4 Generics and collections

Exam objectives covered in this chapter	What you need to know
[4.1] Create a generic class	How to define generic classes, interfaces, and methods with single and multiple type parameters How to define generic methods with a generic or regular class
[4.2] Use the diamond for type inference	How to drop the type from the angle brackets to instantiate generic classes How to use wildcards to create and instantiate generic classes
[4.3] Analyze the interoperability of collections that use raw types and generic types	What happens when you lose type safety by using variables of raw types and objects of generic types How to determine and differentiate scenarios that would generate compilation errors and warnings
[4.4] Use wrapper classes, autoboxing, and unboxing	How and when values are boxed and unboxed when used with wrapper classes
[4.5] Create and use `List`, `Set`, and `Deque` implementations	How to create objects of the `List` interface (`ArrayList`, `LinkedList`), objects of the `Deque` interface (`ArrayDeque`, `LinkedList`), and objects of the `Set` interface (`HashSet`, `LinkedHashSet`, and `TreeSet`) How each implementing class stores data, manipulates it, searches it, and iterates over it How the `List`, `Set`, and `Deque` implementations use methods `hashCode()`, `equals()`, `compare()`, and `compareTo()` Given a set of requirements, how to choose the best interface or its implementing class

Exam objectives covered in this chapter	What you need to know
[4.6] Create and use `Map` implementations	How to instantiate `Map` objects: `HashMap`, `LinkedHashMap`, and `TreeMap` How `Map` implementations use methods `hashCode()`, `equals()`, `compare()`, and `compareTo()`
[4.7] Use `java.util.Comparator` and `java.lang.Comparable`	How to define natural and custom ordering of objects of a class
[4.8] Sort and search arrays and lists	How to sort and search arrays and lists using methods from classes `Arrays` and `Collections` Importance of using sorted collections for searching values

Imagine you need to collect pencils at your workplace. You request all your fellow workers to drop their pencils in a box at the main entrance of the office. When you open the box the next day, you also find ink pens and marker pens (!), which you didn't ask for. Even though you mentioned pencils, people could add pens to the box because no one stopped them from doing so. Now imagine that you could use a box that wouldn't allow adding any item other than a pencil. Would you prefer it? If you answered yes, you'd prefer to use generics. In Java, *generics* empower you to specify the type of objects that you'd like to work with so that you don't work with other types—knowingly or unknowingly.

Now imagine that you need to sort all the collected pencils according to their color and size. Would you like to do that yourself, or would you prefer a magic box that would accept all the pencils and return them to you in a sorted order? If you chose the magic box, you'd like using the collections framework. The Java collections framework includes multiple interfaces and classes to store and manipulate a collection of objects, including the methods that sort and search them.

This chapter covers

- Creating and using generic types
- Using the diamond for type inference
- Analyzing the interoperability of collections that use raw types and generic types
- Using wrapper classes, autoboxing, and unboxing
- Creating and using `List`, `Set`, and `Deque` implementations
- Creating and using `Map` implementations
- Working with the `java.util.Comparator` and `java.lang.Comparable` interfaces
- Sorting and searching arrays and lists

Let's start with an introduction to generics, in the next warm-up section. Feel free to skip it and move to the next section if you're an experienced generics programmer.

4.1 Introducing generics: WARM-UP

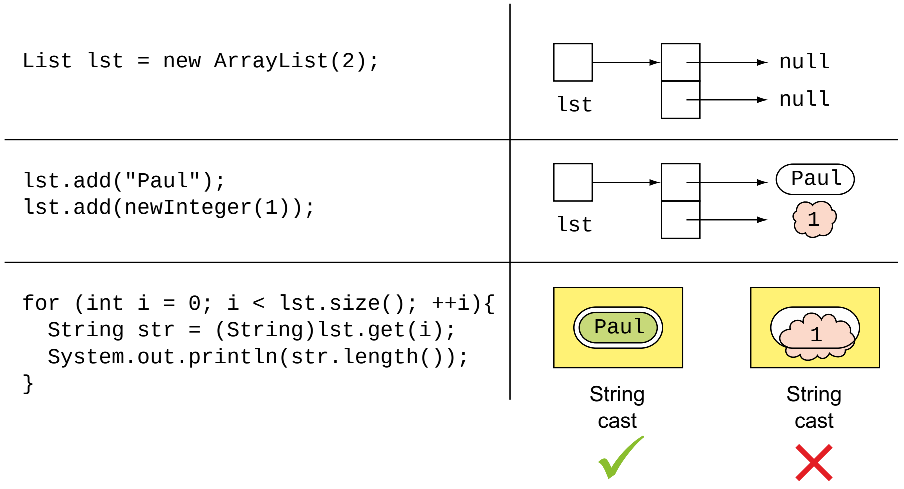
Generics enable you to abstract over types. They add type safety to collection classes. Introduced with Java version 5.0, generics enable developers to detect certain bugs during compilation so they can't creep into the runtime code. Debugging an application is a costly affair, in terms of the time and effort required to find a bug and then fix it. The sooner you can detect a bug, the easier it is to fix it. While developing software, it's easier to fix a bug during unit testing than it is to fix the same bug during integration testing or, say, when it shows up months after an application goes live. A bug is easier to fix in the development phase than in the maintenance phase.

4.1.1 Need for introducing generics

Before generics were introduced, programmers used to *assume* that a class, interface, or method would work with a *certain* data type. For example, figure 4.1 shows how a programmer would *assume* that the `ArrayList` referred to by `lst` would contain `String` objects. But because `lst` is a collection of objects of type `Object`, it can accept any type of data (other than primitives). An issue can creep in when these different types of objects are treated as `String` types during runtime.

NOTE In figures 4.1 and 4.2, `ArrayList-lst` is created with an initial capacity (and not size) of two elements. The size of an `ArrayList` increases as more elements are added to it.

With the introduction of generics, programmers could indicate their intent of using a particular type of data with a class, interface, or method (not enums, because enums can't have generic type parameters). Figure 4.2 shows how you can indicate that an



```
List lst = new ArrayList(2);
```

```
lst.add("Paul");
lst.add(newInteger(1));
```

```
for (int i = 0; i < lst.size(); ++i){
  String str = (String)lst.get(i);
  System.out.println(str.length());
}
```

Figure 4.1 Before generics were added, collection classes like `ArrayList` allowed the addition of any type of data. A programmer's assumption of adding only a particular type of data to a collection was met with a casting exception at runtime.

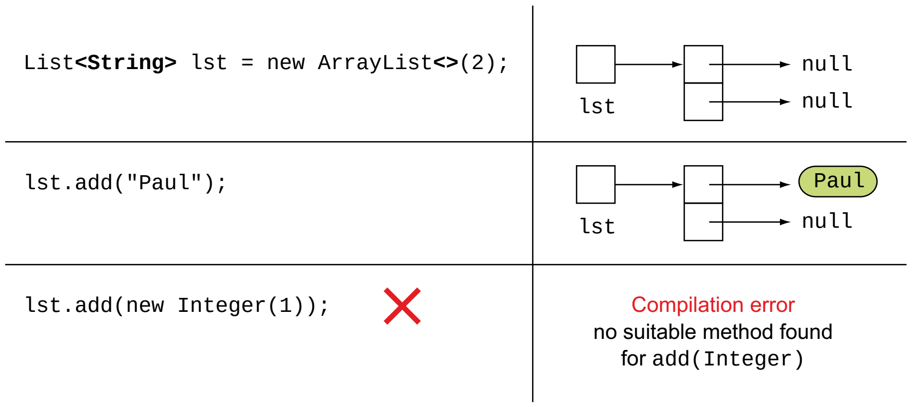

List<String> lst = new ArrayList<>(2);
lst
null
null
lst.add("Paul");
lst
Paul
null
lst.add(new Integer(1));
Compilation error
no suitable method found for add(Integer)

Figure 4.2 Post-generics, you can mark your intent of using a particular data type with a class, method, or interface. If the code doesn't adhere to the restrictions, the code fails to compile.

`ArrayList` referred to by `lst` will accept only objects of type `String`. Code that tries to add an object of any other type won't compile.

As shown in figure 4.2, with generics, the incorrect data type is determined during compilation. This compilation-time safety enables you to identify bugs during development, thus building better code.

EXAM TIP The basic purpose behind using generics is to enable you to mark your intent of using a class, method, or interface with a particular data type. Generics add compile-time safety to collections.

4.1.2 Benefits and complexities of using generics

Apart from compile-time safety, you also get the following benefits with generics:

- *Removing explicit casts*—Prior to generics, you needed to add casts when you had a list with strings and you wanted to get a string out of the list. With generics this isn't needed anymore.
- *Better code readability*—Without explicit casting, code is less cluttered, which improves readability.
- *Developing generic algorithms*—Just as you need not hard-code values when you work with methods and can accept them as method parameters, generics help you parameterize over data types and develop algorithms that work with multiple data types.

But every new concept or approach has its own set of limitations and complexities, and using generics is no exception. As you work through this chapter, you'll see how adding generics to the collections framework created new complexities. (Coverage is limited to the exam topics.)

In the next section, you'll create your own generic entities. If you haven't already worked with generic entities, it might take a while for all the related concepts to sink in.

4.2 Creating generic entities

[4.1] Create a generic class

On the exam, you'll be tested on how to create generic classes, interfaces, and methods—within generic and nongeneric classes or interfaces.

4.2.1 Creating a generic class

In this section, we'll start with an example of a nongeneric class and then modify it to create a generic class. You'll learn how to use a generic class and how important variable naming conventions are for the type parameters.

A NONGENERIC CLASS

To understand how to create a generic class, let's begin with an example of a *non*generic class, `Parcel`:

```
class Parcel {
    private Object obj;
    public void set(Object obj) {
        this.obj = obj;
    }
    public Object get() {
        return obj;
    }
}
```

Class `ParcelNonGeneric` can use class `Parcel`, calling its method `set()` to assign an object of class `Book`. It can retrieve this object by using `get()` and cast it to class `Phone`(!). Even though not desired, it's allowed:

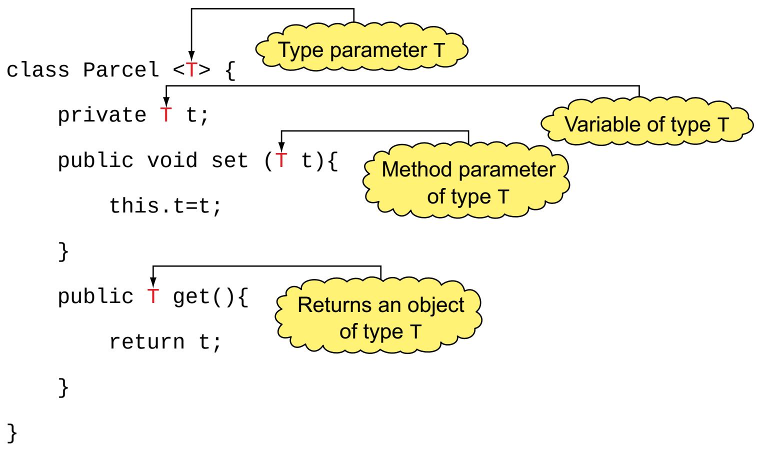
```
class Phone{}
class Book{}

class ParcelNonGeneric {
    public static void main(String args[]) {
        Parcel parcel = new Parcel();
        parcel.set(new Book());
        System.out.println((Phone)parcel.get());
    }
}
```

Assign object of Book

Cast object of Book to Phone; code compiles but throws ClassCastException at runtime.

ADDING TYPE SAFETY TO A NONGENERIC CLASS

Let's see how you add *type safety* to class `Parcel`. Let's define class `Parcel` as a generic class by adding a *type parameter* to it, so that you can retrieve only the object type that you assign to it, as shown in figure 4.3.



```
class Parcel <T> {
    private T t;
    public void set (T t){
        this.t=t;
    }
    public T get(){
        return t;
    }
}
```

Figure 4.3 How to convert a nongeneric class to a generic class by adding type parameters

As shown in the preceding code, the declaration of generic class `Parcel` includes the type parameter T. After adding the type information, it's read as `Parcel<T>` or `Parcel` of T. The generic class `Parcel<T>` defines a private instance variable of type T, and `get()` and `set()` methods to retrieve and set its value. Methods `get()` and `set()` use the parameter type `T` as their method parameter and return type.

EXAM TIP The first occurrence of `T` is different from its remaining occurrences because only the first one is surrounded by `<>`.

Using a generic class

Having seen how to create a generic class, let's see how you can use it. Class `UseGenericParcel` instantiates `Parcel` and calls its methods `get()` and `set()`. Note that you don't need an explicit cast when you use `Book` instance by calling `parcel.get()`:

```
class Book{}
class UseGenericParcel {
    public static void main(String args[]) {
        Parcel<Book> parcel = new Parcel<Book>();
        parcel.set(new Book());
        Book myBook = parcel.get();
    }
}
```

Parameterized type Parcel<Book> indicates Parcel will work with instances of Book.

set() accepts Book instance.

get() returns Book instance; no explicit casts required.

With the generic class `Parcel`, `UseGenericParcel` can use method `set()` to assign an object of type `Book`. But `UseGenericParcel` can't cast the retrieved object to an unrelated class, say, `Phone`. If it tries to do so, the code won't compile (as shown in figure 4.4).

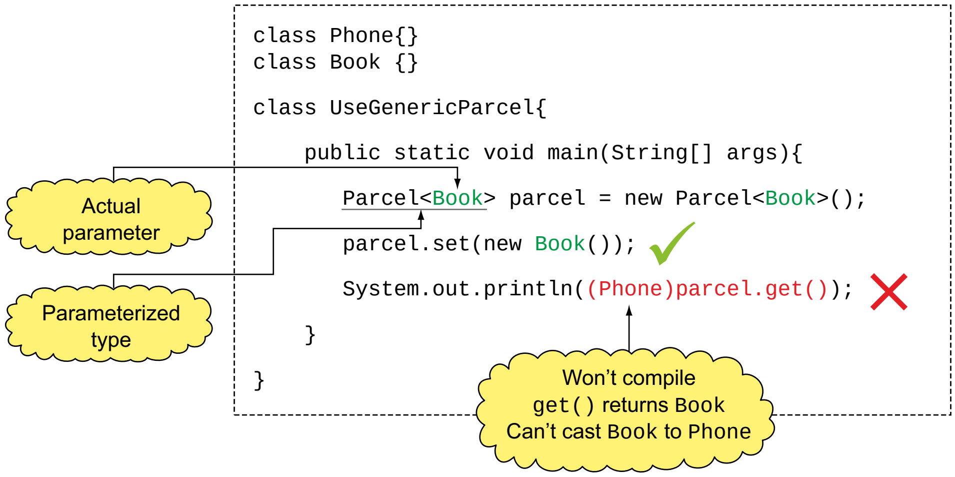

class Phone{}
class Book {}
class UseGenericParcel{
public static void main(String[] args){
Parcel<Book> parcel = new Parcel<Book>();
parcel.set(new Book());
System.out.println((Phone)parcel.get());
}
}
Actual parameter
Parameterized type
Won't compile get() returns Book Can't cast Book to Phone

Figure 4.4 A class that uses a generic class uses a parameterized type, replacing the formal parameter with an actual parameter. Also, invalid casts aren't allowed.

EXAM TIP A type parameter can be used in the declaration of classes, variables, method parameters, and method return types.

Variable names used for type parameters

You must follow the variable naming rules for type parameters; for instance, you can't use Java keywords. As per Oracle's naming conventions, you should use uppercase single characters for type parameters. This also sets them apart from other variables and method parameters, which use *camelCase*. Though the constants use uppercase, they aren't *usually* limited to single characters.

Now, what happens if you don't follow the conventions for naming type parameters? Here's an interesting exam question. For the modified definition of class `Parcel` in the following code, do you think method `set()` can be passed `String` objects?

```
class MyClass{}
class Parcel<MyClass>{
    private MyClass t;
    public void set(MyClass t) {
        this.t = t;
    }
}
```

Yes, it can. In the preceding code, `MyClass` is used as a *placeholder* for a type argument that you pass to class `Parcel`—it doesn't refer to class `MyClass`. So you can instantiate `Parcel`, passing it a type argument, say, `String`, and pass a `String` value to its method `set()`. For example

```
class UseParcel {
    public static void main(String args[]) {
        Parcel<String> parcel = new Parcel<>();
        parcel.set("OCP");
        System.out.println(parcel.get().length());
    }
}
```

Generic class extending another generic class

A generic class can be extended by another generic class. In the following example, generic class `GenericBookParcel<T>` extends generic class `Parcel<T>`:

```
class Parcel<T> {}
class GenericBookParcel<T> extends Parcel<T> {}
```

Generic extended class

In all cases, an extended class must be able to pass type arguments to its base class. For the preceding example, the type argument passed to class `GenericBookParcel` is passed to its base class, `Parcel`, when you instantiate `GenericBookParcel`. For example

```
GenericBookParcel<String> parcel = new GenericBookParcel<>();
```

The preceding example passes argument `String` to `GenericBookParcel`'s type parameter `T`. But if you define `GenericBookParcel` in a way that it can't pass an argument to the parameters of its base class, the code won't compile. Do you think the following code will compile?

```
class Parcel<T> {}
class GenericBookParcel<X> extends Parcel<T> {}
```

Won't compile; no way to pass argument to T

No, it won't. In the preceding code, class `GenericBookParcel` defines a type parameter `X`, but doesn't include `T` in its type parameter list. Because this arrangement prevents `GenericBookParcel` from passing type arguments to its base class `Parcel`, it fails to compile.

You can also define new type parameters for a derived class when you extend a generic base class. In the following example, class `GenericBookParcel` defines two type parameters `X` and `T`:

```
class Parcel<T> {}
class GenericBookParcel<X, T> extends Parcel<T> {}
```

Compiles successfully

Here's another example, in which the derived class passes type arguments to its generic base class in its declaration:

```
class Parcel<T> {}
class GenericBookParcel<X> extends Parcel<Book> {}  //
```

Type argument Book passed to base class Parcel

EXAM TIP A *type argument* must be passed to the *type parameter* of a base class. You can do so while extending the base class or while instantiating the derived class.

Nongeneric class extending a generic class

You can extend a generic base class to define a nongeneric base class. To do so, the derived class doesn't define any type parameters but passes arguments to all type parameters of its generic base class. For example

```
class Parcel<T>{}
class NonGenericPhoneParcel extends Parcel<Phone> {}
```

In the preceding example, `NonGenericPhoneParcel` is a nongeneric class that passes argument `Phone` to its base class `Parcel<T>`.

Watch out for exam questions that try to pass type arguments to a nongeneric class. For class `NonGenericPhoneParcel` defined in the preceding example code, the following code won't compile:

```
NonGenericPhoneParcel<String> var = new NonGenericPhoneParcel<>();
```

Won't compile

EXAM TIP You can't pass type arguments to a nongeneric class.

Multiple type parameters

The example of generic class `Parcel` used in this section defines one type parameter. A generic class with multiple type parameters takes the following form:

```
class ClassName <T1, T2, …, Tn> { /* code */}
```

In the next section on generic interfaces, you'll also work with multiple type parameters.

4.2.2 Working with generic interfaces

A generic interface enables you to abstract over types. In this section, you'll see how to define and implement generic interfaces.

Defining a generic interface

The declaration of a generic interface includes one or more type parameters. Let's look at an example of a generic interface that can accept multiple type parameters: the `MyMap` interface accepts two type parameters and defines methods `put()` and `get()`. You can compare the `MyMap` interface to a simplified version of the `Map` interface, defined in the `java.util` package:

```
interface MyMap<K, V>{
    void put(K key, V value);
    V get(K key);
}
```

MyMap accepts two type parameters—K and V.

put() accepts a key of type K and a value of type V.

For a key of type K, get() returns a value of type V.

Nongeneric class implementing a generic interface

When a nongeneric class implements a generic interface, the type parameters don't follow the class name. For the implemented interface, the type parameters are replaced by actual types:

```
class MapLegendNonGeneric implements MyMap<String, Integer> {
    public void put(String s, Integer i) {}
    public Integer get(String s) { return null; }
}
```

In the preceding example, `MapLegendNonGeneric` is a nongeneric class which implements generic interface `MyMap` (defined in the previous section).

When implementing a generic interface, take note of the type parameters and how they are used in method declarations (method parameters and return types). The methods of an implementing class must implement or override all the interface methods. In the following example, class `MapLegendNonGeneric` won't compile because it doesn't override the abstract method `get(String)` in `MyMap` (the return type of `get()` is declared to be `String`, not `Integer`):

```
class MapLegendNonGeneric implements MyMap<String, Integer> {
    public void put(String s, Integer i) {}
    public String get(String s) { return null; }
}
```

Won't compile

EXAM TIP A nongeneric class can implement a generic interface by replacing its type parameters with actual types.

Generic class implementing a generic interface

Here's an example of declaring a generic class that implements a generic `MyMap` interface. To pass the type parameter information to a class, the type parameters must follow both the name of the class and the interface implemented by the class:

```
interface MyMap<K, V>{
    void put(K key, V value);
    V get(K key);
}
class MapLegendGeneric<K, V> implements MyMap<K, V> {
    public void put(K key, V value) { }
    public V get(K key) { return null; }
}
```

Type parameters are included right after class and interface names.

You might also choose a combination. In the following examples, the classes define only one parameterized type, `V` or `K`. While implementing the `MyMap` interface, the classes pass actual parameters (`String` or `Integer`) to one of the interface's parameterized types (`K` or `V`):

```
class MapLegendGeneric2<V> implements MyMap<String, V> {
    public void put(String key, V value) {}
    public V get(String key) { return null; }
}
```

```
class MapLegendGeneric3<K> implements MyMap<K, String> {
    public void put(K key, String value) {}
    public String get(K key) { return null; }
}
```

It’s important to use a correct combination of type parameters and actual parameters in the method declarations. The following class won’t compile because class `MapLegendGeneric<K>` doesn’t implement method `put(K key, String value)` from the `MyMap` interface:

```
class MapLegendGeneric4<K> implements MyMap<K, String> {
    public void put(Object value, K key) {}
    public String get(K key) { return null; }
}
```

Won’t compile

EXAM TIP Generic classes and interfaces are collectively referred to as *generic types.*

4.2.3 *Using generic methods*

A generic method defines its own formal type parameters. You can define a generic method in a generic or a nongeneric class.

GENERIC METHODS DEFINED IN A NONGENERIC CLASS OR INTERFACE

A nongeneric class doesn’t define type parameters. To define a generic method in a nongeneric class or interface, you must define the type parameters with the method, in its *type parameter section.* A method’s type parameter list is placed just after its access and nonaccess modifiers and before its return type. Because a type parameter could be used to define the return type, it should be known *before* the return type is used. An example

```
abstract class Courier {
    public <E> void deliver(E[] array) {
        for (E item : array) {
            System.out.println("Delivering - " + item);
        }
    }
}
```

Nongeneric class

Generic method

EXAM TIP For a generic method (defined in a nongeneric class or interface), its type parameter list is placed just after the access and nonaccess modifiers and before its return type.

GENERIC METHODS DEFINED IN A GENERIC CLASS OR INTERFACE

The following example defines a generic interface, and a generic method that defines its own type parameter.

Generic interface declaration

```
interface Map<X, Y>{
    <T> void mapMaterial(T t);
}
```

Generic method declaration with its own type parameters

You can also define a generic constructor in a generic class:

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

Generic constructor declaration with type parameter T

Generic class declaration with type parameter X

Instantiating `Phone`

```
Phone<Double> c = new Phone<Double>("Android");
```

In the following "Twist in the Tale" exercise, let's see whether you can determine the difference between the presence and absence of angle brackets in a definition of generic entities.

Twist in the Tale 4.1

Consider this definition of the `Map` interface discussed in a previous section:

```
interface MyMap<K, V>{
    void put(K key, V value);
    V get(K key);
}
```

Now modify that definition to the following:

```
interface MyMap<K, V>{
    void put(K key, V value);
    <V> get(K key);
}
```

Do you think these modifications will make any difference to the definition of the `MyMap` interface?

In the next section, let's see how you can limit the parameter types that you can pass to a generic class, interface, or method.

4.2.4 Bounded type parameters

You can limit the type of objects that can be passed as arguments to generic classes, interfaces, and methods by using bounded type parameters.

Need for bounded type parameter

Without a bounded type parameter (and explicit type casting), you can access only the members defined in the superclass of all classes—that is, class `Object`.

In the following example, the generic class `Parcel` won't be able to access method `getWeight()` of class `Gift`:

```
abstract class Gift{
    abstract double getWeight();
}
```

```
class Book extends Gift{
    public double getWeight() {return 3.2;}
}
class Phone extends Gift{
    public double getWeight() { return 1.1; }
}
class Parcel<T>{
    private T t;
    public void set(T t) {
        this.t = t;
    }
    public void shipParcel() {
        if (t.getWeight() > 10)              <-- Won't compile; type of t is Object.
            System.out.println("Ship by courier ABC");
        else
            System.out.println("Ship by courier XYZ");
    }
}
```

To access members of class `Gift` in `Parcel`, you can limit the type of objects that can be passed to class `Parcel` (to `Gift` and its subclasses) by using bounded parameters (discussed next).

DEFINING BOUNDED TYPE PARAMETERS

You can specify the bounds to restrict the set of types that can be used as type arguments to a generic class, interface, or method. It also enables access to the methods (and variables) defined by the bounds.

Let's restrict the type of objects that can be passed to class `Parcel` to `Gift` so that the methods of class `Parcel` can access the methods and variables of class `Gift`. Because the definitions of classes `Gift`, `Book`, and `Phone` are the same as in the preceding section, they aren't repeated in the following code:

```
class Parcel<T extends Gift>{             <-- ❶ Bounded type parameter
    private T t;
    public void set(T t) {
        this.t = t;
    }
    public void shipParcel() {
        if (t.getWeight() > 10)            <-- ❷ Compiles; type of t is Gift.
            System.out.println("Ship by courier ABC");
        else
            System.out.println("Ship by courier XYZ");
    }
}
```

In the preceding code, the code at ❶ defines a bounded parameter for class `Parcel` with its bounds as class `Gift`. Because the bound of `t` is `Gift`, its method `getWeight()` can be accessed using `t` ❷.

The keyword `implements` isn't used to specify the bound as an interface. The following code won't compile:

```
class Parcel<T implements Serializable>{}        <-- Won't compile
```

EXAM TIP For a bounded type parameter, the bound can be a class, interface, or enum, but not an array or a primitive type. All cases use the keyword `extends` to specify the bound. If the bound is an interface, the `implements` keyword isn't used.

On the exam, you might see a question that tries to instantiate a generic class by passing it a type argument that doesn't comply with its bounded parameter. What do you think happens in this case—a compilation error or a runtime exception? What do you think is the output of the following code, which tries to instantiate class `Parcel` with type parameter `<T extends Gift>`?

```
Parcel<String> p = new Parcel<>();
```

The preceding code will not compile because the type argument `String` isn't within bounds of type variable `T`.

DEFINING MULTIPLE BOUNDS

A type parameter can have multiple bounds. The list of bounds consists of one class and/or multiple interfaces. The following example defines a generic class `Parcel`, the type parameter `T` of which has multiple bounds:

```
interface Wrappable{}
interface Exchangeable{}
class Gift{}
class Parcel <T extends Gift, Exchangeable, Wrappable>{}
```

In this case, the type argument that you pass to the bounded type parameter must be a subtype of all bounds. If you try to pass a type argument that doesn't subtype all the bounds, your code won't compile.

EXAM TIP For a type parameter with multiple bounds, the type argument must be a subtype of all bounds.

4.2.5 Using wildcards

The wildcard ? represents an unknown type. You can use it to declare the type of a parameter; a local, instance, or static variable; and return value of generic types. But you can't use it as a type argument to invoke a generic method, create a generic class instance, or for a supertype.

NEED TO USE AN UNKNOWN TYPE

Before you understand how to use the wildcard, you must know where and why you need it. Say you're given the following class inheritance tree:

```
class Gift{}
class Book extends Gift{}
class Phone extends Gift{}
```

You can assign an object of class `Book` or `Phone` to a reference variable of type `Gift`:

```
Gift gift = new Book();
gift = new Phone();
```

But the following assignment isn't valid:

```
List<Gift> wishList = new ArrayList<Book>();      <- Won't compile
```

You can assign an `ArrayList` to a variable of type `List`. But the type that you pass to it in the angle brackets must be the same. Though `ArrayList<T>` implements `List<T>` for any type `T`, `ArrayList<Book>` implements neither `ArrayList<Gift>` nor `List<>`. So assignment of `ArrayList<Book>` to a variable of type `List<Gift>` isn't allowed with generics.

EXAM TIP You can assign an instance of a subclass, say, `String`, to a variable of its base class, `Object`. But you can't assign `ArrayList<String>` to a variable of type `List<Object>`. Inheritance doesn't apply to the type parameters.

You can use a wildcard to get around this. In the following example, you can assign an `ArrayList` of *any* type to `wishList`:

```
List<?> wishList = new ArrayList<Book>();      <- ? refers to any type
```

Because `?` refers to an unknown type, `wishList` is a list of an unknown type. So it's acceptable to *assign* a list of `Book` objects to it.

Adding objects to collections defined using a wildcard

On the exam, take note of code that tries to add objects to collections that are defined by using the wildcard. Referring to our example, if you try to add or insert a `Book` instance into the `ArrayList` referred by the variable `wishList`, the code won't compile:

```
List<?> wishList = new ArrayList<Book>();     <- ? refers to any type
wishList.add(new Book());                     <- Won't compile
```

Because of the `?` you can invoke method `add()` with literally any object—`String`, `Integer`, `Book`, `Phone`, and others. But `ArrayList<Book>` should only have `Book` instances. Because the compiler can't guarantee it, it forbids adding anything to the list when using a wildcard `?`.

Iterating collections with a wildcard

You can iterate a collection defined using wildcard `?`. Note that the type of the variable used to refer to the list values is `Object`—the base class of all Java classes. Here's an example of using `?` in `wrapGift()` to iterate a `List` of *any* type:

```
class Gift{}
class Book extends Gift{          <- Class Book extends Gift.
    String title;
    Book(String title) {
        this.title = title;
    }
    public String toString() {
        return title;
    }
}
class Courier {
    public static void wrapGift(List<?> list) {
        for (Object item : list) {
            System.out.println("GiftWrap - " + item);
        }
    }
    public static void main(String args[]) {
        List<Book> bookList = new ArrayList<Book>();
        bookList.add(new Book("Oracle"));
        bookList.add(new Book("Java"));
        wrapGift(bookList);

        List<String> stringList = new ArrayList<String>();
        stringList.add("Paul");
        stringList.add("Shreya");
        wrapGift(stringList);
    }
}
```

wrapGift will accept list of any unknown type

Type of variable item is Object, superclass of all objects.

wrapGift will accept a list of Book objects.

wrapGift will accept a list of String objects.

EXAM TIP When you use a wildcard to declare your variables or method parameters, you lose the functionality of adding objects to a collection. In this case, using the `add` method will result in compilation failure.

The wildcard `?` accepts objects of *all* unknown types. Let's use bounded wildcards to limit the types of objects that we can use.

4.2.6 Using bounded wildcards

To restrict the types that can be used as arguments in a parameterized type, you can use bounded wildcards.

Upper-bounded wildcards

You can restrict use of arguments to a type and its subtypes by using `<? extends Type>`, where *Type* refers to a class, interface, or enum.

EXAM TIP In upper-bounded wildcards, the keyword `extends` is used for both a class and an interface.

Consider the following classes:

```
class Gift{}
class Book extends Gift{}
class Phone extends Gift{}
```

For a variable that uses the upper-bounded wildcard `<? extends Gift>`, the following assignments are valid:

```
List<? extends Gift> myList1 = new ArrayList<Gift>();
List<? extends Gift> myList2 = new ArrayList<Book>();
List<? extends Gift> myList3 = new ArrayList<Phone>();
```

Book and Phone extend Gift.

Though Gift doesn't extend itself, this assignment is valid.

Let's see how you can use the upper-bounded wildcard in method parameters. Let's modify the method `wrapGift()`, used in the previous section, to restrict its type arguments to `Gift` or its subclasses (modifications in bold):

```
public static void wrapGift(List<? extends Gift> list) {
    for (Gift item : list) {
        System.out.println("GiftWrap - " + item);
    }
}
```

wrapGift() will accept List of Gift or List of classes that extend Gift.

EXAM TIP In the preceding method `wrapGift()`, the loop variable `item` can be of type `Gift` or its subtype, `Object`.

For the preceding method, you can pass to it `List` of `Gift` or objects that extend class `Gift`. If you try to pass it a list of any other object type, it won't compile.

```
List<Book> bookList = new ArrayList<Book>();
bookList.add(new Book("Oracle"));
bookList.add(new Book("Java"));
wrapGift(bookList);
```

With bounded wildcard <? extends Gift>, wrapGift() will accept List of class Book.

```
List<String> stringList = new ArrayList<String>();
stringList.add("Paul");
stringList.add("Shreya");
wrapGift(stringList);
```

Won't compile; with bounded wildcard <? extends Gift>, wrapGift() won't accept List of class String.

For the exam, you must know the operations that are allowed for variables declared by using upper-bounded wildcards. You can iterate and read values from a collection declared with upper-bounded wildcards. But you can't write any values to the collection. For example, you can't add *any* object to a `List` defined as `List<? extends Gift>` because such a list can refer to a list of either `Gift`, `Book`, or `Phone`. Adding a mismatched object can pollute the list, which isn't allowed.

EXAM TIP For collections defined using upper-bounded wildcards, you can't add any objects. You can iterate and read values from such collections.

It's interesting to note that class `String` is a final class that can't be subclassed. If you try to define a class that extends class `String`, it won't compile:

```
class MyClass extends String {}
```

Won't compile; can't extend final class String.

But it's acceptable to define an upper-bounded wildcard that extends class `String`. Here's the modified code:

```
public static void wrapGift(List<? extends String> list) {
    for (String item : list) {
        System.out.println("GiftWrap - " + item);
    }
}
```

Accept objects of class String or objects of classes that extend String.

EXAM TIP You can use final classes in upper-bounded wildcards. Although `class X extends String` won't compile, `<? extends String>` will compile successfully.

Lower-bounded wildcards

You can restrict use of type arguments to a type and its base or supertypes by using `<? super Type>`, where *Type* refers to a class, interface, or enum. Consider the following classes:

```
class Gift{}
class Book extends Gift{}
class Phone extends Gift{}
```

For a variable that uses the lower-bounded wildcard `<? super Gift>`, note the following assignments:

```
List<? super Gift> myList1 = new ArrayList<Gift>();
List<? super Gift> myList2 = new ArrayList<Object>();
List<? super Gift> myList3 = new ArrayList<Phone>();
List<? super Phone> myList4 = new ArrayList<Gift>();
```

Though Gift isn't its own superclass, this assignment is valid.

Gift extends Object.

Won't compile; gift doesn't extend Phone.

Valid; Phone extends Gift.

So, what can you read from and add to collection objects defined using lower-bounded wildcards? Here's an example:

```
List<? super Gift> list = new ArrayList<Gift>();
list.add(new Gift());
list.add(new Book());
list.add(new Phone());
list.add(new Object());
for (Object obj : list) System.out.println(obj);
```

List<? super Gift> is assigned ArrayList<Gift>.

Can add instances of Gift or its subclasses to List<? super Gift>.

Won't compile

Elements are read as instance Object, superclass of Gift.

EXAM TIP In the preceding example, the loop variable `obj` can't be of type `Gift`.

Table 4.1 lists wildcard and bounded wildcard variables, and the types of values that can be read from and written to them.

Table 4.1 Variables and the values that can be read from or added to them

Variable	Read objects of type	Write objects of type
`List<?>`	`Object`	N/A
`List<? extends Gift>`	`Gift`	N/A
`List<? super Gift>`	`Object`	`Gift` and its subclasses

4.2.7 Type erasure

When class `UseGenericParcel` instantiates `Parcel`, it uses the parameterized type `Parcel<Book>`, replacing the formal type parameter `T` with the actual parameter `Book`. When you do this, you can *assume* to be using the following definition of class `Parcel`, where all references of `T` are replaced with `Book`:

```
class Parcel<Book>{
    private Book t;
    public void set(Book t) {
        this.t = t;
    }
    public Book get() {
        return t;
    }
}
```

This isn't how a generic class is compiled; this is how it behaves.

Though the preceding code can help to a great extent to show how a generic class behaves, it's incorrect. It might make you think that you have access to multiple versions of compiled code, which is incorrect. In this section, you'll see that type information is erased during the compilation process; this is called *type erasure*.

On compilation, the type information of a generic class or an interface is erased. The compilation process generates one class file for each generic class or interface; separate class files aren't created for parameterized types.

EXAM TIP When a generic class is compiled, you don't get multiple versions of the compiled class files. A generic class gets compiled into a single class file, erasing the type information during the compilation process.

The compiler erases the type information by replacing all type parameters in generic types with `Object` (for unbounded parameter types) or their bounds (for bounded parameter types). The compiler might insert type casts to preserve type safety and generate bridge methods to preserve polymorphism in extended generic types.

NOTE Though the exam might not include explicit questions on the contents of a class file after type erasure, it will help you to understand generics better and answer all questions on generics.

ERASURE OF GENERIC TYPE IN CLASSES, INTERFACES, AND METHODS

For a generic class `Parcel`, which uses an unbounded type parameter, say, `T`

```
class Parcel<T>{
    private T t;
    public void set(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
}
```

On compilation, the Java compiler replaces all occurrences of `T` with `Object`:

```
class Parcel {
    private Object t;
    public void set(Object t) {
        this.t = t;
    }
    public Object get() {
        return t;
    }
}
```

Here's an example of an interface that uses both bounded and unbounded type parameters:

```
interface MyMap<K extends String, V>{
    void put(K key, V value);
    V get(K key);
}
```

For the preceding interface, the Java compiler would replace all occurrences of `K` with its first bound class, `String`, and `V` with `Object`:

```
interface MyMap {
    void put(String key, Object value);
    Object get(String key);
}
```

Similarly, for generic methods, the unbounded and bounded type parameters are replaced by `Object` or their first bound class. For the generic method `deliver()` in class `Courier`

```
abstract class Courier {
    public <E> void deliver(E[] array) {
        for (E item : array) {
            System.out.println("Delivering - " + item);
        }
    }
}
```

The Java compiler would replace all occurrences of `E` with `Object`:

```
abstract class Courier {
    public void deliver(Object[] array) {
        for (Object item : array) {
            System.out.println("Delivering - " + item);
        }
    }
}
```

BRIDGE METHODS

The Java compiler might need to create additional methods, referred to as bridge methods, as part of the type erasure process. In the following example, class `BookParcel` extends `Parcel<Book>`:

```
class Book {}
class Parcel<T>{
    private T t;
    public void set(T t) {
        this.t = t;
    }
}
class BookParcel extends Parcel<Book> {
    public void set(Book book) {
        super.set(book);
    }
}
```

For the preceding code, during type erasure, the Java compiler erases and adds a *bridge method* to class `BookParcel`—that is, `set(Object)`. This is to add type safety to class `BookParcel`, ensuring that only `Book` instances can be assigned to its field `t`. Because method `set(Object)` accepts `Object` but casts it to `Book`, it will throw a `ClassCastException` for any other object type:

```
class Parcel {
    private Object t;
    public void set(Object obj) {
        t = obj;
    }
}
class BookParcel extends Parcel {
    public void set(Book book) {
        super.set(book);
    }
    public void set(Object obj) {
        set((Book)obj);
    }
}
```

Throws ClassCastException for objects others than Book

4.2.8 Refreshing the commonly used terms

Table 4.2 lists the new terms that were introduced with generics, which you are sure to see on the exam.

Table 4.2 Commonly used terms with generics and their meanings

Term	Meaning
Generic types	A generic type is a generic class or a generic interface, having one or more type parameters in its declaration.
Parameterized types	An invocation of a generic type is generally known as a parameterized type. For generic type `List<E>`, `List<String>` is a parameterized type.

Table 4.2 Commonly used terms with generics and their meanings

Term	Meaning
Type parameter	You use type parameters to define generic classes, interfaces, or methods. `E` in `List<E>` is a type parameter.
Type argument	A type argument specifies the type of objects to be used for a type parameter. For `List<String>`, `String` is a type argument.
Wildcard	A wildcard is represented by a `?` (a question mark). It refers to an unknown type.
Bounded wildcard	A wildcard is bounded when it is a base or supertype of a type.
Raw type	The name of a generic class, or a generic class without any type arguments, is a raw type. For `List<E>`, `List` is a raw type.

In the next section, you'll learn how the compiler can determine type arguments if you don't specify them while creating instances of generic types.

4.3 Using type inference

[4.2] Use the diamond for type inference

Imagine solving a riddle with multiple constraints in the form of hints. You resolve the constraints to derive the answer. You can compare *type inference* with generating and solving constraints to promote flexibility in a programming language. Here's a simple (nongeneric) example: Java constrains the numeric operands of an addition operator (`+`) to be at least promoted to the `int` data type. So when `+` is used with `int` and `short` types, the type of the resultant value can be *inferred* to be an `int` type. *Type inference* is a Java compiler's capability to determine the argument type passed to an expression or method by examining its declaration and invocation.

With generics, you usually use angle brackets (`<>`, also referred to as the *diamond*) to specify the type of arguments to instantiate a generic class or invoke a generic method. What happens if you don't specify the type arguments? The Java compiler *might* be able to infer the argument type by examining the declaration of the generic entity and its invocation. But if it can't, it'll throw a warning, an error, or an exception. In this section, you'll see how to answer the exam questions on using the diamond for type inference to instantiate generic classes and invoke generic methods.

NOTE By throwing an *unchecked warning*, the compiler states that it can't ensure type safety. The term *unchecked* refers to *operations* that might result in violating type safety. This occurs when the compiler doesn't have enough type information to perform all type checks.

4.3.1 Using type inference to instantiate a generic class

When generics were introduced with Java 5, it was mandatory to include the type arguments to instantiate a generic class. Consider the generic class `Parcel`:

```
class Parcel<T>{
    //..code
}
```

The following code instantiates `Parcel`, passing it type argument `String`:

```
Parcel<String> parcel = new Parcel<String>();
```

Type arguments included to invoke constructor of generic class Parcel.

But with Java 7, you can drop the type arguments required to invoke the constructor of a generic class and use an empty set of type arguments, <>:

```
Parcel<String> parcel = new Parcel<>();
```

With Java 7, empty set of type arguments can invoke constructor of generic class

In the preceding code, the compiler can infer the type argument passed to `Parcel` as `String`. But an attempt to drop the diamond will result in a compilation warning:

```
Parcel<String> parcel = new Parcel();
```

Compilation warning; attempt to assign raw type to generic type

Imagine another situation. What happens if you attempt to try it the other way around? Do you think the following code is valid?

```
Parcel<> parcel = new Parcel<String>();
```

Won't compile

The preceding code won't compile. Imagine what happens if `Parcel` defines a generic constructor:

```
class Parcel<T>{
    <X> Parcel(X x) {}
    public static void main(String[] args) {
        new Parcel<String>(new StringBuilder("Java"));
    }
}
```

Compiler infers type of formal parameter X as StringBuilder.

In the preceding code, `String` is passed as an explicit type argument to the type parameter `T`. The type of the parameter `X` (specified by the constructor) is inferred by the compiler to be `StringBuilder`, which is passed to `Parcel`'s constructor.

4.3.2 Using type inference to invoke generic methods

A Java compiler can't infer the type parameters by using the diamond in the case of generic methods. It uses the type of the actual arguments passed to the method to infer the type parameters. Let's add the generic method `deliver()` to class `Parcel`:

```
class Parcel<T> {
    public <X> void deliver(X x) {
        System.out.println(x.getClass());
    }
    public static void main(String args[]) {
        Parcel<String> parcel = new Parcel<>();
        parcel.<Integer>deliver(new Integer(10));
        //parcel.<>deliver(new Integer(10));
        parcel.deliver("Hello");
    }
}
```

Type parameter X to generic method deliver()

Outputs type of argument passed to deliver().

Type of parameter X is Integer; determined using Integer object passed to deliver().

Won't compile; can't use <> with generic method.

Type of parameter X is String; inferred using actual argument passed to deliver().

Here's the output of the preceding code:

```
class java.lang.Integer
class java.lang.String
```

The next section covers an important topic: mixing generic and raw types. For the exam, you must know how the code behaves when you mix them: compilation warnings, errors, and runtime exceptions. You should also understand that by mixing them, you risk losing type safety.

4.4 Understanding interoperability of collections using raw types and generic types

[4.3] Analyze the interoperability of collections that use raw types and generic types

Before we start with *how* collections that use a raw type operate with generic types, you should know *why* this interoperability was allowed. When generics were introduced with Java 5, there was a *lot* of *existing* Java code, which didn't use generics. Because a new enhancement can't render existing code useless, the existing code that didn't use generics needed to be valid and interoperable, to be made to work with generics. This is also referred to as *migration compatibility*. This, however, introduced multiple complications, including generation of bridge methods and explicit casts.

To recap, when a generic class is used without its type information, it's referred to as its *raw type*. For example, for the generic class `Parcel<T>`, its raw type is `Parcel`.

EXAM TIP Raw types exist only for generic types. Watch out for exam questions that might mention raw types for nongeneric classes and interfaces.

Let's examine the interoperability of code that mixes the assignment of objects of generic types with reference variables of raw types, and vice versa.

4.4.1 Mixing reference variables and objects of raw and generic types

You can assign a parameterized type to its raw type. But the reverse will give a compiler warning. Consider the following generic class:

```
class Parcel<T> {
    private T t;
    public void set(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
}
```

The following assignment is allowed:

```
Parcel parcel = new Parcel<Phone>();
```

But you lose the type information for the class `Parcel` in the preceding code. When you call its method `set()` (passing it a method parameter of *any* type), you'll get a compiler warning:

```
Parcel parcel = new Parcel<Phone>();
parcel.set("harry");
```

Because you lose type information when you use variable of raw type, you can pass String object to set(), instead of Phone object

Here's the detailed warning that the compiler generates when you compile the code using the flag `-Xlint:unchecked`, which informs you that the raw type `Parcel` is unable to comprehend type parameter `T`:

```
warning: [unchecked] unchecked call to set(T) as a member of the raw type
Parcel
        parcel.set(new String("harry"));
                  ^
  where T is a type-variable:
    T extends Object declared in class Parcel
1 warning
```

This happens because the variable `parcel` of raw type `Parcel` doesn't have access to generic type information.

On the exam, watch out for code that mixes raw with generic type. For such code, you'll need to determine whether the code compiles with or without any warning, or throws a runtime exception. In the following code (for class `Parcel` defined in this section), `parcel.set(Phone)` will compile with a warning, but attempt to assign the return value of `parcel.get()` to a variable of type `Phone`:

```
Parcel parcel = new Parcel<Phone>();
parcel.set(new Phone());                  ◁ Compiles with warning
Phone phone = parcel.get();               ◁ Won't compile; with reference variable of raw type
```

EXAM TIP When you mix raw with generic types, you might get a compiler warning or error, or a runtime exception.

Let's get our heads around this with a pictorial representation (see figure 4.5). I've deliberately used interface `List` and class `ArrayList` from the collections framework because you might get to see them in similar code on this exam.

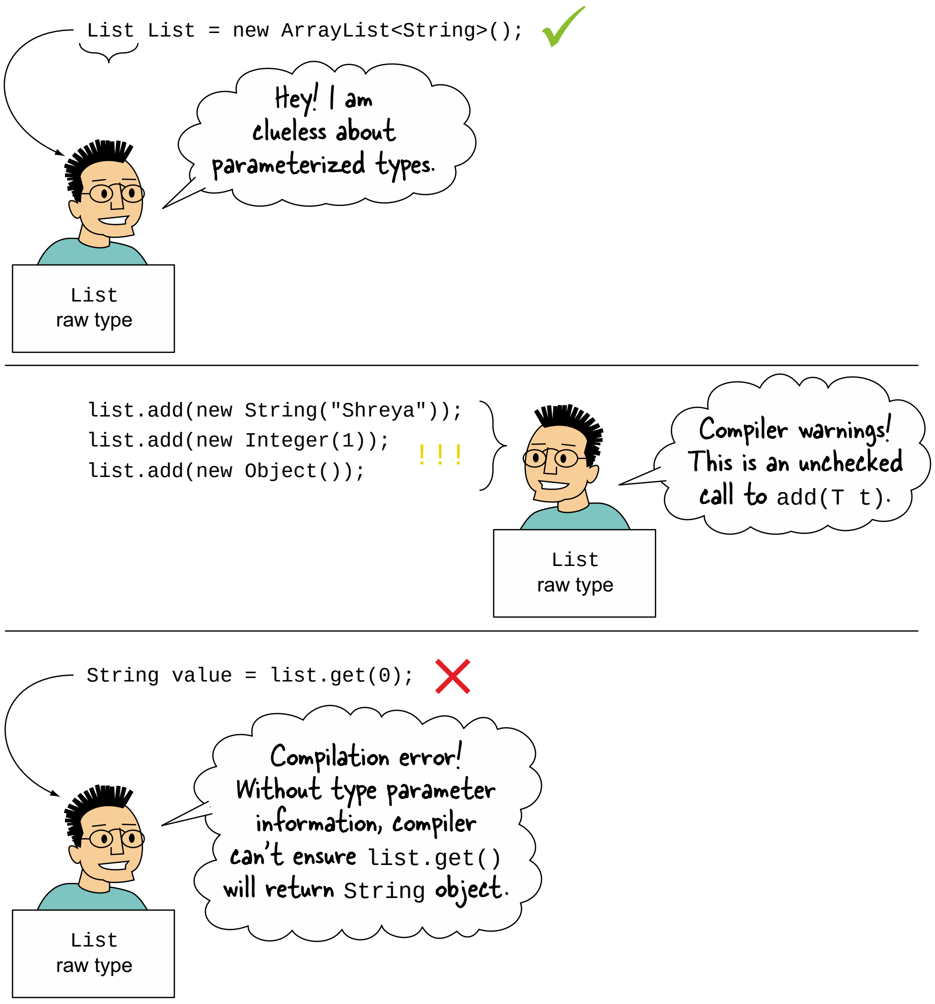

List List = new ArrayList<String>(); ✓
Hey! I am clueless about parameterized types.
List raw type
list.add(new String("Shreya"));
list.add(new Integer(1));
list.add(new Object());
! ! !
Compiler warnings! This is an unchecked call to add(T t).
List raw type
String value = list.get(0); ✗
Compilation error! Without type parameter information, compiler can't ensure list.get() will return String object.
List raw type

Figure 4.5 When you use a reference variable of a raw type, you lose the type information.

Now, let's try to assign a raw type to a parameterized type:

```
Parcel<Phone> parcel = new Parcel();
```

This code generates the following compilation warning:

```
warning: [unchecked] unchecked conversion
        Parcel<Phone> parcel = new Parcel();
                                ^
  required: Parcel<Phone>
  found:    Parcel
```

But it doesn't generate any compiler warning for methods that accept type parameters:

```
Parcel<Phone> parcel = new Parcel();
parcel.set(new Phone());
//parcel.set(new String());
Phone phone = parcel.get();
```

Generates compilation warning

No compiler warnings

Won't compile if uncommented; reference variable parcel knows it's the parameter types.

Compiles successfully

Let me modify this preceding code and use a combination of raw and generic types in the next "Twist in the Tale" exercise. If you can answer this question correctly, you'll answer it correctly on the exam too!

Twist in the Tale 4.2

Consider the definition of the following generic type `MyMap` and class `CustomMap` that implements it:

```
interface MyMap<K, V>{
    void put(K key, V value);
    V get(K key);
}
class CustomMap<K, V> implements MyMap<K, V> {
    K key;
    V value;
    public void put(K key, V value) {
        this.key = key; this.value = value;
    }
    public V get(K key) {
        return value;
    }
}
```

Which options are true about the following code?

```
class Twist4_2 {
    public static void main(String args[]) {
        CustomMap map = new CustomMap<Integer, String>();  //1
        map.put(new String("1"), "Selvan");                //2
        String strVal = map.get(new Integer(1));            //3
        System.out.println(strVal);                         //4
    }
}
```

a. Class `Twist4_2` will compile successfully if you replace line 1 with the following line:

   ```
   CustomMap<Integer, String> map = new CustomMap();
   ```

b. Code on line 2 will generate a compiler warning.
c. Code on line 3 will compile if the type of variable `strVal` is `Object`.
d. The code outputs `null` without any modifications.

On the exam, you might see questions on generics that include other classes from the collection framework, like `Stack`. So the next example uses `Stack` (covered in detail later in this chapter).

Let's look at another scenario, where we mix a method that uses parameters of raw types with actual objects that include generic type information:

```
class Interoperability {
    public static void pushItems(Stack stackParam, Object item) {     ❶
        stackParam.push(item);                                         ❷
    }
    public static void main(String args[]) {
        Stack<String> stackObj = new Stack<String>();                  ❸
        stackObj.push("Paul");                                         ❹
        pushItems(stackObj, new Integer(77));                          ❺
        String value = stackObj.pop();                                 ❻
        System.out.println(value);
    }
}
```

❶ **pushItems() defines parameter of raw type**

❷ **Generates warning: [unchecked] unchecked call to push(E) as member of raw type Stack.**

❸ **Instantiates Stack of String objects.**

❹ **Pushes String object.**

❺ **Calls pushItems(), which pushes Integer to stackObj.**

❻ **Throws ClassCastException at runtime.**

The code at ❶ defines method `pushItems()`, with parameter of raw type `Stack`. Whenever you use a raw type, you lose all the type information. So the code at ❷ throws a compilation warning stating that you made an unchecked call to `push(E)`. It's warning you that you might end up adding incorrect data to your `Stack` object. Code in method `main()` instantiates a `Stack` of `String` objects at ❸. At ❹, the code pushes a `String` object to `stackObj`. So far, so good. A call to method `pushItems()` ❺ pushes an `Integer` object to `stackObj`. Why is this allowed? Because the method parameter `stackParam` is of a raw type, there is no type information, so the compiler doesn't know that the original object (`stackObj`) only allows strings; therefore, the integer is successfully pushed to the stack allowing only strings (`stackObj`). The code at ❻ throws a `ClassCastException` at runtime because the type of the returned object is `Integer` and not `String`.

That's exactly one of the possible problems when you mix generics with nongenerics code. That's also why you get that compiler warning: to warn you that the compiler can't protect you from doing stupid things, like putting an integer in an only-string stack.

As per polymorphism, you can assign an object of a subclass to reference a variable of its base class. But this subtyping rule doesn't work when you assign a collection-of-a-derived-class object to a reference variable of a collection of a base class. Let's see why in the next section.

4.4.2 Subtyping with generics

Because the class `ArrayList` implements the `List` interface, you can assign an object of `ArrayList` to a reference variable of `List`. Similarly, because class `String` extends class `Object`, you can assign an object of `String` to a reference variable of `Object`.

With generics, you must follow certain subtyping rules. The following line is valid because a generic class is a subtype of its raw type:

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

But the following isn't valid:

```
List<Object> list = new ArrayList<String>();      <—— Won't compile
```

This assignment isn't allowed. If you declare a reference variable `List<Object> list`, whatever you assign to the `list` must be of generic type `Object`. A subclass of `Object` is not allowed. Figure 4.6 shows the relationship between the interface `List`, class `ArrayList`, and classes `Object` and `String`. It also shows the related valid and invalid code.

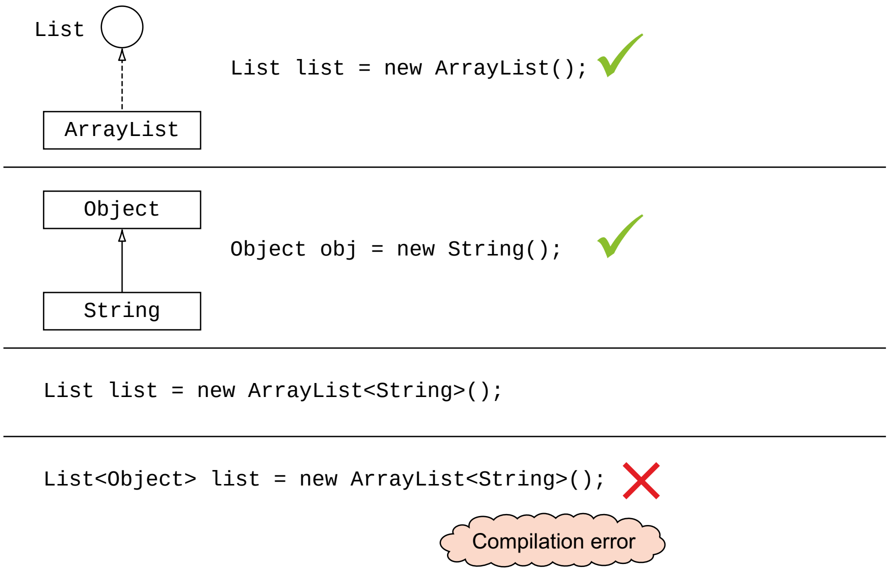

List
ArrayList
List list = new ArrayList(); ✓
Object
String
Object obj = new String(); ✓
List list = new ArrayList<String>();
List<Object> list = new ArrayList<String>(); ✗
Compilation error

Figure 4.6 Object of `ArrayList<String>` isn't compatible with a reference variable of type `List<Object>`.

Apart from generics, this exam will also test you on the collections framework. It'll include many implementations of the interfaces `List`, `Set`, `Deque`, and `Map`, together with classes `Comparator` and `Comparable`. It'll also test you on how to search and sort arrays and lists. The next warm-up section introduces collections and the collections framework. Experienced developers can skip the introduction section.

4.5 Introducing the collections framework: WARM-UP

Imagine that you have to process a list of results submitted by registered users of your website for an opinion poll. You aren't concerned about the order of receiving or processing these results, but you won't accept multiple votes from the same user. Imagine that in another case, you're creating a drawing application that includes an Undo button. You need to keep track of the order in which the drawing commands are selected, so you can undo the last command. Duplicate commands are allowed in a drawing application. Imagine yet another case, when you're looking up a word in a dictionary. The words are ordered alphabetically, but duplicate words don't exist in the dictionary.

These scenarios show examples of needing to *store* and *retrieve* collections of data in various manners. For one collection, you might need to retrieve data in the order in which it was generated. For another collection, you might not allow duplicate values but would prefer data to be sorted on a data item. Figure 4.7 depicts these examples,

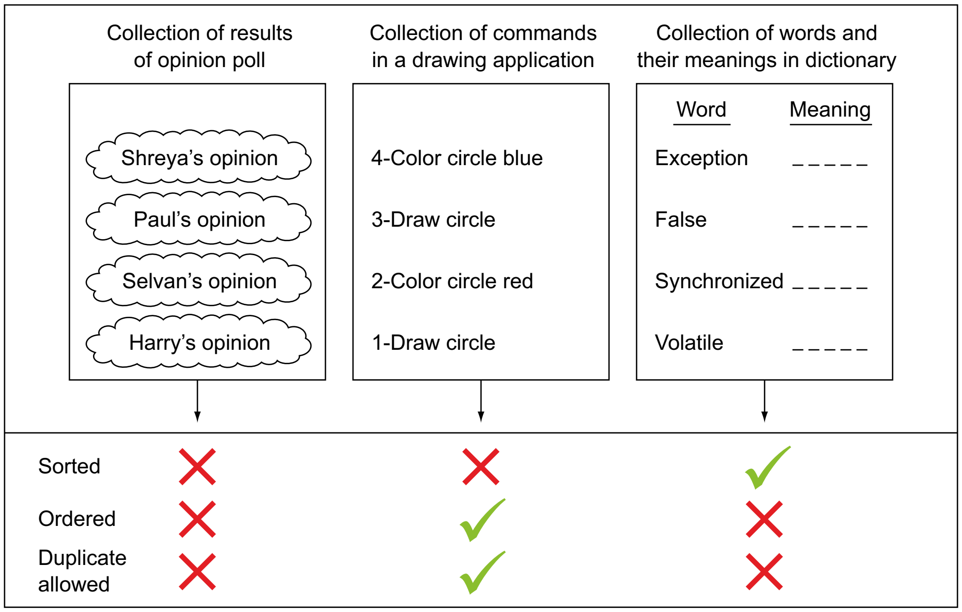


Figure 4.7 Depending on how you need to process a collection of data, you might need a data structure that can sort the collection values, retain the insertion order of the elements, and not allow duplicate elements.

along with a table on their requirements: whether the structure used to store the data needs to be sorted, be ordered, or allow duplicate values.

A *collection* is an object that can group other objects. Each object in a collection is referred to as an *element*. The Java collections framework is architecture for representing and manipulating collections. This framework defines many interfaces to support the need for storing a collection of data in various formats. These collections might need to be ordered, to be sorted, to allow duplicate values, to be immutable, to be of fixed size, and more. The collections framework includes high-performance, high-quality implementations of useful data structures to store a collection of your objects. It includes the following:

- *Interfaces*—Multiple interfaces like `List`, `Set`, `Deque`, and `Map` model the data structures used for storing, accessing, and manipulating a collection of data.
- *Implementations*—Concrete classes like `ArrayList`, `HashSet`, and `TreeMap` implement the interfaces.
- *Algorithms*—Classes like `Collections` and `Arrays` contain utility methods like `sort()` and `search()` for sorting and searching `List` objects or arrays.

NOTE Don't confuse the *interface* `Collection` with the *class* `Collections`. `Collection` is the base interface in the collections framework that is extended by most of the other interfaces. Class `Collections` defines utility methods to operate on or return collections.

Figure 4.8 shows the main interfaces and their implementations in the collections framework (limited to exam coverage). It also shows the classes `Collections` and `Arrays`, which define utility methods to sort and search List or array objects.

The importance of the collection framework

A recruiting manager once asked me why my technical architect places so much emphasis on the ability of a possible recruit to work with data structures and on the recruit's experience with the collections framework. I replied that the collections framework is an *extremely powerful and flexible* framework. A developer who can use its existing classes to store and search data effectively or who can craft out a custom implementation by using the existing framework will be an asset to any organization.

But at the same time, using the collections framework can be overwhelming. The key to optimal use of the collections framework is to get the basics right. So, let's start with the base interface in the Java collections framework: `Collection`.

EXAM TIP All the collection classes are generic; they all define type parameters. Watch out for exam questions that use them without type parameters; these are referred to as raw types.

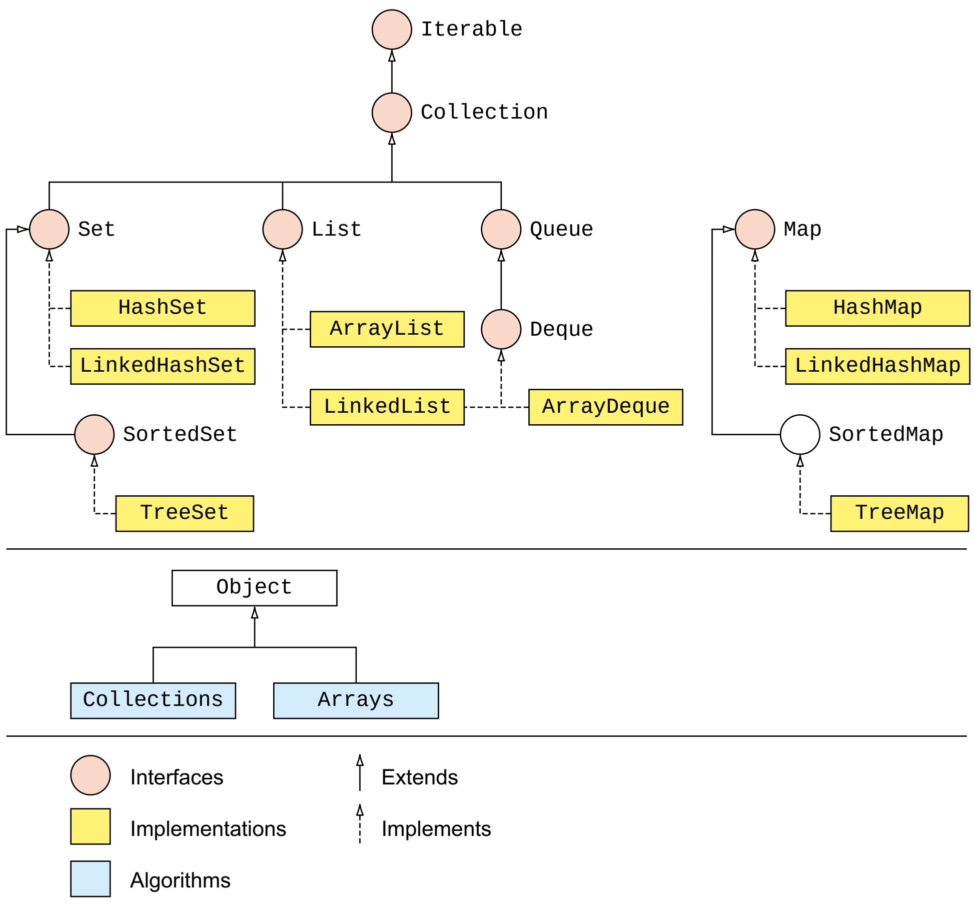


Figure 4.8 The main interfaces and their implementations in the collections framework. Class `Collections`, a utility class that implements various algorithms for searching and sorting, is also a part of the collections framework.

4.6 Working with the Collection interface

[4.5] Create and use List, Set, and Deque implementations

The interface `Collection<E>` represents a group of objects known as its elements. There is no direct implementation of `Collection`; no concrete class implements it. It's extended by more-specific interfaces such as `Set`, `List`, and `Queue`. This collection is used for maximum generality—to work with methods that can accept objects of, say, `Set`, `List`, and `Queue`. Figure 4.9 shows the basic `Collection` interface and its main subinterfaces.

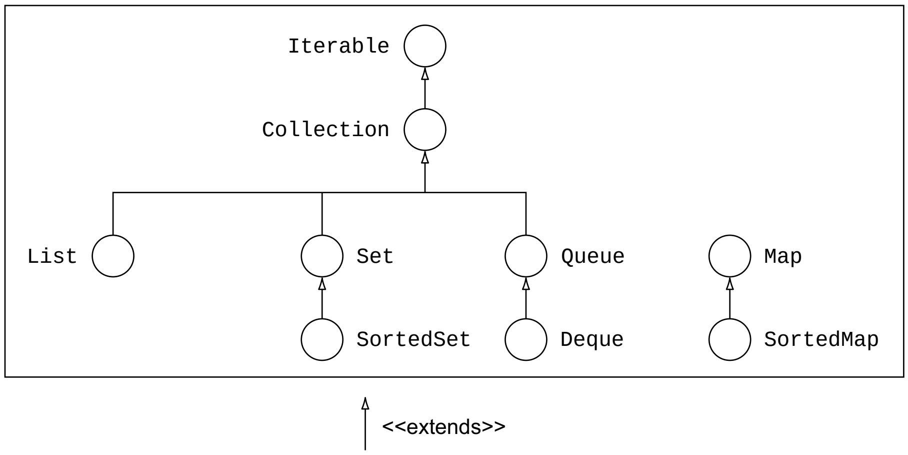


Figure 4.9 The core `Collection` interface and the main interfaces that extend it

All collection classes are generic. Here's the declaration of the `Collection` interface:

```
public interface Collection<E>
           extends Iterable<E>
```

All collection classes are generic.

A thorough understanding of the `Collection` interface will help you absorb a *lot* of concepts and classes that we cover in the rest of this chapter.

EXAM TIP The `Map` interface doesn't extend the core `Collection` interface.

4.6.1 The core Collection interface

The `Collection` interface implements the `Iterable` interface, which defines method `iterator()`, enabling all the concrete implementations to access an `Iterator<E>` to iterate over all the collection objects. So for all the implementing classes, you'd be able to access an `Iterator` to iterate over its elements.

When you work with the concrete implementations of the collections framework, you'll notice that almost all the classes provide two constructors: a void (no-argument) constructor and another that accepts a single argument of type `Collection`. The former creates an empty collection, and the latter creates a new collection with the same elements as its argument, dropping any elements that don't fit the new collection being created. As an interface, `Collection` can't enforce this requirement because it can't define a constructor. But the classes that implement this interface implement it.

The methods of the `Collection` interface aren't marked `synchronized`. The synchronized methods result in a performance drop, even in single-threaded code, and so the creators of the collections framework opted out for them. If you've worked with

Unsupported operations

When specific concrete classes implement the methods defined from the `Collection` interface, the classes might not need to support all its operations specified. For example, a list that's immutable might not support the `Collection`'s method `add()` that adds elements to itself. In this case, this immutable list can choose to return `false` or throw `UnsupportedOperationException` from method `add()`. All methods of the `Collection` interface that modify itself specify (but don't mandate) that classes that don't support these operations *might* throw `Unsupported-OperationException`.

or read about the `Hashtable` or `Vector` classes, which were introduced before the Java collections framework, you'd notice that the methods of these data structures are synchronized. With the existing collections framework, you can get the same functionality (without synchronization) by using the collection classes `HashMap` and `ArrayList`.

NOTE The main target of the exam is to prepare you to write efficient, real-world applications. A solid understanding of the collections framework will go a long way for you, both on the exam and in your career.

4.6.2 Methods of the Collection interface

Figure 4.10 shows the methods of the `Collection` interface, grouped by their functionality: methods that modify `Collection`, methods that query it, and miscellaneous.

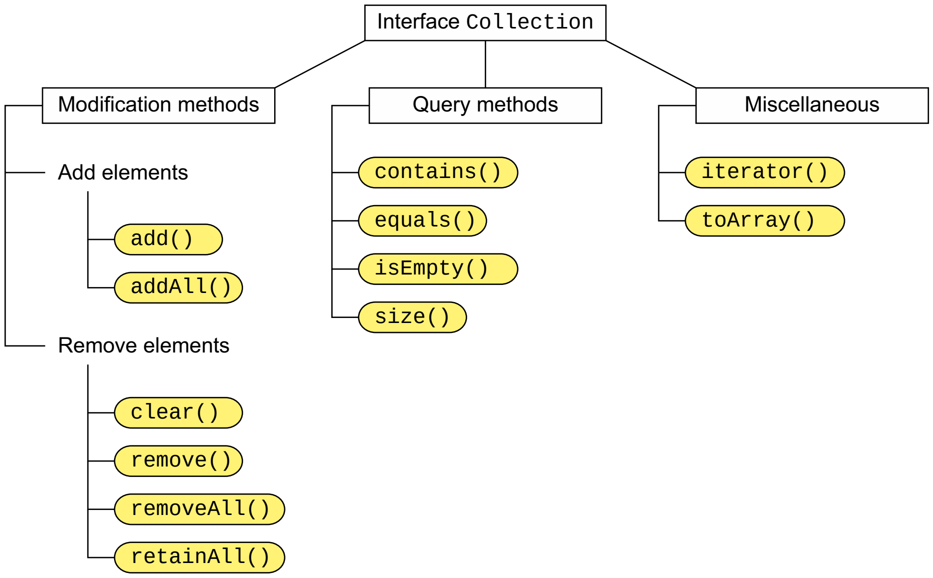


Figure 4.10 Methods defined in the `Collection` interface, grouped by their functionality

Because most of the interfaces implement the `Collection` interface, most of the implementation classes include methods to add, remove, and query its elements and retrieve an `Iterator` to retrieve the collection elements. The implementation, though, varies across classes.

As you proceed and work with the implementation of classes in the collection framework, you'll notice that almost all the classes mention that the `Iterator` returned by method `iterator()` is *fail-fast*. This implies that if you try to modify the structure of a collection by adding or removing elements from it, *after* the `Iterator` is created, the `Iterator` will throw the exception `ConcurrentModificationException`. But this doesn't happen if you modify the collection by using the `Iterator`'s own `add` or `remove` methods. This important behavior prevents collections from returning arbitrary values during concurrent access.

As we move on to the next section to discuss more interfaces (namely, `Set`, `List`, and `Deque`), you'll notice how each of them *might* suggest a different implementation of the methods from the `Collection` interface, to support the specific data structure that they represent.

4.7 Creating and using List, Set, and Deque implementations

[4.5] Create and use List, Set, and Deque implementations

Each of the interfaces `List`, `Set`, and `Deque` model different data structures. The `List` interface allows `null` and duplicate values and retains the order of insertion of objects. `Set` doesn't allow addition of duplicate objects. `Deque` is a linear collection that supports the insertion and removal of elements at both its ends.

In the following sections, when you further explore these interfaces and their implementations, you'll notice the similarities in how each implementation is created. Let's explore the `List` interface and its implementations.

4.7.1 List interface and its implementations

The `List` interface models an ordered collection of objects. It returns the objects to you in the order in which you added them to a `List`. It allows you to store duplicate elements. Figure 4.11 shows valid example data that you'd typically store in a `List`.

In a `List`, you can control the position where you want to store an element. This is the reason that this interface defines overloaded methods to *add*, *remove*, and *retrieve*

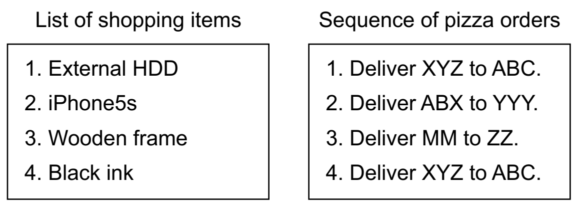


Figure 4.11 Examples of data elements that you could store in a `List`

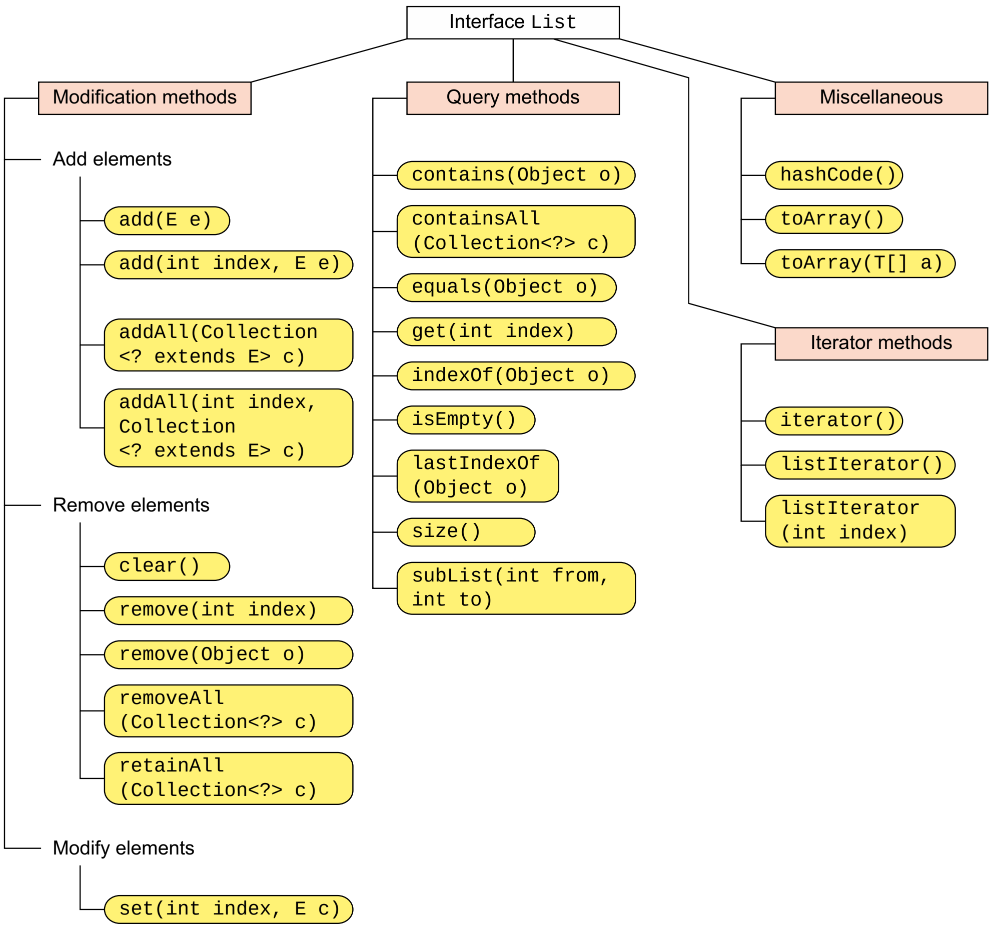


Figure 4.12 Methods of the `List` interface, grouped by their functionality

elements at a particular position. Apart from including the `iterator` method to return an `Iterator`, `List` also includes a method to return a `ListIterator`, to iterate the complete list or a part of it. Figure 4.12 shows the methods of the `List` interface, grouped by their functionality to help you to retain the information better.

In this section, I'll cover only one of the two implementations of interface `List`: `ArrayList` (shown in figure 4.13). Because the other `List` implementation, `LinkedList`, also implements the interface `Deque`, I'll cover it in the next section on `Deque`.

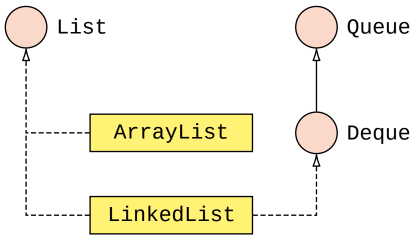


Figure 4.13 The `List` interface and its implementations (on the exam)

ArrayList class

An `ArrayList` is a resizable array implementation of the `List` interface. It's interesting to note that internally, an `ArrayList` uses an array to store its elements. An `ArrayList` defines multiple constructors:

```
ArrayList()
ArrayList(Collection<? extends E> c)
ArrayList(int initialCapacity)
```

Constructs empty list with initial capacity of 10

Constructs list containing elements of specified collection, in the order they're returned by iterator

Constructs empty list with specified initial capacity

Class `ManipulateArrayList` creates an `ArrayList` and manipulates it using methods `add()`, `remove()`, `set()`, and `contains()`:

```
class ManipulateArrayList {
    public static void main(String args[]) {
        ArrayList<String> list = new ArrayList<>();

        list.add("Harry");
        list.add("Selvan");
        list.add("Harry");

        list.add(0, "Paul");

        list.remove("Harry");

        String oldValue = list.set(0, "Shreya");

        list.get(7);

        System.out.println("list contains Harry : " +
                                    list.contains("Harry"));

        ListIterator<String> iterator = list.listIterator();
        while (iterator.hasNext())
            System.out.println(iterator.next());
    }
}
```

Creates ArrayList with default initial capacity of 10.

Adds String objects Harry, Selvan, and Harry; duplicate values allowed

Adds String object Paul at first position, shifting existing list elements to right

Uses equals() to find and remove first occurrence of value matching String Harry

Replaces value at position 0 with String Shreya, retrieving replaced value.

Retrieves element at position 7; throws IndexOutOfBoundsException because only three elements remain in list

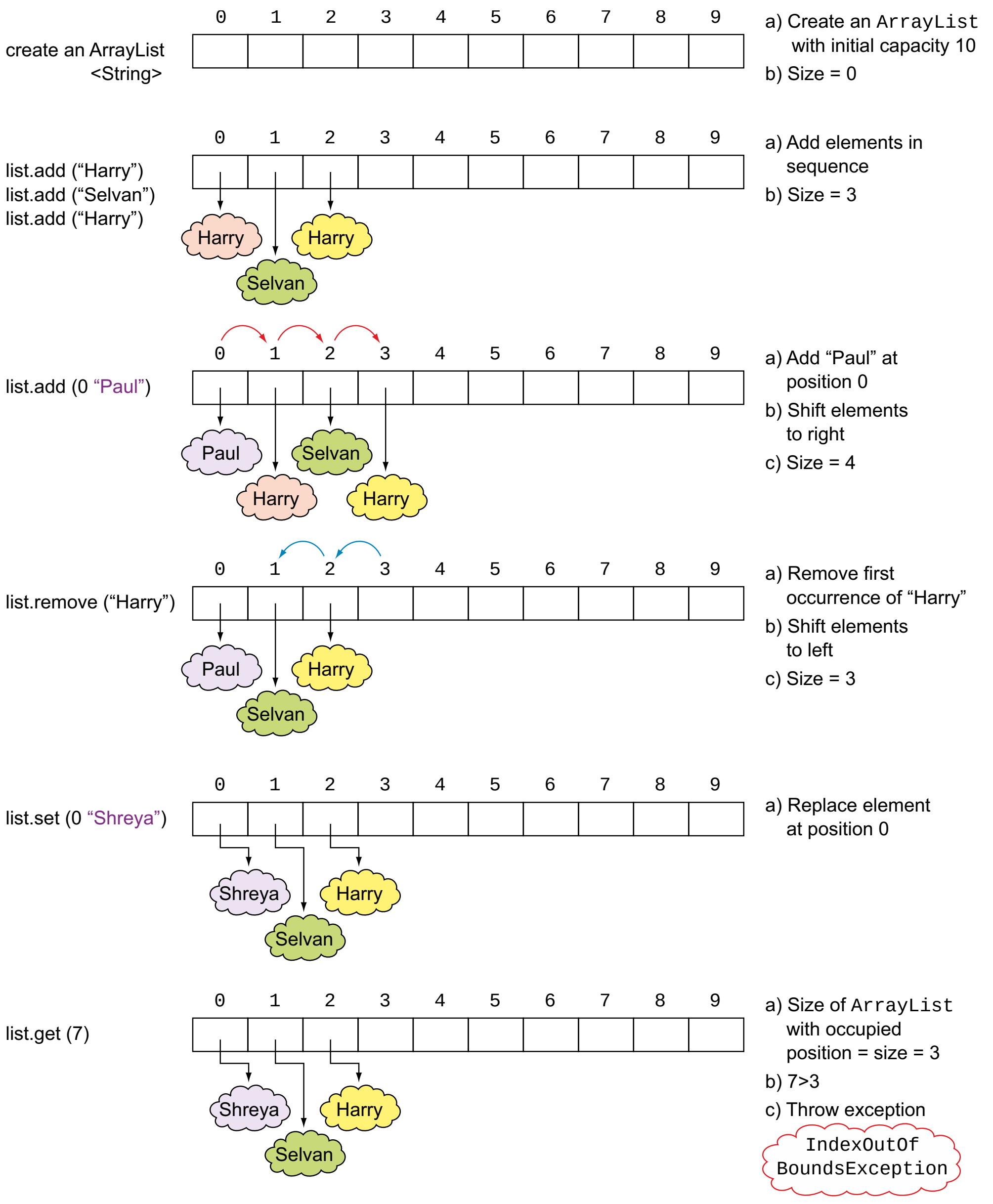
contains() searches sequentially and uses equals() to find first matching occurrence

List can return multiple iterators, Iterator and ListIterator.

hasNext() returns boolean value indicating whether iterator can access more values

next() accesses and returns next value

It's interesting to note that an `ArrayList` uses the `size` variable to keep track of the number of elements inserted in it. By default, an element is added to the first available position in the array. But if you add an element to an earlier location, the rest of the list elements are shifted to the right. Similarly, if you remove an element that isn't the last element in the list, `ArrayList` shifts the elements to the left. As you add more elements to an `ArrayList` that can't be added to its existing array, it allocates a bigger array and copies its elements to the new array. An `ArrayList` maintains a record of its size, so that you can't add elements at arbitrary locations. Figure 4.14 shows how elements are added, removed, and modified in an `ArrayList`.



create an ArrayList <String>
a) Create an ArrayList with initial capacity 10
b) Size = 0
list.add ("Harry")
list.add ("Selvan")
list.add ("Harry")
a) Add elements in sequence
b) Size = 3
list.add (0 "Paul")
a) Add "Paul" at position 0
b) Shift elements to right
c) Size = 4
list.remove ("Harry")
a) Remove first occurrence of "Harry"
b) Shift elements to left
c) Size = 3
list.set (0 "Shreya")
a) Replace element at position 0
list.get (7)
a) Size of ArrayList with occupied position = size = 3
b) 7>3
c) Throw exception
IndexOutOfBoundsException
Harry
Selvan
Paul
Shreya

Figure 4.14 An `ArrayList` offers a resizable array. Internally, it uses an array to store its elements. It manipulates this array to add, remove, or modify `ArrayList` elements. The elements of the internal array are moved to the left or right, when elements are removed from or added to it, respectively. If the `ArrayList` exceeds the existing size of the internal array, its elements are copied to a new array with increased size.

What happens when you ask an ArrayList to remove an object by using method remove(Object obj)? It *sequentially* searches the ArrayList to find the target object. Have you ever wondered how the class ArrayList determines the equality of objects? In the preceding example, you're trying to remove a String object with the value Harry. ArrayList compares the target object and the object that it stores by using method equals(). If a match is found, the ArrayList removes the first occurrence of the String value Harry.

EXAM TIP To remove an element, an ArrayList first searches through its elements to find an element that can be considered equal to the target element. It does so by calling method equals() on the target object and its own objects, one by one. If a matching element is found, remove(Object) removes the *first* occurrence of the match found.

IMPORTANCE OF THE EQUALS METHOD IN FINDING AND REMOVING VALUES FROM AN ARRAYLIST

The example code in the preceding section uses String instances, which override method equals(). Let's work with an example in which the class, whose objects are stored by an ArrayList, doesn't override method equals().

In the following example, class UsingEquals stores Emp instances in an ArrayList. Class UsingEquals tries to remove an Emp object from its ArrayList by using method remove(). Do you think it'll work? Here's the code:

```
class UsingEquals {
    public static void main(String args[]) {
        ArrayList<Emp> list = new ArrayList<Emp>();
        list.add(new Emp(121, "Shreya"));
        list.add(new Emp(55, "Harry"));
        list.add(new Emp(15, "Paul"));
        list.add(new Emp(121, "Shreya"));

        System.out.println(list.size());

        Emp emp = new Emp (121, "Shreya");
        list.remove(emp);

        System.out.println(list.size());
    }
}
class Emp {
    int id;
    String name;
    Emp(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

Create an ArrayList of Emp objects.

Prints "4"

Tries to remove object referred by emp from list

No match found; no objects removed; prints "4".

In the preceding example, no Emp objects were removed from list. This is because Emp doesn't define method equals(), so the default implementation of method equals() of class Object is used. As you already (should) know, this compares the object references for equality and not the object contents. So method remove() fails to find a matching object referred by `emp` in `list`. The answer is to override method `equals()` in class `Emp` (modified code in bold):

```
class Emp {
    int id;
    String name;
    Emp(int id, String name) {
        this.id = id;
        this.name = name;
    }
    public boolean equals(Object obj) {
        if (obj instanceof Emp) {
            Emp emp = (Emp)obj;
            if (emp.id == this.id && emp.name.equals(this.name))
                return true;
        }
        return false;
    }
}
```

equals() returns true when Emp is compared with another Emp that shares same value for instance variables id and name

With the preceding definition of class `Emp`, class `UsingEquals` will be able to find and remove a matching value for the `Emp` instance referred by `emp`.

EXAM TIP If you're adding instances of a user-defined class as elements to an `ArrayList`, override its method `equals()` or else its methods `contains()` or `remove()` might not behave as expected.

I've often observed that when people read the collection framework (which is seemingly complicated), they tend to overlook simple concepts. For example, here's one simple concept: reference variables store a reference to the object that they refer to, and they can be reassigned a new object. In the next "Twist in the Tale" exercise, let's see how this concept can be used on the exam to test you on the collection framework.

Twist in the Tale 4.3

What is the output of the following code?

```
import java.util.*;
class RemoveArrayListElements {
    public static void main(String args[]) {
        ArrayList<Integer> list = new ArrayList<>();
        Integer age1 = 20;
        Integer age2 = 20;
        list.add(age1);
        list.add(age2);
        System.out.print(list.size() + ":");
        age1 = 30;
        list.remove(age1);
        System.out.print(list.size());
    }
}
```

- a 1:1
- b 2:1
- c 2:2
- d 1:0
- e 2:0

EXAM TIP The `ArrayList` methods `clear()`, `remove()`, and `removeAll()` offer different functionalities. `clear()` removes all the elements from an `ArrayList`. `remove(Object)` removes the first occurrence of the specified element, and `remove(int)` removes the element at the specified position. `removeAll()` removes from an `ArrayList` all of its elements that are contained in the specified collection.

The other important methods of class `ArrayList` are `get()` and `contains()`.

The other implementation of the `List` interface, `LinkedList`, also implements the `Deque` interface, covered in the next section.

4.7.2 Deque interface and its implementations

A *queue* is a linear collection of objects. A `Deque` is a double-ended queue, a queue that supports insertion and deletion of elements at both its ends. Let's revisit the hierarchy of the `Deque` interface, as shown in figure 4.15. The `Deque` interface extends the `Queue` interface.

As a double-ended queue, a `Deque` can work as both a *queue* and a *stack*. A queue is a linear collection of elements, in which the elements are added to one end and are processed (or taken off) from the other end. For example, in a queue of people at a

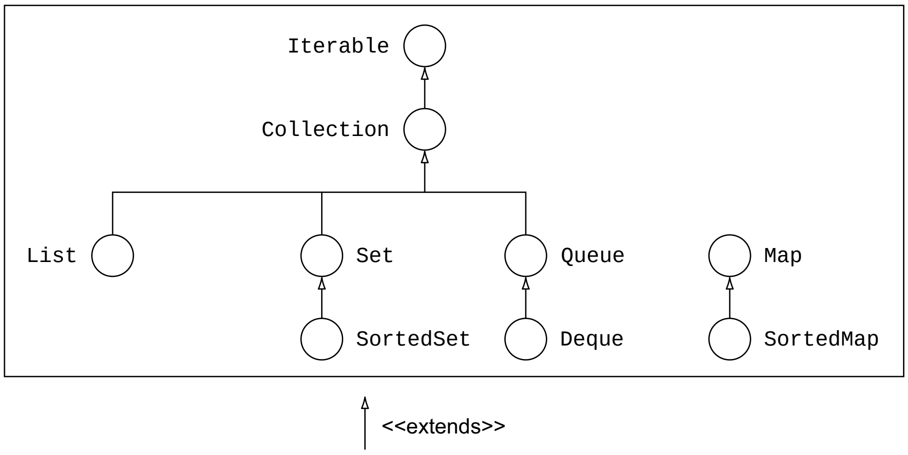


Figure 4.15 Hierarchy of the `Deque` interface

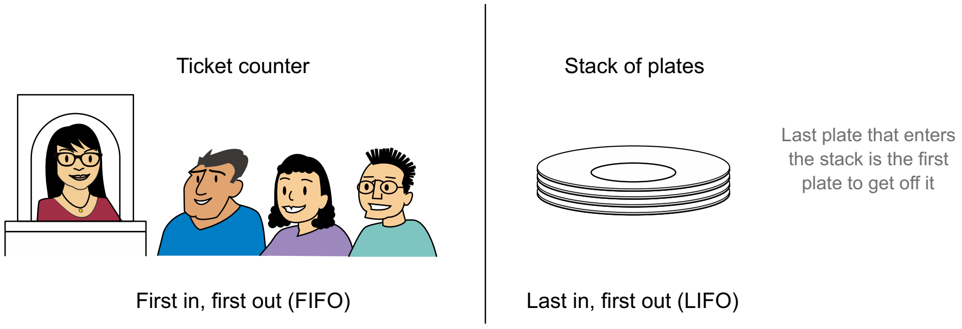

Ticket counter
First in, first out (FIFO)
Stack of plates
Last plate that enters the stack is the first plate to get off it
Last in, first out (LIFO)

Figure 4.16 Real-world examples of a queue and a stack

ticket counter, new persons enter the queue at its *end*. The tickets are issued to the people at its beginning. A queue is also referred to as a first in, first out (FIFO) list.

A *stack* is a linear collection of elements that allows objects to be added and removed at the same end. For example, in a stack of plates, the plates are always added to the top and removed from the top. A stack is also referred to as a last in, first out (LIFO) list data structure. Figure 4.16 shows real-world examples of a queue and a stack.

The `Deque` interface defines multiple methods to add, remove, and query the existence of elements from both its ends. Because `Deque` works as both a *queue* and a *stack*, it's easier to get a hang of its methods if you understand the operations and corresponding methods used for queues and stacks. As I mentioned previously, in a queue, elements are typically added to its tail (or end), and taken off from its head (or beginning). Figure 4.17 shows the `Queue` methods (`Deque` extends `Queue`) used to add elements to the end of a queue and remove or query elements from its beginning.

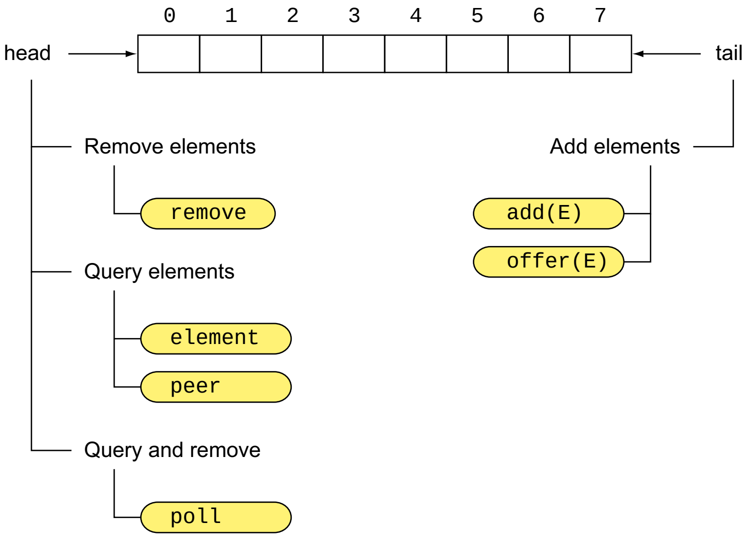


Figure 4.17 `Queue` methods used to work with `Deque` as a FIFO data structure

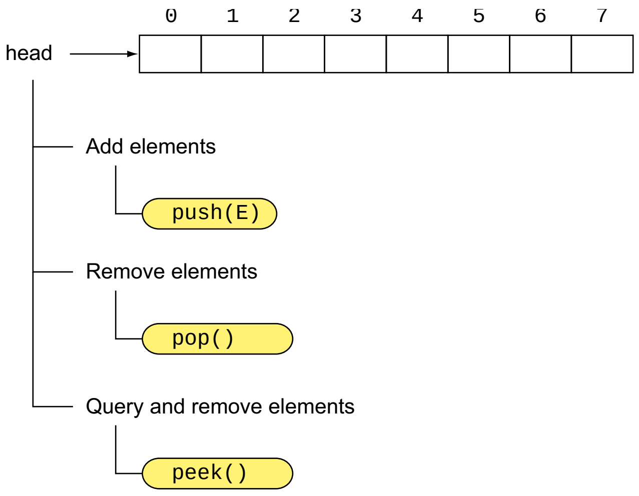


Figure 4.18 The stack methods used to work with Deque as a LIFO data structure

Now, let's see how a Deque works as a stack. In a stack, elements are added and taken off the same end of the queue: its head. Figure 4.18 shows the stack methods used to add and remove elements from its head, using Deque as a LIFO data structure.

For a stack, which allows insertion at only one end, the purpose of method push() is implicit: elements are added at (and removed from) the top. Similarly, with Queue, which usually enables insertion only at its tail, the purpose of method add() or offer() seems implicit: to insert elements at the tail of the list. Now, because Deque supports a double-linked list, supporting insertion at both its ends, method add() or offer() could be ambiguous. So Deque supports other methods with explicit purposes, including addFirst(), addLast(), offerFirst(), and offerLast(). So you can see multiple methods that serve the same purpose. Figure 4.19 shows the Deque interface, representing it as a list with a head and tail. Elements of all implementations of Deque might not be implemented as a contiguous list. It's just to show the beginning and end of a list. This figure shows the methods that are used to add, delete, and query methods at both ends of Deque. The figure also shows other methods that remove and query the Deque elements at random positions.

EXAM TIP The legacy class Stack is used to model a LIFO list. The addition, removal, and query operations of a Stack are named push(), pop(), and peek(). Though the Deque interface isn't related to Stack, Deque supports a double-ended queue and can be used as a Stack. Deque also defines the methods push(), pop(), and peek() to add, remove, and query elements at its beginning.

Though the Deque implementations aren't required to prohibit the addition of null values, it is strongly recommended that they do because certain Deque implementations return null to signal that the underlying list is empty.

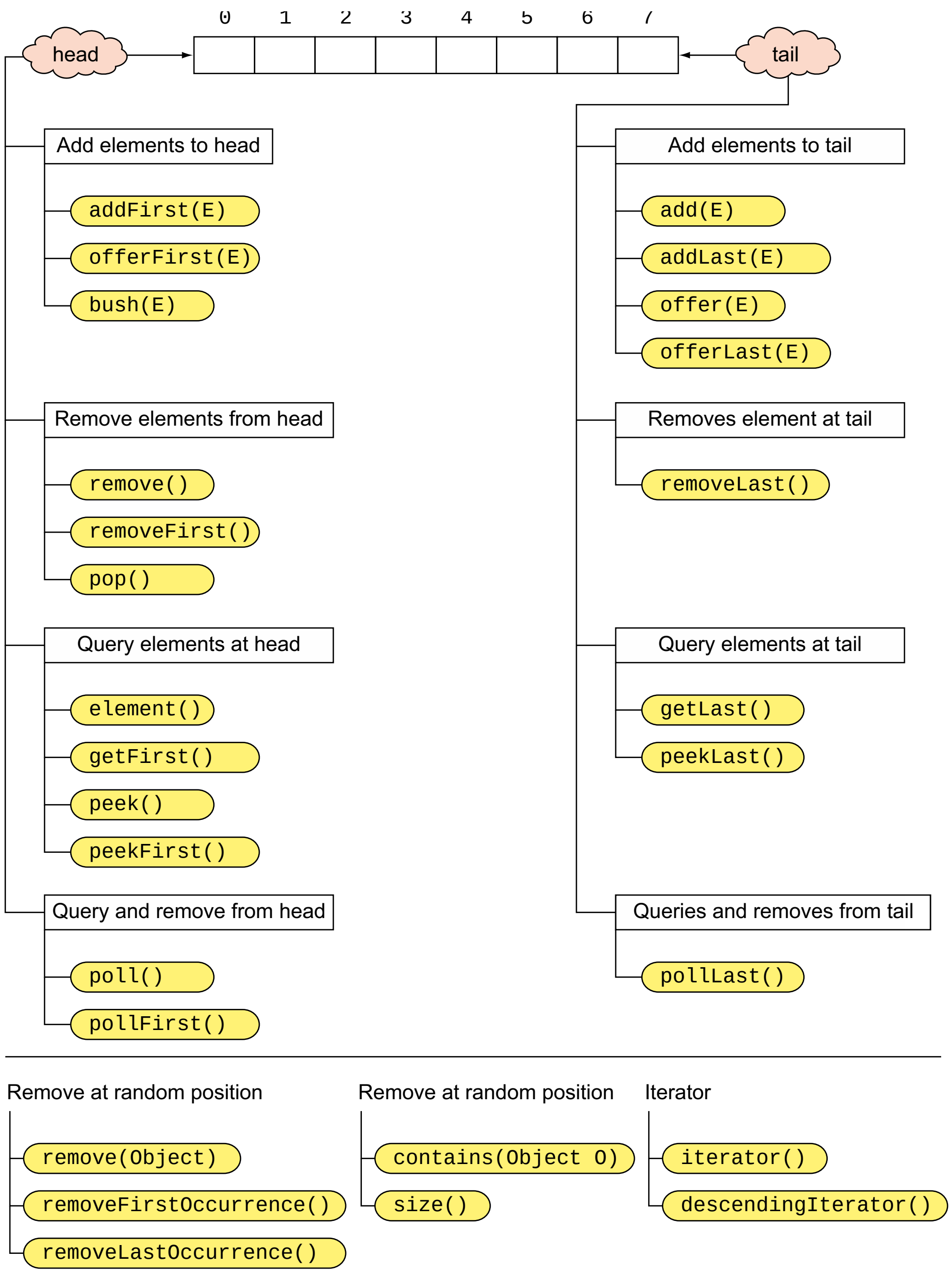


Figure 4.19 Deque methods used to add, remove, and query elements at both its ends. The figure also includes other methods, which operate at random positions.

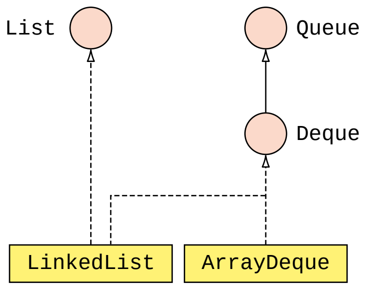


Figure 4.20 The Deque interface and its implementations (on the exam)

IMPLEMENTATIONS OF THE DEQUE INTERFACE

This section covers two main `Deque` implementations: `ArrayDeque` and `LinkedList`, as shown in figure 4.20.

CLASS ARRAYDEQUE

Let's get started with class `ArrayDeque`. It's a resizable array implementation of the `Deque` interface. Here's a list of constructors of this class:

```
ArrayDeque()                               // Constructs empty array deque with initial capacity to hold 16 elements.
ArrayDeque(Collection<? extends E> c)      // Constructs deque containing elements of specified collection, in order of return by collection's iterator
ArrayDeque(int numElements)                // Constructs empty array deque with initial capacity to hold specified number of elements
```

Let's work with an example using the `ArrayDeque` constructor and other methods from this class:

```
import java.util.*;
class TestArrayDeque {
    public static void main(String... args) {
        String strArray[] = {"A1", "B2", "C3"};          // String array

        ArrayDeque<String> arrDeque = new
                     ArrayDeque<String>(Arrays.asList(strArray));   // Creates ArrayDeque from String List; Arrays.asList converts array to List.

        arrDeque.push("D4");                             // push() adds element at Deque beginning
        arrDeque.offer("E5");                            // offer() adds element at Deque end

        //arrDeque.push(null);                           // Can't add null to ArrayDeque; will throw NullPointerException.

        System.out.println(arrDeque.pop());              // pop() returns and removes element at Deque beginning
        System.out.println(arrDeque.remove());           // remove() also returns and removes element at Deque beginning

        arrDeque.add("F6");                              // add() adds an element to end of queue
        System.out.println(arrDeque.peek());             // peek() queries and returns element at beginning of queue

        System.out.println(arrDeque);      #J
    }
}
```

Here's the output of the preceding code:

```
D4
A1
B2
[B2, C3, E5, F6]
```

Whenever `Deque` adds or removes an element, it modifies its pointers to the beginning and end. The take-away from the preceding code is that you need to take note of the methods that add to the beginning or end of the list. You also need to take note of methods like `peek()`, which only queries `Deque`; `remove()`, which removes elements from `Deque`; and `poll()`, which queries and removes an element from `Deque`. Method `poll()` queries and removes, and method `remove()` just removes. Method `poll()` returns `null` when `Deque` is empty and `remove()` throws a runtime exception.

EXAM TIP All the insertion methods (`add()`, `addFirst()`, `addLast()`, `offer()`, `offerFirst()`, `offerLast()`, and `push()`) throw a `NullPointerException` if you try to insert a `null` element into an `ArrayDeque`.

This is a classic example of how to implement a requirement or a recommendation in a concrete class. The `Deque` interface suggests that the implementing classes shouldn't allow `null` elements because some of its special methods return `null` to indicate that the underlying `Deque` is empty. To implement this suggestion, the methods that add elements to class `ArrayDeque` throw `NullPointerException` when you try to add a `null` element to it.

You can iterate over the elements of `Deque` by using an `Iterator`, returned by methods `iterator()` and `descendingIterator()`.

NOTE Together with the `Deque`-specific methods discussed in this section, `ArrayDeque` also implements methods inherited from the `Collection` interface, such as `contains()`, `indexOf()`, and others.

Class LinkedList

Class `LinkedList` implements both the `List` and `Deque` interfaces. So it's a double-linked list implementation of the `List` and `Deque` interfaces. It implements all `List` and `Deque` operations. Unlike `ArrayDeque`, it permits addition of `null` elements.

Here are the constructors of class `LinkedList`:

```
LinkedList()                                  <- Constructs empty list.
LinkedList(Collection<? extends E> c)          <- Constructs list containing elements of specified collection, in order of return by collection's iterator
```

So what happens when you add elements to or remove elements from a `LinkedList`? Let's work with an example:

```
import java.util.*;
class TestLinkedList {
    public static void main(String... args) {
        LinkedList<String> list = new LinkedList<String>();

        list.offer("Java");
        list.push("e");
        list.add(1, "Guru");

        System.out.println(list);

        System.out.println(list.remove("e"));

        Iterator<String> it = list.iterator();
        while(it.hasNext()) System.out.println(it.next());
    }
}
```

Creates empty LinkedList of String objects.

Uses offer() to add String "Java" to end

Uses push() to add String "e" to beginning.

Uses add(1, "Guru") to insert "Guru" at position 1, adjusting references for adjacent values.

Finds and removes first matching occurrence for String object "e"

Iterates in sequential manner, from first element to last

Figure 4.21 shows how a `LinkedList` maintains a reference to its first and last elements. It also shows how, in the absence of using an array, each node in a `LinkedList` maintains a reference to its previous and next element. Whenever you add elements to or remove elements from a `LinkedList`, it modifies the previous and next references of its adjacent elements. As you can see, because each list element maintains a reference to its previous and next element, this list can be traversed in forward and reverse directions.

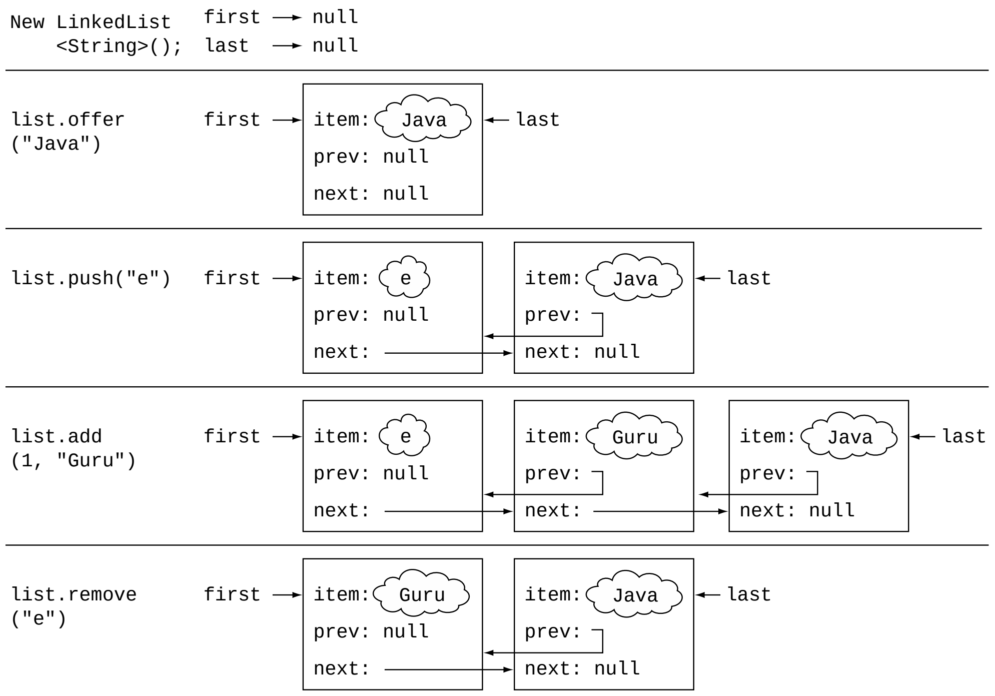


Figure 4.21 Pictorial representation of code that adds elements to and removes elements from a `LinkedList`

Note the difference in internal manipulation of an `ArrayList` or `ArrayDeque` and a `LinkedList`. A `LinkedList` doesn't move a set of elements when you add a new element to it. It modifies the value of the reference variables `prev` and `next`, for adjacent elements, to keep track of its sequence of elements. This is unlike `ArrayList` or `ArrayDeque`, which copy a set of array elements to the right or left, when elements are added to or removed from it.

On the exam you'll get questions to choose the most appropriate interface or class based on a given scenario. A `LinkedList` is like an `ArrayList` (ordered by index) but the elements are double-linked to each other. So besides the methods from `List`, you get a bunch of other methods to add or remove at the beginning and end of this list. So it's a good choice if you need to implement a queue or a stack. A `LinkedList` is useful when you need fast insertion or deletion, but iteration might be slower than an `ArrayList`.

EXAM TIP Because a `LinkedList` implements `List`, `Queue`, and `Deque`, it implements methods from all these interfaces.

In the next "Twist in the Tale" exercise, let me modify the preceding code and see whether you can determine how that affects the code output. Let's see whether you still remember the inheritance concepts covered in chapter 3.

Twist in the Tale 4.4

What is the output of the following code?

```
import java.util.*;
class TestLinkedList {
    public static void main(String... args) {
        List<String> list = new LinkedList<String>();

        list.offer("Java");
        list.push("e");
        list.add(1, "Guru");
        list.remove("e");

        System.out.println(list);
    }
}
```

- **a** `[Guru, Java]`
- **b** `[Java, Guru]`
- **c** `[e, Guru, Java]`
- **d** Compilation error
- **e** Runtime exception

Let's explore another important interface, `Set`, and its implementing classes in the following section.

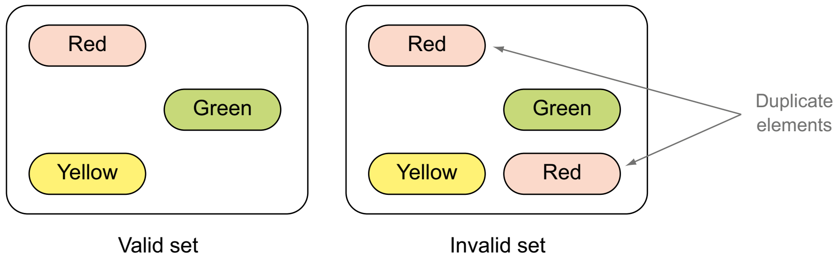


Figure 4.22 Valid and invalid examples of `Set`

4.7.3 Set interface and its implementations

The `Set` interface models the mathematical *Set* abstraction. It's a collection of objects that doesn't contain duplicate elements. Figure 4.22 shows valid and invalid examples of `Set`.

EXAM TIP The `Set` interface doesn't allow duplicate elements and the elements are returned in no particular order.

To determine the equality of objects, `Set` uses its method `equals()`. For two elements, say `e1` and `e2`, if `e1.equals(e2)` returns `true`, `Set` doesn't add both these elements. `Set` defines methods to add and remove its elements. It also defines methods to query itself for the occurrence of specific objects. Because it indirectly implements the `Iterable` interface, it includes method `iterator()` to retrieve an `Iterator`. It also includes methods to convert it into an array. Figure 4.23 shows the methods of the `Set` interface, grouped for convenience.

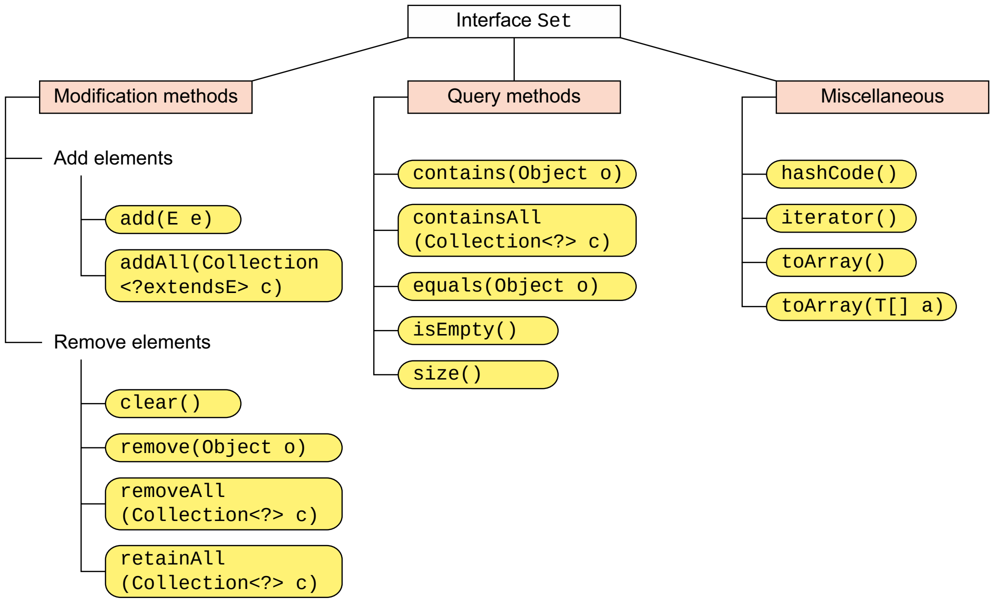


Figure 4.23 Methods of the `Set` interface, grouped for convenience

The exam will query you on the use of Set and its methods. For example, it may query on the appropriate scenarios for using Set or its implementation. It might include a question such as this: When you try to add duplicate String values to a Set, does it throw an exception or simply ignore the duplicate value?

The answer to these questions can vary with the implementing classes. For example, one implementation may return false if you add a duplicate String value, but another may throw an exception. Let's look at implementations of the Set interface, how they're related, and the behavior of their methods in the next section.

4.7.4 Set implementation classes

For the exam, we'll work on the main Set implementation classes: HashSet, LinkedHashSet, and TreeSet, as shown in figure 4.24.

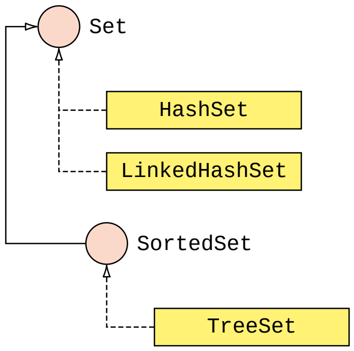


Figure 4.24 The Set interface and its implementations (on the exam)

Class HashSet

Class HashSet implements the Set interface. As required by the Set interface, it doesn't allow duplicate elements. Also, it makes no guarantee to the order of retrieval of its elements. It's implemented using a HashMap. To store and retrieve its elements, a HashSet uses a hashing method, accessing an object's hashCode value to determine the bucket in which it should be stored.

Before I discuss HashSet further, it's important that you understand the meaning of *buckets* and importance of methods hashCode() and equals(). Let's use a simple example of a hotel—when guests leave the hotel they must leave their room key at reception. There the key is put in one big bucket. So when guests arrive they say their room number and the receptionist has to go through all the keys until he finds the matching one (compare it with method equals()). Then a new system is introduced. Instead of having one big bucket, they have some smaller buckets, each with a label (1–9). From now on they apply the same algorithm each time—when the room key is left at reception, the receptionist adds all numbers of the room and repeats this process until just 1 number is left (e.g. 236 -> 2+3+6=11 -> 1+1=2). The key is put in the bucket with that number (hashCode). When guests arrive and want their key back, they say the room number, the receptionist applies the algorithm (compare it with hashCode()), goes to the corresponding bucket, and searches for the matching room key (method equals()).

Let's see what happens when the class AddElementsToHashCode tries to add unique and duplicate objects to a HashSet:

```
class AddElementsToHashSet {
    public static void main(String args[]) {
        String str1 = new String("Harry");
        String str2 = new String("Shreya");
        String str3 = new String("Selvan");
        String str4 = new String("Shreya");

        HashSet<String> set = new HashSet<>();

        set.add(str1);
        set.add(str2);
        set.add(str3);
        set.add(str4);

        for (String e : set) System.out.println(e);
    }
}
```

Create new HashSet<String>

Add str1 to set.

Add str2 to set.

Add str3 to set.

Duplicate String "Shreya". Not added to set.

Prints Harry, Shreya, and Selvan (not always in this order).

In the preceding code, the string object `"Shreya"` referred by the variable `str4` isn't added to `set`. `HashSet` uses the `hashCode()` values of the string objects to determine the appropriate bucket to add them to. A bucket can store multiple objects. Before adding an object, `HashSet` compares its existing elements in the target bucket by using methods `hashCode()` and `equals()` to avoid duplicates being added.

A few important points about working with the preceding example code on `HashSet`:

- Method `hashCode()` doesn't call method `equals()`.
- Method `equals()` doesn't call method `hashCode()`.
- Classes should override their `hashCode()` methods efficiently to enable collection classes like `HashSet` to store them in separate buckets.

To make the concept sink in, here's the next "Twist in the Tale" exercise for you. Let's examine the role of `equals()` and `hashCode()` in storing and retrieving elements in `HashSet`.

Twist in the Tale 4.5

Given the following definition of class `Person`, which options are correct for class `Twist4_5`?

```
class Person {
    String name;
    Person(String name) { this.name = name; }
    public String toString() { return name; }
}
class Twist4_5 {
    public static void main(String args[]) {
        HashSet<Person> set = new HashSet<Person>();
        Person p1 = new Person("Harry");
        Person p2 = new Person("Shreya");
        Person p3 = new Person("Selvan");
        Person p4 = new Person("Shreya");
        set.add(p1);
        set.add(p2);
        set.add(p3);
        set.add(p4);
        for (Person e : set) System.out.println(e);
    }
}
```

a HashSet adds all fours objects, referred to by variables p1, p2, p3, and p4.

b If class Person overrides method hashCode() as follows, only p1 would be added to set:

```
public int hashCode() {
    return 10;
}
```

c If class Person overrides both methods hashCode() and equals() as follows, only p1 would be added to set:

```
public boolean equals(Object o) {
    return true;
}
public int hashCode() {
    return 10;
}
```

d If class Person overrides only method equals() as follows, only p1, p2, and p3 will be added to set:

```
public boolean equals(Object o) {
    if (o instanceof Person) {
        return this.name.equals(((Person)o).name);
    }
    else
        return false;
}
```

The following example shows some of the methods of class HashSet in action:

```
import java.util.*;
class ManipulateHashSet {
     public static void main(String args[]) {
        List<String> list = new ArrayList<String>();      Creates and
        list.add("Shreya");                                populates ArrayList.
        list.add("Selvan");

        HashSet<String> set = new HashSet<String>();      Adds all elements from
        set.add("Harry");                                  list; no duplicate elements.
        set.addAll(list);

        System.out.println(set.contains("Shreya"));        Returns true.
        System.out.println(set.remove("Selvan"));

        for (String e : set) System.out.println(e);        Selvan is removed
    }                                                      from HashSet.
}
```

EXAM TIP Watch out for questions that add `null` to a `HashSet`. A `HashSet` allows storing of only one `null` element. All subsequent calls to storing `null` values are ignored.

Class `HashSet` uses hashing algorithms to store, remove, and retrieve its elements. So it offers constant time performance for these operations, assuming that the hash function disperses its elements properly among its buckets. Covering writing an efficient hash function is beyond the scope of this book. You can find complete books on this topic. Efficient and faster removal, addition, and retrieval of objects have always been a requirement, and many have completed a doctorate on it.

NOTE Access the source code of class `String` from the Java API. Examine how it overrides `hashCode()`, using its individual characters to generate its hash code.

CLASS LINKEDHASHSET

A `LinkedHashSet` offers the benefits of a `HashSet` combined with a `LinkedList`. It maintains a double-linked list running through its entries. As with a `LinkedList`, you can retrieve objects from a `LinkedHashSet` in the order of their insertion. Like a `HashSet`, a `LinkedHashSet` uses hashing to store and retrieve its elements quickly. A `LinkedHashSet` permits `null` values. `LinkedHashSet` can be used to create a copy of a `Set` with the same order as that of the original set.

In the following example code, class `UseLinkedHashSet` creates `LinkedHashSet` of `City`. It uses method `add()` to add individual `City` instances and method `addAll()` to add all objects of the specified collection:

```
class City {
    String name;
    City(String name) {
        this.name = name;
    }
    public String toString() {
        return name;
    }
}
class UseLinkedHashSet {
    public static void main(String args[]) {
        Set<City> route = new LinkedHashSet<>();

        route.add(new City("Seattle"));
        route.add(new City("Copenhagen"));
        route.add(new City("NewDelhi"));

        List<City> extendedRoute = new ArrayList<>();
        extendedRoute.add(new City("Beijing"));
        extendedRoute.add(new City("Tokyo"));

        route.addAll(extendedRoute);

        Iterator<City> iter = route.iterator();
        while (iter.hasNext())
            System.out.println(iter.next());
```

Objects from LinkedAddSet can be retrieved in their insertion order.

List extends Collection.

addAll() accepts Collection object.

Prints "false" because City doesn't override equals().

```
        System.out.println(route.contains(new City("Seattle")));
    }
}
```

The output of the preceding code is:

```
Seattle
Copenhagen
NewDelhi
Beijing
Tokyo
false
```

In the preceding code, note that `addAll()` accepts a `Collection` object. So you can add elements of an `ArrayList` to a `LinkedHashSet`. The order of insertion of objects from `extendedRoute` to `route` is determined by the order of objects returned by `extendedRoute`'s iterator (`ArrayList` objects can be iterated in the order of their insertion). Because you can retrieve objects from a `LinkedHashSet` in the order of their insertion, `iter` iterates the `City` objects in the order of their insertion (as shown in the code output).

EXAM TIP Watch out for exam questions that create a `LinkedHashSet` by using a reference variable of type `List`. A `LinkedHashSet` implements the `Collection` and `Set` interfaces, not `List`.

The next class on the exam is `TreeSet`, which uses a *binary tree* behind the scenes.

Class TreeSet

A `TreeSet` stores all its unique elements in a sorted order. The elements are ordered either on their natural order (achieved by implementing the `Comparable` interface) or by passing a `Comparator`, while instantiating a `TreeSet`. If you fail to specify either of these, `TreeSet` will throw a runtime exception when you try to add an object to it.

Unlike the other `Set` implementations like `HashSet` and `LinkedHashSet`, which use `equals()` to compare objects for equality, a `TreeSet` uses method `compareTo()` (for the `Comparable` interface) or `compare()` (for the `Comparator` interface) to compare objects for equality and their order. As discussed in detail in the next sections on the `Comparator` and `Comparable` interfaces, the implementation of method `compare()` or `compareTo()` should be consistent with method `equals()` of the object instances, which are added to a `TreeSet`. If two object instances are *equal* according to their method `equals()`, but not according to their methods `compare()` or `compareTo()`, the `Set` can exhibit inconsistent behavior.

Constructors of class `TreeSet`

Constructs new, empty tree set, sorted according to natural ordering of its elements

Constructs new tree set containing elements in specified collection, sorted according to natural ordering of elements

Constructs new, empty tree set, sorted according to specified comparator

Constructs new tree set containing same elements and using same ordering as specified sorted set

```
TreeSet()
TreeSet(Collection<? extends E> c)
TreeSet(Comparator<? super E> comparator)
TreeSet(SortedSet<E> s)
```

Behind the scenes, a TreeSet uses a *Black-Red binary tree.* This tree modifies itself as you add more values to it so it has the least number of levels and the values are distributed as evenly as possible. Let's create a TreeSet, using another collection of objects:

```
class TestTreeSet {
    public static void main(String args[]) {
        String[] myNames = {"Shreya", "Harry", "Paul", "Shreya", "Selvan"};
        TreeSet<String> treeSetNames = new
                         TreeSet<String>(Arrays.asList(myNames));
        Iterator it = treeSetNames.descendingIterator();
        while (it.hasNext())
            System.out.println(it.next());
    }
}
```

TreeSet created using List of String values

descendingIterator() returns TreeSet values in descending order.

Prints "Shreya Selvan Paul Harry".

EXAM TIP In the absence of passing a Comparator instance to a TreeSet constructor, the objects that you add to a TreeSet must implement Comparable. In the preceding example, String (which implements Comparable) objects are added to the TreeSet. Watch out for storing objects of wrapper classes, Enum and File in a TreeSet; they all implement Comparable. The natural order of enum constants is the order in which they're declared. StringBuffer and StringBuilder don't implement Comparable.

All the collection classes include constructors to use another collection object to instantiate itself. But depending on how it's implemented, it might not include all the elements from the collection passed to its constructor.

A List allows the addition of duplicate elements, but a Set doesn't. In the preceding example, when you create a TreeSet using a List, which contains duplicate elements, one of the duplicate elements is dropped from the TreeSet. TreeSet also includes iterators to iterate over its values in ascending or descending order.

4.8 Map and its implementations

[4.6] Create and use Map implementations

Unlike the other interfaces from the collections framework, like List and Set, the Map interface doesn't extend the Collection interface. In this section, you'll work with Map and SortedMap interfaces and their implementations like HashMap, LinkedHashMap, and TreeMap.

4.8.1 Map interface

Imagine locking or unlocking the door of your home using a key. This key allows you to restrict access to your home to a key holder. You can compare a Map with a pool of keys, mapped to the values that they can unlock. A key can map to a 0 or a 1 value.

A Map doesn't allow the addition of duplicate keys. Items added to a Map aren't ordered. The retrieval order of items from a Map object isn't guaranteed to be the same as its insertion order. The Map interface declares methods to add or delete a key-value pair or query the existence of a key or value. It also defines methods to retrieve the set of keys, values, and key-value pairs.

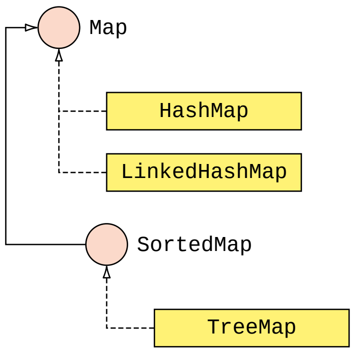


Figure 4.25 The Map interface and its implementations (on the exam)

EXAM TIP Map objects don't allow the addition of duplicate keys.

The addition of a null value as a key or value depends on a particular Map implementation. For example, HashMap and LinkedHashMap allow the insertion of null as a key, but TreeMap doesn't—it throws an exception.

As shown in figure 4.25, the Map implementations on the exam are HashMap, LinkedHashMap, and TreeMap. Let's get started with HashMap.

4.8.2 *HashMap*

A HashMap is a hash-based Map that uses the hash value of its key (returned by hashCode()) to store and retrieve keys and their corresponding values. Each key can refer to a 0 or 1 value. The keys of a HashMap aren't ordered. The HashMap methods aren't synchronized, so they aren't safe to be used in a multithreaded environment.

EXAM TIP The keys of a HashMap aren't ordered. The HashMap methods aren't synchronized, so they aren't safe to be used in a multithreaded environment.

Creating a HashMap and adding values to it

Let's create a HashMap that stores employee names as keys and their salaries as the corresponding values. The following code creates a HashMap with a default initial capacity and adds values to it using method put(Object key, Object value):

```
Map<String, Double> salaryMap = new HashMap<>();
salaryMap.put("Paul", 8888.8);
salaryMap.put("Shreya", 99999.9);
salaryMap.put("Selvan", 5555.5);
```

HashMap defines another constructor (declaration shown below), which accepts a Map object:

```
HashMap(Map<? extends K,? extends V> m)
```

You can use the preceding constructor to create a HashMap by passing it another Map instance:

```
Map<String, Double> salaryMap = new HashMap<>();
Map<String, Object> copySalaryMap = new HashMap<>(salaryMap);
```

The exam might question you on whether the addition or removal of key-value pairs to and from salaryMap will reflect in copySalaryMap. The following example shows that when you delete a key-value pair from salaryMap, it's *not* removed from copySalaryMap:

```
Map<String, Double> salaryMap = new HashMap<>();
salaryMap.put("Paul", 8888.8);
salaryMap.put("Shreya", 99999.9);
Map<String, Object> copySalaryMap = new HashMap(salaryMap);
```

Create copySalaryMap using salaryMap.

```
Set<String> keys = copySalaryMap.keySet();
for (String k : keys)
    System.out.println(k);
```

Outputs two key values.

```
salaryMap.remove("Paul");
```

Remove a key-value pair from salaryMap.

```
keys = copySalaryMap.keySet();
for (String k : keys)
    System.out.println(k);
```

Still outputs two key values.

EXAM TIP You can create a HashMap by passing its constructor another Map object. Additions of new key-value pairs or deletions of existing key-value pairs in the Map object passed to the constructor aren't reflected in the newly created HashMap.

On the exam watch out for the type of the key and value used by the Map object that you pass to the HashMap constructor. The following code won't compile:

```
Map<String, Double> salaryMap = new HashMap<>();
Map<Object, String> copySalaryMap = new HashMap<>(salaryMap);
```

Won't compile

Because a HashMap stores objects as its keys and values, it's common to see code that stores another collection object (like an ArrayList) as a value in a Map (on the exam). For example

```
Map<String, List<Double>> salaryMap = new HashMap<>();
```

When working with generics, note how the type parameters are passed to constructors. You can replace the preceding code with the following:

```
Map<String, List<Double>> salaryMap = new HashMap<String, List<Double>>();
```

But the following are invalid instantiations:

```
Map<String, List<Double>> salaryMap = new HashMap<<>, List<>>();
Map<String, List<Double>> salaryMap = new HashMap<String, List<>>();
Map<String, List<Double>> salaryMap =
                           new HashMap<String, ArrayList<Double>>();
```

Won't compile

Retrieving keys and values

You can call method `get()` on a `HashMap` to retrieve the value for a key. For example

```
enum IceCream {CHOCOLATE, STRAWBERRY, WALNUT};

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

List<IceCream> iceCreamLst = new ArrayList<>();
iceCreamLst.add(IceCream.WALNUT);
iceCreamLst.add(IceCream.CHOCOLATE);

iceCreamMap.put("Shreya", iceCreamLst);
System.out.println(iceCreamMap.get("Shreya"));
```

Prints “[WALNUT, CHOCOLATE]”

On the exam, you might see a code snippet similar to the preceding code, with a difference in type arguments passed to the initialization of `HashMap`. The following code compiles without any warning:

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

Methods `containsKey()` and `containsValue()` check for the existence of a key or a value in a `Map`, returning a `boolean` value. Methods `get()` and `containsKey()` rely on appropriate overriding of key's `hashCode()` and `equals()` methods (discussed in detail in the previous section on `HashSet`). In the following example, class `Emp` doesn't override these methods. Do you think method `get()` will work as expected?

```
class Emp {
    String name;
    String name) {
        this.name = name;
    }
}
Map<Emp, Emp> empMgrMap = new HashMap<>();
empMgrMap.put(new Emp("Shreya"), new Emp("Selvan"));
System.out.println(empMgrMap.get(new Emp("Shreya")));
```

Prints “null”

The preceding code outputs `null`.

EXAM TIP The `String` class and all the wrapper classes override their `hashCode()` and `equals()` methods. So they can be correctly used as keys in a `HashMap`.

Let's see how overriding methods `hashCode()` and `equals()` helps. Here's the modified code, in which class `Emp` overrides methods `hashCode()` and `equals()`:

```
class Emp {
    String name;
    Emp(String name) {
        this.name = name;
    }
    public int hashCode() {
        return name.hashCode();
    }
    public boolean equals(Object o) {
        if (o instanceof Emp)
            return ((Emp)o).name.equals(name);
        else
            return false;
    }
}
class Test {
    public static void main(String args[]) {
        Map<Emp, Emp> empMgrMap = new HashMap<>();
        empMgrMap.put(new Emp("Shreya"), new Emp("Selvan"));
        System.out.println(empMgrMap.get(new Emp("Shreya")));
    }
}
```

EXAM TIP `HashMap` uses hashing functions to add or retrieve key-value pairs. The key must override both methods `equals()` and `hashCode()` so that it can be added to a `HashMap` and retrieved from it.

Do you think methods `containsKey()` and `containsValue()` will work as expected, if class `Emp` overrides only method `equals()` and not method `hashCode()`? Here's the modified code (`Emp` doesn't override `hashCode()`):

```
class Emp {
    String name;
    Emp(String name) {
        this.name = name;
    }
    public boolean equals(Object o) {
        if (o instanceof Emp)
            return ((Emp)o).name.equals(name);
        else
            return false;
    }
}
class Test {
    public static void main(String args[]) {
        Map<Emp, Emp> empMgrMap = new HashMap<>();
        empMgrMap.put(new Emp("Shreya"), new Emp("Selvan"));
        System.out.println(empMgrMap.containsKey(new Emp("Shreya")));      ❶ Prints "false"
        System.out.println(empMgrMap.containsValue(new Emp("Selvan")));    ❷ Prints "true"
    }
}
```

Class `Emp` in the preceding example overrides method `equals()` and not method `hashCode()`. Because method `containsKey()` uses both methods `hashCode()` and `equals()` to determine the equality of keys, the code at ❶ outputs `false`. Because method `containsValue()` uses method `equals()` and not method `hashCode()` to determine the equality of `HashMap` values, the code at ❷ outputs `true`.

EXAM TIP When objects of a class that only overrides method `equals()` and not method `hashCode()` are used as keys in a `HashMap`, `containsKey()` will always return `false`.

Adding duplicate or null keys

What happens if you add a duplicate key to a HashMap? Will the method call be ignored or will its new value replace the key's previous value? The latter is true. At the end of execution of the following code, salaryMap will store 99999.9 as the value for key "Paul".

```
Map<String, Double> salaryMap = new HashMap<>();
salaryMap.put("Paul", 8888.8);
salaryMap.put("Paul", 99999.9);
```

EXAM TIP If you add a key-value pair to a HashMap such that the key already exists in the HashMap, the key's old value will be replaced with the new value.

Do you think you can add a key-value pair to a HashMap with null as the key? The HashMap allows the addition of a maximum of one null key. For example

```
Map<String, Double> salaryMap = new HashMap<>();
salaryMap.put(null, 88.8);
salaryMap.put(null, 99.9);
System.out.println(salaryMap.get(null));
String s = null;
salaryMap.put(null, 77.7);
System.out.println(salaryMap.get(s));
```

Prints "99.9"

Prints "77.7"

EXAM TIP You can add a value with null as the key in a HashMap.

Removing HashMap entries

You can use method remove(key) or clear() to remove one or all key-value pairs of a Map. Method remove() removes the mapping for the specified key from a Map if it is present. It returns the value associated with the key, or null if the key doesn't exist in the map. Method remove() is simple to work with. For example

```
Map<String, Double> salaryMap = new HashMap<>();
salaryMap.put("Paul", 88.8);
System.out.println(salaryMap.remove("Paul"));
```

Removes Paul, 88.8 pair and prints "88.8".

EXAM TIP Method remove() can return a null value, irrespective of whether the specified key exists in a HashMap. It might return null if a matching key isn't present in HashMap, or if null is stored as a value for the specified key.

What happens if you try to remove a key-value pair from a HashMap that uses List as a key? Here's an example:

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

List<IceCream> iceCreamLst = new ArrayList<>();
iceCreamLst.add(IceCream.WALNUT);
iceCreamLst.add(IceCream.CHOCOLATE);
```

```
flavorNameMap.put(iceCreamLst, "Shreya");

List<IceCream> iceCreamLst2 = new ArrayList<>();
iceCreamLst2.add(IceCream.WALNUT);
iceCreamLst2.add(IceCream.CHOCOLATE);

System.out.println(flavorNameMap.remove(iceCreamLst2));
```

Matches key referred by iceCreamLst and removes value "Shreya".

Because the size and order of elements in lists `iceCreamLst` and `iceCreamLst2` are the same, they're considered equal by their `equals()` methods. The `ArrayList` also overrides its `hashCode()`, returning the same hash-code values for equal lists. This enables method `remove()` to find the specified list and remove its corresponding values.

EXAM TIP For a `HashMap`, methods that query or search a key use the key's methods `hashCode()` and `equals()`.

Method `clear()` doesn't accept any method arguments and returns `void`. At the end of execution of the following code, `salaryMap` wouldn't have any key-value pairs:

```
Map<String, Double> salaryMap = new HashMap<>();
salaryMap.put("Paul", 88.8);
salaryMap.put("Shreya", 88.8);
salaryMap.clear();
```

EXAM TIP Method `remove()` removes a maximum of one key-value pair from a `HashMap`. Method `clear()` clears *all* the entries of a `HashMap`. Method `remove()` accepts a method parameter but `clear()` doesn't.

DETERMINING THE SIZE OF HASHMAP

You can use methods `size()` and `isEmpty()` to query a `HashMap`'s size. Method `size()` returns an `int` value representing the count of key-value mappings in a `HashMap`. Method `isEmpty()` returns a boolean value—`true` represents a `HashMap` with no key-value mappings.

COPYING ANOTHER MAP OBJECT

You can use method `putAll()` to copy all the mappings from the specified map to a `HashMap`. What happens if the source and target `HashMap` have the same keys? If the map reference passed to `putAll()` defines keys that already exist in this map, then the values in this map are replaced. For example

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "Shreya");
map.put(11, "Paul");

Map<Integer, String> anotherMap = new HashMap<>();
anotherMap.put(1, "Harry");

anotherMap.putAll(map);
```

For Integer value 1, anotherMap has value "Shreya".

EXAM TIP Method `putAll()` accepts an argument of type `Map`. It copies all the mappings from the specified map to the map that calls `putAll()`. For common keys, the values of the map that calls `putAll()` are replaced with the values of the `Map` object passed to the `putAll()` method.

Retrieving keys, values, and key-value pairs

The `Map` interface defines methods `keySet()`, `values()`, and `entrySet()` to access keys, values, and key-value pairs of a `Map`. The following example shows these methods in action:

```
enum Color {RED, BLUE, YELLOW};

Map<Color, String> colorMap = new HashMap<>();
colorMap.put(Color.RED, "Passion");
colorMap.put(Color.BLUE, "Stability");
colorMap.put(Color.YELLOW, "Energy");

Collection<String> mood = colorMap.values();
Set<Color> colors = colorMap.keySet();
Set<Map.Entry<Color, String>> colorsMood = colorMap.entrySet();

for (String s : mood)
    System.out.println(s);

for (Color c : colors)
    System.out.println(c);

for (Map.Entry pair : colorsMood)
    System.out.println(pair.getKey() + ":" + pair.getValue());
```

Because the order of iteration of a `HashMap` might change, the following is one of the probable outputs of the preceding code:

```
Passion
Energy
Stability
RED
YELLOW
BLUE
RED:Passion
YELLOW:Energy
BLUE:Stability
```

EXAM TIP Method `values()` returns a `Collection` object, method `keySet()` returns a `Set` object, and method `entrySet()` returns a `Map.Entry` object.

Class HashTable legacy code

Class HashTable wasn't a part of the collections framework initially. It was retrofitted to implement the Map interface in Java 2, making it a member of the Java Collection framework. But it's considered legacy code. It's roughly equivalent to a HashMap, with some differences. The operations of a HashMap aren't synchronized, whereas the operations of a HashTable are synchronized. But if you need to work with a HashMap in a multithreaded environment (which needs synchronized methods), you can use class ConcurrentHashMap (covered in chapter 11).

Unlike a HashMap, a HashTable doesn't allow the addition of null keys or values.

4.8.3 LinkedHashMap

The LinkedHashMap IS-A HashMap with a predictable iteration order. Like a LinkedList (covered previously in this chapter), a LinkedHashMap maintains a double-linked list that runs through all its entries. This linked list is used to retrieve the LinkedHashMap elements in the order they were inserted. Like a HashMap, the methods of a LinkedHashMap aren't synchronized.

The following example shows that (unlike a HashMap) a LinkedHashMap would always iterate over its elements in their order of insertion:

```
Map<String, Integer> colorMap = new HashMap<>();
colorMap.put("Red", 1);
colorMap.put("Blue", 2);
colorMap.put("Yellow", 3);
colorMap.put("Purple", 4);
colorMap.put("Orange", 5);

for (Integer i : colorMap.values())
    System.out.print(i);

System.out.println("");

Map<String, Integer> linkedColorMap = new LinkedHashMap<>();
linkedColorMap.put("Red", 1);
linkedColorMap.put("Blue", 2);
linkedColorMap.put("Yellow", 3);
linkedColorMap.put("Purple", 4);
linkedColorMap.put("Orange", 5);

for (Integer i : linkedColorMap.values())
    System.out.print(i);
```

Iteration order of map elements can vary with each execution

Prints "12345"

Here's a probable output of the code:

```
21345
12345
```

NOTE Methods to add, retrieve, remove, or query the elements of a LinkedHashMap work in a similar manner as discussed in the previous section on HashMap.

4.8.4 TreeMap

A TreeMap is sorted according to the natural ordering of its keys or as defined by a Comparator passed to its constructor. It implements the SortedMap interface. Like HashMap and LinkedHashMap, the operations of a TreeMap aren't synchronized, which makes it unsafe to be used in a multithreaded environment.

NOTE The Comparable and Comparator interfaces are discussed in detail in the next section.

Because the key-value pairs of a TreeMap are always sorted, querying a TreeMap (using methods containsKey() and get()) is faster in comparison to querying keys of other unsorted implementations of the Map interface.

In one of the previous sections, you learned how HashMap uses methods hashCode() and equals() of its key to add, remove, or query it. But TreeMap performs all key comparisons by using method compareTo() or compare() of its keys. Two keys are considered equal by a TreeMap if the key's method compareTo() or compare() considers them equal.

Let's get started by creating some TreeMap objects.

CREATING TREEMAP OBJECTS

When you create a TreeMap object, you should specify how its keys should be ordered. A key might define its natural ordering by implementing the Comparable interface. If it doesn't you should pass a Comparator object to specify the key's sort order.

Because this is an important point to note for the exam, I'll cover multiple scenarios here. Let's start with instantiating a TreeMap, which uses enum objects as its keys (enums define their natural order by implementing the Comparable interface):

```
enum IceCream {STRAWBERRY, CHOCOLATE, WALNUT};

Map<IceCream, String> flavorMap = new TreeMap<>();
flavorMap.put(IceCream.CHOCOLATE, "Paul");
flavorMap.put(IceCream.STRAWBERRY, "Shreya");

for (String s : flavorMap.values())
       System.out.println(s);
```

Natural order of enums is their sequence of declaration

The output of the preceding code is:

```
Shreya
Paul
```

In the preceding output, note that IceCream.STRAWBERRY precedes IceCream.CHOCOLATE. The natural order of enum elements is the sequence in which they're defined. The set of *values* that you retrieve from a TreeMap is sorted on its *keys* and not on its *values*.

EXAM TIP The natural order of enum elements is the sequence in which they're defined. The set of *values* that you retrieve from a TreeMap is sorted on its *keys* and not on its *values*.

All the wrapper classes and `String` class implement the `Comparable` interface, so you can use their objects as `TreeMap` keys. Let's see what happens when you use objects of a user-defined class, say, `Flavor`, which doesn't define its natural sort order, as keys to `TreeMap`:

```
class Flavor {
    String name;
    Flavor(String name) {
        this.name = name;
    }
}
class CreateTreeMap {
    public static void main(String args[]) {
        Map<Flavor, String> flavorMap = new TreeMap<>();
        flavorMap.put(new Flavor("Chocolate"), "Paul");
    }
}
```

Flavor class doesn't implement Comparable.

TreeMap instantiation doesn't throw an exception.

Throws ClassCastException.

In the preceding code, note that you can instantiate a `TreeMap` by neither passing it a `Comparator` object, nor using keys that implement the `Comparable` interface. But an attempt to add a key-value pair to such a `TreeMap` will throw a runtime exception.

EXAM TIP You can create a `TreeMap` without passing it a `Comparator` object or without using keys that implement the `Comparable` interface. But adding a key-value pair to such a `TreeMap` will throw a runtime exception, `ClassCastException`.

Now, what happens if the keys used in a `TreeMap` define a natural order and a `Comparator` object is also passed to a `TreeMap` constructor? What happens if the natural order of the keys doesn't match with the order defined by the `Comparator` object? Or, is the natural order of keys ignored if a `Comparator` object is passed to a `TreeMap` object? Let's answer these questions using the next example:

```
class Flavor implements Comparable<Flavor> {
    String name;
    Flavor(String name) {
        this.name = name;
    }
    public int compareTo(Flavor f) {
        return this.name.compareTo(f.name);
    }
}
```

Natural order of Flavor instances is alphabetical order of its names.

```
class MyComparator implements Comparator<Flavor> {
    public int compare(Flavor f1, Flavor f2) {
        return f2.name.compareTo(f1.name);
    }
}
```

MyComparator orders Flavor instances in reverse alphabetical order of its names.

TreeMap creation

```
class CreateTreeMap {
    public static void main(String args[]) {
        Map<Flavor,String> flavorMap = new TreeMap<>(new MyComparator());
        flavorMap.put(new Flavor("Chocolate"), "Paul");
        flavorMap.put(new Flavor("Vanilla"), "Selvan");

        for (Flavor flavor : flavorMap.keySet())
            System.out.println(flavor.name);
    }
}
```

The output of the preceding code is:

```
Vanilla
Chocolate
```

The preceding code shows that when you pass a `Comparator` object to a `TreeMap` constructor, the natural order of its keys is ignored.

EXAM TIP When you pass a `Comparator` object to a `TreeMap` constructor, the natural order of its keys is ignored.

Class `TreeMap` implements the `SortedMap` interface. Watch out for similar code on the exam that tries to instantiate a `SortedMap`. It won't compile. For example

```
Map<String, String> map = new SortedMap<String, String>();   <-- Won't compile
```

Comparing keys: TreeMap versus HashMap

Unlike a `HashMap`, a `TreeMap` uses method `compare()` or `compareTo()` to determine the equality of its keys. In the following example, a `TreeMap` can access the value associated with a key, even though its key doesn't override its method `equals()` or `hashCode()`:

```
class Flavor implements Comparable<Flavor> {
    String name;
    Flavor(String name) {
        this.name = name;
    }
    public int compareTo(Flavor f) {
        return this.name.compareTo(f.name);
    }
}
class CreateTreeMap {
    public static void main(String args[]) {
        Map<Flavor, String> flavorMap = new TreeMap<>();
        flavorMap.put(new Flavor("Chocolate"), "Paul");
        flavorMap.put(new Flavor("Apple"), "Harry");
        System.out.println(flavorMap.get(new Flavor("Apple")));   <-- Prints "Harry"
    }
}
```

In this section on `TreeMap`, you learned how user-defined classes can use the `Comparable` and `Comparator` interfaces to define a natural or custom order of objects. The next section covers these interfaces in detail.

4.9 Using java.util.Comparator and java.lang.Comparable

[4.7] Use java.util.Comparator and java.lang.Comparable

Until now, you have used method `equals()` to compare objects for equality. But when it comes to sorting a collection of objects, you must also compare objects to determine whether an object is less than or greater than another object. To do so, you can use two interfaces: `java.lang.Comparable` and `java.util.Comparator`.

4.9.1 Comparable interface

The `Comparable` interface is used to define the *natural order* of the objects of the class that implements it. It is a generic interface (using `T` as type parameter) and defines only one method, `compareTo(T object)`, which compares the object to the object passed to it as a method parameter. It returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than the specified object. Here's the definition of the `Comparable` interface:

```
package java.lang;
public interface Comparable<T> {            Generic
    public int compareTo(T o);              interface
}
```

Here's an example of class `Person` that implements the `Comparable` interface:

```
class Person implements Comparable<Person> {
    String name;
    int age;                                      Person constructor
                                                  accepts name and age.
    Person (String name, int age) {
        this.name = name;
        this.age = age;
    }
    public int compareTo(Person person) {         Natural order of instances of Person
        return (this.age-person.age);             is based on int value of its age
    }
    public String toString() {            Overridden toString()
        return name;                      to return name
    }
}
```

EXAM TIP The `Comparable` interface is used to define the *natural order* of the objects of the class that implements it.

Some collection classes, like `TreeSet` and `TreeMap`, store their elements in a sorted order. You can specify the sort order of the elements by making their class implement the `Comparable` interface. Here's an example in which `TreeSet` stores instances of class `Person`, which implements `Comparable`:

```
class TestComparable {
    public static void main(String args[]) {
        TreeSet<Person> set = new TreeSet<>();
        set.add(new Person("Shreya", 12));
        set.add(new Person("Harry", 40));
        set.add(new Person("Paul", 30));

        Iterator<Person> iterator = set.iterator();
        while(iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

The `TreeSet` values are returned in ascending order of `age` of class `Person`. Here's the output of the preceding code:

```
Shreya
Paul
Harry
```

EXAM TIP Method `compareTo()` returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than the specified object.

It's important to note that the implementation of method `compareTo()` should be consistent with the implementation of method `equals()`. This rule is recommended, but not required.

For any two object instances `a` and `b`, if `a.compareTo(b)` returns a value `0`, then `a.equals(b)` should return `true`. Let's see what happens if we implement `compareTo()` in an inconsistent manner in class `Person` and add its instances to a `TreeSet` (changes in bold):

```
class Person implements Comparable<Person> {
    String name;
    int age;

    Person (String name, int age) {
        this.name = name;
        this.age = age;
    }
    public int compareTo(Person person) {
        return 0;
    }
    public String toString() {
        return name;
    }
}
class TestComparable {
    public static void main(String args[]) {
        TreeSet<Person> set = new TreeSet<>();

        Person p1 = new Person("Shreya", 12);
        Person p2 = new Person("Harry", 40);
        Person p3 = new Person("Paul", 30);
```

compareTo returns 0.

```
        set.add(p1);
        set.add(p2);
        set.add(p3);

        Iterator<Person> iterator = set.iterator();
        while(iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

p2 and p3 aren't added to set because Set doesn't allow duplicate values.

Prints only one value, Shreya.

Classes like `TreeSet` and `TreeMap` store their elements in a sorted order. Before `set` adds the second element, `p2`, it compares it to the existing element, `p1`. Because `p1.compareTo(p2)` returns `0`, `set` doesn't add the *duplicate element* and returns `false`. The same steps are repeated when `set` tries to add `p3`. At the end, only one element, `p1`, is added to `set`.

QUICK EXERCISE Modify method `compareTo()` in the preceding example so that `TreeSet` returns the values in descending order of `Person`'s `age`.

What if you want to sort the elements of class `Person` based on its instance variable, `name`? Also, can you do this if you can't modify the source code of class `Person`? Yes, it's possible by using the `Comparator` interface, as discussed in the next section.

4.9.2 Comparator interface

The `Comparator` interface is used to define the *sort order* of a collection of objects, without requiring them to implement this interface. This interface defines methods `compare()` and `equals()`. You can pass `Comparator` to sort methods like `Arrays.sort` and `Collections.sort`. It's also passed to collection classes like `TreeSet` and `TreeMap` that require ordered elements.

The `Comparator` interface is used to specify the sort order for classes that

- Don't define a natural sort order
- Need to work with an alternate sort order
- Don't allow modification to their source code so that natural ordering can be added to them

EXAM TIP Unlike the `Comparable` interface, the class whose objects are being compared need not implement the `Comparator` interface.

Here's the source code for this interface:

```
package java.util;
public interface Comparator<T> {
    int compare(T o1, T o2);
    boolean equals(Object obj);
}
```

Like the Comparable interface, method compare() in Comparator returns a negative integer, zero, or a positive integer if o1 is less than, equal to, or greater than o2. Let's modify the example used in the preceding section to use Comparator instead of Comparable:

```
import java.util.*;
class TestComparator {
    public static void main(String args[]) {
        TreeSet<Person> set = new TreeSet<>(
            new Comparator<Person>(){
                public int compare(Person p1, Person p2) {
                    return (p1.age-p2.age);
                }
            }
        );
        set.add(new Person("Shreya", 12));
        set.add( new Person("Harry", 40));
        set.add(new Person("Paul", 30));

        Iterator<Person> iterator = set.iterator();
        while(iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

Class TreeSet is passed an anonymous inner class.

```
class Person {
    String name;
    int age;
    Person (String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String toString() {
        return name;
    }
}
```

Class Person doesn't implement the Comparator interface.

The output of the preceding code is:

```
Shreya
Paul
Harry
```

As you noticed, class Person no longer needs to implement Comparable. Class TreeSet accepts Comparator to define the sort order of its elements. What happens if class Person implements the Comparable interface, which sorts it on name, and the Comparator interface sorts it on age? What do you think is the output of the code in the next "Twist in the Tale" exercise?

Twist in the Tale 4.6

What is the output of the following class?

```
class Twist4_6 {
    public static void main(String args[]) {
        TreeSet<Person> set = new TreeSet<>(new Comparator<Person>(){
            public int compare(Person p1, Person p2) {
                return (p1.age-p2.age);
            }
        });
        Person p1 = new Person("Shreya", 12);
        Person p2 = new Person("Harry", 40);
        Person p3 = new Person("Paul", 30);
        set.add(p1);
        set.add(p2);
        set.add(p3);
        Iterator<Person> iterator = set.iterator();
        while(iterator.hasNext()) {
            System.out.print(iterator.next()+":");
        }
    }
}
class Person implements Comparable<Person>{
    String name;
    int age;
    Person (String name, int age) {
        this.name = name;
        this.age = age;
    }
    public int compareTo(Person person) {
        return name.compareTo(person.name);
    }
    public String toString() { return name; }
}
```

- **a** `Shreya:Paul:Harry:`
- **b** `Harry:Paul:Shreya:`
- **c** `Paul:Shreya:Harry:`
- **d** `Harry:Shreya:Paul:`

Like the `Comparable` interface, the implementation of method `compare()` in `Comparator` should be consistent with the implementation of method `equals()`. For any two object instances `a` and `b`, if `compare(a, b)` returns a value `0`, then `a.equals(b)` should return `true`.

In the next section, let's see why you need sorted data and how to use `Comparable` and `Comparator` to sort and search arrays and lists.

4.10 Sorting and searching arrays and lists

[4.8] Sort and search arrays and lists

How do you view the list of names in a phone directory or the list of selected candidates for admission to a university? Usually, these lists are sorted on their names or on their registration numbers (for university students). Do you think it's easier and faster to find a particular name or a candidate in a sorted list? Yes, it is.

You might need data in a sorted order for multiple reasons: to display information in an ascending or descending order, or to search for particular data. Searching data is always faster in a sorted list. Searching an unsorted list requires comparing *all* list elements with the target element, resulting in a time- and processing-intensive task. In today's world, when we're overwhelmed with data, fast searching and retrieval of data is crucial.

For the exam, you need to know how to search and sort arrays and `List` by using the existing methods from the collections framework classes, `Arrays` and `Collections` to be specific. The OCP Java SE 7 Programmer II exam won't ask you to create or write your own sorting methods. Let's get started with the sorting methods that are accessible using classes `Arrays` and `Collections`.

4.10.1 Sorting arrays

The class `Arrays` in the collections framework defines multiple methods to sort arrays of primitive data types and objects. You can use these methods to sort either a complete array or a part of it. Table 4.3 lists the sorting methods for arrays of `byte`, `int`, and `Object`. The class `Arrays` defines similar methods for other primitive data types: `char`, `short`, `long`, `float`, and `double`. *Please note that I've deliberately excluded them from this list to keep the table short.*

Table 4.3 Class `Arrays` defines sort methods for arrays of `Object` and all primitive data types (excluding type `boolean`)

Method name	Method description
`static void sort(byte[] a)`	Sorts the specified array into ascending numerical order
`static void sort(byte[] a, int fromIndex, int toIndex)`	Sorts the specified range of the array into ascending order
`static void sort(int[] a)`	Sorts the specified array into ascending numerical order
`static void sort(int[] a, int fromIndex, int toIndex)`	Sorts the specified range of the array into ascending order
`static void sort(Object[] a)`	Sorts the specified array of objects into ascending order, according to the natural ordering of its elements

Table 4.3 Class `Arrays` defines sort methods for arrays of `Object` and all primitive data types (excluding type `boolean`) *(continued)*

Method name	Method description
`static void sort(Object[] a, int fromIndex, int toIndex)`	Sorts the specified range of the specified array of objects into ascending order, according to the natural ordering of its elements
`static <T> void sort(T[] a, Comparator<? super T> c)`	Sorts the specified array of objects according to the order induced by the specified comparator
`static <T> void sort(T[] a, int fromIndex, int toIndex, Comparator<? super T> c)`	Sorts the specified range of the specified array of objects according to the order induced by the specified comparator

EXAM TIP All the methods in table 4.3 that sort a partial array accept `fromIndex` and `toIndex` values. The element stored at position `fromIndex` is sorted, but the element stored at position `toIndex` isn't.

Let's look at a simple example of sorting an `int` array:

```
class SortArrays {
    public static void main(String args[]) {
        int[] intArray = {20, 14, 4, 10, 5, 3};
        for (int a:intArray) System.out.print(a + " ");
        Arrays.sort(intArray);
        System.out.println();
        for (int a:intArray) System.out.print(a + " ");

        System.out.println();

        intArray = new int[]{20, 14, 4, 10, 5, 3};
        for (int a:intArray) System.out.print(a + " ");
        Arrays.sort(intArray, 1, 5);
        System.out.println();
        for (int a:intArray) System.out.print(a + " ");
    }
}
```

int array with 6 elements

Sorts all elements of array intArray.

Reinitializes intArray.

Sorts elements at positions 1, 2, 3, and 4, excluding element at position 5.

EXAM TIP A quick reminder that the index of an array is 0-based.

The output of the preceding code is as follows:

```
20 14 4 10 5 3
3 4 5 10 14 20
20 14 4 10 5 3
20 4 5 10 14 3
```

When you sort an array of objects using the `sort` method from class `Arrays`, it uses the natural sort order of the instances. If the objects don't specify a natural sort order, an overloaded version of `sort` can be passed a `Comparator`. A lot of classes like `String` and wrapper classes implement `Comparable` and define a natural sort order. The `String` values are sorted in alphabetical or lexicographic order. On the exam you might be queried about the natural sorting order of `String` values, which differ only in the case of their letters. What do you think is the output of the following sorting operation?

```
String[] strArray = {"ocP", "oCP", "OcP", "OCp", "Ocp"};
for (String str:strArray) System.out.print(str + " ");
Arrays.sort(strArray);
System.out.println();
for (String str:strArray) System.out.print(str + " ");
```

Literal String values that differ only in their case

sort() sorts strArray

Prints OCp OcP Ocp oCP ocP

Each character has a corresponding ASCII or Unicode value. The uppercase letters have a lower ASCII value than their lowercase counterparts.

EXAM TIP Watch out for exam questions that sort string objects starting with a space. A space has a lower ASCII or Unicode value than lowercase or uppercase letters. Let's see how you can use a comparator to sort the objects of a user-defined class:

```
class SortObjects {
   public static void main(String args[]) {
        Person p1 = new Person("Shreya", 32);
        Person p2 = new Person("Harry", 40);
        Person p3 = new Person("Paul", 30);

        Person[] objArray = new Person[]{p1, p2, p3};

        Arrays.sort(objArray,
            new Comparator<Person>(){
                public int compare(Person p1, Person p2) {
                    return (p1.age-p2.age);
                }
            }
        );
        for (Person p:objArray) System.out.print(p + " ");
    }
}
class Person {
    String name;
    int age;
    Person (String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String toString() {
        return name+":"+age;
    }
}
```

sort() is passed array of instances of Person and comparator that defines sort order for Person instances

The preceding code sorts the `Person` instances on the increasing order of their ages, printing this:

```
Paul:30 Shreya:32 Harry:40
```

Imagine what happens if you neither use a `Comparator` nor define a natural ordering for your class. Find out by attempting the next "Twist in the Tale" exercise.

Twist in the Tale 4.7

What is the output of the following class?

```
import java.util.*;
class Twist4_7 {
    public static void main(String args[]) {
        Person p1 = new Person("Shreya", 32);
        Person p2 = new Person("Harry", 40);
        Person p3 = new Person("Paul", 30);

        Person[] objArray = new Person[]{p1, p2, p3};

        Arrays.sort(objArray);

        for (Person p:objArray) System.out.print(p + " ");
    }
}
class Person {
    String name;
    int age;
    Person (String name, int age) {
        this.name = name;
        this.age = age;
    }
    public int compareTo(Person person) {
        return (this.age-person.age);
    }
    public String toString() {
        return name+":"+age;
    }
}
```

- **a** `Shreya:32 Paul:30 Harry:40`
- **b** `Paul:30 Shreya:32 Harry:40`
- **c** `Shreya:32 Harry:40 Paul:30`
- **d** Compilation error
- **e** Runtime exception

4.10.2 Sorting List using Collections

Class `Collections` defines two sorting methods to sort objects of `List`:

Sorts specified list into ascending order, according to natural ordering of elements

```
static <T extends Comparable<? super T>> void sort(List<T> list)
static <T> void sort(List<T> list, Comparator<? super T> c)
```

Sorts specified list according to order induced by specified comparator

Here's an example of sorting a list using method `Collections.sort()`:

```
class SortList {
    public static void main(String args[]) {
        List<Integer> integers = new ArrayList<>();
        integers.add(new Integer(200));
        integers.add(new Integer(87));
        integers.add(new Integer(999));

        for (Integer i : integers) {
            System.out.println(i);
        }

        System.out.println("After calling Collections.sort()");
        Collections.sort(integers);

        for (Integer i : integers) {
            System.out.println(i);
        }
    }
}
```

The output of the preceding code is:

```
200
87
999
After calling Collections.sort()
87
200
999
```

What would happen if we add another item to a list after it was sorted? Will it be sorted too? Let's find out using the next example:

```
class SortList {
    public static void main(String... args) {
        Star s1 = new Star("Sun", 7777.77);
        Star s2 = new Star("Sirius", 999999.99);
        Star s3 = new Star("Pilatim", 222.22);

        List<Star> list = new ArrayList<>();
        list.add(s1); list.add(s2); list.add(s3);
```

Creates new ArrayList, referred by list.

Adds Star instances to list.

```
        Collections.sort(list);
        list.add(new Star("Litmier", 4444.44));
        Collections.reverse(list);

        for (Star star:list) System.out.println(star);
    }
}
class Star implements Comparable<Star> {
    String name;
    double mass;
    Star(String name, double mass) {
        this.name = name;
        this.mass = mass;
    }
    public int compareTo(Star other) {
        return (int)(this.mass - other.mass);
    }
    public String toString(){
        return name + ":" + mass;
    }
}
```

Sorts list.

Adds another Star instance to list; this isn't sorted.

Reverses order of list; doesn't sort in descending order.

Here's the output of the preceding code:

```
Litmier:4444.44
Sirius:999999.99
Sun:7777.77
Pilatim:222.22
```

EXAM TIP Once sorted, new elements are added to a list according to the specific algorithm used by the underlying data structure. After you sort elements of an `ArrayList`, the new elements are added to its end.

4.10.3 *Searching arrays and List using collections*

Classes `Arrays` and `Collections` define method `binarySearch()` to search a sorted array or a `List` for a matching value using the binary search algorithm. The list or array *must* be sorted according to the natural order of its elements or as specified by `Comparator`. If you pass this method an unsorted list, the results are undefined. If more than one value matches the target key value to be searched, this method can return any of these values. Method `binarySearch()` returns the index of the search key, if it is contained in the list; otherwise it returns (-(insertion point) - 1). The insertion point is defined as the point at which the key would be inserted into the list: the index of the first element greater than the key, or `list.size()` if all elements in the list are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

Following is the declaration of the overloaded method `binarySearch()`, which searches the specified array for the specified value using the binary search algorithm:

```
static int binarySearch(byte[] a, byte key)
static int binarySearch(int[] a, int key)
static int binarySearch(Object[] a, Object key)
static <T> int binarySearch(T[] a, T key, Comparator<? super T> c)
```

The preceding list includes the searching methods for the primitive data types `byte` and `int` and objects. I've deliberately not included the overloaded methods for the rest of the primitive data types (`char`, `short`, `long`, `float`, and `double`) to keep it manageable. For example

```
public class SortSearch {
    static final Comparator<Integer> INT_COMPARATOR =
                               new Comparator<Integer>() {
        public int compare (Integer n1, Integer n2) {
            return n2.compareTo(n1);
        }
    };

    public static void main(String args[]) {
        ArrayList<Integer> list = new ArrayList<>();
        list.add(9999);
        list.add(10);
        list.add(55);
        list.add(28);

        Collections.sort(list, null);
        System.out.println(Collections.binarySearch(list, 55));

        Collections.sort(list,INT_COMPARATOR);
        System.out.println(Collections.binarySearch(list, 55));
    }
}
```

The output of the preceding code is

```
2
1
```

Here's the list of the overloaded method `binarySearch()`, which searches a range of the specified array for the specified value by using the binary search algorithm. Again, I've deliberately excluded the overloaded version of these methods for the rest of the primitive data types (`char`, `short`, `long`, `float`, and `double`) to keep the list manageable:

```
static int binarySearch(byte[] a, int fromIndex, int toIndex, byte key)
static int binarySearch(int[] a, int fromIndex, int toIndex, int key)
static int binarySearch(Object[] a, int fromIndex, int toIndex, Object key)
static <T> int binarySearch(T[] a, int fromIndex, int toIndex, T key,
Comparator<? super T> c)
```

Here's the list of methods defined in class `Collections` to search the specified list for the specified object using the binary search algorithm:

```
static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)
static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super
T> c)
```

Similar to method binarySearch(), which accepts List objects, method binarySearch()v that accepts arrays requires the array to be sorted in an ascending order, or else the results are undefined. The output value in the following example is undefined:

```
import java.util.*;
public class SearchArray {
    public static void main(String[] args) {
        Object[] myArray = new Object[3];
        myArray[0] = "Java";
        myArray[1] = "EJava";
        myArray[2] = "Guru";
        int position = Arrays.binarySearch(myArray, "Java");
        System.out.println(position);
    }
}
```

On the exam you might see a question that stores different object types in an array of type Object[]. What do you think is the output of the following code?

```
import java.util.*;
public class SearchArray2 {
    public static void main(String[] args) {
        Object[] myArray = new Object[3];
        myArray[0] = "Java";
        myArray[1] = 10;
        myArray[2] = 'z';
        int position = Arrays.binarySearch(myArray, "Java");
        System.out.println(position);
    }
}
```

The preceding code throws a ClassCastException at runtime when it tries to convert Integer value 10 to String.

4.11 Using wrapper classes

[4.4] Use wrapper classes, autoboxing, and unboxing

Java defines a wrapper class for each of its primitive data types. The wrapper classes are used to *wrap* primitives in an object, so they can be added to a collection object. Wrapper classes help you write cleaner code, which is easy to read. For this exam, you should be able to use these wrapper classes and understand how boxing and unboxing applies to these classes.

4.11.1 Class hierarchy of wrapper classes

All the wrapper classes are immutable. They share multiple usage details and methods. Figure 4.26 shows their hierarchy.

All the numeric wrapper classes extend the class java.lang.Number. Classes Boolean and Character directly extend class Object. All the wrapper classes implement the

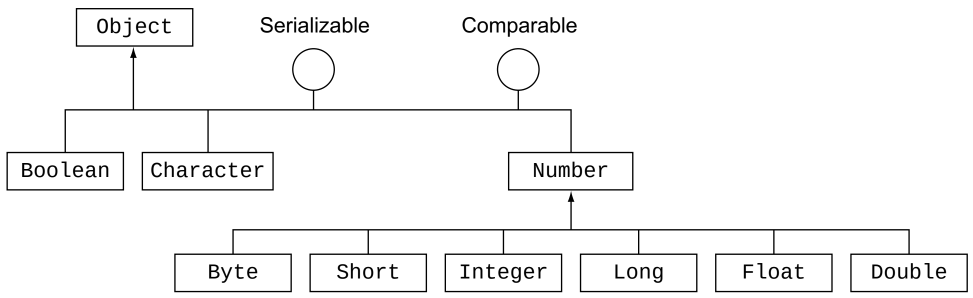


Figure 4.26 Hierarchy of wrapper classes

interfaces `java.io.Serializable` and `java.lang.Comparable`. All these classes can be serialized to a stream, and their objects define a natural sort order.

4.11.2 Creating objects of the wrapper classes

You can create objects of all the wrapper classes in multiple ways:

- *Assignment*—By assigning a primitive to a wrapper class variable
- *Constructor*—By using wrapper class constructors
- *Static methods*—By calling the static method of wrapper classes, like `valueOf()`

For example

```
Boolean bool1 = true;
Character char1 = 'a';
Byte byte1 = 10;
Double double1 = 10.98;
```

Autoboxing

```
Boolean bool2 = new Boolean(true);
Character char2 = new Character('a');
Byte byte2 = new Byte((byte)10);
Double double2 = new Double(10.98);
```

Constructors that accept primitive value

Won't compile

```
//Character char3 = new Character("a");
Boolean bool3 = new Boolean("true");
Byte byte3 = new Byte("10");
Double double3 = new Double("10.98");
```

Constructors that accept String

```
Boolean bool4 = Boolean.valueOf(true);
Boolean bool5 = Boolean.valueOf(true);
Boolean bool6 = Boolean.valueOf("TrUE");
Double double4 = Double.valueOf(10);
```

Using static method valueOf()

You can create objects of the rest of the wrapper classes (`Short`, `Integer`, `Long`, and `Float`) in a similar manner. All the wrapper classes define constructors to create an object using a corresponding primitive value or as a `String`.

Another interesting point to note is that neither of these classes defines a default no-argument constructor. Because wrapper classes are immutable, it doesn't make sense to initialize the wrapper objects with the default primitive values if they can't be modified later.

EXAM TIP All wrapper classes (except `Character`) define a constructor that accepts a `String` argument representing the primitive value that needs to be wrapped. Watch out for exam questions that include a call to a no-argument constructor of a wrapper class. None of these classes defines a no-argument constructor.

You can assign a primitive value directly to a reference variable of its wrapper class type—thanks to *autoboxing*. The reverse is *unboxing*, when an object of a primitive wrapper class is converted to its corresponding primitive value. I'll discuss autoboxing and autounboxing in detail in the next section.

4.11.3 Retrieving primitive values from the wrapper classes

All wrapper classes define methods of the format *`primitive`*`Value()`, where *`primitive`* refers to the exact primitive data type name. Table 4.4 shows a list of the classes and their methods to retrieve corresponding primitive values.

Table 4.4 Methods to retrieve primitive values from wrapper classes

Boolean	Character	Byte, Short, Integer, Long, Float, Double
`booleanValue()`	`charValue()`	`byteValue()`, `shortValue()`, `intValue()`, `longValue()`,`floatValue()`, `doubleValue()`

It's interesting to note that all numeric wrapper classes define methods to retrieve the value of the primitive value they store, as a `byte`, `short`, `int`, `long`, `float`, and `double`.

4.11.4 Parsing a string value to a primitive type

To get a primitive data type value corresponding to a string value, you can use the static utility method `parseDataType()`, where `DataType` refers to the type of the return value. Each wrapper class (except `Character`) defines a method, to parse a `String` to the corresponding primitive value, listed as follows:

Table 4.5 Parsing methods defined by wrapper classes

Class name	Method
`Boolean`	`public static boolean parseBoolean(String s)`
`Character`	no corresponding parsing method
`Byte`	`public static byte parseByte(String s)`
`Short`	`public static short parseShort (String s)`
`Integer`	`public static int parseInt(String s)`

Table 4.5 Parsing methods defined by wrapper classes

Class name	Method
`Long`	`public static long parseLong(String s)`
`Float`	`public static float parseFloat(String s)`
`Double`	`public static double parseDouble(String s)`

All these parsing methods throw a `NumberFormatException` for invalid values. Here are some examples:

```
Long.parseLong("12.34");
```

Throws NumberFormatException: 12.34 isn't valid long.

```
Byte.parseByte("1234");
```

Throws NumberFormatException: 1234 is out of range for byte.

```
Boolean.parseBoolean("true");
```

Returns Boolean true.

```
Boolean.parseBoolean("TrUe");
```

No exceptions; the String argument isn't case-sensitive.

4.11.5 Difference between using method valueOf() and constructors of wrapper classes

Method `valueOf()` returns an object of the corresponding wrapper class when it's passed an argument of a primitive type or `String`. Then what is the difference between method `valueOf()` and constructors of these classes, which also accept method arguments of a primitive type and `String`?

Wrapper classes `Character`, `Byte`, `Short`, `Integer`, and `Long` cache objects with values in the range of –128 to 127. These classes define inner static classes that store objects for the primitive values –128 to 127 in an array. If you request an object of any of these classes, from this range, method `valueOf()` returns a reference to a predefined object; otherwise, it creates a new object and returns its reference:

```
Long var1 = Long.valueOf(123);
Long var2 = Long.valueOf("123");
System.out.println(var1 == var2);
```

Prints "true"; var1 and var2 refer to same cached object.

```
Long var3 = Long.valueOf (223);
Long var4 = Long.valueOf (223);
System.out.println (var3 == var4);
```

Prints "false"; var3 and var4 refer to different objects.

4.11.6 Comparing objects of wrapper classes

The wrapper classes correctly implement methods `hashCode()` and `equals()`, so you can use them in collection framework classes as keys in a map. In the following example, you can use `Double` objects as keys in a `HashMap`:

```
public class UseWrapperAsKeysInMap {
    public static void main(String[] args) {
        Map<Double, String> map = new HashMap<>();
        map.put(6.6, "OCA");
        map.put(7.7, "OCP");

        System.out.println(map.get(6.6));                    Prints "OCA"
        System.out.println(map.get(new Double(7.7)));        Prints "OCP"
    }
}
```

EXAM TIP Integer literal values are implicitly converted to `Integer` objects and decimal literal values are implicitly converted to `Double` objects.

Let's modify the preceding code and try to retrieve the string value "`OCP`" using a `Float` object with value `7.7`. Do you think objects of `Double` and `Float` with the same values are considered equal?

```
public class UseWrapperAsKeysInMap {
    public static void main(String[] args) {
        Map<Double, String> map = new HashMap<>();
        map.put(6.6, "OCA");
        map.put(7.7, "OCP");
        System.out.println(map.get(6.6));                        Outputs 'OCA'
        System.out.println(map.get(new Float((float)7.7)));      Outputs 'null'
    }
}
```

In the preceding code, a `Float` object with a value can't be used to retrieve the value that was added to a `HashMap` using a `Double` instance. Their values don't matter.

EXAM TIP The objects of different wrapper classes with the same values are not equal.

All the wrapper classes also implement the `Comparable` interface. You can compare them using method `compareTo()` and use them in collection framework classes that use natural ordering (like `TreeSet`). Method `compareTo()` returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object. What do you think is the output of the following code that adds `Boolean` instances to a `HashSet`?

```
public class UseTreeSetWithWrapperClasses {
    public static void main(String[] args) {
        TreeSet<Boolean> set = new TreeSet<Boolean>();
        set.add(new Boolean(true));
        set.add(new Boolean("FaLSe"));
        set.add(Boolean.valueOf("TrUe"));
        for (Boolean b : set)
            System.out.println(b);
    }
}
```

The output of the preceding code is

```
false
true
```

EXAM TIP When arranged in natural sort order, `false` precedes `true`.

In the preceding code, you can add `Boolean` instances to a `HashSet` because `Boolean` implements the `Comparable` interface. Because `HashSet` ignores the addition of duplicate values, only one `Boolean.false` object is added to `HashSet`. When instances of a class that doesn't implement `Comparable` are added to a `HashSet`, a `ClassCastException` is thrown at runtime.

The next section covers autoboxing and unboxing, used by a compiler to convert primitive values to wrapper objects and vice versa.

4.12 Autoboxing and unboxing

[4.4] Use wrapper classes, autoboxing, and unboxing

Autoboxing is the automatic conversion of a primitive data type to an object of the corresponding wrapper class (you *box* the primitive value). *Unboxing* is the reverse process (you *unbox* the primitive value), as shown in figure 4.27.

The wrapper classes use autoboxing and unboxing features quite frequently:

```
Double d1 = new Double(12.67);
System.out.println(d1.compareTo(21.68));
```

Prints -1, since 12.67 < 21.68

Compare the use of the preceding method against the following method defined by class `Double`:

```
public int compareTo(Double anotherDouble)
```

Wait—did I just mention that method `compareTo()` defined in the class `Double` accepts an object of class `Double` and not a `double` primitive data type? Then why does the preceding code compile? The answer is autoboxing. Java converted the primitive `double` to an object of class `Double` (by using method `valueOf()`), so it works correctly. The Java compiler converted it to the following at runtime:

```
Double d1 = new Double(12.67D);
System.out.println(d1.compareTo(Double.valueOf(21.68D)));
```

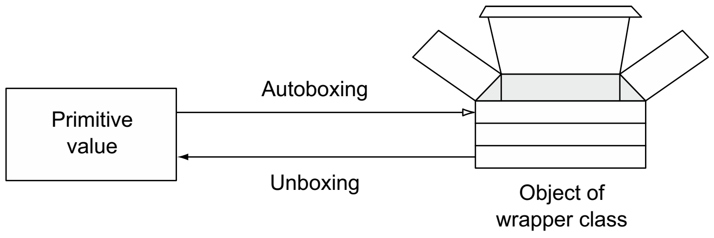


Figure 4.27 Autoboxing and unboxing

Now examine the following code (an example of unboxing with autoboxing):

```
public class Unboxing {
    public static void main (String args[]) {                    List of
        ArrayList<Double> list = new ArrayList<Double>();   <--   Double
        list.add(12.12);
        list.add(11.24);                  Autoboxing-Add double
        Double total = 0;
        for (Double d : list)
            total += d;               <-- Unbox to use operator
    }                                     += with total
}
```

In the preceding code, at the end of execution of the `for` loop, `total` will be assigned a `Double` value of `23.36`. The arithmetic operators like `+=` can't be used with objects. So why do you think the code compiles? In this example, the Java compiler converted the preceding code to the following at runtime:

```
public class Unbox {
    public static void main(String args[]) {
        ArrayList list = new ArrayList();
        list.add(new Double(12.12D));
        list.add(new Double(87.98D));
        Double total = Double.valueOf(0.0D);
        for(Iterator iterator = list.iterator(); iterator.hasNext();)
        {
            Double d = (Double)iterator.next();
            total += total.doubleValue() + d.doubleValue();
        }
    }
```

In the previous section, I mentioned that wrapper classes are immutable. So, what happens when you *add* a value to the variable `total`, a `Double` object? In this case, the variable `total` refers to a *new* `Double` object.

EXAM TIP Wrapper classes are immutable. Adding a primitive value to a wrapper class variable doesn't modify the value of the object it refers to. The wrapper class variable is assigned a new object.

Here's another interesting question. What happens if you pass `null` as an argument to the following method?

```
public int increment(Integer obj) {
    return ++i;
}
```

Because the Java compiler would call `obj.intValue()` to get `obj`'s `int` value, passing `null` to method `increment()` will throw a `NullPointerException`.

EXAM TIP Unboxing a wrapper reference variable, which refers to `null`, will throw a `NullPointerException`.

With the preceding exam tip, you've completed the coverage of generics and collections topics for the exam. With an understanding of all the related nuances under your belt, you'll be able to write better code in real projects. Good luck to you.

4.13 Summary

We started this chapter with a warm-up section on generics, including the need for their introduction and the benefits and complexities of using them. The chapter covered the creation of generic classes, interfaces, and methods. It included how to define and use single and multiple type parameters. You can use bounded type parameters to limit the type of objects that can be passed as arguments to generic classes, interfaces, and methods. We also covered the wildcard `?` to declare a type of a variable or a return type of a method. Bounded wildcards enable you to restrict the types that can be used as arguments in a parameterized type. You learned how type erasure removes the type information during the compilation process so that you get only one class file for each generic class on the interface on compilation. Type erasure also creates bridge methods.

By using type inference, a compiler can determine type arguments if you don't specify them while creating instances of generic types. But if it can't, it'll throw a warning, an error, or an exception. Mixing of raw and generic types was allowed to use code that existed before generics were introduced. If not used correctly, you might get a compiler warning or error, or a runtime exception, when you mix raw with generic types. With generics, you must follow certain subtyping rules. A generic class is a subtype of its raw type. An object of `ArrayList<String>` isn't compatible with a reference variable of type `List<Object>`.

You also worked with the collections framework, an architecture for representing and manipulating collections. You worked with the main interfaces, their implementations, and the algorithms used to manipulate collection objects. The `Collection` interface is extended by the `List`, `Deque`, and `Set` interfaces (but not by the `Map` interface). The `Collection` interface defines methods to manipulate and query its elements.

The `List` interface models an ordered collection of objects. It returns the objects to you in the order in which you added them to a `List`. It allows you to store duplicate elements. The `List` implementations on the exam are `ArrayList` and `LinkedList`.

A `Queue` is a linear collection of objects. A `Deque` is a double-ended queue, a queue that supports the insertion and deletion of elements at both its ends. The `Deque` implementations on the exam are `LinkedList` and `ArrayDeque`.

The `Set` interface models the mathematical `Set` abstraction. It's a collection of objects that doesn't contain duplicate elements. Set implementations on the exam are `HashSet`, `LinkedHashSet`, and `TreeSet`.

A `Map` stores a pool of key-value pairs. It doesn't allow the addition of duplicate keys. Items added to a `Map` aren't ordered. The retrieval order of items from a `Map` object isn't guaranteed to be the same as its insertion order. `Map` implementations on the exam are `HashMap`, `LinkedHashMap`, and `TreeMap`.

Class HashTable wasn't a part of the collections framework initially. It was retrofitted to implement the Map interface in Java 2, making it a member of the Java Collection framework. But it's considered legacy code. It's roughly equivalent to a HashMap, with some differences. The operations of a HashMap aren't synchronized, whereas the operations of a HashTable are synchronized.

The Comparable interface is used to define the natural order of objects. The Comparator interface is used to define the custom order for objects when you don't want to use their natural order, can't define or redefine their natural order, or need a custom order. These interfaces are used by multiple interfaces and classes to sort objects like TreeSet, TreeMap, Collections.sort(), and Arrays.sort(). Classes Arrays and Collections define methods to sort and search arrays and lists.

The wrapper classes are used to wrap primitive types so that they can be used with collection classes. Autoboxing is the automatic conversion of a primitive data type to an object of the corresponding wrapper class (you box the primitive value). Unboxing is the reverse process.

REVIEW NOTES

This section lists the main points covered in this chapter.

Creating generic entities

- You define a generic class, interface, or method by adding one or more type parameters to it.
- A class that uses a generic class uses a parameterized type, replacing the formal parameter with an actual parameter. Also, invalid casts aren't allowed.
- Java's naming conventions limit the use of single uppercase letters for type parameters. Though not recommended, using any valid identifier name for type parameters is acceptable code.
- A generic class can be extended by another generic or nongeneric class.
- An extended class must be able to pass type arguments to its generic base class. If it doesn't, the code won't compile.
- When a nongeneric class extends a generic class, the derived class doesn't define any type parameters but passes arguments to all type parameters of its generic base class.
- A generic interface is defined by including one or more type parameters in its declaration.
- When a nongeneric class implements a generic interface, the type parameters follow the interface name.
- When a generic class implements a generic interface, the type parameters follow both the class and the interface name.
- A generic method defines its own formal type parameters. You can define a generic method in a generic or a nongeneric class.

- To define a generic method in a nongeneric class or interface, you must define the type parameters with the method in its type parameter section.
- A method's type parameter list is placed just after its access and nonaccess modifiers and before its return type. Because a type parameter could be used to define the return type, it should be known before the return type is used.
- You can define a generic method in a generic class or interface, defining its own type parameters.
- You can also define a generic constructor in a generic class.
- You can specify the bounds to restrict the set of types that can be used as type arguments to a generic class, interface, or method. It also enables access to the methods (and variables) defined by the bounds.
- For a bounded type parameter, the bound can be a class, an interface, or an enum, but not an array or a primitive type. All cases use the keyword `extends` to specify the bound. If the bound is an interface, the `implements` keyword isn't used.
- A type parameter can have multiple bounds. The list of bounds consists of one class or multiple interfaces.
- For a type parameter with multiple bounds, the type argument must be a subtype of all bounds.
- The wildcard `?` represents an unknown type. You can use it to declare the type of a parameter; a local, instance, or static variable; and a return value of generic types. But you can't use it as a type argument to invoke a generic method, create a generic class instance, or for a supertype.
- You can assign an instance of a subclass, say, `String`, to a variable of its base class, `Object`. But you can't assign `ArrayList<String>` to a variable of type `List<Object>`. Inheritance doesn't apply to type parameters.
- When you use a wildcard to declare your variables or method parameters, you lose the functionality of adding objects to a collection.
- To restrict the types that can be used as arguments in a parameterized type, you can use bounded wildcards.
- In upper-bounded wildcards, the keyword `extends` is used for both a class and an interface.
- For collections defined using upper-bounded wildcards, you can't add any objects. You can iterate and read values from such collections.
- You can use final classes in upper-bounded wildcards. Although `class X extends String` won't compile, `<? extends String>` will compile successfully.
- You can restrict the use of type arguments to a type and its supertypes or base types by using `<? super Type>`, where `Type` refers to a class, interface, or enum.
- Type information is erased during the compilation process; this is called type erasure.
- When a generic class is compiled, you don't get multiple versions of the compiled class files.

- The compiler erases the type information by replacing all type parameters in generic types with `Object` (for unbounded parameter types) or their bounds (for bounded parameter types).
- The Java compiler might need to create additional methods, referred to as bridge methods, as part of the type erasure process.

Using type inference

- If you don't specify the type of type arguments to instantiate a generic class or invoke a generic method, the Java compiler might be able to infer the argument type by examining the declaration of the generic entity and its invocation. If the type can't be inferred, you might get a compilation warning, an error, or an exception.
- By throwing an unchecked warning, the compiler states that it can't ensure type safety. The term *unchecked* refers to operations that might result in violating type safety. This occurs when the compiler doesn't have enough type information to perform all type checks.
- Starting with Java 7, you can drop the type arguments required to invoke the constructor of a generic class and use a diamond—that is, <>. But an attempt to drop the diamond will result in a compilation warning.
- A Java compiler can't infer the type parameters by using the diamond in the case of generic methods. It uses the type of the actual arguments passed to the method to infer the type parameters.

Understanding interoperability of collections using raw types and generic types

- Raw types exist only for generic types.
- You can assign a parameterized type to its raw type, but the reverse will give a compiler warning.
- When you assign a parameterized type to its raw type, you lose the type information.
- When you mix raw types with generic types, you might get a compiler warning or error or a runtime exception.
- You can assign an object of a subclass to reference a variable of its base class. But this subtyping rule doesn't work when you assign a collection-of-a-derived-class object to a reference variable of a collection of a base class.
- If you declare a reference variable `List<Object>` to a list, whatever you assign to the list must be of generic type `Object`. A subclass of `Object` isn't allowed.

Working with the Collection interface

- The `Collection<E>` interface represents a group of objects known as its elements.
- There's no direct implementation of `Collection`; no concrete class implements it. It's extended by more specific interfaces such as `Set`, `List`, and `Queue`.

- This collection is used for maximum generality—to work with methods that can accept objects of, say, `Set`, `List`, and `Queue`.
- All collection classes are generic.
- The `Map` interface doesn't extend the core `Collection` interface.
- The `Collection` interface implements the `Iterable` interface, which defines method `iterator()`, enabling all the concrete implementations to access an `Iterator<E>` to iterate over all the collection objects.
- The methods of the `Collection` interface aren't marked as synchronized.

Creating and using List, Set, and Deque implementations

- The `List` interface models an ordered collection of objects. It returns the objects to you in the order in which you added them. It allows you to store duplicate elements.
- In a `List` you can control the position where you want to store an element. This is the reason that this interface defines overloaded methods to add, remove, and retrieve elements at a particular position.
- Method `listIterator()` of `List` can be used to iterate the complete list or a part of it.
- An `ArrayList` is a resizable array implementation of the `List` interface.
- An `ArrayList` uses the size variable to keep track of the number of elements inserted in it. By default, an element is added to the first available position in the array. But if you add an element to an earlier location, the rest of the list elements are shifted to the right.
- If you remove an element that isn't the last element in the list, `ArrayList` shifts the elements to the left.
- An `ArrayList` maintains a record of its size so that you can't add elements at arbitrary locations.
- `ArrayList`'s method `remove()` sequentially searches the `ArrayList` to find the target object, using method `equals()` to compare its elements with the target object.
- If a matching element is found, `remove(Object)` removes the first occurrence of the match found.
- If you're adding instances of a user-defined class as elements to an `ArrayList`, override its method `equals()` or else its method `contains()` or `remove()` might not behave as expected.
- The `ArrayList` methods `clear()`, `remove()`, and `removeAll()` offer different functionalities. `clear()` removes all the elements from an `ArrayList`. `remove (Object)` removes the first occurrence of the specified element, and `remove(int)` removes the element at the specified position. `removeAll()` removes from an `ArrayList` all of its elements that are contained in the specified collection.
- A `Deque` is a double-ended queue, a queue that supports the insertion and deletion of elements at both its ends.

- As a double-ended queue, a `Deque` can work as both a queue and a stack.
- The `Deque` interface defines multiple methods to add, remove, and query the existence of elements from both its ends.
- Methods `addFirst()`, `addLast()`, `offerFirst()`, and `offerLast()` add and remove elements from the top and tail.
- `Deque` also defines methods `push()`, `pop()`, and `peek()` to add, remove, and query elements at its beginning.
- `ArrayDeque` and `LinkedList` implement the `Deque` interface.
- `ArrayDeque` is a resizable array implementation of the `Deque` interface.
- `Deque`'s method `peek()` only queries elements, it doesn't remove them.
- `Deque`'s method `remove()` just removes an element.
- `Deque`'s method `poll()` returns `null` when `Deque` is empty and `remove()` throws a runtime exception.
- All the insertion methods (`add()`, `addFirst()`, `addLast()`, `offer()`, `offerFirst()`, `offerLast()`, and `push()`) throw a `NullPointerException` if you try to insert a `null` element into an `ArrayDeque`.
- You can iterate over the elements of `Deque` by using an `Iterator`, returned by methods `iterator()` and `descendingIterator()`.
- Class `LinkedList` implements both the `List` and `Deque` interfaces. So it's a double-linked list implementation of the `List` and `Deque` interfaces.
- Unlike `ArrayDeque`, `LinkedList` permits addition of `null` elements.
- A `LinkedList` is like an `ArrayList` (ordered by index) but the elements are double-linked to each other. So besides the methods from `List`, you get a bunch of other methods to add or remove at the beginning and end of this list. So it's a good choice if you need to implement a queue or a stack. A `LinkedList` is useful when you need fast insertion or deletion, but iteration might be slower than an `ArrayList`.
- Because a `LinkedList` implements `List`, `Queue`, and `Deque`, it implements methods from all these interfaces.
- The `Set` interface models the mathematical `Set` abstraction.
- The `Set` interface doesn't allow duplicate elements and the elements are returned in no particular order.
- To determine the equality of objects, `Set` uses their method `equals()`. For two elements, say `e1` and `e2`, if `e1.equals(e2)` returns `true`, `Set` doesn't add both these elements.
- `Set` defines methods to add and remove its elements. It also defines methods to query itself for the occurrence of specific objects.
- Class `HashSet` implements the `Set` interface. It doesn't allow the addition of duplicate elements and makes no guarantee to the order of retrieval of its elements.
- `HashSet` is implemented using a `HashMap`.

- To store and retrieve its elements, a `HashSet` uses a hashing method, accessing an object's `hashCode()` value to determine the bucket in which it should be stored.
- Method `hashCode()` doesn't call method `equals()`.
- Method `equals()` doesn't call method `hashCode()`.
- Classes should override their `hashCode()` methods efficiently to enable collection classes like `HashSet` to store them in separate buckets.
- A `HashSet` allows storing of only one `null` element. All subsequent calls to storing `null` values are ignored.
- Class `HashSet` uses hashing algorithms to store, remove, and retrieve its elements. So it offers constant time performance for these operations, assuming that the hash function disperses its elements properly among its buckets.
- A `LinkedHashSet` offers the benefits of a `HashSet` combined with a `LinkedList`. It maintains a double-linked list running through its entries.
- As with a `LinkedList`, you can retrieve objects from a `LinkedHashSet` in the order of their insertion.
- Like a `HashSet`, a `LinkedHashSet` uses hashing to store and retrieve its elements quickly.
- A `LinkedHashSet` permits `null` values.
- `LinkedHashSet` can be used to create a copy of a `Set` with the same order as that of the original set.
- `LinkedHashSet`'s method `addAll()` accepts a `Collection` object. So you can add elements of an `ArrayList` to a `LinkedHashSet`. The order of insertion of objects from `ArrayList` to `LinkedHashSet` is determined by the order of objects returned by `ArrayList`'s iterator (`ArrayList` objects can be iterated in the order of their insertion).
- A `TreeSet` stores all its unique elements in a sorted order. The elements are ordered either on their natural order (achieved by implementing the `Comparable` interface) or by passing a `Comparator` while instantiating a `TreeSet`. If you fail to specify either of these, `TreeSet` will throw a runtime exception when you try to add an object to it.
- Unlike the other `Set` implementations like `HashSet` and `LinkedHashSet`, which use `equals()` to compare objects for equality, a `TreeSet` uses method `compareTo()` (for the `Comparable` interface) or `compare()` (for the `Comparator` interface) to compare objects for equality and their order.
- If two object instances are equal according to their method `equals()`, but not according to their method `compare()` or `compareTo()`, a `Set` can exhibit inconsistent behavior.
- Classes `Enum` and `File` implement the `Comparable` interface. The natural order of enum constants is the order in which they're declared. Classes `StringBuffer` and `StringBuilder` don't implement the `Comparable` interface.

Map and its implementations

- Unlike the other interfaces from the collections framework, like List and Set, the Map interface doesn't extend the Collection interface.
- A Map defines key-values pairs, where a key can map to a 0 or 1 value.
- Map objects don't allow the addition of duplicate keys.
- The addition of a null value as a key or value depends on a particular Map implementation. A HashMap and LinkedHashMap allow insertion of null as a key, but TreeMap doesn't—it throws an exception.
- A HashMap is a hash-based Map that uses the hash value of its key (returned by hashCode()) to store and retrieve keys and their corresponding values. Each key can refer to a 0 or 1 value. The keys of a HashMap aren't ordered. The HashMap methods aren't synchronized, so they aren't safe to be used in a multi-threaded environment.
- You can create a HashMap by passing its constructor another Map object. Additions of new key-value pairs or deletions of existing key-value pairs in the Map object passed to the constructor aren't reflected in the newly created HashMap.
- Because a HashMap stores objects as its keys and values, it's common to see code that stores another collection object (like an ArrayList) as a value in a Map.
- You can call method get() on a HashMap to retrieve the value for a key.
- Methods containsKey() and containsValue() check for the existence of a key or a value in a HashMap, returning a boolean value. Methods get() and containsKey() rely on appropriate overriding of a key's methods hashCode() and equals().
- Class String and all the wrapper classes override their methods hashCode() and equals(), so they can be correctly used as keys in a HashMap.
- HashMap uses hashing functions to add or retrieve key-value pairs. The key must override both methods equals() and hashCode() so that it can be added to a HashMap and retrieved from it.
- When objects of a class that only overrides method equals() (and not method hashCode()) are used as keys in a HashMap, containsKey() will always return false.
- If you add a key-value pair to a HashMap such that the key already exists in the HashMap, the key's old value will be replaced with the new value.
- You can add a value with null as the key in a HashMap.
- You can use method remove(key) or clear() to remove one or all key-value pairs of a HashMap.
- Method remove() can return a null value, irrespective of whether the specified key exists in a HashMap. It might return null if matching a key isn't present in HashMap, or if null is stored as a value for the specified key.
- For a HashMap, methods that query or search a key use the key's methods hashCode() and equals().

- Method `remove()` removes a maximum of one key-value pair from a `HashMap`. Method `clear()` clears all the entries of a `HashMap`. Method `remove()` accepts a method parameter but `clear()` doesn't.
- You can use methods `size()` and `isEmpty()` to query a `HashMap`'s size.
- You can use method `putAll()` to copy all the mappings from the specified map to a `HashMap`.
- Method `putAll()` accepts an argument of type `Map`. It copies all the mappings from the specified map to the map that calls `putAll()`. For common keys, the values of map that call `putAll()` are replaced with the values of the `Map` object passed to `putAll()`.
- The `Map` interface defines methods `keySet()`, `values()`, and `entrySet()` to access keys, values, and key-value pairs of a `HashMap`.
- Method `values()` returns a `Collection` object, method `keySet()` returns a `Set` object, and method `entrySet()` returns a `Map.Entry` object.
- Class `HashTable` wasn't a part of the collections framework initially. It was retrofitted to implement the `Map` interface in Java 2, making it a member of the Java Collection framework. But it's considered legacy code. It's roughly equivalent to a `HashMap` with some differences. The operations of a `HashMap` aren't synchronized, whereas the operations of a `HashTable` are synchronized.
- The `LinkedHashMap` IS-A `HashMap` with a predictable iteration order. Like a `LinkedList`, a `LinkedHashMap` maintains a double-linked list, which runs through all its entries.
- A `LinkedHashMap` will always iterate over its elements in their order of insertion.
- A `TreeMap` is sorted according to the natural ordering of its keys or as defined by a `Comparator` passed to its constructor.
- `TreeMap` implements the `SortedMap` interface. Like `HashMap` and `LinkedHashMap`, the operations of a `TreeMap` aren't synchronized, which makes it unsafe to be used in a multithreaded environment.
- The `TreeMap` performs all key comparisons by using method `compareTo()` or `compare()`. Two keys are considered equal by a `TreeMap` if the key's method `compareTo()` or `compare()` considers them equal.
- When you create a `TreeMap` object, you should specify how its keys should be ordered. A key might define its natural ordering by implementing the `Comparable` interface. If it doesn't you should pass a `Comparator` object to specify the key's sort order.
- The set of values that you retrieve from a `TreeMap` is sorted on its keys and not on its values.
- You can create a `TreeMap` without passing it a `Comparator` object or without using keys that implement a `Comparable` interface. But adding key-value pairs to such a `TreeMap` will throw a runtime exception, `ClassCastException`.

- When you pass a Comparator object to TreeMap constructor, the natural order of its keys is ignored.
- Because a TreeMap uses method compare() or compareTo() to determine the equality of its keys, it can access the value associated with a key, even though its key doesn't override its method equals() or hashCode().

Using java.util.Comparator and java.lang.Comparable

- The Comparable interface is used to define the natural order of the objects of the class that implements it.
- Comparable is a generic interface (using T as type parameter) and defines only one method, compareTo(T object), which compares the object to the object passed to it as a method parameter.
- Method compareTo() returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than the specified object.
- The Comparator interface is used to define the sort order of a collection of objects, without requiring them to implement this interface.
- The Comparator interface defines methods compare() and equals().
- You can pass Comparator to sort methods like Arrays.sort() and Collections .sort().
- A Comparator object is also passed to collection classes like TreeSet and TreeMap that require ordered elements.
- The Comparator interface is used to specify the sort order for classes that
  - Don't define a natural sort order
  - Need to work with an alternate sort order
  - Don't allow modification to their source code so that natural ordering can be added to them

Sorting and searching arrays and lists

- Class Arrays in the collections framework defines multiple methods to sort complete or partial arrays of primitive data types and objects.
- When method Arrays.sort() accepts fromIndex and toIndex values to sort a partial array, the element stored at position fromIndex is sorted, but the element stored at position toIndex isn't.
- A space has a lower ASCII or Unicode value than lowercase or uppercase letters. When arranged in an ascending order, a String value that starts with a space is placed before the String values that don't start with a space.
- Class Collections defines method sort() to sort objects of List.
- Classes Arrays and Collections define method binarySearch() to search a sorted array or a List for a matching value using the binary search algorithm. The array or List must be sorted according to the natural order of its elements or as specified by Comparator. If you pass this method an unsorted list, the

results are undefined. If more than one value matches the target key value to be searched, this method can return any of these values.

- Method `binarySearch()` returns the index of the search key if it's contained in the list; otherwise it returns (-(insertion point) - 1). The insertion point is defined as the point at which the key would be inserted into the list: the index of the first element greater than the key, or `list.size()` if all elements in the list are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

Using wrapper classes

- All the wrapper classes are immutable.
- All the wrapper classes implement the `Comparable` interface. All these classes define their natural order.
- You can create objects of all the wrapper classes in multiple ways:
  - *Assignment*—By assigning a primitive to a wrapper class variable
  - *Constructor*—By using wrapper class constructors
  - *Static methods*—By calling the static method of wrapper classes, like `valueOf()`
- All wrapper classes (except `Character`) define a constructor that accepts a `String` argument representing the primitive value that needs to be wrapped. Watch out for exam questions that include a call to a no-argument constructor of a wrapper class. None of these classes defines a no-argument constructor.
- To get a primitive data-type value corresponding to a string value, you can use the static utility method `parseDataType()`, where `DataType` refers to the type of the return value.
- Wrapper classes `Character`, `Byte`, `Short`, `Integer`, and `Long` cache objects with values in the range of –128 to 127. These classes define inner static classes that store objects for the primitive values –128 to 127 in an array. If you request an object of any of these classes, from this range, method `valueOf()` returns a reference to a predefined object; otherwise, it creates a new object and returns its reference.
- Integer literal values are implicitly converted to `Integer` objects and decimal literal values are implicitly converted to `Double` objects.
- The objects of different wrapper classes with the same values aren't equal.
- When arranged in natural sort order, `false` precedes `true`.

Autoboxing and Unboxing

- Autoboxing is the automatic conversion of a primitive data type to an object of the corresponding wrapper class (you box the primitive value). Unboxing is the reverse process (you unbox the primitive value).
- Wrapper classes are immutable. Adding a primitive value to a wrapper class variable doesn't modify the value of the object it refers to. The wrapper class variable is assigned a new object.

- Unboxing a wrapper reference variable, which refers to `null`, will throw a `NullPointerException`.

SAMPLE EXAM QUESTIONS

Q 4-1. Which of the following options creates a generic class that can be passed multiple generic types? (Choose all that apply.)

a
```
class EJavaMap<A , B> {}
```
b
```
class EJavaMap<A a, B b> {}
```
c
```
class EJavaMap<Aa extends String, Bb extends Object> {
    void add(Aa a) {}
    void add(Bb a) {}
}
```
d
```
class EJavaMap<Aa, Bb> {
    void add(Aa a, Bb b) {}
}
```

Q 4-2. Which of the following statements are true about generic classes, interfaces, and methods?

- **a** If you define a generic class, you must define its corresponding raw class explicitly.
- **b** On compilation, type information is erased from a generic class.
- **c** A generic method can be defined within a generic class or a regular class.
- **d** Generic interfaces might not accept multiple generic type parameters.

Q 4-3. Which of the following options when inserted at `//INSERT CODE HERE` would compile successfully without any warning? (Choose all that apply.)

```
class Box<T> {
    T t;
    Box(T t) {
        this.t = t;
    }
    T getValue() {
        return t;
    }
}
class Test {
    public static void main(String args[]) {
    //INSERT CODE HERE
    }
}
```

- **a** `Box box = new Box("abcd");`
- **b** `Box<String> box = new Box<>("String");`
- **c** `Box<String> box = new Box<String>("Object");`
- **d** `Box<Object> box = new Box<String>("String");`

Q 4-4. Consider this pre-generics implementation of method `concat()` in class `MyString`:

```
class MyString {
    public static String concat(List list) {                       //1
        String result = new String();                              //2
        for (Iterator iter = list.iterator(); iter.hasNext(); ) {  //3
            String value = (String)iter.next();                    //4
            result += value;                                       //5
        }
        return result;
    }
}
```

Which three of the following changes together will allow method `concat()` to be used with generics without generating unchecked warnings?

- **a** Replace line 1 with `public static String concat(List<String> list) {`.
- **b** Replace line 1 with `public static String concat(List<Integer> list) {`.
- **c** Remove code on line 3.
- **d** Remove code on line 4.
- **e** Change code on line 3 to `for (String value : list) {`.
- **f** Change code on line 3 to `for (String value : list.listIterator()) {`.

Q 4-5. What happens when you try to compile and execute the following class with Java 7? (Choose all that apply.)

```
class EJava {
    public static void main(String args[]) {
        ArrayList list = new ArrayList();
        list.add("ABCD");
        list.add(1);
        list.add(new Thread());

        for (Object obj:list) System.out.println(obj);
    }
}
```

- **a** Class `EJava` fails to compile with Java 7.
- **b** Class `EJava` compiles with a compilation warning when compiled with Java 7.
- **c** Class `EJava` iterates though all the objects of the list and prints their values as returned by their method `toString()`.
- **d** Class `EJava` prints the first list value and throws a `ClassCastException` while trying to print the second list element.

Q 4-6. What is the output of the following code?

```
import java.util.*;
public class MyHashSet {
    public static void main (String [] args) {
        Set<Phone> set = new TreeSet<>();
        set.add(new Phone("Harry"));
        set.add(new Phone("Paul"));
        set.add(new Phone("Harry"));
        set.add(new Phone("Paul"));

        Iterator <Phone> iterator = set.iterator ();

        while (iterator.hasNext()) {
            Phone ph = iterator.next();
            switch (ph.toString()){
                case "Harry": System.out.print("?Harry? ");
                        break;
                case "Paul": System.out.print("<Paul> ");
                        break;
            }
        }
        System.out.print("Set size=" + set.size());
    }
}
class Phone{
    String manufacturer;
    Phone(String value) {
        manufacturer = value;
    }
    public String toString() {
        return manufacturer;
    }
}
```

- **a** `<Paul> ?Harry? ?Harry? <Paul> Set size=4`
- **b** `<Paul> ?Harry? <Paul> ?Harry? Set size=4`
- **c** `?Harry? ?Harry? <Paul> <Paul> Set size=4`
- **d** `<Paul> ?Harry? Set size=2`
- **e** `?Harry? <Paul> Set size=2`
- **f** Compilation error
- **g** Runtime exception
- **h** The output is unpredictable.

Q 4-7. Given the following code, which code options when inserted at `//INSERT CODE HERE` will sort the keys in `props`?

```
class EMap {
    public static void main(String... args) {
        HashMap props = new HashMap();
        props.put("Harry", "Manth");
        props.put("Paul", "Rosen");
        props.put("Alm", "Bld");
        Set keySet = props.keySet();
        //INSERT CODE HERE
    }
}
```

a `Arrays.sort(keySet);`
b `Collections.sort(keySet);`
c `Collection.sort(keySet);`
d `Collections.arrange(keySet);`
e `keySet = new TreeSet(keySet);`
f `keySet = new SortedSet(keySet);`

Q 4-8. Which code option(s) when inserted at `//INSERT CODE HERE` will make class `EJava` print `Harry`?

```
class EJava {
    public static void main(String args[]) {
            String myArray[] = {"Harry", "Shreya",
                                            "Selvan", "Paul"};
            //INSERT CODE HERE
           System.out.println(myArrayList.get(0));
    }
}
```

a `List <?> myArrayList = new LinkedList<?>(Arrays.asList(myArray));`
b `List <?> myArrayList = new LinkedList<>(Arrays.asList(myArray));`
c `List <? extends String> myArrayList = new LinkedList<>(Arrays.asList (myArray));`
d `List myArrayList = new LinkedList(Arrays.asList(myArray));`
e `List myArrayList = new LinkedList<String>(Arrays.asList(myArray));`

Q 4-9. Which statements are true about method `hashCode()`?

a Method `hashCode()` is used by classes such as `HashMap` to determine inequality of objects.
b Method `hashCode()` is used by classes such as `HashSet` to determine equality of objects.
c Method `hashCode()` is used by class `Collections.sort` to order the elements of a collection.
d Method `hashCode()` is used by classes like `HashSet`, `TreeSet`, and `HashMap`, which use hashing to group their elements into hash buckets.
e An efficient `hashCode()` method includes use of a particular algorithm recommended by Java.

Q 4-10. What is the output of the following code? (Choose all that apply.)

```
import java.util.*;
class MyHash {
    public static void main(String args[]) {
        Person p1 = new Person("Shreya");
        Person p2 = new Person("Harry");
        Person p3 = new Person("Paul");
        Person p4 = new Person("Paul");
        HashSet<Person> set = new HashSet<>();
        set.add(p1);
        set.add(p2);
        set.add(p3);
        set.add(p4);
        System.out.println(set.size());
    }
}
class Person {
    String name;
    Person(String name) {
        this.name = name;
    }
    public int hashCode() {
        return 20;
    }
    public boolean equals(Object obj) {
        return true;
    }
}
```

- a 0
- b 1
- c 2
- d 3
- e 4
- f Compilation error
- g Runtime exception

Q 4-11. Which code option when inserted at `//INSERT CODE HERE` will enable you to sort instances of class `Student` using their natural order and add them to a `TreeSet`?

```
class Student implements Comparator<Student> {
    String id;
    String name;
    //INSERT CODE HERE
}
```

- a `public boolean compare(Object obj1, Object obj2) {/* relevant code here */}`
- b `public int compare(Object obj1, Object obj2) {/* relevant code here */}`
- c `public boolean compareTo(Student s1, Student s2) {/* relevant code here */}`
- d `public boolean compare(Object obj1) {/* relevant code here */}`
- e `public int compare(Student obj1) {/* relevant code here */}`
- f None of the above

Q 4-12. Select true statements about method `hashCode()`.

- **a** Classes `HashSet` and `HashMap` use method `hashCode()` to store and retrieve their values.
- **b** Class `TreeSet` can use method `hashCode()` to store and retrieve its elements.
- **c** Method `hashCode()` is used to test for object equality and inequality for a class.
- **d** If `hashCode()` for two objects of the same class returns the same value, the objects are considered equal.
- **e** For a class, method `hashCode()` can be used to test for object inequality.

ANSWERS TO SAMPLE EXAM QUESTIONS

A 4-1. a, c, d

[4.1] Create a generic class

Explanation: Though Java recommends using single letters like `T` or `V` to specify the type, using the letters `A` and `B` is correct in option (a) as per the syntax.

Option (b) is incorrect because it uses invalid syntax to specify the type parameters to a class. To specify multiple type parameters in a class declaration, you need to specify a placeholder for only the type—not its variables.

Option (c) and (d) are correct. It's acceptable to define the type parameters as a subtype of an existing Java class. Though not recommended, it's acceptable to use type parameters with more than one letter: `Aa` and `Bb`.

A 4-2. b, c

[4.1] Create a generic class

Explanation: Option (a) is incorrect. A raw type doesn't include the generic information. For the generic type `List<T>`, its raw type is `List`. You don't need to define a raw type explicitly for any generic class or interface. You can access the raw type of all the generic types.

Option (d) is incorrect. Like generic classes, generic interfaces can define any number of generic type parameters.

A 4-3. b, c

[4.2] Use the diamond for type inference
[4.3] Analyze the interoperability of collections that use raw types and generic types

Explanation: Option (a) generates a compilation warning, because it uses generic code without its type information.

Options (b) and (c) are correct. The type that you use for declaring a variable of class `Box` is `String`, as in `Box<String> box`. Class `Box` defines only one constructor that accepts an object of its type parameter. Even though you can just use the angle brackets and drop the type parameter `String` from it, you must pass a `String` object to the constructor of class `Box` or a subclass. The following code also compiles without warning (and I pass an instance of a subclass of the generic type parameter into the constructor):

```
Box<Object> box3 = new Box<Object>("Object");
```

Option (d) fails to compile. Even though class `String` subclasses class `Object`, the reference variable `box` of type `Box<Object>` can't refer to objects of `Box<String>`.

A 4-4. a, d, e

[4.3] Analyze the interoperability of collections that use raw types and generic types

Explanation: The options (a), (d), and (e), when implemented together, will allow method `concat()` to be used with generics without generating any warnings.

Option (b) is incorrect. Replacing line 1 with `public static String concat (List<Integer> list) {` would generate a `ClassCastException` at runtime, if a list other than a list of integer objects is passed to method `concat()`.

Option (c) is incorrect because a `for` loop is required to iterate through the list objects.

Options (d) and (e) are correct. With generics, you can use an advanced `for` loop to iterate through list elements. Because the object type is already specified (as `String`), the advanced `for` loop returns `String` objects, which don't require an explicit cast.

Option (f) is incorrect and won't compile.

A 4-5. b, c

[4.3] Analyze the interoperability of collections that use raw types and generic types

Explanation: The code executes, printing all the values of the objects added to the `List`. Method `toString()` is implicitly called when you try to print the value of an object.

Using a raw type of interface is allowed post-introduction of generics. This is allowed for backward compatibility with nongenerics code.

But all uses of the `add` methods with `List`'s raw type will compile with the following compilation warning:

```
warning: [unchecked] unchecked call to add(E) as a member of the raw type
     ArrayList
         list.add("ABCD");
              ^
  where E is a type-variable:
    E extends Object declared in class ArrayList
```

A 4-6. g

[4.5] Create and use List, Set, and Deque implementations

Explanation: The code fails at runtime with the following message because class `Phone` doesn't implement the `java.lang.Comparable` interface:

```
Exception in thread "main" java.lang.ClassCastException: Phone
can't be cast to java.lang.Comparable.
```

This exception is thrown when the code tries to add a value to `set`. A `TreeSet` should be able to sort its elements either by using their natural order or by using a `Comparator` object passed to `TreeSet`'s constructor. A class defines its natural sort order by implementing the `Comparable` interface. Class `Phone` doesn't define its natural order. Also, while instantiating `set`, no `Comparator` object is passed to `TreeSet`'s constructor.

You can instantiate a `TreeSet` that neither uses elements with a natural sort order nor is passed a `Comparator` object. But such a `TreeSet` will throw a runtime exception when you try to add an element to it.

A 4-7. e

[4.6] Create and use Map implementations

Explanation: Option (a) is incorrect because method `sort()` of class `Arrays` sorts only arrays, not `HashMap`.

Option (b) is incorrect. Method `sort()` of class `Collections` sorts `List` objects, not `HashMap`.

Option (c) is incorrect because `Collection` isn't defined in the Java API.

Option (d) is incorrect because method `arrange()` isn't defined in class `Collections`.

Option (f) is incorrect because `SortedSet` is an interface, which can't be instantiated.

A 4-8. b, c, d, e

[4.2] Use the diamond for type inference
[4.3] Analyze the interoperability of collections that use raw types and generic types
[4.5] Create and use List, Set, and Deque implementations

Explanation: Option (a) is incorrect. This code won't compile. When Java runtime instantiates a `LinkedList` object, it must know the type of objects that it stores—either implicitly or explicitly. But, in this option, the type of the `LinkedList` object is neither stated explicitly nor can it be inferred.

Option (b) is correct. The wildcard `?` is used to refer to any type of object. Here you're creating a reference variable `myArrayList`, which is a `List` of any type. This reference variable is initialized with `LinkedList` object, whose type is inferred by the argument passed to the constructor of `LinkedList`.

Option (c) is correct. This option uses the bounded wildcard `<? extends String>`, restricting the unknown type to be either class `String` or any of its subclasses. Even though `String` is a final class, `? extends String` is acceptable code.

Option (d) generates a compiler warning because it uses raw types.

Option (e) doesn't generate a compiler warning because the object creation uses generics.

A 4-9. a

[4.5] Create and use List, Set, and Deque implementations

Explanation: Option (b) is incorrect. Method `equals()`is used to determine the equality of objects.

Option (c) is incorrect. Class `Collections` defines two overloaded versions of method `sort()`. Both accept a `List` object, with or without a `Comparator` object. Method `sort()` sorts a `List` passed to it into ascending order, according to the natural ordering of its elements, or by using the order specified by a `Comparator` object.

Option (d) is incorrect. Though `HashSet` and `HashMap` use `hashCode()` for hashing, `TreeSet` doesn't.

Option (e) is incorrect. Java doesn't recommend any particular algorithm for writing an efficient `hashCode()` method. But Java does recommend writing an efficient algorithm.

A 4-10. b

[4.5] Create and use List, Set, and Deque implementations

Explanation: Method `HashSet()` uses method `hashCode()` to determine an appropriate bucket for its element. If it adds a new element to a bucket that already contains an element, `HashSet` calls `equals` on the elements to determine whether they're equal. `HashSet` doesn't allow duplicate elements. When it adds a `Person` object, the same `hashCode` value makes it land in the same bucket. Calling the `equals()` method returns `true`, signaling that an attempt is being made to add a duplicate object, which isn't allowed by `HashSet`.

A 4-11. f

[4.5] Create and use List, Set, and Deque implementations
[4.7] Use java.util.Comparator and java.lang.Comparable

Explanation: Instances of a class are sorted using its *natural order,* if the class implements the `Comparable` interface and not `Comparator`.

The `Comparator` interface is used to define how to compare two objects for sorting (less than, equal to, or greater than). Unlike the `Comparable` interface, the `Comparator` interface need not be implemented by the class whose objects are to be sorted. The `Comparator` interface can be used to define an order for objects if the objects don't define their natural order. It can also be used to define a custom order for objects. You can use a `Comparator` object to define an order for objects, the natural order of which you can't define or modify. When you pass a `Comparator` object to the instantiation of a collection class like `TreeMap`, the `TreeMap` uses the order as defined by the `Comparator` object, ignoring the natural order of its keys.

For example, the following class defines a custom (descending) order for `String` objects:

```
class DescendingStrings implements Comparator<String> {
    public int compare(String s1, String s2) {
        return s2.name.compareTo(s1.name);
    }
}
```

A 4-12. a, e

[4.5] Create and use List, Set, and Deque implementations
[4.6] Create and use Map implementations

Explanation: In option (a), classes `HashSet` and `HashMap` use hashing to store and retrieve their values. Hashing uses the `hashCode` value to determine the bucket in which the values should be stored.

Option (b) is incorrect. `TreeSet` ignores the `hashCode` values. A `TreeSet` stores its elements based on its key's natural ordering or the ordering defined by a `Comparator`.

Options (c) and (d) are incorrect, and (e) is correct. The `hashCode` value is used to test for object inequality. If two objects return different `hashCode` values, they can never be equal. But if your objects return the same `hashCode` values, they can be unequal (if their `equals()` returns `false`).