Introduction to Programming

Week 12

Magnus Madsen

Week 12: Outline

Generics

- Generic types, classes, and interfaces
- Type parameters and diamond operator
- Generic methods
- Boxing and unboxing

Collections

- List, Set, and Map implementations
- ArrayList vs. LinkedList
- HashSet and HashMap
- Enhanced for-loop
- Iterator and Iterable
- Comparable and Comparator

Quote of the Week

"If someone claims to have the perfect programming language, he is either a fool or a salesman or both."

— Bjarne Stroustrup

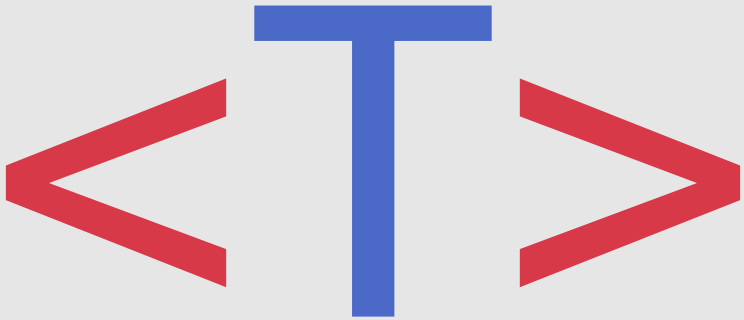
Epigram of the Week

"A language that doesn't affect the way you think about programming, is not worth knowing."

— Alan Perlis

Generics

Mnemonic: Generics



What are Generic Types?

Generics enable **types**, i.e. classes and interfaces, to be **parameters** when defining classes, interfaces, and methods.

Key Concepts:

Abstraction	enables **parametric polymorphism**: *types as arguments*
Code Reuse	write one implementation, use with multiple types
Type Safety	ensures type correctness at compile time

Q: What is type safety?

Motivation: Duplicate Code

A list of fruits:

```
class FruitList {
    private Fruit[] items = new Fruit[10];
    private int size = 0;

    public void add(Fruit fruit) {
        if (size < items.length) {
            items[size] = fruit;
            size++;
        }
    }

    public boolean contains(Fruit fruit) {
        for (int i = 0; i < size; i++) {
            if (items[i].equals(fruit)) {
                return true;
            }
        }
        return false;
    }
}
```

Motivation: Duplicate Code

A list of fruits:

```
class FruitList {
    private Fruit[] items = new Fruit[10];
    private int size = 0;

    public void add(Fruit fruit) {
        if (size < items.length) {
            items[size] = fruit;
            size++;
        }
    }

    public boolean contains(Fruit fruit) {
        for (int i = 0; i < size; i++) {
            if (items[i].equals(fruit)) {
                return true;
            }
        }
        return false;
    }
}
```

A list of students:

```
class StudentList {
    private Student[] items = new Student[10];
    private int size = 0;

    public void add(Student student) {
        if (size < items.length) {
            items[size] = student;
            size++;
        }
    }

    public boolean contains(Student student) {
        for (int i = 0; i < size; i++) {
            if (items[i].equals(student)) {
                return true;
            }
        }
        return false;
    }
}
```

Motivation: A Generic List Class

Instead, we can write **one generic** class:

```
class MyList<T> {
    private T[] items = (T[]) new Object[10];
    private int size = 0;

    public void add(T item) {
        if (size < items.length) {
            items[size] = item;
            size++;
        }
    }

    public boolean contains(T item) {
        for (int i = 0; i < size; i++) {
            if (items[i].equals(item)) {
                return true;
            }
        }
        return false;
    }
}
```

Motivation: A Generic List Class

Instead, we can write **one generic** class:

```
class MyList<T> {
    private T[] items = (T[]) new Object[10];
    private int size = 0;

    public void add(T item) {
        if (size < items.length) {
            items[size] = item;
            size++;
        }
    }

    public boolean contains(T item) {
        for (int i = 0; i < size; i++) {
            if (items[i].equals(item)) {
                return true;
            }
        }
        return false;
    }
}
```

And reuse it as much as we want:

```
MyList<Fruit> fruits = new MyList<>();
fruits.add(new Apple());
fruits.add(new Orange());

MyList<Student> students = new MyList<>();
students.add(new Student("Alice"));
students.add(new Student("Bob"));

MyList<Animal> animals = new MyList<>();
animals.add(new Dog("Rex"));
animals.add(new Cat("Whiskers"));

MyList<String> names = new MyList<>();
names.add("John");
names.add("Mary");

...
```

Creating a Generic Array

Java does not allow direct instantiation of generic arrays due to **type erasure**.

```
T[] arr = new T[10];  // Compile error!
```

Instead, we must create an `Object` array and cast it to `T[]`:

```
T[] arr = (T[]) new Object[10];
```

Note: This produces an **unchecked cast warning**, but c'est la vie.

Remark: This awkwardness stems from generics being a later addition to Java (Java 5).

Type Safety

Generics provide **compile-time type safety** by preventing type mismatches.

Valid: Matching types

```
MyList<Fruit> fruits = new MyList<>();
fruits.add(new Apple());
fruits.add(new Orange());

MyList<Student> students = new MyList<>();
students.add(new Student("Alice"));
students.add(new Student("Bob"));
```

Type Safety

Generics provide **compile-time type safety** by preventing type mismatches.

Valid: Matching types

```
MyList<Fruit> fruits = new MyList<>();
fruits.add(new Apple());
fruits.add(new Orange());

MyList<Student> students = new MyList<>();
students.add(new Student("Alice"));
students.add(new Student("Bob"));
```

Invalid: Mixed types

```
MyList<Fruit> fruits = new MyList<>();
MyList<Student> students = new MyList<>();

fruits.add(new Student("Alice"));
```

```
Error: incompatible types:
Student cannot be converted to Fruit
```

Upshot: We did not lose any type safety when we dropped `FruitList` and `StudentList`.

Generic Classes

A **generic class** is a class that can work with different types through type parameters.

Syntax:

```
class C<T> {
    // Use T throughout the class
}
```

Key points:

- Type parameters are declared in angle brackets `<T>` after the class name.
- Type parameters can be used as **field types**, **parameter types**, and **return types**.
- Multiple type parameters are separated by commas: `<T, U, V>`.

Generic Interfaces

Like classes, interfaces can also be generic.

Syntax:

```
interface I<T> {
    // Use T throughout the interface
}
```

Key points:

- Type parameters are declared in angle brackets `<T>` after the interface name.
- Type parameters can be used as **parameter types** and **return types**.
- Multiple type parameters are separated by commas: `<T, U, V>`.

Diamond operator <> (Java 7+)

The diamond operator allows **type inference** on the right side of assignments.

Before Java 7:

```
ArrayList<String> names = new ArrayList<String>();
HashMap<String, Integer> ages = new HashMap<String, Integer>();
```

Java 7+ with diamond operator:

```
ArrayList<String> names = new ArrayList<>();
HashMap<String, Integer> ages = new HashMap<>();
```

Generic Methods

A **generic method** is a method with its own type parameters, independent of the class.

Syntax:

```
public <T> void m(T param) {
    // Use T in the method
}
```

Key points:

- Type parameters are declared in angle brackets `<T>` before the return type.
- Generic methods can appear in both **generic** and **non-generic** classes.
- Type parameters can be used as **parameter types** and **return types**.

Example: Generic Methods

Without generics, we need separate methods:

```
public void print(Fruit[] arr) {
    for (int i = 0; i < arr.length; i++) {
        System.out.println(arr[i]);
    }
}
```

```
public void print(Student[] arr) {
    for (int i = 0; i < arr.length; i++) {
        System.out.println(arr[i]);
    }
}
```

Example: Generic Methods

Without generics, we need separate methods:

```
public void print(Fruit[] arr) {
    for (int i = 0; i < arr.length; i++) {
        System.out.println(arr[i]);
    }
}
```

```
public void print(Student[] arr) {
    for (int i = 0; i < arr.length; i++) {
        System.out.println(arr[i]);
    }
}
```

With generics, we write **one** method:

```
public <T> void print(T[] arr) {
    for (int i = 0; i < arr.length; i++) {
        System.out.println(arr[i]);
    }
}
```

Example: Calling a Generic Method

Method definition:

```
public class Example {
    public static <T> void m(T param) {
        System.out.println(param);
    }
}
```

With explicit type parameter:

```
Example.<String>m("Hello");
Example.<Fruit>m(new Fruit(...));
Example.<Student>m(new Student(...));
```

Without type parameter (inferred):

```
Example.m("Hello");          // T is String
Example.m(new Fruit(...));   // T is Fruit
Example.m(new Student(...)); // T is Student
```

Mixing Generic Classes and Methods

A **generic class** can have **generic methods** with their own type parameters.

```java
public class Box<T> {
    private T item;

    public Box(T item) {
        this.item = item;
    }

    public T getItem() {
        return item;
    }

    // Generic method with its own type parameter U
    public <U> void printWith(U other) {
        System.out.println(item + " and " + other);
    }
}
```

```java
Box<String> box = new Box<>("A");

box.printWith(42);
// T=String, U=int*
// Output: A and 42

box.printWith("B");
// T=String, U=String
// Output: A and B
```

Naming Conventions for Type Parameters

Java has established **naming conventions** for type parameters:

Letter	Meaning	Common Usage
`T`	Type	Generic classes and methods
`E`	Element	Collections (ArrayList, HashSet)
`K`	Key	Maps (HashMap, TreeMap)
`V`	Value	Maps (HashMap, TreeMap)
`N`	Number	Numeric operations

Recall that Java prefers long PascalCase names, e.g., `BankAccount`, for classes and interfaces.

What can you say about this program fragment?

Q:

```
public class Container<T> {
    private T item;

    public Container(T item) {
        this.item = item;
    }

    public <T> void swap(T newItem) {
        this.item = newItem;
    }
}
```

Boxing and Unboxing

Boxing is the conversion of a primitive type to its corresponding wrapper class

The Problem: Generics work only with **reference types**, not **primitive types**.

```
ArrayList<int> numbers = new ArrayList<>();  // Error! int is primitive
```

The Solution: Java provides **wrapper classes** for each primitive type.

```
ArrayList<Integer> numbers = new ArrayList<>();  // Correct: Using wrapper
```

Wrapper Classes

Java provides a **wrapper class** for each primitive type:

Primitive Type	Wrapper Class
`byte`	`java.lang.Byte`
`short`	`java.lang.Short`
`int`	`java.lang.Integer`
`long`	`java.lang.Long`
`float`	`java.lang.Float`
`double`	`java.lang.Double`
`char`	`java.lang.Character`
`boolean`	`java.lang.Boolean`

We often call a wrapper type **"a box"** since it holds a primitive value inside.

Autoboxing and Unboxing (1/2)

Autoboxing: Java automatically converts between primitive types and their corresponding wrapper classes.

Autoboxing: `int` ⟶ `Integer`

```
ArrayList<Integer> l = new ArrayList<>();

l.add(5);       //  5 auto-boxed to Integer
l.add(10);      // 10 auto-boxed to Integer
l.add(15);      // 15 auto-boxed to Integer
```

Autoboxing and Unboxing (1/2)

Autoboxing: Java automatically converts between primitive types and their corresponding wrapper classes.

Autoboxing: `int` ⟶ `Integer`

```
ArrayList<Integer> l = new ArrayList<>();

l.add(5);      //  5 auto-boxed to Integer
l.add(10);     // 10 auto-boxed to Integer
l.add(15);     // 15 auto-boxed to Integer
```

Unboxing: `Integer` ⟶ `int`

```
int first = l.get(0);     // Unboxed to 5
int second = l.get(1);    // Unboxed to 10
int sum = first + second; // sum is 15
```

Autoboxing and Unboxing (2/2)

Automatic Boxing:

When you write:

```
l.add(5);
```

Java converts this to:

```
l.add(Integer.valueOf(5));
```

Autoboxing and Unboxing (2/2)

Automatic Boxing:

When you write:

```
l.add(5);
```

Java converts this to:

```
l.add(Integer.valueOf(5));
```

Automatic Unboxing:

When you write:

```
int first = l.get(0);
```

Java converts this to:

```
int first = l.get(0).intValue();
```

Remark: This conversion happens automatically, but has a small performance cost.

Q:

What can you say about this program fragment?

```
Integer a = 127;
Integer b = 127;
Integer c = 128;
Integer d = 128;
System.out.println(a == b);
System.out.println(c == d);
```

Inheritance and Generics

Combining Inheritance and Generics

A **generic class** can implement a **generic interface**.

Interface with type parameters:

```
public interface Cache<K, V> {
    void put(K key, V value);
    V get(K key);
    boolean contains(K key);
}
```

Implementation with same type parameters:

```
public class SimpleCache<K, V>
    implements Cache<K, V> {

    public void put(K key, V value) {
        // ... implementation ...
    }

    public V get(K key) {
        // ... implementation ...
    }

    public boolean contains(K key) {
        // ... implementation ...
    }
}
```

Example: Specialized Implementations

We can **specialize** type parameters when implementing a generic interface:

Partial specialization:

```
public class StringCache<V>
    implements Cache<String, V> {

    public void put(String key, V value) {
        // ... implementation ...
    }

    public V get(String key) {
        // ... implementation ...
    }

    public boolean contains(String key) {
        // ... implementation ...
    }
}
```

Full specialization:

```
public class IntCache
    implements Cache<Integer, Integer> {

    public void put(Integer key, Integer
value) {
        // ... implementation ...
    }

    public Integer get(Integer key) {
        // ... implementation ...
    }

    public boolean contains(Integer key) {
        // ... implementation ...
    }
}
```

Generics with Inheritance

We can use a **supertype** as the type parameter and add different subtypes:

```
class Animal {
    String makeSound() { return "..."; }
}

class Dog extends Animal {
    String makeSound() { return "Woof!"; }
}

class Cat extends Animal {
    String makeSound() { return "Meow!"; }
}
```

```
MyList<Animal> animals = new MyList<>();

animals.add(new Dog());
animals.add(new Cat());
animals.add(new Animal());

Animal[] arr = animals.toArray();
for (int i = 0; i < arr.length; i++) {
    System.out.println(arr[i].makeSound());
}
```

This works! We can add any subtype of `Animal` to a `MyList<Animal>`.

Invariance

In Java, generics are **invariant**: `MyList<Dog>` is **NOT** a subtype of `MyList<Animal>`, even though `Dog` is a subtype of `Animal`.

This does NOT work:

```
class Animal { }
class Dog extends Animal { }
class Cat extends Animal { }

MyList<Dog> dogs = new MyList<>();
MyList<Animal> animals = dogs;  // Error!
```

```
Error: incompatible types
```

Invariance

In Java, generics are **invariant**: `MyList<Dog>` is **NOT** a subtype of `MyList<Animal>`, even though `Dog` is a subtype of `Animal`.

This does NOT work:

```
class Animal { }
class Dog extends Animal { }
class Cat extends Animal { }

MyList<Dog> dogs = new MyList<>();
MyList<Animal> animals = dogs;  // Error!
```

```
Error: incompatible types
```

Why? Type safety:

```
// If this worked:
MyList<Dog> dogs = new MyList<>();
MyList<Animal> animals = dogs;

// We could do this:
animals.add(new Cat());

// Now dogs contains a Cat!
Dog d = dogs.get(0);  // Runtime error!
```

Upshot: Java prevents this assignment to maintain type safety.

Q:

What can you say about this program fragment?

```
public static <T> void copy(List<T> src, List<T> dst) {
    // ... copies all elements in src into dst.
}

public static void main(String[] args) {
    List<Animal> animals = new ArrayList<>();
    List<Dog> dogs = new ArrayList<>();
    copy(dogs, animals);
}
```

We have now seen two forms of **polymorphism** in Java:

Subtype Polymorphism

- Based on **inheritance**
- Objects of a subtype can be used where a supertype is expected

We have now seen two forms of **polymorphism** in Java:

Subtype Polymorphism

- Based on **inheritance**
- Objects of a subtype can be used where a supertype is expected

Parametric Polymorphism

- Based on **generics**
- Same code works with different types

Two Types of Polymorphism: Examples

Subtype Polymorphism (Inheritance)

```
class Animal {
    void makeSound() { ... }
}

class Dog extends Animal {
    void makeSound() {
        System.out.println("Woof!");
    }
}

Animal a = new Dog();
a.makeSound();  // Calls Dog's method
```

Two Types of Polymorphism: Examples

Subtype Polymorphism (Inheritance)

```
class Animal {
    void makeSound() { ... }
}

class Dog extends Animal {
    void makeSound() {
        System.out.println("Woof!");
    }
}

Animal a = new Dog();
a.makeSound();  // Calls Dog's method
```

Parametric Polymorphism (Generics)

```
class ArrayList<T> {
    void add(T item) { ... }
    T get(int i) { ... }
}

ArrayList<Dog> dogs = new ArrayList<>();
dogs.add(new Dog());

ArrayList<Cat> cats = new ArrayList<>();
cats.add(new Cat());
// Same ArrayList code for both
```

Live Programming

Live Programming

- The Box Class
- Autoboxing and unboxing (inspect the heap)
- Generic Classes vs. Generic Methods

Collections

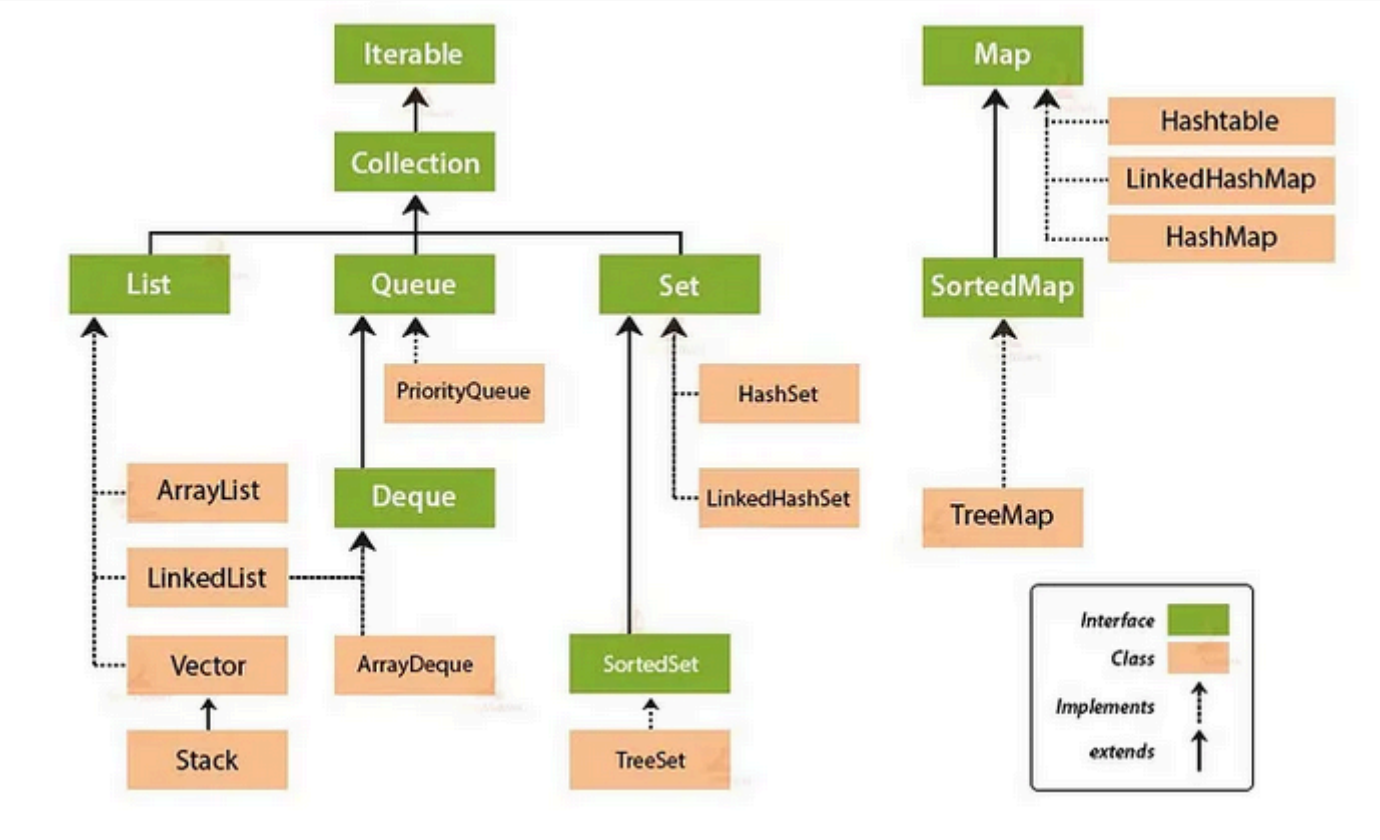
Collections are objects that group multiple elements into a single unit, providing powerful operations for managing data.

Key Benefits:

Rich Operations	search, sort, filter, and transform data
Dynamic Sizing	grow and shrink as needed
Flexibility	more powerful and convenient than arrays
Type Safety	compile-time type checking with generics

Remark: Collections are part of the `java.util` package.

Collection Interfaces



Lists

List Implementations

A **list** is an indexed sequence of elements that allows duplicates.

- **ArrayList**: Resizable array implementation.
  - Fast *random access* – $\mathcal{O}(1)$ for get/set.
  - Slow insertion/deletion in middle – $\mathcal{O}(n)$.
  - Good for frequent access, infrequent modifications.
- **LinkedList**: Doubly-linked list implementation.
  - Fast insertion/deletion at any position – $\mathcal{O}(1)$.
  - Slow random access – $\mathcal{O}(n)$.
  - Good for frequent additions/removals.

Both `ArrayList` and `LinkedList` implement the `List<E>` interface.

Memory Layout

Draw the memory layout for:

- `ArrayList`
- `LinkedList`

Example: ArrayList

We can use an `ArrayList<T>` to store a dynamic list of elements of type `T`:

```
ArrayList<String> fruits = new ArrayList<>();
fruits.add("Apple");
fruits.add("Banana");

System.out.println("Size: " + fruits.size());
fruits.add("Mango");
fruits.add("Strawberry");

String first = fruits.get(0);
System.out.println(first);

fruits.set(0, "Cherry");

for (int i = 0; i < fruits.size(); i++) {
    System.out.println(fruits.get(i));
}
```

Key Points:

- **Dynamic size** – grows and shrinks as needed.
- **Access by index** – use `get(i)` to retrieve element at position `i`.
- **Modify by index** – use `set(i)` to update element at position `i`.
- **Ordered sequence** – maintains insertion order, allows duplicates.

Example: LinkedList

We can use a `LinkedList<T>` when we need to insert or remove from the middle of a list.

```
LinkedList<String> tasks = new LinkedList<>();
tasks.add("Read email");
tasks.add("Write report");
tasks.add("Review code");

// Add high priority task at beginning
tasks.addFirst("Fix critical bug");

// Add task at specific position
tasks.add(2, "Call client");

// Process tasks from beginning
while (!tasks.isEmpty()) {
    String task = tasks.removeFirst();
    System.out.println(task);
}
```

Key Points:

- **Dynamic size** – grows and shrinks as needed.
- **Efficient modifications** – fast insertion/removal at any position.
- **Deque operations** – can act as both stack and queue.
- **Ordered sequence** – maintains insertion order, allows duplicates.

Q:

You are implementing a web browser's history feature that needs to:

- Support back/forward navigation
- Add visited pages to the end of the history
- Remove old pages from the front of the history

Which list would you use?

In real-life, you should always use **ArrayList**.

(You use ArrayList 99% of the time and LinkedList in 1% of special cases.)

ArrayList >>> LinkedList

Why `ArrayList`s are excellent:

Cache locality – elements stored contiguously in memory
- CPU can prefetch adjacent elements
- Fewer cache misses

Memory efficiency – no extra node objects
- ArrayList: just the array + size
- LinkedList: each element needs a node with two pointers

Modern CPU optimization – processors love arrays
- Vectorization and parallelization
- Predictable memory access patterns

Real-world performance – $\mathcal{O}(n)$ shifts are fast
- Most lists are small (< 100 elements)
- Array copying is highly optimized

When LinkedList Wins

Music player with queue and limited history:

```
LinkedList<Song> playQueue = new LinkedList<>();
LinkedList<Song> recentlyPlayed = new LinkedList<>();
int maxRecent = 50;

void playNext() {
    if (!playQueue.isEmpty()) {
        // Dequeue current song
        Song nowPlaying = playQueue.removeFirst();  // O(1)

        // Add to recently played
        recentlyPlayed.addFirst(nowPlaying);        // O(1)

        // Maintain history limit
        if (recentlyPlayed.size() > maxRecent) {
            recentlyPlayed.removeLast();            // O(1)
        }
    }
}
```

Q:

What can you say about this program fragment?

```
ArrayList<int> numbers = new ArrayList<>();
numbers.add(5);
numbers.add(10);
```

Q:

What can you say about this program fragment?

```
ArrayList<Integer> numbers = new ArrayList<>();
numbers.add(null);
int x = numbers.get(0);
System.out.println(x);
```

Sets and Maps

Set Implementations

A **set** is an unordered collection that contains no duplicate elements.

- **HashSet**: Hash table implementation.
  - Fast operations – $\mathcal{O}(1)$ for add/remove/contains.
  - No ordering guarantee.
  - Good for fast membership testing.
- **TreeSet**: Red-black tree implementation.
  - Sorted elements – maintains natural ordering.
  - Slower operations – $\mathcal{O}(\log n)$ for add/remove/contains.
  - Good when you need sorted elements.
- **LinkedHashSet**: Hash table with linked list.
  - Fast operations – $\mathcal{O}(1)$ like HashSet.
  - Maintains insertion order.
  - Good when you need predictable iteration order.

Example: HashSet

We can use a `HashSet<T>` to store unique elements and check membership efficiently:

```java
// Track visited pages
HashSet<String> visited = new HashSet<>();

// Visit some pages
visited.add("home.html");
visited.add("about.html");
visited.add("home.html");
visited.add("contact.html");

// Check if page was visited
if (visited.contains("about.html")) {
    System.out.println("Already visited");
}
```

Key Points:

- **No duplicates** – automatically ensures uniqueness.
- **Fast lookups** – `contains()` runs in $\mathcal{O}(1)$ time.
- **Unordered** – no guarantee on iteration order.

Map Implementations

A **map** is an object that maps keys to values, with no duplicate keys allowed.

- **HashMap**: Hash table implementation.
  - Fast operations – $\mathcal{O}(1)$ for get/put/remove.
  - No ordering guarantee.
  - Good for fast key-value lookups.
- **TreeMap**: Red-black tree implementation.
  - Sorted by keys – maintains natural ordering.
  - Slower operations – $\mathcal{O}(\log n)$ for get/put/remove.
  - Good when you need sorted keys.

Example: HashMap

We can use a `HashMap<K,V>` to store key-value pairs for fast lookups:

```
// Student grades database
HashMap<String, Integer> grades = new HashMap<>();

// Store grades
grades.put("Alice", 95);
grades.put("Bob", 87);
grades.put("Charlie", 92);
grades.put("Bob", 89);

// Retrieve a grade
Integer bobGrade = grades.get("Bob");
System.out.println("Bob: " + bobGrade);

// Check if student exists
if (grades.containsKey("Alice")) {
    Integer grade = grades.get("Alice");
    System.out.println("Alice: " + grade);
}
```

Key Points:

- **Key-value pairs** – maps unique keys to values.
- **Fast access** – `get()` runs in $\mathcal{O}(1)$ time.
- **Updates allowed** – `put()` replaces existing values.

Example: Word Frequency

We can use a `HashMap<K,V>` to count word occurrences in text:

```
public static void main(String[] args) {
    HashMap<String, Integer> wordCount = new HashMap<>();

    for (int i = 0; i < args.length; i++) {
        String word = args[i];
        if (wordCount.containsKey(word)) {
            // Increment existing count
            Integer count = wordCount.get(word);
            wordCount.put(word, count + 1);
        } else {
            // First occurrence
            wordCount.put(word, 1);
        }
    }

    System.out.println("'the': " + wordCount.get("the"));
    System.out.println("'cat': " + wordCount.get("cat"));
}
```

Q:

Which of these assignments are valid?

```
class Shape { }
class Circle extends Shape { }
class Rectangle extends Shape { }

List<Shape> l1         = new ArrayList<Shape>();     // (1)
LinkedList<Shape> l2   = new LinkedList<Circle>();   // (2)
List<Shape> l3         = new ArrayList<Circle>();    // (3)
List<Circle> l4        = new LinkedList<Circle>();   // (4)
ArrayList<Shape> l5    = new ArrayList<Circle>();    // (5)
List<Circle> l6        = new ArrayList<Shape>();     // (6)
```

Enhanced For-Loop

Enhanced For Loop (For-Each)

The **enhanced for loop** provides a simple way to iterate through collections without explicit indices or iterators.

```
for (Type element : collection) {
    // Use element
}
```

Key Benefits:

More Readable	intent is clear: process each element
Cleaner Code	no index variables or iterators needed
Less Error-Prone	no off-by-one errors
Universal	works with any Iterable: arrays, lists, sets, etc.

Example: Enhanced For-Loop (1/2)

We can use a for-each loop to iterate through an array:

```
int[] numbers = {10, 20, 30, 40, 50};
for (int num : numbers) {
    System.out.println(num);
}
```

We can use a for-each loop to iterate through an ArrayList:

```
ArrayList<String> fruits = new ArrayList<>();
fruits.add("Apple");
fruits.add("Banana");
for (String fruit : fruits) {
    System.out.println(fruit);
}
```

Example: Enhanced For-Loop (2/2)

We can use a for-each loop to iterate through a HashSet:

```
HashSet<String> cities = new HashSet<>();
cities.add("Paris");
cities.add("London");
for (String city : cities) {
    System.out.println(city);
}
```

We can use a for-each loop to iterate through HashMap entries:

```
HashMap<String, Integer> ages = new HashMap<>();
ages.put("Alice", 25);
ages.put("Bob", 30);
for (Map.Entry<String, Integer> entry : ages.entrySet()) {
    System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

Q:

What can you say about this program fragment?

```
HashSet<String> animals = new HashSet<>();
animals.add("Bear");
animals.add("Cat");
animals.add("Dog");
animals.add("Eagle");
animals.add("Fox");

for (String animal : animals) {
    System.out.println(animal);
}
```

Q:

What can you say about this program fragment?

```
ArrayList<String> l = new ArrayList<>();
l.add("A");
l.add("B");
l.add("C");
for (String s : l) {
    if (s.equals("B")) {
        l.remove(s);
    }
}
```

Iterator and Iterable

What is an Iterator?

An **iterator** is an object that provides a standard way to traverse through a collection, visiting each element once.

Key Characteristics:

Sequential Access	visit elements one at a time in order
Implementation-Agnostic	same interface for ArrayList, LinkedList, HashSet, etc.
Safe Modification	allows removing elements during iteration without errors

Iterator Interface

The **Iterator** interface allows *traversing* through a collection:

```
public interface Iterator<E> {
    boolean hasNext();  // Returns true if more elements exist
    E next();           // Returns next element and advances
    void remove();      // Removes last returned element
}
```

Example: Using Iterator

We can use an iterator to traverse through a collection:

```
ArrayList<String> planets = new ArrayList<>();
planets.add("Mercury");
planets.add("Venus");
planets.add("Earth");
planets.add("Mars");

Iterator<String> it = planets.iterator();
while (it.hasNext()) {
    String planet = it.next();
    System.out.println("Planet: " + planet);
}
```

Example: Safe Removal with Iterator

We can use iterators for *safe removal* during iteration:

```
ArrayList<String> inbox = new ArrayList<>();
inbox.add("Meeting at 3pm");
inbox.add("WIN FREE MONEY NOW!!!");
inbox.add("Project update");
inbox.add("CLICK HERE FOR PRIZES");
inbox.add("Lunch tomorrow?");

Iterator<String> it = inbox.iterator();
while (it.hasNext()) {
    String email = it.next();
    if (email.contains("!!!") || email.contains("CLICK HERE")) {
        it.remove();  // Remove spam
    }
}
```

Iterable Interface

The **Iterable** interface enables the *enhanced for loop*:

```
public interface Iterable<E> {
    Iterator<E> iterator();  // Returns an iterator over elements
}
```

Key points:

- Any class implementing `Iterable<E>` works with for-each loops
- All Java collections implement this interface
- Compiler uses `iterator()` method behind the scenes

Example: How For-Each Works

What you write:

```
ArrayList<String> colors = new ArrayList<>();
colors.add("Red");
colors.add("Green");
colors.add("Blue");

for (String color : colors) {
    System.out.println(color);
}
```

What the compiler generates:

```
ArrayList<String> colors = new ArrayList<>();
colors.add("Red");
colors.add("Green");
colors.add("Blue");

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

Example: Implementing Iterable

We can make our own class work with for-each by implementing `Iterable`:

```
class PokerHand implements Iterable<Card> {
    private Card card1;
    private Card card2;
    private Card card3;
    private Card card4;
    private Card card5;

    public Iterator<Card> iterator() {
        Card[] cards = {card1, card2, card3, card4, card5};
        return Arrays.asList(cards).iterator();
    }
}
```

Comparable and Comparator

Motivation: Comparable and Comparator

When working with collections of objects, we often need to *sort* them in a meaningful order.

The challenge:

- Java can sort numbers and strings, but what about custom objects like Student or Product?
- Different situations require different orderings – by name, by price, by date
- We need a way to tell Java *how* to compare our objects

Two approaches:

- **Comparable** – objects know how to compare themselves (natural ordering)
- **Comparator** – external objects that compare two items (custom ordering)
- Use Comparable for the *one obvious way* to sort (e.g., students by ID)
- Use Comparator for *alternative ways* to sort (e.g., by grade or by name)

The Comparable Interface

The `Comparable` interface allows an object to define its **natural ordering** by comparing itself to another object of the same type.

```
public interface Comparable<T> {
    // Returns a negative number if this < other
    // Returns zero if this == other
    // Returns a positive number if this > other
    int compareTo(T other);
}
```

Example: Implementing Comparable (1/2)

We can implement `Comparable` to define **natural ordering** by age (eldest first), then last name, then first name:

```java
public class Person implements Comparable<Person> {
    private String firstName;
    private String lastName;
    private int age;

    public int compareTo(Person other) {
        // First compare by age (eldest first)
        int ageCmp = other.age - this.age;
        if (ageCmp != 0) return ageCmp;

        // If ages are equal, compare by last name
        int lastCmp = this.lastName.compareTo(other.lastName);
        if (lastCmp != 0) return lastCmp;

        // If last names are equal, compare by first name
        return this.firstName.compareTo(other.firstName);
    }
}
```

Example: Implementing Comparable (2/2)

We can create a list of people and sort them using their **natural ordering**:

```
ArrayList<Person> people = new ArrayList<>();
people.add(new Person("Bill", "Gates", 69));
people.add(new Person("Taylor", "Swift", 34));
people.add(new Person("Elon", "Musk", 53));
people.add(new Person("Oprah", "Winfrey", 70));
people.add(new Person("Mark", "Zuckerberg", 40));
people.add(new Person("Bill", "Clinton", 78));
Collections.sort(people);

for (Person p : people) {
    System.out.println(p.firstName + " " + p.lastName);
}
```

Prints: Bill Clinton, Oprah Winfrey, Bill Gates, Elon Musk, Mark Zuckerberg, Taylor Swift.

The Comparator Interface

The `Comparator` interface allows us to define **custom orderings** by comparing two objects from the outside.

```
public interface Comparator<T> {
    // Returns a negative number if o1 < o2
    // Returns zero if o1 == o2
    // Returns a positive number if o1 > o2
    int compare(T o1, T o2);
}
```

Key differences from Comparable:

- **External comparison** – comparator is a separate object, not part of the class
- **Multiple orderings** – can create different comparators for different sort orders
- **No modification needed** – can sort classes you don't control

Example: Comparator (1/2)

We can define a comparator to sort by last name, then first name:

```
class PersonByLastNameAndFirstName implements Comparator<Person> {
    public int compare(Person p1, Person p2) {
        int lastCmp = p1.lastName.compareTo(p2.lastName);
        if (lastCmp != 0) return lastCmp;
        return p1.firstName.compareTo(p2.firstName);
    }
}
```

Usage:

```
Collections.sort(people, new PersonByLastNameAndFirstName());
```

Prints: Bill Clinton, Bill Gates, Elon Musk, Taylor Swift, Oprah Winfrey, Mark Zuckerberg.

Example: Comparator (2/2)

We can define another comparator to sort by age (youngest first), then first name:

```
class PersonByAgeAndFirstName implements Comparator<Person> {
    public int compare(Person p1, Person p2) {
        int ageCmp = p1.age - p2.age;
        if (ageCmp != 0) return ageCmp;
        return p1.firstName.compareTo(p2.firstName);
    }
}
```

Usage:

```
Collections.sort(people, new PersonByAgeAndFirstName());
```

Prints: Taylor Swift, Mark Zuckerberg, Elon Musk, Bill Gates, Oprah Winfrey, Bill Clinton.

Epilogue

Error of the Week

What is the problem here?

```
public class Box<T> {
    private T[] items = new T[10];
    private int size = 0;

    public void add(T item) {
        if (size < items.length) {
            items[size] = item;
            size++;
        }
    }
}
```

Error of the Week

What is the problem here?

```
public class Box<T> {
    private T[] items = new T[10];
    private int size = 0;

    public void add(T item) {
        if (size < items.length) {
            items[size] = item;
            size++;
        }
    }
}
```

```
Box.java:2: error: generic array creation
    private T[] items = new T[10];
                        ^
1 error
```

Live Programming

Live Programming

- Generics (classes and methods)
- Equality and hash code
- Mutable objects as keys
- Iterator and Iterable
- Comparable and Comparator
- Invariance
- Mutable Hash Keys