

OBJECT-ORIENTED PROGRAMMING Using JAVA

GENERIC PROGRAMMING

QUAN THAI HA

HUS

2024



- 1 Introduction
- 2 Generic Classes
- 3 Generic Methods
- 4 Wildcard Types
- 5 Bounded Types
- 6 References

- **Generic programming** is a programming style in which algorithms and data structures are written at the most abstract possible level independent of the form of the data.
 - ▶ For example, the Java library programmers who implemented the `ArrayList` class used the technique of generic programming. As a result, you can form array lists that collect elements of different types, such as `ArrayList<String>`, `ArrayList<BankAccount>`, and so on.
- Different terms (and implementation) → similar concept.
 - ▶ **Generics**
 - Ada, Eiffel, Java, C#, VisualBasic.NET
 - ▶ **Parametric polymorphism**
 - ML, Scala, Haskell
 - ▶ **Templates**
 - C++

■ Generic programming

- ▶ **Functions** (methods) or **types** (classes) that **differ only in the set of types** on which they operate.
- ▶ Generic programming is a way to make a language more expressive, while still maintaining full static type-safety.
- ▶ Reduce duplication of code.
- ▶ Algorithms are written in terms of generic types.
 - **Types** are passed as **parameters** later when needed.

■ Generic types

- ▶ Store values and perform operation on different data types.

■ Generic functions

- ▶ Performs the same operation on different data types.



```
1 public class Pair {  
    private Object first;  
3    private Object second;  
  
5    public Pair(Object first , Object second) {  
        this.first = first;  
7        this.second = second;  
    }  
9  
    public Object getFirst() {  
11        return first;  
    }  
13  
    public Object getSecond() {  
15        return second;  
    }  
17  
    public String toString() {  
19        return "(" + first + ", " + second + ")";  
    }  
21 }
```



```
1 public class Main {  
    public static void main(String[] args) {  
3        Pair pair1 = new Pair("alpha", 1);  
        // String and Integer (auto-boxing) - Implicit upcasting to Object  
5        String name = (String) pair1.getFirst();  
        // Explicit downcasting from Object to String  
7        Integer value = (Integer) pair1.getSecond();  
        System.out.println("Name: " + name + ", value: " + value);  
9  
        Pair pair2 = new Pair(3.2, 5.5); // Double and Double (auto-boxing)  
11       Double x = (Double) pair2.getFirst();  
        double y = (double) pair2.getSecond(); // auto-unboxing  
13       System.out.println("x: " + x + ", y: " + y);  
  
15       x = (Double) pair1.getFirst(); // Run-time error  
        // Exception in thread "main" java.lang.ClassCastException: java.lang.String  
17       // Cannot be cast to java.lang.Double  
    }  
19 }
```



```
1 public class Pair<T, S> {  
    private T first;  
    private S second;  
  
    5 public Pair(T first, S second) {  
        this.first = first;  
        7 this.second = second;  
    }  
  
    9 public T getFirst() {  
        11 return first;  
    }  
  
    13 public S getSecond() {  
        15 return second;  
    }  
  
    17 public String toString() {  
        19 return "(" + first + ", " + second + ")";  
    }  
    21 }
```



```

1 public class Main {
    public static void main(String[] args) {
2         // Explicit actual type paramethers
3         Pair<String, Integer> pair1 = new Pair<String, Integer>("alpha", 1);
4         String name = pair1.getFirst();
5         Integer value = pair1.getSecond();
6         System.out.println("Name: " + name + ", value: " + value);
7
8         // Implicit actual type paramethers
9         Pair<Double, Double> pair2 = new Pair(3.2, 5.5);
10        Double x = pair2.getFirst();
11        double y = pair2.getSecond();
12        System.out.println("x: " + x + ", y: " + y);
13
14        x = pair1.getFirst();
15        // Compile-time error: Type mismatch: cannot convert from String to Double
16    }
17 }
    
```


- The first possible solution: **Overloading**

- **Overloading**

- ▶ Set of methods all having the **same name**, but with a **different arguments** list (signature).



```
public class ArrayUtil {  
2   public static String getCentral(String[] arr) {  
    if (arr == null || arr.length == 0)  
4       return null;  
    return (arr[arr.length / 2]);  
6   }  
  
8   public static Integer getCentral(Integer[] arr) {  
    if (arr == null || arr.length == 0)  
10      return null;  
    return (arr[arr.length / 2]);  
12  }  
}
```



```
1 public class Main {  
    public static void main(String[] args) {  
3        String[] strings = {"alpha", "beta", "charlie"};  
        String centralString = ArrayUtil.getCentral(strings); // "beta"  
5  
        Integer[] integers = {4 , 8 , 15 , 16 , 23 , 42};  
7        int centralInteger = ArrayUtil.getCentral(integers); // 16 (auto-unboxing)  
9  
        Double[] doubles = {1.1 , 2.3 , 5.8 , 13.21};  
        Double centralDouble = ArrayUtil.getCentral(doubles);  
11       // Compile time error: No suitable method found for getCentral(Double[])  
        }  
13 }
```

- **auto-boxing** is the automatic conversion that the Java compiler makes from the primitive types to their corresponding object wrapper classes.
- **auto-unboxing** is the conversion converting of an object of a wrapper type to its corresponding primitive value.

- The second possible solution: **Inheritance** and **Polymorphism**



```
1 public class ArrayUtil {  
    // Get the central element of the array and return central element.  
3     public static Object getCentral(Object[] arr) {  
        if (arr == null || arr.length == 0) {  
5             return null;  
        }  
7  
        return (arr[arr.length / 2]);  
9     }  
}
```

- We can write a method that **takes a base class** (or **interface**) as an argument, and then use that method with **any class derived from that base class**. This method is more general and can be used in more places.



```
public class Main {
2   public static void main(String[] args) {
        String[] strings = {"alpha", "beta", "charlie"};
4       String centralString = (String) ArrayUtil.getCentral(strings);
        // Downcast from Object to String

6       Integer[] integers = {4 , 8 , 15 , 16 , 23 , 42};
7       int centralInteger = (int) ArrayUtil.getCentral(integers);
        // Downcast and auto-unboxing

10      Double[] doubles = {1.1, 2.3, 5.8, 13.21};
12      Double centralDouble = (Double) ArrayUtil.getCentral(doubles); // 5.8

14      Integer stringToInteger = (Integer) ArrayUtil.getCentral(strings);
        // No compile-time error, but run-time exception
16      // Exception ... ClassCastException ... Cannot be cast to java.lang.Integer ...
    }
18 }
```

- **Downcasting** from base class can generate **no type-safe** code. Run-time exception occurs in wrong cast operations.

- 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.
- You can think of it as a template for a set of methods that differ only by one or more types.
- When you call the generic method, you need not specify which type to use for the type parameter. Simply call the method with appropriate parameters, and the compiler will match up the type parameters with the parameter types.



```
public static <T> T getCentral(T[] arr) {  
2   if (arr == null || arr.length == 0) {  
       return null;  
4   }  
   return (arr[arr.length / 2]);  
6 }
```



```

1 public class Main {
2     public static void main(String[] args) {
3         String[] strings = {"alpha", "beta", "charlie"};
4         String centralString = ArrayUtil.getCentral(strings); // "beta"
5
6         Character[] characters = {'h', 'a', 'l'};
7         Character centralChar = ArrayUtil.<Character>getCentral(characters); // 'a'
8         // (Explicit type (Character) parameter)
9
10        Integer[] integers = {4, 8, 15, 16, 23, 42};
11        int centralInteger = ArrayUtil.getCentral(integers); // 16
12        // (Implicit type parameter and auto-unboxing)
13
14        Double[] doubles = {1.1, 2.3, 5.8, 13.21};
15        Double centralDouble = (Double) ArrayUtil.getCentral(doubles); // 5.8
16
17        Integer charToInteger = ArrayUtil.getCentral(characters);
18        // Compile-time error: incompatible types
19    }
20 }
    
```

- A **generic type** is a type with formal **type parameters**.



```
interface Collection<E> {  
2   public void add (E x);  
   public Iterator<E> iterator();  
4 }
```

- The interface **Collection** has one **type parameter E**. The **type parameter E** is a place holder that will later be replaced by a type argument when the generic type is instantiated and used.
- The instantiation of a generic type with actual type arguments is called a **parameterized type**.



```
Collection<String> coll = new LinkedList<String>();
```

- Naming guidelines: **Type parameter** names are single, uppercase letters, which is quite different from the variable naming convention — the difference between type variable and an ordinary class or interface name should be very clear.
- The most commonly used type parameter names:
 - ▶ **E**: Element type in a collection
 - ▶ **K**: Key type in a map
 - ▶ **V**: Value type in a map
 - ▶ **T**: General type
 - ▶ **S**, **U**: Additional general types
- Syntax enhancement in JDK 7: use the diamond operator `<>`:



```
Map<String , List<Trade>> trades = new TreeMap<>();
```


- We can **restrict the types** that can be used as type arguments in a parameterized type by using **bounded type parameter**.
 - For example, a method that operates on numbers might only want to accept instances of **Number** or its subclasses.
- To declare a **bounded type parameter**, list the type parameter's name, followed by the **extends** keyword, followed by its **upper bound**.
 - Note that, in this context, **extends** is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).



```
1 public class Box<T extends Number> {  
    private T data;  
3  
    Sample(T data) {  
5        this.data = data;  
    }  
7  
    public void display() {  
9        System.out.println(this.data);  
    }  
11 }  
  
13 public class BoundsExample {  
    public static void main(String args[]) {  
15        Box<Integer> box1 = new Box<>(20);  
        box1.display();  
17  
        Box<Double> box2 = new Box<>(20.22);  
19        box2.display();  
    }  
21 }
```

- Bounded type parameters are key to the implementation of generic algorithms. Consider the following method that counts the number of elements in an array `T[]` that are greater than a specified element `elem`.



```
1 public static <T> int countGreaterThan(T[] anArray, T elem) {  
    int count = 0;  
3    for (T e : anArray) {  
        if (e > elem) { // compiler error  
5            ++count;  
        }  
7    }  
    return count;  
9 }
```

- The implementation of the method is straightforward, but it does not compile because the greater than operator (`>`) applies only to primitive types such as `short`, `int`, `double`, `long`, `float`, `byte`, and `char`. You cannot use the `>` operator to compare objects.

- To fix the problem, use a type parameter bounded by the `Comparable<T>` interface:



```
1 public interface Comparable<T> {  
    public int compareTo(T o);  
3 }
```



```
1 public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem) {  
    int count = 0;  
3    for (T e : anArray) {  
        if (e.compareTo(elem) > 0) {  
5            ++count;  
        }  
7    }  
    return count;  
9 }
```

- It is often necessary to formulate **constraints** of type parameters.
- In generic code, the question mark (?), called the wildcard, represents an unknown type.
- The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable; sometimes as a return type.
- There are three kinds of wildcard types:

Name	Syntax	Meaning
Wildcard with upper bound	? extends T	Any subtype of T
Wildcard with lower bound	? super T	Any supertype of T
Unbounded	?	Any type

- You can use an **Upper Bounded Wildcard** to relax the restrictions on a variable. An upper bounded wildcard restricts the unknown type to be a specific type or a subtype of that type.
 - ▶ For example, if you want to write a method that works on `List<Integer>`, `List<Double>`, and `List<Number>`, you can achieve this by using an upper bounded wildcard.
- To declare an upper bounded wildcard, use the wildcard character ('?'), followed by the **extends** keyword, followed by its **upper bound**.
 - ▶ Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).
 - ▶ To write the method that works on lists of `Number` and the subtypes of `Number`, such as `Integer`, `Double`, and `Float`, you would specify `List<? extends Number>`.
 - ▶ The term `List<Number>` is **more restrictive** than `List<? extends Number>` because the former matches a list of type `Number` only, whereas the latter matches a list of type `Number` or any of its subclasses.



```
1 public class ArrayUtil {  
    public static double sumOfList(List<? extends Number> list) {  
3        double sum = 0.0;  
        for (Number element : list) {  
5            sum += element.doubleValue();  
        }  
        return sum;  
7    }  
  
9  
    public static double productOfList(List<? extends Number> list) {  
11        double product = 1.0;  
        for (Number element : list) {  
13            product *= element.doubleValue();  
        }  
15        return product;  
    }  
17 }
```



```
1 public class Main {  
    public static void main(String[] args) {  
3        List<Integer> integers = Arrays.asList(1, 2, 3);  
        System.out.println("Sum = " + ArrayUtil.sumOfList(integers)); // Sum = 6  
5        System.out.println("Product = " + ArrayUtil.productOfList(integers));  
  
6        List<String> strings1 = Arrays.asList("alpha", "beta", "charlie");  
        System.out.println("Sum = " + ArrayUtil.sumOfList(strings1));  
9        // Compile-time error: The method sumOfList(List<? extends Number>) in the  
        // type ArrayUtil is not applicable for the arguments (List<String>) ...  
11  
        List strings2 = Arrays.asList("alpha", "beta", "charlie");  
13        System.out.println("Sum = " + ArrayUtil.sumOfList(string2));  
        // Run-time error: Exception in thread "main" java.lang.ClassCastException:  
15        // java.lang.String cannot be cast to java.lang.Number  
    }  
17 }
```

- The **Unbounded Wildcard Type** is specified using the wildcard character (?), for example, `List<?>`. This is called a list of **unknown type**.
- There are two scenarios where an unbounded wildcard is a useful approach:
 - ▶ If you are writing a method that can be implemented using functionality provided in the `Object` class.
 - ▶ When the code is using methods in the generic class that **don't depend** on the type parameter.
- Example of printing a list of any type.



```
1 public static void printList(List<?> list) {  
    for (Object element : list) {  
3        System.out.print(element + " ");  
    }  
5    System.out.println();  
}
```


- A **Lower Bounded Wildcard** restricts the unknown type to be a **specific type or a super type of that type**.
 - ▶ Say you want to write a method that puts **Integer** objects into a list. To maximize flexibility, you would like the method to work on **List<Integer>**, **List<Number>**, and **List<Object>** - anything that can hold **Integer** values.
- A lower bounded wildcard is expressed using the wildcard character ('?'), following by the **super** keyword, followed by its **lower bound**: **<? super A>**.
 - ▶ To write the method that works on lists of **Integer** and the supertypes of **Integer**, such as **Integer**, **Number**, and **Object**, you would specify **List<? super Integer>**.
 - ▶ The term **List<Integer>** is more restrictive than **List<? super Integer>** because the former matches a list of type **Integer** only, whereas the latter matches a list of any type that is a supertype of **Integer**.

- Example of adding the numbers 1 through 10 to the end of a list:



```
public static void addNumbers(List<? super Integer> list) {  
2   for (int i = 1; i <= 10; i++) {  
       list.add(i);  
4   }  
}
```

- One of the more confusing aspects when learning to program with generics is determining when to use an **upper bounded wildcard** and when to use a **lower bounded wildcard**.
- For purposes of this discussion, it is helpful to think of variables as providing one of two functions:
 - ▶ **An "In" Variable:** An "in" variable serves up data to the code. Imagine a copy method with two arguments: `copy(src, dest)`. The `src` argument provides the data to be copied, so it is the "in" parameter.
 - ▶ **An "Out" Variable:** An "out" variable holds data for use elsewhere. In the copy example, `copy(src, dest)`, the `dest` argument accepts data, so it is the "out" parameter.
- **Wildcard Guidelines:**
 - ▶ An **"in" variable** is defined with an upper bounded wildcard, using the **extends** keyword.
 - ▶ An **"out" variable** is defined with a lower bounded wildcard, using the **super** keyword.
 - ▶ In the case where the **"in" variable** can be accessed using methods defined in the **Object** class, use an **unbounded wildcard**.
 - ▶ In the case where the code needs to access the variable as **both an "in" and an "out" variable**, **do not use a wildcard**.

- As you already know, it is possible to assign an object of one type to an object of another type provided that the types are compatible.



```
1 Object someObject = new Object();  
  Integer someInteger = new Integer(10);  
3 someObject = someInteger; // Ok
```

- In object-oriented terminology, this is called an "IS-A" relationship. Since an **Integer** is a kind of **Object**, the assignment is allowed. But **Integer** is also a kind of **Number**, so the following code is valid as well:



```
1 public void someMethod(Number n) { /* ... */ }  
  
3 someMethod(new Integer(10)); // Ok  
  someMethod(new Double(10.1)); // Ok
```

- You can also perform a generic type invocation, passing `Number` as its type argument, and any subsequent invocation of `add` will be allowed if the argument is compatible with `Number`:



```
1 Box<Number> box = new Box<Number>();  
  box.add(new Integer(10));    // Ok  
3 box.add(new Double(10.1));  // Ok
```

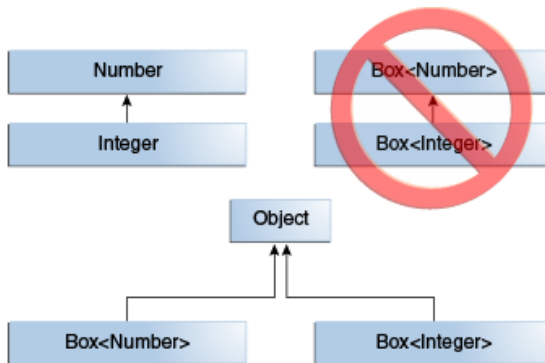
- Now consider the following method:



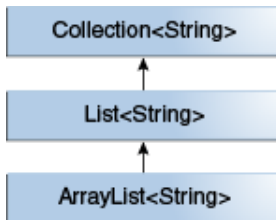
```
1 public void boxTest(Box<Number> n) { /* ... */ }
```

- You can see that it accepts a single argument whose type is `Box<Number>`. But you can not pass in `Box<Integer>` and `Box<Double>`, because they are not subtypes of `Box<Number>`.

- Given two concrete types *A* and *B* (for example, *Number* and *Integer*), *MyClass<A>* has no relationship to *MyClass*, regardless of whether or not *A* and *B* are related. The common parent of *MyClass<A>* and *MyClass* is *Object*.



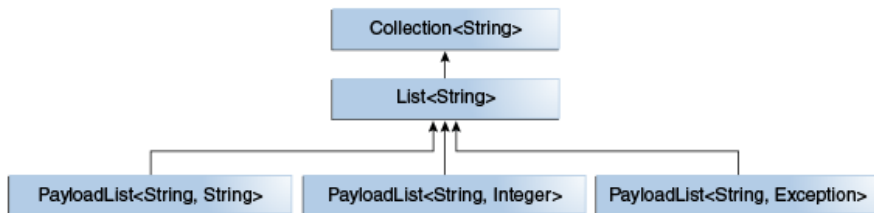
- You can subtype a generic class or interface by extending or implementing it. The relationship between the type parameters of one class or interface and the type parameters of another are determined by the extends and implements clauses.
- Using the **Collections** classes as an example, `ArrayList<E>` implements `List<E>`, and `List<E>` extends `Collection<E>`. So `ArrayList<String>` is a subtype of `List<String>`, which is a subtype of `Collection<String>`. So long as you do not vary the type argument, the subtyping relationship is preserved between the types.



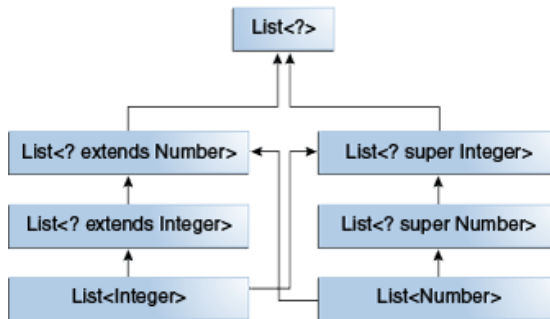
- Imagine we want to define our own list interface, `PayloadList`, that associates an optional value of generic type `P` with each element. Its declaration might look like:



```
1 interface PayloadList<E,P> extends List<E> {  
    void setPayload(int index, P val);  
3    ...  
    }  
5 // The following parameterizations of PayloadList are subtypes of List<String>:  
   PayloadList<String,String>; PayloadList<String,Integer>; PayloadList<String,Exception>
```



- You can use wildcards to create a relationship between generic classes or interfaces.



```
1 List<? extends Integer> intList = new ArrayList<>();  
2 List<? extends Number> numList = intList; // OK. List<? extends Integer> is a  
   ↳ subtype of List<? extends Number>
```

- Generics were introduced to the Java language to provide **tighter type checks at compile time** and to support generic programming.
- In Java, there is a process called type **erasure**, through which, type information is **removed** during compilation and there is no way to tell what was the type of a generic when it was instantiated during run-time.
- The Java compiler applies type erasure to:
 - ▶ Replace all type parameters in generic types with **their bounds** or **Object** if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
 - ▶ Insert type casts if necessary to preserve type safety.
 - ▶ Generate bridge methods to preserve polymorphism in extended generic types.
- Type erasure ensures that no new classes are created for parameterized types; consequently, generics incur no runtime overhead.

- Because the type parameter **T** is unbounded, the Java compiler replaces it with **Object**:



```
1 public class Node<T> {  
    private T data;  
3     private Node<T> next;  
  
5     public Node(T data, Node<T> next) {  
        this.data = data;  
7        this.next = next;  
    }  
  
9  
    public T getData() {  
11        return data;  
    }  
13 }
```



```
1 public class Node {  
    private Object data;  
3     private Node next;  
  
5     public Node(Object data, Node next) {  
        this.data = data;  
7        this.next = next;  
    }  
  
9  
    public Object getData() {  
11        return data;  
    }  
13 }
```

- The Java compiler replaces the bounded type parameter **T** with the first bound class, Comparable:



```
1 public class Node<T extends  
    ↳ Comparable<T>> {  
    private T data;  
3    private Node<T> next;  
  
5    public Node(T data, Node<T> next) {  
        this.data = data;  
7        this.next = next;  
    }  
  
9    public T getData() {  
11       return data;  
    }  
13 }
```



```
1 public class Node {  
    private Comparable data;  
3    private Node next;  
  
5    public Node(Comparable data, Node  
        ↳ next) {  
        this.data = data;  
7        this.next = next;  
    }  
  
9    public Comparable getData() {  
11       return data;  
    }  
13 }
```



```
1 // Source code
  public class ArrayUtil {
3     public static <T> T getCentral(T[] arr) {
        if (arr == null || arr.length == 0) {
5         return null;
        }
7     return (arr[arr.length / 2]);
    }
9 }

11 // Code after erasure
  public class ArrayUtil {
13     public static Object getCentral(Object arr[]) {
        if (arr == null || arr.length == 0) {
15         return null;
        } else {
17         return arr[arr.length / 2];
        }
19     }
  }
```



```
public class Main {
2   public static void main(String args[]) {
    String strings[] = {"alpha", "beta", "charlie"};
4   Character characters[] = {Character.valueOf('h'),
    Character.valueOf('a'), Character.valueOf('l')};
6   Integer integers[] = {Integer.valueOf(4), Integer.valueOf(8),
    Integer.valueOf(15), Integer.valueOf(16),
8   Integer.valueOf(23), Integer.valueOf(42) };
    Double doubles[] = {Double.valueOf(1.1000000000000001D),
10   Double.valueOf(2.2999999999999998D),
    Double.valueOf(5.7999999999999998D),
12   Double.valueOf(13.2100000000000001D)};

14   String centralString = (String) ArrayUtil.getCentral(strings);
    Character centralChar = (Character) ArrayUtil.getCentral(chars);
16   int centralInteger = ((Integer) ArrayUtil.getCentral(integers)).intValue();
    Double centralDouble = (Double) ArrayUtil.getCentral(doubles);
18 }
}
```

- It keeps things simple, in that generics do not add anything fundamentally new.
- It keeps things small, for example, there is exactly one implementation of [List](#), not one version for each type.
- It eases evolution, since the same library can be accessed in both nongeneric and generic forms.
 - ▶ You don't get problems due to maintaining two versions of the libraries:
 - A nongeneric legacy version that works with Java 1.4 or earlier.
 - A generic version that works with Java 5 or next.
- Cast-iron guarantee: The implicit casts added by the compilation of generics never fail.

- The Java compiler erases type parameters, replacing them with their bounds or Objects.
- Because the Java compiler erases all type parameters in generic code, you cannot verify which parameterized type for a generic type is being used at runtime.
- The term erasure is a slight misnomer, since the process erases type parameters but adds casts.
- Knowing about type erasure helps you understand limitations of Java generics:
 - ▶ Cannot instantiate generic types with primitive types.
 - ▶ Cannot create instances of type parameters.
 - ▶ Cannot declare static fields whose types are type parameters.
 - ▶ Cannot use casts or **instanceof** with parameterized types.
 - ▶ Cannot create arrays of parameterized types.
 - ▶ Cannot create, catch, or throw Objects of parameterized types.
 - ▶ Cannot overload a method where the formal parameter types of each overload erase to the same raw type.

- Cannot instantiate generic types with primitive types.



```
1 public class Pair<K, V> {  
    private K key;  
3    private V value;  
  
5    public Pair(K key, V value) {  
        this.key = key;  
7        this.value = value;  
    }  
9  
    // ...  
11 }  
  
13 Pair<int, char> p = new Pair<>(8, 'a'); // Compile-time error  
15 Pair<Integer, Character> p = new Pair<>(8, 'a'); // Ok
```

- Cannot create instances of type parameters



```
1 public static <E> void append(List<E> list) {  
    E elem = new E(); // Compile-time error  
3    list.add(elem);  
}
```

- As a workaround, you can create an object of a type parameter through reflection:



```
public static <E> void append(List<E> list, Class<E> cls) throws Exception {  
2    E elem = cls.newInstance(); // Ok  
    list.add(elem);  
4 }
```

- Cannot declare static fields whose types are type parameters.
 - ▶ A class's static field is a class-level variable shared by all non-static objects of the class. Hence, static fields of type parameters are not allowed.



```
public class MobileDevice<T> {  
2   private static T os;  
  
4   // ...  
}  
  
6  
// If static fields of type parameters were allowed ,  
8 //   then the following code would be confused:  
MobileDevice<Smartphone> phone = new MobileDevice<>();  
10 MobileDevice<Pager> pager = new MobileDevice<>();  
MobileDevice<TabletPC> pc = new MobileDevice<>();
```

- Cannot use casts or **instanceof** with parameterized types.
 - ▶ Because the Java compiler erases all type parameters in generic code, you cannot verify which parameterized type for a generic type is being used at runtime:



```
1 public static <E> void rtti(List<E> list) {  
    if (list instanceof ArrayList<Integer>) { // Compile-time error  
3      // ...  
    }  
5 }
```

- ▶ The set of parameterized types passed to the rtti method is:
 $S = \text{ArrayList<Integer>}, \text{ArrayList<String>}, \text{LinkedList<Character>}, \dots$

The runtime does not keep track of type parameters, so it cannot tell the difference between an `ArrayList<Integer>` and an `ArrayList<String>`.

- Cannot create arrays of parameterized types.



```
1 List<Integer>[] arrayOfLists = new List<Integer>[2]; // Compile-time error
```

- The following code illustrates the problems.



```
1 Object[] strings = new String[2];  
  strings[0] = "hi"; // OK  
3 strings[1] = 100; // An ArrayStoreException is thrown.  
  
5 Object[] stringLists = new List<String>[2]; // Compiler error, but pretend it's allowed  
  stringLists[0] = new ArrayList<String>(); // Ok  
7 stringLists[1] = new ArrayList<Integer>(); // An ArrayStoreException should be thrown,  
                                              // but the runtime can't detect it.
```

- Cannot create, catch, or throw objects of parameterized types.



```
// Extends Throwable indirectly
2 class MathException<T> extends Exception { /* ... */ } // Compile-time error

4 // Extends Throwable directly
class QueueFullException<T> extends Throwable { /* ... */ // Compile-time error
6

8 /*
 * A method cannot catch an instance of a type parameter:
 */
10 public static <T extends Exception, J> void execute(List<J> jobs) {
12     try {
14         for (J job : jobs)
16             // ...
18     } catch (T e) { // Compile-time error
19         // ...
20     }
21 }
```

- Cannot overload a method where the formal parameter types of each overload erase to the same raw type.
 - ▶ A class cannot have two overloaded methods that will have the same signature after type erasure. The overloads would all share the same classfile representation and will generate a compile-time error.



```
public class Example {  
2   public void print(Set<String> strSet) { }  
   public void print(Set<Integer> intSet) { }  
4 }
```

- Java Collection Framework is a unified architecture for **representing and manipulating collections**, enabling collections to be manipulated independently of implementation details.
 - ▶ Reduces programming effort by **providing data structures and algorithms** so you don't have to write them yourself.
 - ▶ Increases performance by providing **high-performance implementations** of data structures and algorithms.
 - ▶ Fosters software reuse by providing a standard interface for collections and algorithms with which to manipulate them.



```
import java.util.ArrayList;
2 import java.util.Iterator;

4 public class PreJDK5Test {
    public static void main(String[] args) {
6         ArrayList list = new ArrayList(); // ArrayList contains instances of Object
        list.add("alpha"); // add() takes Object. String upcast to Object implicitly
8         list.add("beta");
        list.add("charlie");
10        list.add(new Integer(10)); // Integer upcast to Object implicitly
        System.out.println(list); // [alpha, beta, charlie, 10]

12
        Iterator iter = list.iterator();
14        while (iter.hasNext()) {
            // Explicitly downcast from Object to String
16            String str = (String) iter.next(); // Error
            System.out.println(str);
18        }
20    }
```



```
public class PostJDK5Test {
2   public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>(); // Inform compiler about type.
4       list.add("alpha"); // Compiler checks if argument's type is String.
        list.add("beta");
6       list.add("charlie");
        System.out.println(list); // [alpha, beta, charlie]

8
        Iterator<String> iter = list.iterator(); // Iterator of Strings
10       while (iter.hasNext()) {
            String str = iter.next(); // Compiler inserts downcast operator
12         System.out.println(str);
        }

14
        list.add(new Integer(1234)); // Error: Compiler can detect wrong type.
16         // Error: no suitable method found for add(Integer)

18         Integer integer = list.get(0); // Error: Compiler can detect wrong type
        // Error: incompatible types: String cannot be converted to Integer
20     }
}
```

- 1 Introduction
- 2 Generic Classes**
- 3 Generic Methods
- 4 Wildcard Types
- 5 Bounded Types
- 6 References

- Suppose we need to implement a data structure queue of strings, we can create a new class that contains the linked list as a private instance variable



```

1 public class StringQueue {
    private List<String> items = new LinkedList<>();
3
    public void enqueue(String item) {
5        items.addLast(item);
    }
7
    public String dequeue() {
9        return items.removeFirst();
    }
11
    public boolean isEmpty() {
13        return (items.size() == 0);
    }
15 }
    
```

- But, if this is how we write queue classes, and if we want queues of **Integers** or **Doubles** or **Colors** or any other type, then **we will have to write a different class for each type**. The code for all of these classes will be almost identical, which seems like a lot of redundant programming.
- To avoid the redundancy, **we can write a generic Queue class that can be used to define queues of any type of object**.
- The syntax for writing the **generic class** is straightforward: We replace the specific type **String** with a type parameter such as **T**, and we add the type parameter to the name of the class



```
1 public class Queue<T> {  
    private List<T> items =  
3         new LinkedList<>();  
  
5     public void enqueue(T item) {  
        items.addLast(item);  
7     }  
  
9     public T dequeue() {  
        return items.removeFirst();  
11    }  
  
13    public boolean isEmpty() {  
        return (items.size() == 0);  
15    }  
}
```

- Note that you don't have to use "T" as the name of the type parameter in the definition of the generic class. Type parameters are like formal parameters in subroutines.
- You can make up any name you like in the definition of the class. The name in the definition will be replaced by an actual type name when the class is used to declare variables or create objects.
- If you prefer to use a more meaningful name for the type parameter, you might define the `Queue` class as following.



```
public class Queue<ItemType> {  
2   private List<ItemType> items =  
       new LinkedList<>();  
4  
   public void enqueue(ItemType item) {  
6       items.addLast(item);  
   }  
8  
   public ItemType dequeue() {  
10      return items.removeFirst();  
   }  
12  
   public boolean isEmpty() {  
14      return (items.size() == 0);  
   }  
16 }
```

- It's also easy to define generic classes and interfaces that have two or more type parameters, as is done with the standard interface `Map<K, V>`.
- A typical example is the definition of a "`Pair`" that contains two objects, possibly of different types.



```
1 public class Pair<T,S> {  
2     public T first;  
3     public S second;  
4  
5     public Pair(T a, S b) {  
6         first = a;  
7         second = b;  
8     }  
9 }  
10  
11 /* main */  
12 Pair<String, Color> colorName =  
13     new Pair<>("Red", Color.RED);  
14  
15 Pair<Double, Double> coordinates  
16     = new Pair<>(17.3, 42.8);
```

- 1 Introduction
- 2 Generic Classes
- 3 Generic Methods**
- 4 Wildcard Types
- 5 Bounded Types
- 6 References

- In addition to **generic classes**, Java also has **generic methods**. An example is the method *Collections.sort()*, which can sort collections of objects of any type.
- To see how to write generic methods, let's start with a non-generic method for counting the number of times that a given **string occurs in an array of strings**.
- **How about another types?**



```
public static int countOccurrences(  
2   String[] list, String item) {  
    if (item == null) {  
4       int count = 0;  
        for (String listItem : list) {  
6           if (listItem == null)  
                count++;  
8       }  
        return count;  
10    }  
  
12    int count = 0;  
    for (String listItem : list) {  
14        if (item.equals(listItem))  
                count++;  
16    }  
    return count;  
18 }  
  
20 int ct = countOccurrences(wordList, word); // OK  
    int ct = countOccurrences(numbers, 17); // ???
```

- By writing a **generic method**, we get to write a single method definition that will **work for objects of any non-primitive type**.
- **We need to replace the specific type `String` with type parameter, such as `T`.** However, if that's the only change we make, the compiler will think that "`T`" is the name of an actual type, and it will mark it as an undeclared identifier.
- We need some way of telling the compiler that "`T`" is a type parameter. That's what the "`<T>`" goes just before the name of the return type of the method.



```
1 public static <T> int countOccurrences(  
    T[] list, T item) {  
3     if (itemToCount == null) {  
        int count = 0;  
5         for (T listItem : list) {  
             if (listItem == null)  
3                 count++;  
6         }  
9         return count;  
10    }  
11  
    int count = 0;  
13    for (T listItem : list) {  
        if (item.equals(listItem))  
15            count++;  
17    }  
18    return count;  
19 }  
20  
21 int ct = countOccurrences(wordList, word);  
    int ct = countOccurrences(numbers, 17);
```

- A generic method can have one or more type parameters, such as the "T" in `countOccurrences()`.
- The compiler can deduce the type from the types of the actual parameters in the method call. Since `wordlist` is of type `String[]`, the compiler can tell that the type that replaces `T` is `String`.
- The `countOccurrences` method operates on an array. We could also write a similar method to count occurrences of an object in any collection.



```
1 public static <T> int countOccurrences(  
    Collection<T> collection, T item) {  
3     if (item == null) {  
        int count = 0;  
5         for (T item : collection) {  
             if (item == null)  
3                 count++;  
7         }  
9         return count;  
    }  
11  
    int count = 0;  
13    for (T item : collection) {  
        if (itemToCount.equals(item))  
15            count++;  
    }  
17    return count;  
}
```

- 1 Introduction
- 2 Generic Classes
- 3 Generic Methods
- 4 Wildcard Types**
- 5 Bounded Types
- 6 References

- There is a limitation on the sort of generic classes and methods that we have looked at so far:
 - ▶ The type parameter in our examples, usually named **T**, can be any type at all. This is Ok in many cases, but it means that the only things that you can do with **T** are things that can be done with every type, and the only things that you can do with objects of type **T** are things that you can do with every object.
 - ▶ Suppose that we want to write a generic method that compare objects with the `compareTo()` method. With the techniques that we have covered so far, you can't, since that method is not defined for all objects.
 - ▶ The `compareTo()` method is defined in the `Comparable` interface. What we need is a way of specifying that a generic class or method only applies to objects of type `Comparable` and not to arbitrary objects. With that restriction, we should be free to use `compareTo()` in the definition of the generic class or method.

- There are two different but related syntaxes for putting restrictions on the types that are used in generic programming.
 - ▶ **Bounded type parameters**, which are used as formal type parameters in generic class and method definitions; a **bounded type parameter would be used in place of the simple type parameter `T`** in "class `GenericClass<T>` ..." or in "public static `<T>` void `genericMethod`(...".
 - ▶ **Wildcard types**, which are used as type parameters in the declarations of variables and of formal parameters in method definitions; a **wildcard type could be used in place of the type parameter `String`** in the declaration statement "`List<String> list;`" or in the formal parameter list "void `concat(Collection<String> c)`".

- Suppose that we want a method that can draw all the shapes in a collection of `Shapes`.



```
public static void drawAll(Collection<Shape> shapes) {  
2   for (Shape s : shapes) {  
    s.draw();  
4   }  
}
```

- This method works fine if we apply it to a variable of type `Collection<Shape>`, or `ArrayList<Shape>`, or any other collection class with type parameter `Shape`.
- Suppose that `Shape` has subclasses such as `Rect` and `Oval`. Suppose that you have a list of `Rects` stored in a variable named `rectangles` of type `Collection<Rect>`. Since `Rects` are `Shapes`, you expect to be able to call `drawAll(rectangles)`.
- Unfortunately, this will not work; **a collection of `Rects` is not considered to be a collection of `Shapes`!** The variable `rectangles` cannot be assigned to the formal parameter `shapes`.

- The solution is to replace the type parameter "**Shape**" in the declaration of *shapes* with the wildcard type "**? extends Shape**":



```
1 public static void drawAll(Collection<? extends Shape> shapes) {  
    for (Shape s : shapes) {  
3         s.draw();  
    }  
5 }
```

- The wildcard type "**? extends Shape**" means roughly "any type that is either equal to **Shape** or that is a subclass of **Shape**". We can pass actual parameters to *drawAll* of types **Collection<Rect>** since **Rect** is a subclass of **Shape** and therefore matches the wildcard.
- We could also pass actual parameters to *drawAll* of type **ArrayList<Rect>** or **Set<Oval>**. And we can still pass variables of type **Collection<Shape>** or **ArrayList<Shape>**, since the class **Shape** itself matches "**? extends Shape**".

- You might be interested in knowing why Java does not allow a collection of **Rects** to be used as a collection of **Shapes**, even though every **Rect** is considered to be a **Shape**. Consider the rather silly but legal method that adds an oval to a list of shapes:



```
1 static void addOval(List<Shape> shapes, Oval oval) {  
    shapes.add(oval);  
3 }
```

- Suppose that *rectangles* is of type **List<Rect>**. It's illegal to call *addOval(rectangles, oval)*, because of the rule that a list of **Rects** is not a list of **Shapes**.
- If we dropped that rule, then *addOval(rectangles, oval)* would be legal, and it would add an **Oval** to a list of **Rects**. This would be bad: since **Oval** is not a subclass of **Rect**, an **Oval** is not a **Rect**, and a list of **Rects** should never be able to contain an **Oval**. The method call *addOval(rectangles, oval)* does not make sense and should be illegal.)

- In method *addAll*, we don't want to require *coll* to be a collection of objects of type *T*, we want to allow collections of any subclass of *T*.
- This makes sense because any object of type *S* is automatically of type *T* and so can legally be added to *coll*.



```
1 public interface Queue<T> {  
    public void enqueue(T item);  
3    public T dequeue();  
    public boolean isEmpty();  
5    public void addAll(Collection<T> coll);  
}
```



```
public class Queue<T> {  
2   private List<T> items = new LinkedList<T>();  
  
4   public void enqueue(T item) {  
    items.addLast(item);  
6   }  
  
8   public T dequeue() {  
    return items.removeFirst();  
10  }  
  
12  public boolean isEmpty() {  
    return (items.size() == 0);  
14  }  
  
16  public void addAll(Collection<? extends T> coll) {  
    for (T item : coll) {  
18      enqueue(item); // Add all the items to the end of the queue  
    }  
20  }  
}
```

- Here, **T** is a type parameter in the generic class definition. We are combining wildcard types with generic classes. Inside the generic class definition, "**T**" is used as if it is a specific, though unknown, type.
- The wildcard type "**? extends T**" means some type that is equal to or extends that specific type. When we create a queue of type `Queue<Shape>`, "**T**" refers to "**Shape**", and the wildcard type "**? extends T**" in the class definition means "**? extends Shape**". This ensures that the *addAll* method of the queue can be applied to collections of **Rects** and **Ovals** as well as to collections of **Shapes**.
- The for-each loop in the definition of *addAll* iterates through the collection using a variable, *item*, of type **T**. Now, collection can be of type `Collection<S>`, where **S** is a subclass of **T**. Since *item* is of type **T**, not **S**, do we have a problem here? No, no problem. As long as **S** is a subclass of **T**, a value of type **S** can be assigned to a variable of type **T**. The restriction on the wildcard type makes everything work nicely.

- The *addAll* method adds all the items from a collection to the queue. Suppose that we wanted to do the opposite: Add all the items that are currently in the queue to a given collection. An instance method defined as



```
1 public class Queue<T> {  
    private List<T> items = new LinkedList<T>();  
3  
    ...  
5  
    public void addAllTo(Collection<T> collection) { }  
7 }
```

- Would only work for collections whose base type is exactly the same as `T`. This is too restrictive.
- We need some sort of wildcard.

- We think about something like "? extends T". However, "? extends T" won't work. Suppose we try it:



```
1 public void addAllTo(Collection<? extends T> collection) {  
    // Remove all items currently on the queue and add them to collection  
3 while (!isEmpty()) {  
    T item = dequeue(); // Remove an item from the queue.  
5    collection.add(item); // Add it to the collection. ILLEGAL!!  
    }  
7 }
```

- The problem is that we can't add an item of type T to a collection that might only be able to hold items belonging to some subclass, S, of T. The containment is going in the wrong direction: An item of type T is not necessarily of type S. For example, if we have a queue of type Queue<Shape>, it doesn't make sense to add items from the queue to a collection of type Collection<Rect>, since not every Shape is a Rect.

- If we have a `Queue<Rect>`, it would make sense to add items from that queue to a `Collection<Shape>` or indeed to any collection `Collection<S>` where `S` is a superclass of `Rect`. To express this type of relationship, we need a new kind of type wildcard: `"? super T"`. This wildcard means, roughly, "either `T` itself or any class that is a superclass of `T`".
- For example, `Collection<? super Rect>` would match the types `Collection<Shape>`, `Set<Rect>`, and `ArrayList<Object>`.



```
1 class Queue<T> {  
  ...  
3  public void addAllTo(Collection<? super T> collection) {  
    // Remove all items currently on the queue and add them to collection  
5    while (!isEmpty()) {  
      T item = dequeue();    // Remove an item from the queue.  
7      collection.add(item); // Add it to the collection.  
    }  
9  }  
}
```

- 1 Introduction
- 2 Generic Classes
- 3 Generic Methods
- 4 Wildcard Types
- 5 Bounded Types**
- 6 References

- Wildcard types don't solve all of our problems.
 - ▶ They allow us to generalize method definitions so that they can work with collections of objects of various types, rather than just a single type.
 - ▶ However, they do not allow us to restrict the types that are allowed as formal type parameters in a generic class or method definition. Bounded types exist for this purpose.
- Suppose that you would like to create groups of GUI components using a generic class named **ControlGroup**. The class will include methods that can be called to apply certain operations to all components in the group at once.



```
public void disableAll () {  
2   .  
   .    // Call c.setDisable(true) for every control, c, in the group.  
4   .  
   }  
}
```

- The problem is that the *setDisable()* method is defined in a **Control** object, but not for objects of arbitrary type. It wouldn't make sense to allow types such as **ControlGroup<String>** or **ControlGroup<Integer>**, since **Strings** and **Integers** don't have *setDisable()* methods.
- We need some way to restrict the type parameter **T** in **ControlGroup<T>** so that only **Control** and subclasses of **Control** are allowed as actual type parameters.
- We can do this by using the bounded type "**T extends Control**" instead of a plain "**T**" in the definition of the class.



```
1 public class ControlGroup<T extends Control> {  
    private List<T> components;  
  
3     public void disableAll( ) {  
5         for (Control c : components) {  
7             if (c != null)  
                c.setDisable(true);  
9         }  
  
11        public void enableAll( ) {  
12            for (Control c : components) {  
13                if (c != null)  
                    c.setDisable(false);  
15            }  
16        }  
  
17        public void add(T c) {  
18            components.add(c);  
19        }  
20        ...  
21    }
```

- In general, a bounded type parameter "`T` extends `SomeType`" means roughly.
 - ▶ A type, `T`, that is either equal to `SomeType` or is a subclass of `SomeType`, the upshot is that any object of type `T` is also of type `SomeType`, and any operation that is defined for objects of type `SomeType` is defined for objects of type `T`.
 - ▶ The type `SomeType` doesn't have to be the name of a class. It can be any name that represents an actual object type. For example, it can be an interface or even a parameterized type.
- Bounded types and wildcard types are clearly related. They are, however, used in very different ways.
 - ▶ A **bounded type** can be used only as a **formal type parameter in the definition of a generic method, class, or interface**.
 - ▶ A **wildcard type** is used most often to **declare the type of a formal parameter in a method and cannot be used as a formal type parameter**.
 - ▶ One other difference, by the way, is that, in contrast to wildcard types, bounded type parameters can only use "extends", never "super".

- Bounded type parameters can be used when declaring generic methods. For example, as an alternative to the generic `ControlGroup` class, one could write a free-standing generic static method that can disable any collection of `Controls` as follows:



```
public static <T extends Control> void disableAll(Collection<T> components) {  
2   for (Control c : components) {  
    if (c != null) {  
4       c.setDisable(true);  
    }  
6   }  
}
```

- Using "<T extends Control>" as the formal type parameter means that the method can only be called for collections whose base type is `Control` or some subclass of `Control`, such as `Button` or `Slider`.

- Note that we don't really need a generic type parameter in this case. We can write an equivalent method using a wildcard type:



```
1 public static void disableAll(Collection<? extends Control> components) {  
    for (Control c : components) {  
3        if (c != null) {  
            c.setDisable(true);  
5        }  
        }  
7    }
```

- In this situation, the version that uses the wildcard type is to be preferred, since the implementation is simpler.

- However, there are some situations where a generic method with a bounded type parameter cannot be rewritten using a wildcard type.
- Note that a generic type parameter gives a name, such as `T`, to the unknown type, while a wildcard type does not give a name to the unknown type. The name makes it possible to refer to the unknown type in the body of the method that is being defined. If a generic method definition uses the generic type name more than once or uses it outside the formal parameter list of the method, then the generic type parameter cannot be replaced with a wildcard type.



```
1 static void sortedInsert(List<String> sortedList, String newItem) {  
    ListIterator<String> iter = sortedList.listIterator();  
3 while (iter.hasNext()) {  
    String item = iter.next();  
5    if (newItem.compareTo(item) <= 0) {  
        iter.previous();  
7        break;  
    }  
9 }  
    iter.add(newItem);  
11 }
```

- This method works fine for lists of strings, but it would be nice to have a generic method that can be applied to lists of other types of objects.
- The problem is that the code assumes that the *compareTo()* method is defined for objects in the list, so the method can only work for lists of objects that implement the [Comparable](#) interface. We can't simply use a wildcard type to enforce this restriction.

- Suppose we try to do it, by replacing `List<String>` with `List<? extends Comparable>`:



```
1 static void sortedInsert(List<? extends Comparable> sortedList, ?? newItem) {  
    ListIterator<??> iter = sortedList.listIterator();  
3    ...
```










- We immediately run into a problem, because we have no name for the unknown type represented by the wildcard. We need a name for that type because the type of `newItem` and of `iter` should be the same as the type of the items in the list.

- The problem is solved if we write a generic method with a bounded type parameter, since then we have a name for the unknown type, and we can write a valid generic method



```
1 static <T extends Comparable<T>> void sortedInsert(  
    List<T> sortedList, T newItem) {  
3     ListIterator<T> iter = sortedList.listIterator();  
    while (iter.hasNext()) {  
5         T item = iter.next();  
        if (newItem.compareTo(item) <= 0) {  
7             iter.previous();  
            break;  
9         }  
    }  
11  
    iter.add(newItem);  
13 }
```

- 1 Introduction
- 2 Generic Classes
- 3 Generic Methods
- 4 Wildcard Types
- 5 Bounded Types
- 6 References**

-  ALLEN B. DOWNEY, CHRIS MAYFIELD, ***THINK JAVA***, (2016).
-  GRAHAM MITCHELL, ***LEARN JAVA THE HARD WAY***, 2ND EDITION, (2016).
-  CAY S. HORSTMANN, ***BIG JAVA - EARLY OBJECTS***, 7E-WILEY, (2019).
-  JAMES GOSLING, BILL JOY, GUY STEELE, GILAD BRACHA, ALEX BUCKLEY, ***THE JAVA LANGUAGE SPECIFICATION - JAVA SE 8 EDITION***, (2015).
-  MARTIN FOWLER, ***UML DISTILLED - A BRIEF GUIDE TO THE STANDARD OBJECT MODELING LANGUAGE***, (2004).
-  RICHARD WARBURTON, ***OBJECT-ORIENTED VS. FUNCTIONAL PROGRAMMING - BRIDGING THE DIVIDE BETWEEN OPPOSING PARADIGMS***, (2016).
-  NAFTALIN, MAURICE WADLER, PHILIP ***JAVA GENERICS AND COLLECTIONS***, O'REILLY MEDIA, (2009).
-  ERIC FREEMAN, ELISABETH ROBSON ***HEAD FIRST DESIGN PATTERNS - BUILDING EXTENSIBLE AND MAINTAINABLE OBJECT***, ORIENTED SOFTWARE-O'REILLY MEDIA, (2020).
-  ALEXANDER SHVETS ***DIVE INTO DESIGN PATTERNS***, (2019).

THANK YOU!