# Homework 06 – Choo Choo!

Topics: generics, ADTs, Iterator, Iterable

## Problem Description

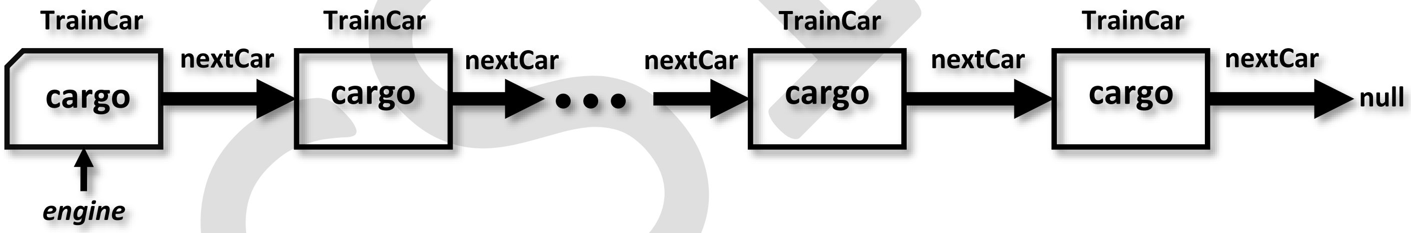
Break is right around the corner, and you've been planning a vacation for a while. You decide that you're going to travel via train, and you quickly realize that a train behaves similarly to a linked list! Using what you've learned about the List ADT, you decide to implement it as a Train.

## Solution Description

We highly recommend that you **read this document in its entirety** and familiarize yourself with the requirements of this assignment **before writing any code**!

**Important**: All classes created in this assignment should be **generic classes**. All instances of generic classes should specify a concrete type and should never use raw types. If a method returns a reference type, it must use generic types, and should never use raw types.

A train is an ordered sequence of train cars. Each individual train car stores cargo and a reference to the next train car in the sequence. The first train car is called the engine and a reference to it should be kept. When there are no train cars attached to the train, the engine should be *null.*



We have provided you with a file named `List.java`. This file is an interface called `List` that you will implement in Train.java. It represents the List ADT and extends from the `Iterable` interface. You will create three classes: `TrainCar.java`, `TrainIterator.java`, and `Train.java`.

**Note:**

1. All variables should be inaccessible from other classes and must require an instance to be accessed through, unless specified otherwise.
2. All methods should be accessible from everywhere and must require an instance to be accessed through, unless specified otherwise.
3. Reuse code when possible and private helper methods are optional.
4. When throwing exceptions, provide descriptive and specific messages.
5. Add Javadoc comments to all public methods and classes, including in provided files where Javadoc comments are missing.

**Tip from your TAs:** There are a lot of edge cases to consider when completing this assignment. Make sure to perform extensive testing in a Driver file! We encourage sharing your tests and outputs in the Discussion Thread on Ed Discussion.

## *List.java* (Provided File)

This generic interface defines the List ADT and describes the methods that the `Train` class must implement. The Javadocs in this file will contain more information about each of the methods you must implement, including what the method does, what should be returned, and when to throw exceptions.

This interface extends from the `Iterable` interface, so the `iterator()` method must also be implemented. Refer to the Java documentation on **Iterable** for more information about this interface.

**Methods:**

***In addition to*** the method(s) required by the `Iterable` interface, the `List` interface requires the following methods:

- `void add(E element)`
- `void add(int index, E element)`
- `E remove()`
- `E remove(int index)`
- `E remove(E element)`
- `E set(int index, E element)`
- `E get(int index)`
- `boolean contains(E element)`
- `void clear()`
- `boolean isEmpty()`
- `int size()`

**Note**: You do not need to implement the `forEach()` and `spliterator()` methods of the `Iterable` interface for this assignment.

## *TrainCar.java*

This generic class represents an individual train car that will be part of a train. It should be able to carry cargo of any type.
**Note:** You **may not** create any instance fields nor instance methods that are not required.

**Type Parameter**:

- `T` – the type of cargo this train car carries

**Variables:**

- `cargo` – the cargo this train car carries
  - If we ever attempt to set its value to null, throw an `IllegalArgumentException`.
- `nextCar` – a reference to the next train car, which should also carry the same type of cargo as *this* train car

**Constructors:**

- A constructor that takes in `cargo` and `nextCar`.
- A constructor that takes in `cargo` and sets `nextCar` to null.

**Methods:**

- Getters and setters for each of the variables.

## *TrainIterator.java*

This is a generic class that represents an iterator to iterate over a `Train`. It implements the `Iterator` interface using generics. Refer to the Java documentation on **Iterator** for more about this interface.

For `Train` to successfully implement the `Iterable` interface (which the provided `List` interface extends from), you must provide an implementation for the `iterator` method. The `iterator` method returns an instance of an `Iterator` that knows how to iterate over the Iterable to retrieve its elements.

To successfully implement the `iterator` method, we will create our own iterator, `TrainIterator`. A TrainIterator should iterate over every TrainCar in a Train, and in the same order as the elements.

**Type Parameter**:
- `T` – the type of cargo carried by the train that this iterator iterates over

**Variables**:
- `nextCar` – the next train car to iterate over
  - ***Hint***: When a `TrainIterator` is first instantiated, `nextCar` should store a reference to the first train car in the train (the engine of the train).

**Constructor**:
- A constructor that takes in the `Train` that this iterator will iterate over.
  - If the train passed in is null, throw an `IllegalArgumentException`.

**Methods**:
- `hasNext`
  - This method should override the `hasNext` method required by `Iterator`.
  - Returns a boolean specifying whether there are more train cars to iterate over.
    - ***Hint***: What value of `nextCar` indicates the existence of a train car?
- `next`
  - This method should override the `next` method required by `Iterator`.
  - Returns the `cargo` in the next train car and advances the iterator to the next train car.
    - **Note**: The return type of this method should **not** be `Train`, but instead the type of `cargo` in the train car.
  - ***Hint***: The first time `next` is called, it should return the cargo in the first train car of the train and advance the iterator to the second train car. The second time `next` is called, it should return the cargo in the second train car of the train and advance the iterator to the third person in line, and so on.
  - If there are no more train cars to iterate over, throw a `NoSuchElementException`.

**Functionality**:

- After properly implementing `TrainIterator`, you should be able to use the iterator to print out all the cargo in a `Train` using this block of code. Assume that `train` is an instance of `Train`, and that `T` is replaced with a concrete type.

```
Iterator<T> itr = train.iterator();
while (itr.hasNext()) {
    System.out.println(itr.next());
}
```

## *Train.java*

This is a generic class that represents a train. A `Train` is comprised of `TrainCar` objects that are *linked* together. This class implements all the methods required by the provided `List` interface (don't forget about the methods required by the `Iterable` interface) and utilizes the `TrainCar` and `TrainIterator` classes to do so.

**Note:** Train.java is a provided file with fields and method stubs. Download it and complete it by providing an implementation for each method. The `toString` method has already been implemented to help you visualize the train.

**Type Parameter**:
- `T` – the type of cargo this train carries

**Variables**:
- `engine` – a reference to the first train car of the train
  - ***Hint*:** Think about which part of a linked list this refers to.
- `size` – the number of cargo this Train carries
  - Don't forget to update its value when a train car is added to or removed from the train!

**Constructors**:
- A constructor that takes in an array of cargo of type `T`.
  - The order of the cargo in this train must be in the same order as they were in the array.
  - If the array passed in is null or any elements of the array is null, throw an `IllegalArgumentException`.
- A no-argument constructor that initializes an empty `Train`.

**Methods**:
***In addition to*** *the methods required by the List interface*, implement the following method:

- `toArray`
  - Return the cargo that this train carries in an array.
  - You must implement this method by explicitly using an iterator (i.e., by explicitly utilizing `Iterator`'s `hasNext` and `next` methods) Do not use a for-each loop.
  - The order of the cargo in the array must be in the same order as the cargo on this train.
  - **Note:** The intuitive approach of creating a generic array in Java by writing `T[] arr = new T[size];` causes a compiling error. Instead, create an Object array and cast that to a generic array: `T[] arr = (T[]) new Object[size];`
  - Note that this cast is not checked by the compiler, so it will display a warning (not an error) to notify you of an unchecked or unsafe operation. It is acceptable if the only warnings you receive are related to performing this unchecked cast.

- You can recompile and use the `-Xlint` flag before the file name to see more specific details about the warnings.

## Checkstyle

You must run Checkstyle on your submission (To learn more about Checkstyle, check out cs1331-style-guide.pdf under the Checkstyle Resources module on Canvas.) **The Checkstyle cap for this assignment is 50 points.** This means there is a maximum point deduction of 50. If you don't have Checkstyle yet, download it from Canvas → Modules → Checkstyle Resources → checkstyle-8.28.jar. Place it in the same folder as the files you want to run Checkstyle on. Run Checkstyle on your code like so:

```
$ java -jar checkstyle-8.28.jar yourFileName.java
Starting audit...
Audit done.
```

The message above means there were no Checkstyle errors. If you had any errors, they would show up above this message, and the number at the end would be the points we would take off (limited by the Checkstyle cap). In future homeworks we will be increasing this cap, so get into the habit of fixing these style errors early!

Additionally, you must Javadoc your code.

Run the following to only check your Javadocs:

```
$ java -jar checkstyle-8.28.jar -j yourFileName.java
```

Run the following to check both Javadocs and Checkstyle:

```
$ java -jar checkstyle-8.28.jar -a yourFileName.java
```

For additional help with Checkstyle see the CS 1331 Style Guide.

## Turn-In Procedure

### *Submission*

To submit, upload the files listed below to the corresponding assignment on Gradescope:

- `TrainCar.java`
- `TrainIterator.java`
- `Train.java`

Make sure you see the message stating the assignment was submitted successfully. From this point, Gradescope will run a basic autograder on your submission as discussed in the next section. **Any autograder test are provided as a courtesy to help "sanity check" your work and you may not see all the test cases used to grade your work.** You are responsible for thoroughly testing your submission on your own to ensure you have fulfilled the requirements of this assignment. If you have questions about the requirements given, reach out to a TA or Professor via the class forum for clarification.

You can submit as many times as you want before the deadline, so feel free to resubmit as you make substantial progress on the homework. We will only grade your latest submission. **Be sure to submit every file each time you resubmit**.

## *Gradescope Autograder*

If an autograder is enabled for this assignment, you may be able to see the results of a few basic test cases on your code. Typically, tests will correspond to a rubric item, and the score returned represents the performance of your code on those rubric items only. If you fail a test, you can look at the output to determine what went wrong and resubmit once you have fixed the issue.

The Gradescope tests serve two main purposes:

- Prevent upload mistakes (e.g., non-compiling code)
- Provide basic formatting and usage validation

In other words, the test cases on Gradescope are by no means comprehensive. Be sure to thoroughly test your code by considering edge cases and writing your own test files. You also should avoid using Gradescope to compile, run, or Checkstyle your code; you can do that locally on your machine.

Other portions of your assignment can also be graded by a TA once the submission deadline has passed, so the output on Gradescope may not necessarily reflect your grade for the assignment.

### Allowed Imports

To prevent trivialization of the assignment, you are only allowed to import the following:

- `java.util.Iterator`
- `java.util.NoSuchElementException`

### Feature Restrictions

There are a few features and methods in Java that overly simplify the concepts we are trying to teach or break our auto grader. For that reason, do not use any of the following in your final submission:

- `var` (the reserved keyword)
- `System.exit`
- `System.arraycopy`

### Collaboration

Only discussion of the assignment at a conceptual high level is allowed. You can discuss course concepts and assignments broadly, that is, at a conceptual level to increase your understanding. If you find yourself dropping to a level where specific Java code is being discussed, that is going too far. Those discussions should be reserved for the instructor and TAs. To be clear, you should never exchange code related to an assignment with anyone other than the instructor and TAs.

You **MAY NOT** use code generation tools to complete this assignment. This includes generative AI tools like ChatGPT.

### Important Notes (Don't Skip)

- Non-compiling files will receive a 0 for all associated rubric items

- Do not submit `.class` files
- Test your code in addition to the basic checks on Gradescope
- Submit every file each time you resubmit
- Read "Allowed Imports" and "Feature Restrictions" to avoid losing points
- **Check on Ed Discussion for a note containing all official clarifications and sample outputs**

It is expected that everyone will follow the Student-Faculty Expectations document, and the Student Code of Conduct. The professor expects a **positive, respectful, and engaged academic environment** inside the classroom, outside the classroom, in all electronic communications, on all file submissions, and on any document submitted throughout the duration of the course. **<u>No inappropriate language is to be used</u>, and any assignment, deemed by the professor, to contain inappropriate, offensive language or threats will get a zero**. You are to use professionalism in your work. Violations of this conduct policy will be turned over to the Office of Student Integrity for misconduct.