Static nested classes :

* Related to the enclosing class and not its instances
* They are like top level classes.
* Can only access static members of enclosing class directly.
* Can access instance members of enclosing class only through objects.
* Can have both static members.
* Can have instance members.
* Instantiated using the syntax -

OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();

Inner classes :

* Related to instances of enclosing class.
* Can access both static and instance members of enclosing class directly.
* Cannot have static members of its own other than constant variables(A *constant variable* is a variable of primitive type or type String that is declared final and initialized with a compile-time constant expression. A compile-time constant expression is typically a string or an arithmetic expression that can be evaluated at compile time).
* Cannot have interface declarations.
* Instantiated using the syntax:

OuterClass.InnerClass innerObject = outerObject.new InnerClass()

* Used to handle user interface events
* To avoid shadowing:

ParentClass.this.x(Parent class field) || this.x(inner class field) || |x(inner class local variable)

Local classes:

* Within static method: Can only access static members of enclosing class directly.
* Within instance method: Can access both static and instance members of enclosing class directly.
* Cannot have static members of its own other than constant variables(A *constant variable* is a variable of primitive type or type String that is declared final and initialized with a compile-time constant expression. A compile-time constant expression is typically a string or an arithmetic expression that can be evaluated at compile time).
* Cannot have interface declarations.
* Can access parameters and effectively final local members of enclosing block.
* Can be instantiated only within the method.
* To avoid shadowing:

ParentClass.this.x(Parent class field) || ParentClass.x(Parent class local variable) || this.x(inner class field) || x(inner class local variable)

Anonymous classes:

* Same as local classes
* Don’t have a name
* Are defined and instantiated at the same time using syntax:

HelloWorld frenchGreeting = new HelloWorld() {  
 // class definition };  
 Where HelloWorld is the name of the interface to implement or class to extend

* It does not have a constructor as it has no name.
* Used in GUI Applications

Inheritance:

Super class A, Sub class B.

* Static methods from superclass are inherited but get hidden when the methods with the same name are defined in the sub class.
* Static methods are called by the specification type of the memory location that calls the method.
* The binding of a Static method to its invocation happens at compile-time.
* The super keyword is used to access hidden superclass variables and hidden/overrriden superclass methods.
* Super() calls the method from a virtual scope in which the specification is that of the super class but the instance is of the current child class. Therefore, the static methods will be called from superclass even if they have been hidden by the child class but the instance methods will be called from the child class - the overridden versions.
* Static methods of interfaces are never inherited by implementation classes.
* Instance methods in superclass get overridden when the methods with the same name are defined in the sub class.
* Static methods are called by the instance that calls the method.
* The binding of an instance method its invocation happens at run time.
* If an interface has a default method xyz and a class has instance method xyz, a class which implements the interface and extends the above class will use the implementation of the superclass and not the interface. Because Instance methods are preferred over interface default methods.
* When multiple interfaces {B,C} extend a default method from a common ancestor A, and B overrides the default method while C does not, any class implementing B and C wil inherit the overridden version of the method(i.e. The definition present in B will be inherited and not the one in A.)
* If two or more independently defined default methods conflict, then the Java compiler produces a compiler error. You must explicitly override the supertype methods as in the example:

public int commonDefaultMethodName() {  
 InterfaceName1.super.commonDefaultMethodName();  
 InterfaceName2.super.commonDefaultMethodName();  
 }

* The access specifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the superclass can be made public, but not private, in the subclass.
* In a subclass, you can overload the methods inherited from the superclass. Such overloaded methods neither hide nor override the superclass instance methods—they are new methods, unique to the subclass.
* If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error. Object *does* have such a constructor, so if Object is the only superclass, there is no problem. If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that there will be a whole chain of constructors called, all the way back to the constructor of Object. In fact, this is the case. It is called *constructor chaining*, and you need to be aware of it when there is a long line of class descent.
* Abstract class can allow methods from an interface it implements to be implemented in its subclasses by marking the definition as abstract.
* If a subtype instance is referenced by a supertype variable, then the supertype variable is allowed to call all methods inherited or overridden from the supertype. But it cannot be used to call a method exclusive to the subtype.
* A class A has a private field p. It has a public method pm() which sets the value of the field p to 5 and then returns p. Now we create a class B that extends A and we do not override the method pm(). Now we create an object of B and call pm() using this object. It returns 5.

In this case, how does the method fuction properly when called from object of B if the private field p is not inherited at all by the subclass B?

The JLS states this:

Members of a class that are declared private are not inherited by subclasses of that class. Only members of a class that are declared protected or public are inherited by subclasses declared in a package other than the one in which the class is declared.

Now this means since the subclass can't access or modify the private fields, then, in other words, they are not inherited. But this only applies to the class.

But when an object is created, it is supposed to be an object of both A and B. Hence, the object does contain the private fields as well. They are just not accessible from the class.

It is better to say that the private field of superclass are not accessible from child class. But they are indeed present in any object of the child class.

Example:

**public class** InheritencePrivateFields {

**private int c**;

**public int** getc(){

**c** = 10;

**return c**;

}

}

**public class** InheritanceChildClass **extends** InheritencePrivateFields {

**public static void** main(String args[]){

InheritanceChildClass obj = **new** InheritanceChildClass();

System.***out***.println(obj.getc());

}

}

The output from InheritanceChildClass will be 10.

* Storing a subtype into variable of a supertype ---> casting not needed.

Storing a supertype into variable of a subtype ---> casting needed

Storing a type into a variable of a type that is neither its subtype of supertype ---> casting not possible. Store one of them into a third variable of a common supertype (e.g. Object) and then cast that third variable down to the other type.

Eg.

List<? **extends** Number> unboundList = **new** ArrayList<>();

List<Integer> intList = **new** ArrayList<>();

intList.add(9000);

unboundList = intList;

List<Float> numList = (List<Float>) unboundList;  
Here List<Integer> and List<Float> aren’t related. But both of them are subtypes of List<? extends Number>.

* Marker interface. Does not have any methods or constants. It is just used to mark an operation. It says that the class which implements it have special needs. Example : clonable, serializable. In order to create a marker interface, just create an empty interface and use it.

Example:

public interface IMarkerEntity { }   
If any class which implement this interface will be taken as database entity by your application.  
Sample Code:  
public boolean save(Object object) throws InvalidEntityException { if(!(object instanceof IMarkerEntity)) { throw new InvalidEntityException("Invalid Entity Found, cannot proceed); } database.save(object); }

* Methods inherited from Object class:

**clone**: Allows to create clone of an object. Prerequisite: use the cloneable interface at the method declaration. And override the clone method.

class B implements Cloneable{

@Override

protected Object clone() throws CloneNotSupportedException

{

return super.clone();

}

}

In the main method, to create clone of object use:

B obj = new B**();**

B obj1 = (B) obj.clone();

**equals**: The equals() method provided by Object tests whether the object *references* are equal—that is, if the objects compared are the exact same object.

To test whether two objects are equal in the sense of *equivalency* (containing the same information), you must override the equals() method. Here is an example of a Book class that overrides equals():

public class Book {  
 ...  
 public boolean equals(Object obj) {  
 if (obj instanceof Book)  
 return ISBN.equals((Book)obj.getISBN());   
 else  
 return false;  
 }  
 }

**finalize:** The Object class provides a callback method, finalize(), that *may be* invoked on an object when it becomes garbage. Object's implementation of finalize() does nothing—you can override finalize() to do cleanup, such as freeing resources.

**getClass**: You cannot override getClass.

The getClass() method returns a Class object, which has methods you can use to get information about the class, such as its name (getSimpleName()), its superclass (getSuperclass()), and the interfaces it implements (getInterfaces()). For example, the following method gets and displays the class name of an object:

void printClassName(Object obj) {  
 System.out.println("The object's" + " class is " +  
 obj.getClass().getSimpleName());  
 }  
The [Class](https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html) class, in the java.lang package, has a large number of methods (more than 50). For example, you can test to see if the class is an annotation (isAnnotation()), an interface (isInterface()), or an enumeration (isEnum()). You can see what the object's fields are (getFields()) or what its methods are (getMethods()), and so on.

**tostring:** You should always consider overriding the toString() method in your classes.

The Object's toString() method returns a String representation of the object, which is very useful for debugging. The String representation for an object depends entirely on the object, which is why you need to override toString() in your classes.

You can use toString() along with System.out.println() to display a text representation of an object, such as an instance of Book:

System.out.println(firstBook.toString());  
which would, for a properly overridden toString() method, print something useful, like this:

ISBN: 0201914670; The Swing Tutorial; A Guide to Constructing GUIs, 2nd Edition

* Final methods: Cannot be overridden. Final class: cannot be subclassed. Final fields: constants.
* Abstract classes can be subclassed. But cannot be instantiated.
* If a class declares even one abstract method, it should be marked as abstract.
* Not all methods need to be abstract.
* A subclass that extends the abstract class must implement all of its abstract methods otherwise it should also be marked as abstract.
* An abstract class can implement multiple interfaces, but it does not need to define every abstract method present in the interface.
* An abstract class can have non-static and non-final fields, unlike interfaces.
* Static methods of abstract class will be called with class name as usual.

Generics:

* Introduces type parameters while defining classes, interfaces and methods.
* Type parameters enable us to reuse the same class with different input.
* Inputs to type parameters are types.(Unlike inputs to formal parameters which are values)
* A **generic type** is a class or interface that has type parameters.
* String objects are immutable.

This is how String works: String str = "knowledge".

This, as usual, creates a string containing "knowledge" and assigns it a reference str. Simple enough? Let's perform some more functions:

String s = str; // assigns a new reference to the same string "knowledge"

Let’s see how the below statement works:

str = str.concat(" base");

This appends a string " base" to str. But wait, how is this possible, since String objects are immutable? Well to your surprise, it is.

When the above statement is executed, the VM takes the value of String str, i.e. "knowledge"and appends " base", giving us the value "knowledge base". Now, since Strings are immutable, the VM can't assign this value to str, so it creates a new String object, gives it a value "knowledge base", and gives it a reference str.

An important point to note here is that, while the String object is immutable, **its reference variable is not.** So that's why, in the above example, the reference was made to refer to a newly formed String object.

At this point in the example above, we have two String objects: the first one we created with value "knowledge", pointed to by s, and the second one "knowledge base", pointed to by str. But, technically, we have three String objects, the third one being the literal "base" in the concat statement.

Java maintains a String pool which contains all String objects. Whenever a new String object is created, JVM tries to look for a matching String in the String pool. If a match is found, the new String variable is made to reference the existing String object in the String pool. Any String object in the String pool can have multiple references. But all of these String objects are immutable. So there will be no concurrency issues even when the same String object is being referenced by multiple variables. Because every time, a reference tries to change a String object, a new String object is formed.

* Non generic vs generic class:

Non generic class:

public class Box {  
 private Object object;  
 public void set(Object object) { this.object = object; }  
 public Object get() { return object; }  
 }

Generic class:

public class Box<T> {  
 // T stands for "Type"  
 private T t;  
 public void set(T t) { this.t = t; }  
 public T get() { return t; }  
 }

Types that can be passed to T: interface type, class type, array type, another type variable.

* 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

* To reference the generic Box class from within your code, you must perform a *generic type invocation*, which replaces T with some concrete value, such as Integer:

Box<Integer> integerBox;

Instantiation:

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

* Autoboxing and Unboxing:

*Autoboxing* is the automatic conversion that the Java compiler makes between primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called *unboxing*.

Autoboxing is applied by Java compiler when:

* A primitive value is passed as argument to a method expecting an object of the respective wrapper class.
* A primitive value is assigned to a variable of the corresponding Wrapper class

Unboxing is applied when:

* A Wrapper class instance is passed to a method expecting the corresponding primitive type.
* A Wrapper class instance is assigned to the variable of the corresponding primitive type.
* A generic class can have multiple type parameters.
* You can also substitute a type parameter (i.e., K or V) with a parameterized type (i.e., List<String>).
* A *raw type* is the name of a generic class or interface without any type arguments.

Box<Integer> intBox = new Box<>(); // generic type

Box rawBox = new Box(); // raw type

These raw types will show up a lot in legacy code which used certain APIs before they were made generics. This is allowed for backward compatibilty.

A generic type instance can be assigned to a raw type variable.

Box<String> stringBox = new Box<>();  
 Box rawBox = stringBox;  
 rawBox.set(8);

But this will issue a warning because Java compiler will no longer apply type checks on the generic type instances once it has been assigned to a raw type.

* When a raw type is assigned to a generic type variable, the Java compiler will issue an Unchecked warning
* Generic methods: Methods which have their own type paraemters. Scope of these type parameters will be limited within the method.

public class Util {  
 **public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2)** {  
 return p1.getKey().equals(p2.getKey()) &&  
 p1.getValue().equals(p2.getValue());  
 }  
}  
  
public class Pair<K, V> {  
  
 private K key;  
 private V value;  
  
 public Pair(K key, V value) {  
 this.key = key;  
 this.value = value;  
 }  
  
 public void setKey(K key) { this.key = key; }  
 public void setValue(V value) { this.value = value; }  
 public K getKey() { return key; }  
 public V getValue() { return value; }  
}

The complete syntax for invoking this method would be:

Pair<Integer, String> p1 = new Pair<>(1, "apple");  
Pair<Integer, String> p2 = new Pair<>(2, "pear");  
boolean same = Util.**<Integer, String>**compare(p1, p2);

The type has been explicitly provided, as shown in bold. Generally, this can be left out and the compiler will infer the type that is needed:

Pair<Integer, String> p1 = new Pair<>(1, "apple");  
Pair<Integer, String> p2 = new Pair<>(2, "pear");  
boolean same = Util.compare(p1, p2);

This feature, known as *type inference*, allows you to invoke a generic method as an ordinary method, without specifying a type between angle brackets.

* Sometimes constructors can have their own type parameters irrespective of the type parameters of the class or even if the class is not a generic class at all.

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

In these cases the invocation can be like:

MyClass<Integer> myObject = new MyClass<>("");

The Java compiler will infer that X is Integer and T is string.

* Sometimes the type parameter can be related to the return type of the method. In these cases type inference happens by looking at the type of the variable that stores the return value of the method or the **Target Type**:

static <T> List<T> emptyList(); // Example Method Signature. Method belongs to Collections class

The above method can be invoked like:

List<String> listOne = Collections.emptyList(); // type inference will happen automatically, T will be inferred to be String by looking at the type of listOne

Instead of:

List<String> listOne = Collections.<String>emptyList();

Similarly if you have a method like:

void processStringList(List<String> stringList) {  
 // process stringList  
 }

You can call it like this:

processStringList(Collections.emptyList());

Instead of,

processStringList(Collections.<String>emptyList());

In this case the target type is the type of the parameter and T can be

* Bounded types.

Bounded types specify an upper bound for the types that can be used to instantiate a generic class or invoke a generic method.

The extends keyword is used for this purpose.

public class NaturalNumber<T extends Integer> {  
 private T n;  
 public NaturalNumber(T n) { this.n = n; }

public boolean isEven() {  
 return **n.intValue()** % 2 == 0;  
 }  
 }

In this example either Integer or a subclass of Integer can be used to instantiate the NaturalNumber class. Also because T extends Integer, the instance methods of Integer class can be used with the field n as shown. In the above example, intValue is an instance method of Integer.

* Multiple bounds:

<T extends B1 & B2 & B3>

A type variable with multiple bounds means a subtype of all types listed in the bound. It can have one class and multiple interfaces in the list. If it has a class, the class should be listed first.

* A class can implement a generic type interface by passing itself as the type argument. Example comparable interface:

**class** Student **implements** Comparable<Student>{   
**int** rollno;

String name;

**int** age;

Student(**int** rollno,String name,**int** age){

**this**.rollno=rollno;

**this**.name=name;

**this**.age=age;

}

**public** **int** compareTo(Student st){

**if**(age==st.age)

**return** 0;

**else** **if**(age>st.age)

**return** 1;

**else**

**return** -1;

}

}

Now if we want to create a method that expects this class as the type argument and then use its compareTo method, it will be like this:

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

Now this class could also be passed as the type argument to this method, even if the method’s type parameter looked like <T > and not <T extends Comparable<T>>. However, in that case we could not have used the compareTo method in the method definition.

Hence if we want to call methods of a certain interface using the instances of type T then we have to make sure that the type T extends that interface in the type parameter list.

* If we have a class like:

Class Box<T>{

Public void add( T t){

//lines of code

}

}

In this case,

Box<Number> box = new Box<Number>();  
 box.add(new Integer(10)); // OK  
 box.add(new Double(10.1)); // OK

The above statements are true because both Integer and Double are subclasses of numbers.

However, If we have a method like:

public void boxTest(Box<Number> n)

We cannot pass Box<Integer> to it as Box<Integer> is not a subtype of Box<Number>.

* However we can extend a generic class of implement a generic interface to create subtypes of the generic class.

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>.

* Wild cards(in generics) are only used for :
* parameters that accepts objects of generic classes.
* A variable that is used to store objects of generic classes.
* Upper bounded wildcards:

public static void process(List<? extends Foo> list) {  
 for (Foo elem : list) {  
 // ...  
 }  
}

It means you can call this method and pass a list of any of the subtypes of Number class to it. For example, objects of List<Integer>, List<Double> etc up to List<Number> can be passed to it.

Though this is same as writing:

public static <T extends Foo> void process(List<T> list) {  
 for (T elem : list) {  
 // ...  
 }  
 }

There is no difference between the two code snippets. In the second one we introduce a type parameter that can be used throughout the body of the method for every list element. In the first one, we have to use the type Foo for representing the list elements.

Nothing can be added to a List<? Extends Number> because we don’t know what type of values are allowed to be stored in this list because this list can point to List<Integer>, List<Float> or List<Number> and we don’t know exactly which one it is actually pointing to.

* Unbounded wildcard

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(the parameter type) that don't depend on the type parameter. For example, List.size or List.clear. In fact, Class<?> is so often used because most of the methods in Class<T> do not depend on T.

For example the below code can accept a list of any kind of object.  
 public static void printList(List<?> list) {  
 for (Object elem: list)  
 System.out.print(elem + " ");  
 System.out.println();  
 }

}  
This would not have been possible if the parameter of the method was List<Object> list. Because List<Object> can only accept List<Object> and cannot accept List<Integer>, List<Double> etc.

Nothing can be added to a List<?> except NULL.

* Lower bounded wildcard  
  We know that an upper bounded wildcard restricts the unknown type to be a specific type or a subtype of that type and is represented using the extends keyword. In a similar way, a *lower bounded* wildcard restricts the unknown type to be a specific type or a *super type* of that type.  
  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>.

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

The elements of a List<? Super Integer> cannot be displayed because this list can point to List<Number> or List<Object>. But we can add a subclass of Integer to this list because anyway every subclass of Integer will also be an integer, a number and an object also.

* Q) I have a List which is declared like this :  
   List<? extends Number> foo3 = new ArrayList<Integer>();  
   I tried to add 3 to foo3. However I get an error message. Why?

The wildcard declaration of List<? extends Number> foo3 means that the variable foo3 can hold any value from a family of types (rather than any value of a specific type). It means that any of these are legal assignments:  
List<? extends Number> foo3 = new ArrayList<Number>; //   
List<? extends Number> foo3 = new ArrayList<Double>; //   
List<? extends Number> foo3 = new ArrayList<Double>;

**Now, you can't add any object to List<? extends T> because you can't guarantee what kind of List it is really pointing to, so you can't guarantee that the object is allowed in that List. The only "guarantee" is that you can only read from it and you'll get a T or subclass of T.**

The reverse logic applies to super, e.g. List<? super T>. These are legal:

List<? super Number> foo3 = new ArrayList<Number>;   
List<? super Number> foo3 = new ArrayList<Object>;

**You can't read the specific type T (e.g. Number) from List<? super T> because you can't guarantee what kind of List it is really pointing to. The only "guarantee" you have is you are able to add a value of type T (or any subclass of T) without violating the integrity of the list being pointed to.**

* Use of helper methods:  
  public class WildcardError {  
   void foo(List<?> i) {  
   i.set(0, i.get(0));  
   }  
  }

This will throw an error because i is of type List<?> and nothing apart from null can be added to it. In other words, nothing can be added to a list unless a value for its generic type parameter E is provided.

Helper class can be used to fix it:

public class WildcardFixed {  
 void foo(List<?> i) {  
 fooHelper(i);  
 }  
 // Helper method created so that the wildcard can be captured  
 // through type inference.  
 **private <T> void fooHelper(List<T> l) {  
 l.set(0, l.get(0));  
 }**  
 }

* Guidelines for using wildcards:  
  Two types of generic type parameters can be used in methods : **in parameters**(supplies data to code; usually provides get methods) and **out parameters**(stores data from code; usually provides set methods)  
  **In parameters:** Use upper bounded wildcards.   
  For example, imagine a parameter “abcClass<? extends Number> param”. Mostly get methods of param will be used. Those get methods will return values that are either of a fixed specified type or a type that is a subclass of Number. Any method that returns a subclass of Number can be easily stored into a Number data type. Hence, there will be no type incompatibility.

**Out parameters:** Use lower bounded wildcards. For example, imagine a parameter “abcClass<? super Number> param”. Mostly set methods will be used. Since set methods might accept parameters that are Number or superclasses of Number, Number types can be easily passed to them without type incompatibility issues.

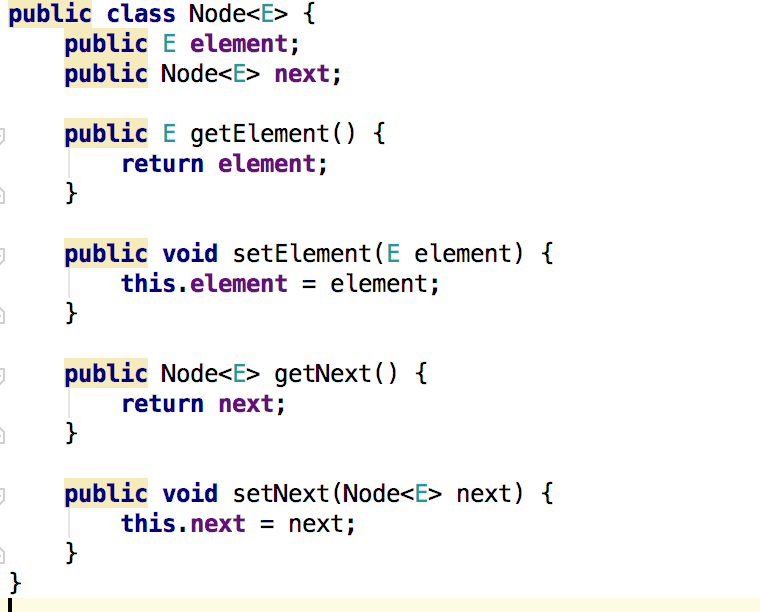
**In parameters that can be used by methods defined in Object class**: Use unbounded wildcards.

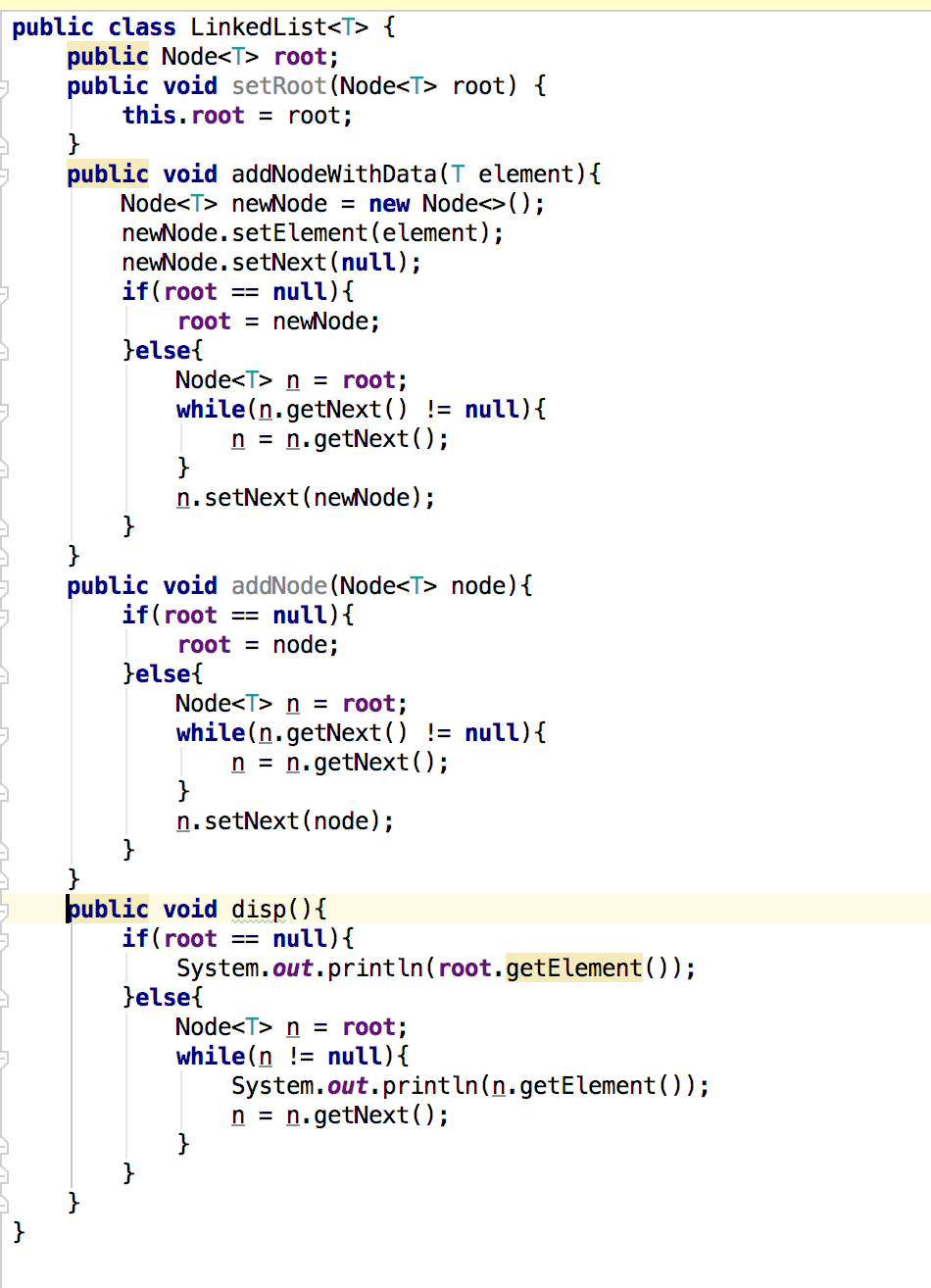
**Parameters that serve as both in and out parameters:** Do not use wildcards.

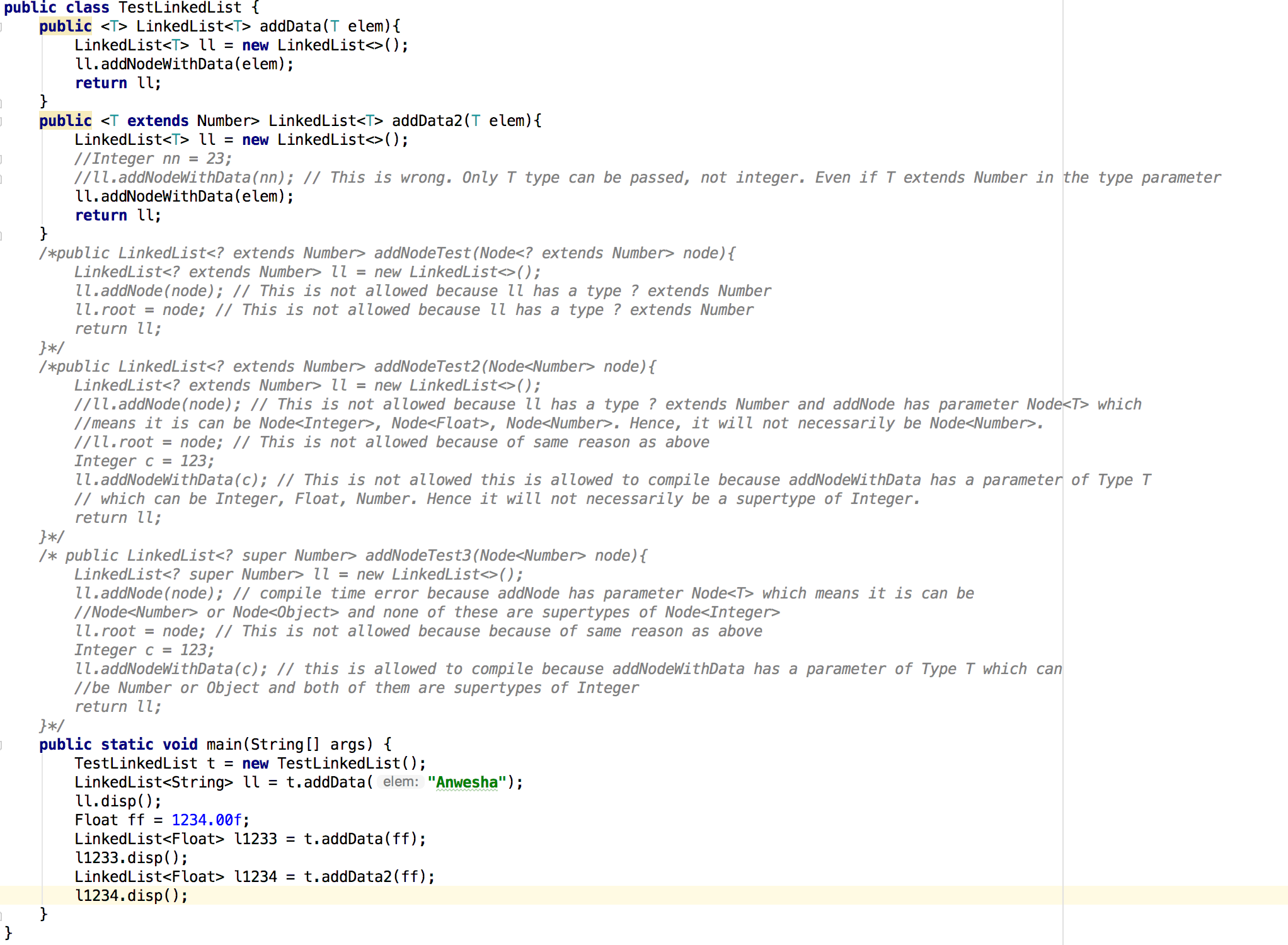
These guidelines are only for parameters. Not for return types. It is better to avoid wildcards in return types as that would force unnecessary constraints upon the code calling these methods.

* Type Erasure
* 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.  
  public class Node<T> {  
   private T data;  
   private Node<T> next;  
   public Node(T data, Node<T> next) {  
   this.data = data;  
   this.next = next;  
   }  
   public T getData() { return data; }  
   // ...  
  }  
  Gets replaced as:  
  public class Node {  
   private Object data;  
   private Node next;  
   public Node(Object data, Node next) {  
   this.data = data;  
   this.next = next;  
   }  
   public Object getData() { return data; }  
   // ...  
  }  
  And  
  public class Node<T extends Comparable<T>> {  
   private T data;  
   private Node<T> next;  
   public Node(T data, Node<T> next) {  
   this.data = data;  
   this.next = next;  
   }  
   public T getData() { return data; }  
   // ...  
  }  
  Gets replaced as:  
  public class Node {  
   private Comparable data;  
   private Node next;  
   public Node(Comparable data, Node next) {  
   this.data = data;  
   this.next = next;  
   }  
   public Comparable getData() { return data; }  
   // ...  
  }
* Insert type casts if necessary to preserve type safety.
* Generate bridge methods to preserve polymorphism in extended generic types. Example:  
  public class Node<T> {  
   public T data;  
   public Node(T data) { this.data = data; }  
    
   public void setData(T data) {  
   System.out.println("Node.setData");  
   this.data = data;  
   }  
  }  
    
  public class MyNode extends Node<Integer> {  
   public MyNode(Integer data) { super(data); }  
   public void setData(Integer data) {  
   System.out.println("MyNode.setData");  
   super.setData(data);  
   }  
  }
* The above two classes will get replaced by type Erasure as below:  
  public class Node {  
   public Object data;  
   public Node(Object data) { this.data = data; }  
   public void setData(Object data) {  
   System.out.println("Node.setData");  
   this.data = data;  
   }  
  }  
    
  public class MyNode extends Node {  
   public MyNode(Integer data) { super(data); }  
   public void setData(Integer data) {  
   System.out.println("MyNode.setData");  
   super.setData(data);  
   }  
  }  
  Now as we have two methods setData(Object data) and setData(Integer data), the later will not override the former though this was the intention initially. Therefore a bridge method will be created by the compiler that will replace the body of every superclass method like this:  
  **public void setData(Object data) {  
   setData((Integer) data); // the type casting is done to Integer because** MyNode **extends** Node<Integer> **}**
* Restrictions on generics:  
   - Cannot Instantiate Generic Types with Primitive Types.  
   - Cannot create an instance of a type parameter.  
   E.g.   
   public static <E> void append(List<E> list) {  
   E elem = new E(); // compile-time error  
   list.add(elem);  
   }  
   Workaraound:  
   public static <E> void append(List<E> list, Class<E> cls) throws Exception {  
   E elem = cls.newInstance(); // OK  
   list.add(elem);  
   }

List<String> ls = new ArrayList<>();  
 append(ls, String.class);  
 - Cannot Declare Static Fields Whose Types are Type Parameters.  
 - Cannot instanceof with Parameterized Types  
 public static <E> void rtti(List<E> list) {  
 if (list instanceof ArrayList<Integer>) { // compile-time error  
 // ...  
 }  
 }  
 However, this is allowed:  
 public static void rtti(List<?> list) {  
 if (list instanceof ArrayList<?>) { // OK; instanceof requires a reifiable type  
 // ...  
 }  
 }  
 - Cannot create arrays of parameterized types.  
 Arrays of generic types are not allowed because they're not sound. The  
 problem is due to the interaction of Java arrays, which are not statically   
 sound but are dynamically checked, with generics, which are statically   
 sound and not dynamically checked. Here is how you could exploit the   
 loophole:  
 class Box<T> {  
 final T x;   
 Box(T x) { this.x = x; }   
 }   
 class Loophole {   
 public static void main(String[] args) {   
 Box<String>[] bsa = new Box<String>[3];   
 Object[] oa = bsa; oa[0] = new Box<Integer>(3); //  
 error not caught by array store check   
 String s = bsa[0].x; // BOOM! }   
 }  
 - A generic class cannot extend the Throwable class directly or indirectly.  
 E.g.  
 // Extends Throwable indirectly  
 class MathException<T> extends Exception { /\* ... \*/ } // compile-time error  
  
 // Extends Throwable directly  
 class QueueFullException<T> extends Throwable { /\* ... \*/ // compile-time error  
 - A method cannot catch an instance of a type parameter.  
 public static <T extends Exception, J> void execute(List<J> jobs){  
 try {  
 for (J job : jobs)  
 // ...  
 } catch (T e) { // compile-time error  
 // ...  
 }  
 }  
 However, a type parameter can be used in a throws clause:  
 class Parser<T extends Exception> {  
 public void parse(File file) throws T { // OK  
 // …  
 }  
 }  
 - A class cannot have two overloaded methods that will have the same signature after type erasure.  
 public class Example {  
 public void print(Set<String> strSet) { }  
 public void print(Set<Integer> intSet) { }  
 }  
The overloads would all share the same classfile representation and will generate a compile-time error.

* Some confusing areas:  
  Calling methods of a generic type. **public <T extends Number> LinkedList<T> addData2(T elem){  
   LinkedList<T> ll = new LinkedList<>();  
   Integer nn = 23;ll.addNodeWithData(nn);ll.addNodeWithData(elem);  
   return ll;  
  }**In this example, the line **ll.addNodeWithData(nn);**will produce compile-time error.  
  **LinkedList<T> has a method addNodeWithData(T elem).** This means that only type T can be passed to this method if the object is LinkedList<T>. If the object was LinkedList<String>, only String could be passed to this method and so on. If we are creating a generic type with a given Type argument. The type argument can be 1. a fixed type, 2. a generic Type or 3. a wildcard.   
  - **For 1(a fixed type) -->** the method has to be called by passing the exact type that is used to create the generic object or its subtypes.  
  - **For 2(a generic type like T ) --->** the method has to be called by passing the exact type that is used to create the generic object i.e. T. Even if the object is created using a type T that extends some class like Number, we cannot call its methods directly by passing subclasses of Number. The above example demonstrates this.  
  **For the wildcard scenario check below examples:  
  **





In the above example addNodeTest method throws compile time error because of this reason:  
- LinkedList<? extends Number> ll means the type ll can only reference instances of LinkedList<Number> or LinkedList<Integer> and call only those methods of ll that do not take generic types of the class parameter T as parameters.   
- Because when ? extends Number is sent as an argument to T in LinkedList class while creating object ll, T doesnt get any specific value that it can use. None of the fields, variables, return types or parameters of type T in object ll are of a fixed type because they can be anything e.g Integer, Float, Number, etc. Therefore we cannot store pass any value to those fields or parameters because we don’t know the type. However, we can call the get methods and store the returned values in a Number variable because whatever T is, it will always be subclass of Number.  
-Therefore when we instantiate a class having a type parameter T with ? extends Number, we cannot use pass any value to its method parameters or fields that are of type T. However, we can call their methods that return T and store the value in a Number.

This is why addNodeTest and addNodeTest2 will not compile in the above example.

* If I want to instantiate a generic class, I can either pass a concrete type, a range of types(wildcard usage or a generic type) or another generic as arguments to a type parameter. If we pass another generic type as argument to a type parameter, the generic type should be listed at the enclosing class or method declaration line. And it will be the responsibility of the code using this method or class to supply the concrete types.  
  For example Node has type parameter E. It is instantiated with type parameter T inside linked list class. Now, whenever we instantiate linkedlist class for actual usage , at that time we will provide the concrete types which will pass down to node.

**Collection framework**

* Consists of:   
  - interfaces  
  - implementations  
  - polymorphic algorithms
* Interfaces - Collection, set, list, queue, deque, map, sortedset, sortedmap
* Collection Interface:  
  - Has elements  
  - root of the collection framework.  
  - A generic interface.  
  - Jdk does not provide direct implementations of collection interface. It provides implementations of more specific subinterfaces of collection like list, set, etc.  
  - Any general purpose collection implementation must have two standard constructors - a void (no arguments) constructor, which creates an empty collection, and a constructor with a single argument of type Collection, which creates a new collection with the same elements as its argument.   
  In effect, the latter constructor allows the user to copy any collection, producing an equivalent collection of the desired implementation type. This constructor, known as a *conversion constructor*, initializes the new collection to contain all of the elements in the specified collection, whatever the given collection's subinterface or implementation type. In other words, it allows you to *convert* the collection's type.

Suppose, for example, that you have a Collection<String> c, which may be a List, a Set, or another kind of Collection. This idiom creates a new ArrayList (an implementation of the List interface), initially containing all the elements in c.  
List<String> list = new ArrayList<>(c);

- Traversing collections →   
 **Aggregate operations**: Done by using aggregate operations and lambda  
 expressions on streams or parallel streams of collections. Eg:  
 myShapesCollection.stream()

.filter(e -> e.getColor() == Color.RED)

.forEach(e -> System.out.println(e.getName()));

OR

myShapesCollection.parallelStream()

.filter(e -> e.getColor() == Color.RED)

.forEach(e -> System.out.println(e.getName()));

**For each constructs:** for (Object o : collection)

System.out.println(o);

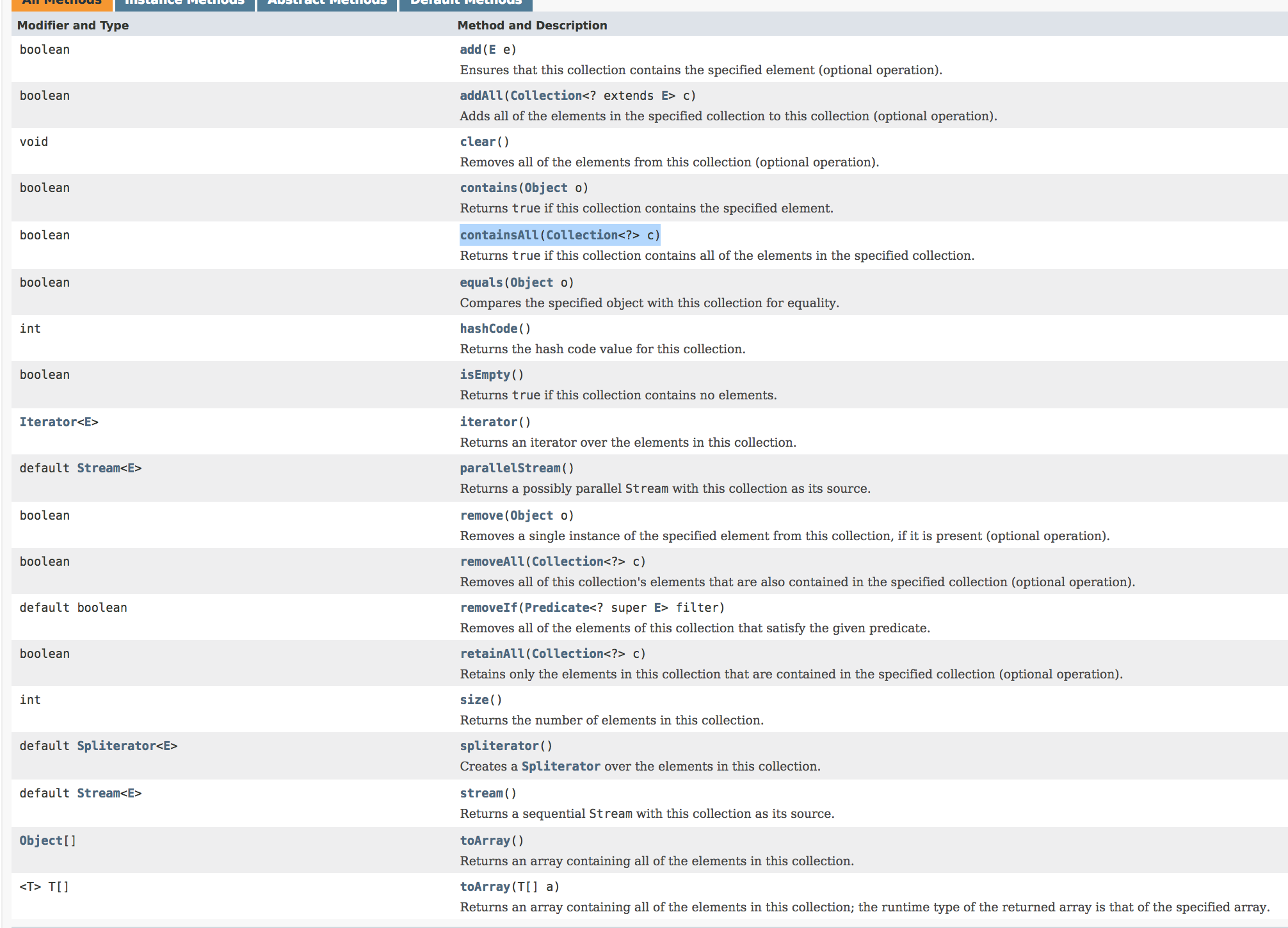
**Iterator:** Note that Iterator.remove is the *only* safe way to modify a collection   
 during iteration; the behavior is unspecified if the underlying collection is modified   
 in any other way while the iteration is in progress.

Use Iterator instead of the for-each construct when you need to:

* Remove the current element. The for-each construct hides the iterator, so you cannot call remove. Therefore, the for-each construct is not usable for filtering.
* Iterate over multiple collections in parallel.

Iterator interface:  
 public interface Iterator<E> {  
 boolean hasNext();  
 E next();  
 void remove(); //optional  
 }  
 The remove method removes the last element that was returned by next from the   
 underlying Collection. The remove method may be called only once per call to  
 next and throws an exception if this rule is violated.  
 - Collections.singleton: a static factory method that returns an immutable Set containing  
 only the specified element.  
 c.removeAll(Collections.singleton(e)); // e is an element. This statement removes all instances of e from c  
 OR  
 c.removeAll(Collections.singleton(null)); // This statement removes all Null values from c

- Why is it necessary to pass an array of given type to the method <T> T[] toArray(T[] a) of collections?  
If you look at the implementation of toArray(T[] a) of [ArrayList<E>](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html) class, it is like:  
public <T> T[] toArray(T[] a) {   
 if (a.length < size)   
 return (T[]) Arrays.copyOf(elementData, size, a.getClass());  
 System.arraycopy(elementData, 0, a, 0, size);  
 if (a.length > size)   
 a[size] = null;   
 return a;   
}  
Problem with this method is that you need to pass an array of the same generic type. Now consider if this method did not take any argument then the implementation would be something similar to:  
public <T> T[] toArray() {   
 T[] t = new T[size]; // compilation error   
 return Arrays.copyOf(elementData, size, t.getClass());   
}  
But the problem here is that **you can not create generic arrays in Java** because the compiler does not know exactly what T represents. In other words **creation of array of a non-reifiable type** **is not allowed in Java**.

- Methods :   


* Set Interface:
* Has all the methods of collection.
* Add the restriction that duplicate elements cannot exist.
* Add, Remove, Contains operations happen in unique time.
* Has three implementations - HashSet, LinkedHashSet, TreeSet.
* TreeSet stores elements in sorted order.
* HashSet stores elements in a HashTable internally.
* Set is not sequential i.e. it does not store the elements in the order they were entered.
* Set iterations take time and should be done carefully.
* Always refer to a collection by its interface type (Set) rather than by its implementation type(HashSet). This provides flexibility to change the implementation by just changing the constructor.
* Set operations:  
  Union : SetA.addAll(SetB)  
  Intersection: SetA.retainAll(SetB)  
  Difference: SetA.removeAll(SetB)  
  Subset: SetA.containsAll(SetB)  
  Exclusive difference:   
   Set<Type> symmetricDiff = new HashSet<Type>(s1);

symmetricDiff.addAll(s2);

Set<Type> tmp = new HashSet<Type>(s1);

tmp.retainAll(s2);

symmetricDiff.removeAll(tmp);

* List Interface:
* Sequential
* Can contain duplicates
* Two implementations - ArrayList & LinkedList
* All methods of Collections plus additional features.
* Positional access — manipulates elements based on their numerical position in the list. This includes methods such as get, set, add, addAll, and remove.
* Search — searches for a specified object in the list and returns its numerical position. Search methods include indexOf and lastIndexOf.
* Iteration — Has the same iterator method. Plus two listIterator methods that extend Iterator semantics to take advantage of the list's sequential nature.
* Range-view — The sublist method performs arbitrary *range operations* on the list.
* Shuffling:  
  public static void shuffle(List<?> list, Random rnd) {

for (int i = list.size(); i > 1; i--)

swap(list, i - 1, rnd.nextInt(i));

}

* Arrays.asList() returns a special type of list. Any changes to this list will reflect on the array and vice versa. However, this list doesn’t support add and remove operations.
* ListIterator:

Methods: next(), hasNext(), nextIndex(), previous(), hasPrevious(), previousIndex(), add, remove

Valid indexes for an array of size 5: 0,1,2,3,4,5 (n+1 indexes → 0 to n)

* ListIterator<E> it = list.listIterator(0): cursor = -0.5, prevIndex = -1; nextIndex = 0, hasPrev = False, hasNext = False, prev = exception, next = first element. After subsequent next calls, cursor moves from -0.5 → 0.5 → 1.5 → 2.5 → 3.5 → 4.5 and then exception.
* ListIterator<E> it = list.listIterator(5): cursor = 4.5, prevIndex = 4; nextIndex =5, hasPrev = True, hasNext = False, prev = last element, next = exception. After subsequent previous calls, cursor moves from 4.5 → 3.5 → 2.5 → 1.5 → 0.5 → -0.5 then exception.
* The add method adds an element before the current cursor position.
* Range-view operation:

sublist(toindex, from index) → The range-view operation, subList(int fromIndex, int toIndex), returns a List view of the portion of this list whose indices range from fromIndex, inclusive, to toIndex, exclusive. This *half-open range* mirrors the typical for loop.

for (int i = fromIndex; i < toIndex; i++) {

…

}

As the term *view* implies, the returned List is backed up by the List on which subList was called, so changes in the former are reflected in the latter.

For example, the following idiom removes a range of elements from a List:

list.subList(fromIndex, toIndex).clear();

However,

int i = list.subList(fromIndex, toIndex).indexOf(o);

int j = list.subList(fromIndex, toIndex).lastIndexOf(o);  
These statements return the index in the **sublist**, not the index in the backing list.  
Although the subList operation is extremely powerful, some care must be exercised when using it. The semantics of the List returned by subListbecome undefined if elements are added to or removed from the backing List in any way other than via the returned List. Thus, it's highly recommended that you use the List returned by subList only as a transient object — to perform one or a sequence of range operations on the backing List. The longer you use the subList instance, the greater the probability that you'll compromise it by modifying the backing List directly or through another subListobject. Note that it is legal to modify a sublist of a sublist and to continue using the original sublist (though not concurrently).

* Queue Interface:
* Implementations: LinkedList, PriorityQueue
* 6 main methods. Other methods are inherited from Collection.  
  - boolean add(E e): add element. In case of bounded queues, throws IllegalStateException exception if capacity is full.

- boolean offer(E e): add element. In case of bounded queues, returns false if capacity is full.

- E remove() - Removes and returns the head of the queue. Which element gets removed depends on Object ordering. Throws NoSuchElementException exception, if queue is empty.

- E poll() - Removes and returns the head of the queue. Which element gets removed depends on Object ordering. Throws Null if queue is empty.

- E element() - Returns but does not remove the head of the queue. Which element gets removed depends on Object ordering. Throws NoSuchElementException exception, if queue is empty.

- E peek() - Returns but does not remove the head of the queue. Which element gets removed depends on Object ordering. Throws Null if queue is empty.

* Bounded queues: Queues in which capacity is fixed. Some queues in java.util.concurrent are bounded, but the implementations in java.util are not.
* The Queue interface does not define the blocking queue methods, which are common in concurrent programming. These methods, which wait for elements to appear or for space to become available, are defined in the interface [java.util.concurrent.BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html), which extends Queue.
* Queue implementations generally do not define element-based versions of the equals and hashCode methods but instead inherit the identity-based versions from Object.
* In the following example program, a queue is used to implement a countdown timer.

import java.util.\*;

public class Countdown {

public static void main(String[] args) throws InterruptedException {

int time = Integer.parseInt(args[0]);

Queue<Integer> queue = new LinkedList<Integer>();

for (int i = time; i >= 0; i--)

queue.add(i);

while (!queue.isEmpty()) {

System.out.println(queue.remove());

Thread.sleep(1000);

}

}

}

* In the following example, a priority queue is used to sort a collection of elements.

static <E> List<E> heapSort(Collection<E> c) {

Queue<E> queue = new PriorityQueue<E>(c);

List<E> result = new ArrayList<E>();

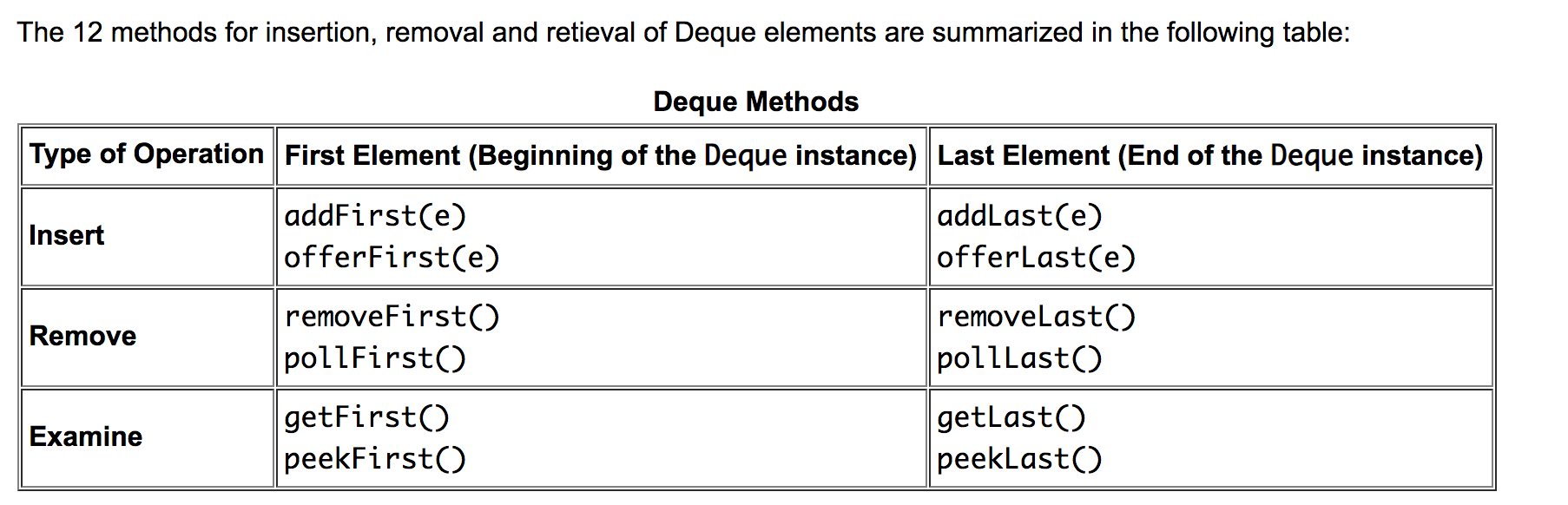
while (!queue.isEmpty())

result.add(queue.remove());

return result;

}

* Deque:
* Allows addition and removal on both ends.
* Superinterfaces: Iterator, Collection, Queue
* Implements both stack and queue.
* In addition to collection methods, below methods are available:



Two additional predefined methods:  
removeFirstOccurence: This method removes the first occurence of the specified element if it exists in the Deque instance.

removeLastOccurence: This method removes the last occurence of the specified element if it exists in the Deque instance.

* Implementations: ArrayDeque, LinkedList
* Map interface:
* Three implementations - HashMap, LinkedHashMap, TreeMap

HashMap : Like HashSet

LinkedHashMap: Sequential

TreeMap: Sorted

* Map interface basic operations: put, get, containsKey, containsValue, size, and isEmpty, putAll
* Collection views:   
  keySet — the Set of keys contained in the Map.

values — The Collection of values contained in the Map. This Collection is not a Set, because multiple keys can map to the same value.  
entrySet — the Set of key-value pairs contained in the Map. The Map interface provides a small nested interface called Map.Entry, the type of the elements in this Set.

* Iterating through Keyset:

for-each construct:

for (KeyType key : m.keySet())

System.out.println(key);

and with an iterator:

// Filter a map based on some

// property of its keys.

for (Iterator<Type> it = m.keySet().iterator(); it.hasNext(); )

if (it.next().isBogus())

it.remove();

* Iterating through EntrySet:

for (Map.Entry<KeyType, ValType> e : m.entrySet())

System.out.println(e.getKey() + ": " + e.getValue());

* With **all three Collection views, calling an Iterator's remove operation removes the associated entry from the backing Map**, assuming that the backing Map supports element removal to begin with.   
  With the entrySet view, it is also possible to change the value associated with a key by calling a Map.Entry's setValue method during iteration (again, assuming the Map supports value modification to begin with).

These are the *only* safe ways to modify a Map during iteration; the behavior is unspecified if the underlying Map is modified in any other way while the iteration is in progress.

* Employee -> manager map example:

Consider a map managers where all employees of an organization are keys and all of these keys are mapped to the respective managers.

* Operation to get only Non-manager employees:

Set<Employee> individualContributors = new HashSet<Employee>(managers.keySet());

individualContributors.removeAll(managers.values();

* Remove all employees who directly report to a given manager Simon:  
  Employee simon = ... ;

managers.values().removeAll(Collections.singleton(simon));

* Once the second step is done, you may have a bunch of employees whose managers no longer work for the company (if any of Simon's direct-reports were themselves managers). The following code will tell you which employees have managers who no longer works for the company.

Map<Employee, Employee> m = new HashMap<Employee, Employee>(managers);

m.values().removeAll(managers.keySet());

Set<Employee> slackers = m.keySet();

* A *multimap* is like a Map but it can map each key to multiple values. The Java Collections Framework doesn't include an interface for multimaps because they aren't used all that commonly. It's a fairly simple matter to use a Map whose values are List instances as a multimap.