic Classes

Note that constructors can be generic (in other words, declare their own formal type parameters) in both generic and non-generic classes.

Consider the following example:

```
class MyClass<X> {
 <T> MyClass(T t) {
   // ...
 }
}
```

Consider the following instantiation of the class MyClass:

```
new MyClass<Integer>("")
```

# Type Inference and Generic Constructors of Generic and Non-Generic Classes

- Consider the following instantiation of the class MyClass:
  - new MyClass<Integer>("");
- This statement creates an instance of the parameterized type MyClass<Integer>
- The statement explicitly specifies the type Integer for the formal type parameter, X, of the generic class MyClass<X>.
- The constructor for this generic class contains a formal type parameter, T.
- The compiler infers the type String for the formal type parameter, T, of the constructor of this generic class (because the actual parameter of this constructor is a String object).
- Compilers from releases prior to Java SE 7 are able to infer the actual type parameters of generic constructors, similar to generic methods.

# Type Inference and Target Types

- The Java compiler takes advantage of target typing to infer the type parameters of a generic method invocation.
- The target type of an expression is the data type that the Java compiler expects depending on where the expression appears.
- Consider the method Collections.emptyList, which is declared as follows:
- static <T> List<T> emptyList();
- Consider the following assignment statement:
- List<String> listOne = Collections.emptyList();
- This statement is expecting an instance of List<String>; this data type is the target type.

# Restrictions on Generics

- Cannot Use Primitives: Generics work only with objects, not primitives (e.g., int, double). Use wrapper classes (Integer, Double).
- No Static Fields: Generic types cannot be used in static fields or methods.
- Cannot Create Instances of Type Parameters:
  - T obj = new T(); // Not allowed
- Type Erasure: Generic type information is removed during runtime, so List<String> and List<Integer> look the same.

Example:

```
List<String> list = new ArrayList<>();
list.add("Hello");
// list.add(123); // Compile-time error
```

# Functional Interfaces

- A functional interface is an interface with a single abstract method (SAM), suitable for use in lambda expressions.
- Functional interfaces are the backbone of functional programming in Java.
- Key Functional Interfaces
- a) Function<T, R>:
- Represents a function that accepts one argument and produces a result.
- Functional Method: R apply(T t)
- Use Case: Transformation or mapping functions.
- Example:
- Function<Integer, String> intToString = i -> "Number: " + i;
- System.out.println(intToString.apply(5)); // Output: Number: 5

# Functional Interfaces

b) BiFunction<T, U, R>

Description: Represents a function that accepts two arguments and produces a result.

Functional Method: R apply(T t, U u)

Use Case: Operations involving two inputs.

Example:

```
BiFunction<Integer, Integer, Integer> adder = (a, b) -> a + b;
System.out.println(adder.apply(3, 5)); // Output: 8
```

# Functional Interfaces

c) Predicate<T>

Description: Represents a predicate (boolean-valued function) of one argument.

Functional Method: boolean test(T t)

Use Case: Filtering or condition-based logic.

Example:

```
Predicate<String> isLongString = s -> s.length() > 5;
System.out.println(isLongString.test("Hello")); // Output: false
```

# Functional Interfaces

d) Supplier<T>

Description: Represents a supplier of results; takes no input and returns a value.

Functional Method: T get()

Use Case: Deferred execution or object creation.

Example:

```
Supplier<Double> randomSupplier = () -> Math.random();
System.out.println(randomSupplier.get()); // Output: A random number
```

# Lambda Expression

A lambda expression is a concise way to represent an anonymous function (a function without a name).

It provides an implementation of a functional interface.

Syntax of Lambda Expressions

(parameters) -> expression

(parameters) -> { statements }

Examples

Single-line Expression:

(x) -> x * x

Multiple Statements:

(x, y) -> { int result = x + y;

# Block Lambda Expressions

Block lambda expressions allow multiple statements within a lambda body.

Example:

```
Function<Integer, Integer> factorial = n -> {
   int result = 1;
   for (int i = 1; i <= n; i++) {
       result *= i;
   }
   return result;
};
System.out.println(factorial.apply(5)); // Output: 120
```

# Passing Lambda Expressions as Arguments

Lambda expressions can be passed as arguments to methods that accept functional interfaces.

Example:

```
public static void printResult(Function<Integer, String> func, int value) {
   System.out.println(func.apply(value));
}
public static void main(String[] args) {
   printResult(x -> "Result: " + (x * 2), 10); // Output: Result: 20
}
```

# Lambda Expressions and Exceptions

Lambda expressions can throw exceptions if allowed by the functional interface.

Example:

```
Function<String, Integer> stringToInt = s -> {
   try {
      return Integer.parseInt(s);
   } catch (NumberFormatException e) {
      System.out.println("Invalid number: " + s);
      return 0;
   }
};
System.out.println(stringToInt.apply("123")); // Output: 123
```

# Variable Capture

Definition: A lambda expression can "capture" variables from its surrounding scope.

Rules:

Can use final or effectively final variables (variables not modified after initialization).

Cannot modify non-final variables from the enclosing scope.

Example:

```
public static void main(String[] args) {
  int num = 10; // Effectively final
  Supplier<Integer> supplier = () -> num + 5;
  System.out.println(supplier.get()); // Output: 15
}
```

# Method References

Method references are shorthand notations for calling methods directly, used when a lambda expression simply calls an existing method.

Types of Method References

Static Method Reference: ClassName::methodName

```
Function<String, Integer> parseInt = Integer::parseInt;
System.out.println(parseInt.apply("123")); // Output: 123
```

# Method References

Instance Method Reference (on a particular object): instance::methodName

```
String str = "Hello";
Supplier<Integer> lengthSupplier = str::length;
System.out.println(lengthSupplier.get()); // Output: 5
```

# Method References

Instance Method Reference (on an arbitrary object of a type):
ClassName::methodName

```
Function<String, String> toUpperCase = String::toUpperCase;
System.out.println(toUpperCase.apply("hello")); // Output: HELLO
```

# Method References

Constructor Reference: ClassName::new

```
Supplier<List<String>> listSupplier = ArrayList::new;
List<String> list = listSupplier.get();
```