Lecture 5: Generics and Collections

Learning Outcomes

- Be able to understand the difference between concrete class, abstract class, and interface with default methods
- Appreciate why generics is helpful
- Be able to create a generic class with one or more type parameters, and instantiate the parameterized type(s) by passing in type argument(s).
- Understand the subtype relationships between generic classes
- Understand type erasure, generic methods, wildcard types, bounded wildcard types.
- Familiar with wrapper classes with primitives and autoboxing/unboxing; when to use primitive types and when to use wrapper classes
- Familiar with Java collection frameworks: `Set` , `List` , `Map` and their concrete class `HashSet` , `LinkedList` , `ArrayList` , and `HashMap` .
- Aware of the other classes in Java Collection and is comfortable to look them up by reading the Java documentation.
- Understand there are differences between the collection classes and know when to use which one

Abstract Class and Interface with Default Methods

We have seen how a class can inherit from a parent class, and implement one or more interfaces. So far, the parent class that we have seen is a *concrete* class -- it has fields and methods, complete with method implementation. Such concrete parent class can be instantiated into objects.

On the other hand, we have *pure* interfaces, which is completely virtual or abstract. It declares what public methods it should provide -- together with, for each method, the return type, the exception(s) it throws, and its method signature. It can only have constant fields [1] and no implementation.

Between these two extremes, there are two other possibilities in Java:

- An *abstract class*, which is just like a class, but it is declared as `abstract` , and some of its methods are declared as `abstract` , without implementation. An abstract class cannot be instantiated, and any subclass who wish to be concrete needs to implement these abstract methods.

```
abstract class PaintedShape {
  Color fillColor;
    :
  void fillWith(Color c) {
    fillColor = c;
  }
    :
  abstract double getArea();
  abstract double getPerimeter();
    :
}
```

- An interface with default implementations. Introduced only in Java 8, with the goal of allowing an interface to evolve, an interface can now contain default implementation of its methods. Such interface still cannot be instantiated into objects, but classes that implement such interface need not provide an implementation for a method where a default implementation exists. For instance, we can have:

```
interface Shape {
  public double getArea();
  public double getPerimeter();
  public boolean contains(Point p);
  default public boolean cover(Point p) {
    return contains(p);
  }
}
```

  where `cover` is a new method with default implementation, denoted with keyword `default` .

The reason Java 8 introduced default methods for interface is for backward compatibility. For instance, the implementer of `Shape` can add `cover` to a new version of interface `Shape` without breaking all the existing code that implements `Shape` .

Abstract class, however, should be used in general in the design of a class hierarchy -- in a case where it is more meaningful for the subclass to implement its own method than the superclass. For example, in the `PaintedShape` abstract class above, it can implement the methods related to styles, but it is not meaningful to provide an implementation of `getArea` and `getPerimeter` . We will see more examples of abstract class later.

Generics

Now let's move on to the topic of generics. Suppose you want to create a new class that encapsulates a queue of circles. You wrote:

```
class CircleQueue {
  private Circle[] circles;
   :
  public CircleQueue(int size) {...}
  public boolean isFull() {...}
  public boolean isEmpty() {...}
  public void enqueue(Circle c) {...}
  public Circle dequeue() {...}
}
```

Later, you found that you need a new class that encapsulates a queue of points. You wrote:

```
class PointQueue {
  private Point[] points;
   :
  public PointQueue(int size) {...}
  public boolean isFull() {...}
  public boolean isEmpty() {...}
  public void enqueue(Point p) {...}
  public Point dequeue() {...}
}
```

And you realize that there are actually a lot of similar code. Invoking the *abstraction principle*, which states that *"Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts*", you decided to create a queue of Objects to replace the two classes above.

```java
class ObjectQueue {
  private Object[] objects;
   :
  public ObjectQueue(int size) {...}
  public boolean isFull() {...}
  public boolean isEmpty() {...}
  public void enqueue(Object o) {...}
  public Object dequeue() {...}
}
```

Now you have a very general class, that you can use to store objects of any kind, including a queue of strings, a queue of colors, etc. You are quite pleased with yourself, as you should! The early Java collection library contains many such generic data structures that stores elements of type `Object`.

To create a queue of 10 circles and add some circles, you just need:

```java
ObjectQueue cq = new ObjectQueue(10);
cq.enqueue(new Circle(new Point(0, 0), 10));
cq.enqueue(new Circle(new Point(1, 1), 5));
 :
```

Getting a circle out of the queue is a bit more troublesome:

```java
Circle c = cq.dequeue();
```

Would generate a compilation error, since we are trying to perform a narrowing reference conversion -- we cannot assign a variable of type `Object` to a variable of type `Circle` without type casting:

```java
Circle c = (Circle)cq.dequeue();
```

As we have seen in Lecture 4 [../lec04/index.html], the code above might generate a runtime `ClassCastException` if there is an object in the queue that is not `Circle` or its subclass. To avoid runtime error, we should check the type first:

```java
Object o = cq.dequeue();
if (o instanceof Circle) {
    Circle c = (Circle)o;
}
```

Wouldn't it be nice if we can still have code that is general, but we let the compiler generates an error if we try to add a non- `Circle` into our queue of `Circle` objects, so that we don't have to check for the type of an object all the time?

Java 5 introduces generics, which is a significant improvement to the type systems in Java. It allows a *generic class* or a *generic interface* of some type `T` to be written:

```java
class Queue<T> {
  private T[] objects;
   :
  public Queue(int size) {...}
  public boolean isFull() {...}
  public boolean isEmpty() {...}
  public void enqueue(T o) {...}
  public T dequeue() {...}
}
```

`T` is known as *type parameter*. The same code as before can be written as:

```java
1 Queue<Circle> cq = new Queue<Circle>(10);
2 cq.enqueue(new Circle(new Point(0, 0), 10));
3 cq.enqueue(new Circle(new Point(1, 1), 5));
4 Circle c = cq.dequeue();
```

Here, we pass `Circle` as the *type argument* to `T`, creating a *parameterized type* `Queue<Circle>`.

In Line 4, we no longer need to cast, and there is no danger of runtime error due to an object of the wrong class being added to the queue, for doing this:

```java
Queue<Circle> cq = new Queue<Circle>(10);
cq.enqueue(new Point(1, 3));
```

will generate a compile-time error.

Generic typing is a type of polymorphism as well and is also known as *parametric polymorphism.*

> **In Other Languages**
> Many modern programming languages support generics: C++ (known as *template*), Swift, Kotlin, C#, etc. For dynamically typed languages such as Python, one can already write generic code without regards to the type of the variables, so generic is kinda "build in". In C (the 2011 version, C11), there is a `_Generic` expression, which is not to be confused with generics we are talking about here.

We can use parameterized type anywhere a type is used, including as type argument. If we want to have a queue of a queue of circles, we can:

```java
Queue<Queue<Circle>> cqq = new Queue<>(10);
```

Variance of Generic Types

In Lecture 4 [../lec04/index.html], we discuss about subtypes and the variance of types. Recall that T is a subtype of S if everywhere we use S, we can substitute with T.

Suppose a class or interface `B` is a subtype of `A`, then `B<T>` is also a subtype of `A<T>`, i.e., they are covariant.

Generics, however, are *invariant*, with respect to the type parameter. That is, if a class or interface `B` is a subtype of `A`, then neither is `C<B>` a subtype of `C<A>`, nor is `C<A>` a subtype of `C<B>`. A parameterized type must be used with exactly with same type argument. For instance:

```java
Queue<Circle> qc = new Queue<Shape>();  // error
Queue<Shape> qs = new Queue<Circle>(); // error
Queue<Shape> qs = (Queue<Shape>) new Queue<Circle>(); // error
```

will lead to compile-time error. Attempting to type cast just like when we do a narrowing reference conversion will fail as well.

Wildcards

Subtyping among generic types are achieved through wildcard types `?`. `?` itself is not a type, but a notation to denote variance of generic types. We can replace type parameter `T` with `?`, to indicate that `T` can be any type. This creates a subtype relationship. For instance, `Queue<Shape>` is a subtype of `Queue<?>`, `Queue<Object>` is a subtype of `Queue<?>`.

```
Queue<?> qc = new Queue<Shape>();   // OK
```

Since the wildcard `?` is not a type, we cannot declare a class like this:

```
class Bar<?> {
    private ? x;
}
```

We only use `?` when specifying the type of a variable, field, or parameter.

Wildcards can be bounded. Java uses the keyword `extends` and `super` to specify the bound. For instance, `Queue<Circle>` and `Queue<Shape>` are both subtypes of `Queue<? extends Shape>`, but `Queue<Object>` is not a subtype of ~~`Queue<?>`~~ `Queue<? extends Shape>`.

To be more general, if `T` is a subtype of `S`, then `Queue<T>` is a subtype of `Queue<? extends S>`. In other words, the use of `? extends` is covariant. Further, `Queue<? extends T>` is a subtype of `Queue<? extends S>`.

The `super` keyword is used to specify the contravariant relationship. If `T` is a subtype of `S`, then `Queue<S>` is a subtype of `Queue<? super T>`.

More intuitively, we can replace `? extends X` with type X or any subtype that extends X; `? super X` with type X or any supertype of X.

> **Wildcard type in Other Languages**
> The use of wildcard type `?` is not common in other programming languages. Partly because Java introduces generics late and has to use type erasure to implement generic types.

Type Erasure

In Java, for backward compatibility, generic typing is implemented with a process called *type erasure*. The type argument is erased during compile time, and the type parameter `T` is replaced with either `Object` (if it is unbounded) or the bound (if it is bounded). For instance, `Queue<Circle>` will be replaced by `Queue` and `T` will be replaced by `Object`, just like how Java code is written before generic type is introduced in Java 5. `Queue<? extends Shape>` will be replaced with `Queue` and `T` will be replaced with `Shape`. The compiler also inserts type casting and additional methods [2][#fn:2] to preserve the semantics of generic type.

Java's design decision to use type erasure to implement generics has several important implications.

First, unlike other languages, such as C++, Java does not support generics of a primitive type. We cannot create a `Queue<int>`, for instance. We can only use reference types as type argument (anything that is a subtype of `Object`).

Second, we cannot have code that looks like the following:

```
class A {
  void foo(Queue<Circle> c) {}
  void foo(Queue<Point> c) {}
}
```

Even though both `foo` above seem like they have different method signatures and is a valid application of method overloading, the compiler actually translates them both to:

```
class A {
  void foo(Queue c) {}
  void foo(Queue c) {}
}
```

and bummer, we have a compilation error!

Third, using static methods or static fields in generic class becomes trickier.

Consider the example

```
class Queue<T> {
  static int x = 1;
  static T y;
  static T foo(T t) {};
}
```

`Queue<Circle>` and `Queue<Point>` both gets compiled to `Queue`. So they share the same `x`, even though you might think `Queue<CIrcle>` and `Queue<Point>` as two different distinct classes.

Further, the next two liens `static T y` and `static T foo(T t) {}` will generate a compilation error. Since there will only be a copy of `y` shared by both `Queue<Circle>` and `Queue<Point>`, the compiler does not know whether to replace `T` with `Circle` or `Point`, and will complain.

Fourth, we cannot create an array of parameterized types.

```
Queue<Circle>[] twoQs = new Queue<Circle>[2]; // compiler error
twoQs[0] = new Queue<Circle>();
twoQs[1] = new Queue<Point>();
```

After type erasure, we get

```
Queue[] twoQs = new Queue[2]; // compiler error
twoQs[0] = new Queue();
twoQs[1] = new Queue();
```

If Java would have allowed us to create an array of a parameterized type, then we could have added an element of incompatible type `Queue<Point>` into an array of `Queue<Circle>` without raising a run-time error.

Generics in Java is added as an afterthought and to maintain backward compatibility, the compiler has to erase the type parameter during compile type. During runtime, the type information is no longer available, and this design causes the quirks above.

Raw Types

One final note: for backward compatibility, Java allows us to use a generic class to be used without the type argument. For instance, even though we declare `Queue<T>`, we can just use `Queue`.

```
Queue<Circle> cq = new Queue();
```

After all, during compile time, `Queue<Circle>` is translated into `Queue` anyway. In Java 5 or later, if we just use `Queue`, we get a `Queue` of `Object` s. This is called a *raw type*. Recent Java compilers will warn you if you use a raw type in your code.

Wrapper Classes

We can get around the problem of not being able to create generics of primitive types using wrapper classes. Java provides a set of wrapper classes, one for each primitive type: `Boolean`, `Byte`, `Character`, `Integer`, `Double`, `Long`, `Float`, and `Short`.

```
Queue<Integer> iq = new Queue<Integer>(10);
cq.enqueue(new Integer(4));
cq.enqueue(new Integer(8));
cq.enqueue(new Integer(15));
```

Java 5 introduces something called *autoboxing* and *unboxing*, which creates the wrapper objects automatically (autoboxing) and retrieves its value (unboxing) automatically. With autoboxing and unboxing, we can just write:

```
Queue<Integer> iq = new Queue<Integer>(10);
cq.enqueue(4);
cq.enqueue(8);
cq.enqueue(15);
```

Note that `enqueue` expects an `Integer` object, but we pass in an `int`. This would cause the `int` variable to automatically be boxed (i.e., be wrapped in Integer object) and put onto the call stack of `enqueue`.

Performance Penalty

If the wrapper class is so great, why not use it all the time and forget about primitive types?

The answer: performance. Because using an object comes with the cost of allocating memory for the object and collecting of garbage afterward, it is less efficient than primitive types. Consider the following two programs:

```
Double sum;
for (int i = 0; i < Integer.MAX_VALUE; i++)
{
    sum += i;
}
```

vs.

```
double sum;
for (int i = 0; i < Integer.MAX_VALUE; i++)
{
    sum += i;
}
```

The second one is 2 times faster! Due to autoboxing and unboxing, the cost of creating objects become hidden and often forgotten.

All primitive wrapper class objects are immutable -- once you create an object, it cannot be changed. Thus, every time `sum` in the example above is updated, a new object gets created.

Generic Methods

We can get around the problem of not being able to define static methods using generic methods. Generic methods are just like generic type, but the scope of the type parameter is limited to only the method itself.

```
class Queue<T> {
  static <T> T foo(T t) { return t; };
}
```

The above would compile. Note that the `<T>` in `static <T> T foo(T t)` effectively declares a new type parameter `T`, scoped to the method `foo`, and has nothing to do with the type parameter `<T>` for `Queue<T>`.
(Recall that, since `foo` is static and is associated with the class `Queue`).

As a convention, it is common to use the single letter `T` here, which may be confusing. We can actually use any other single letter to represent the type parameter.

```
class Queue<T> {
  static <X> X foo(X t) { return t; };
}
```

To invoke a generic method, we can call it like this:

```
Queue.<Point>foo(new Point(0, 0));
```

or simply:

```
Queue.foo(new Point(0, 0));
```

Type Inference

In the last example, Java compiler uses *type inference* to determine what `T` should be: in this case, we are passing in a `Point` object, so `T` should be replaced with `Point`.

Type inference can also be used with constructors.

```
Queue<Point> q = new Queue<>();
```

In the line above, we use the *diamond operator* `<>` to ask Java to fill in the type for us. Again, type inference allows Java to infer that we are creating a `Point` object.

In cases where the type inference finds multiple matching types, the most specific type is chosen.

> **Local variable type inference in Java 10**
> Java 10, which is scheduled to be released next month, supports type inference for a local variable. We can write
>
> ```
> var q = new Queue<Point>();
> ```
>
> and Java 10 compiler can infer that q should have the type `Queue<Point>` .

Java Collections

Now, we turn our attention to the Java Collection Framework. Java provides a rich set of classes for managing and manipulating data. They efficiently implement many useful data structures (hash tables, red-black trees, etc.) and algorithms (sorting, searching, etc.) so that we no longer have to. As computer scientists, it is still very important for us to know how these data structures and algorithms can be implemented, how to prove some behaviors (such as running time) and their correctness, how certain trade-offs are made, etc. They are so important that we have two modules dedicated to them, CS2040 and CS3230, in the core CS curriculum.

In CS2030, however, we will only use them but not study how to implement them. Most importantly, Java Collection Framework is mostly well-designed and serves as real-world examples of how all the OO principles and programming language concepts are used. We can see how all the things that we have learned so far is applied.

Collection

One of the basic interfaces in Java Collection Framework is `Collection<E>` , it looks like:

```
1  public interface Collection<E> extends Iterable<E> {
2    boolean add(E e);
3    boolean contains(Object o);
4    boolean remove(Object o);
5    void clear();
6    boolean isEmpty();
7    int size();
8
9    Object[] toArray();
10   <T> T[] toArray(T[] a);
11
12   boolean addAll(Collection<? extends E> c);
13   boolean containsAll(Collection<?> c);
14   boolean removeAll(Collection<?> c);
15   boolean retainAll(Collection<?> c);
16     :
17 }
```

There are some newly added methods in Java 8 that we will visit in the second half of this module, but first, let's try to understand what the definition above means. First, like a generic class that you have seen, `Collection` is a *generic interface* parameterized with a type parameter `E` . It extends a generic `Iterable<E>` interface (we will get to this later).

The first six methods of `Collection<E>` should be self-explanatory. `add` adds an element into the collection; `contains` checks if a given object is in the collection; `remove` removes a single instance of the given object from the collection; `clear` removes all objects from the collection; `isEmpty()` checks if the collection has no elements or not; and finally, `size` returns the number of elements.

Note that `contains()` relies on the implementation of `equals()` to check if the object exists in the collection or not. Similarly, `remove()` relies on `equals()` to find the matching objects. We said earlier that it is useful to override the `equals` methods of `Object` instead of implementing our own `equals` , because the overridden `equals()` will be called elsewhere. This is one of the "elsewhere" I mentioned. The documentation of `contains(o)` mentions that it is guaranteed to return `true` if there exists an element `e` such that `e.equals(o)` or `e == null` (if `o == null`). Ditto for `remove(o)` .

> **Non-generic Methods**
> You might notice that, instead of `contains(E e)` and `remove(E e)` , the `Collection` interface uses `contains(Object o)` and `remove(Object o)` . This little inconsistency, however, is harmless. For instance, if you have a collection intended for circles only, adding a non-circle could be disastrous. Trying to remove a non-circle or checking for a non-circle, would just return false. More information can be found on this StackOverflow [https://stackoverflow.com/questions/104799/why-arent-java-collections-remove-methods-generic] thread.

Java Collection Framework allows classes that implement an interface to throw an `UnsupportedOperationException` if the implementation decides not to implement one of the operations (but still need to have the method in the class).

The method `toArray()` on Line 9 returns an array containing all the elements inside this collection. The second overloaded `toArray` method takes in an array of generic type `T` . If the collections fit in `a` , `a` is filled and returned. Else, it allocates a new array of type `T` and returned.

The second `toArray` method is an example of a generic method. It is the caller responsibility to pass in the right type, otherwise, an `ArrayStoreException` will be thrown.

The next group of methods operates on another collection. `addAll` add all the elements of collection `c` into the current collection; `containsAll` checks if all the elements of collection `c` are contained in the current collection; `removeAll` removes all elements from collection `c` , and finally, `retainsAll` remove all elements not in `c` .

What is more interesting about the methods is the type of `c` . In `containsAll` , `removeAll` , and `retainAll` , the collection `c` has the type `Collection<?>` . Recall that `Collection<?>` is a supertype of `Collection<T>` , whatever `T` is. So, we can pass in a `Collection` of any reference type to check for equality. In this case, we are not adding anything to the collection, so the type of `c` is set to be as general as possible.

In `addAll` , however, `c` is declared as `Collection<? extends E>` . Since we are adding to the collection, we can only add elements that either has the type `E` or a type that is a subtype of `E` . This constraint is enforced during compile time through the covariance relationship between `Collection<? extends E>` and `Collection<E>` .

> **Iterable**
> The `Iterable<E>` interface provides only a single method, `Iterator<E> iterator()` , which returns a generic interface called `Iterator<E>` over the collection. An `Iterator` is another interface that allows us to go through all the elements in a `Collection<E>` . Useful methods include `hasNext()` , which returns if there is a next element in the `Collection<E>` ; `next()` , which returns the next element (with parameterized type `E`); and `remove()` , which removes the last returned element from the `Collection<E>` .

So far we have not seen any example code using `Collection` . This is because Java Collection Framework does not provide a class that implements the `Collection<E>` directly. The documentation recommends that we implement the `Collection<E>` interface [3] [#fn:3] if we want a collection of objects that allows duplicates and does not care about the orders.

Set and List

The `Set<E>` and `List<E>` interfaces extend the `Collection<E>` interface. `Set<E>` is meant for implementing a collection of objects that does not allow duplicates (but still does not care about order of elements), while `List<E>` is for implementing a collection of objects that allow duplicates, but the order of elements matters.

Mathematically, a `Collection<E>` is used to represent a bag, `Set<E>` , a set, and `List<E>` , a sequence.

The `List<E>` interface has additional methods for adding and removing elements. `add(e)` by default would just add to the end of the list. `add(i, e)` inserts `e` to position `i`. `get(i)` returns the element at position `i`, `remove(i)` removes the elements at position `i`; `set(i,e)` replace the `i`-th element with `e`.

Useful classes in Java collection that implement `List<E>` include `ArrayList` and `LinkedList`, and useful classes that implement `Set<E>` include `HashSet`.

Let's see some examples:

```
1 List<String> names = new ArrayList();
2 names.add("Cersei");
3 names.add("Joffrey");
4 names.add(0, "Gregor");
5 System.out.println(names.get(1));
```

Line 1 above creates an empty array list. The second line adds two strings into the list, each appending them to the list. After executing Line 3, it would contain the sequence `<"Cersei","Joffrey">`. Line 4 inserts the string `"Gregor"` to position 0, moving the rest of the list down by 1 position. The sequence is now `<"Gregor","Cersei","Joffrey">`. Finally, calling `get(1)` would return the string `"Cersei"`.

Note that we declare `names` with the interface type `List<String>`. We should always do this to keep our code flexible. If we want to change our implementation to `LinkedList`, we only need to change Line 1 to:

```
List<String> names = new LinkedList();
```

Comparator

The `List<E>` interface also specifies a `sort` method, with the following specification:

```
default void sort(Comparator<? super E> c)
```

`List<E>` is an example of an interface with default method. The keyword `default` indicates that the interface `List<E>` comes with a default `sort` implementation, so a class that implements the interface needs not implement it again unless the class wants to override the method.

This method specification is also interesting and worth looking closer. It takes in an object `c` with generic interface `Comparator<? super E>`. The `Comparator` interface allows us to specify how to compare two elements, by implementing a `compare()` method. `compare(o1,o2)` should return 0 if the two elements are equals, a negative integer if o1 is "less than" o2, and a positive integer otherwise.

Let's write `Comparator` class [4] [#fn:4]:

```
class NameComparator implements Comparator<String> {
    public int compare(String s1, String s2) {
        return s1.compareTo(s2);
    }
}
```

In the above, we use the `compareTo` method provided by the `String` class to do the comparison. With the above, we can now sort the `names`:

```
names.sort(new NameComparator());
```

This would result in the sequence being changed to `<"Cersei","Gregor","Joffrey">`. We can easily change how we want to sort the names, for instance, to

```
class NameComparator implements Comparator<String> {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

if we want to sort by the length of the names.

One last thing to note about the method `sort` is that it takes in `Comparator<? super E>` that is contravariant. Recall that `Comparator<Object>` is a subtype of `Comparator<? super E>`, so we can pass in more general `Comparator` object to `sort` (e.g., a class that implements `Comprator<Object>`

Map

One of the more powerful data structures provided by Java Collection is maps (also known as a dictionary in other languages). A map allows us to store a (unique key, value) pair into the collection, and retrieve the value later by looking up the key.

The `Map<K,V>` interface is again generic, but this time, has two type parameters, `K` for the type of the key, and `V` for the type of the value. These parameters make the `Map` interface flexible -- we can use any type as the key and value.

The two most important methods for `Map` is `put` and `get`:

```
    V put(K key, V value);
    V get(Object k);
```

A useful class that implements `Map` interface is `HashMap`:

```
Map<String,Integer> population = new HashMap<String,Integer>();
population.put("Oldtown",500000);
population.put("Kings Landing",500000);
population.put("Lannisport",300000);
```

Later, if we want to lookup the value, we can:

```
population.get("Kings Landing");
```

Which Collection Class?

Java provides many collection classes, more than what we have time to go through. It is important to know which one to use to get the best performance out of them. For the few classes we have seen:

- Use `HashMap` if you want to keep a (key, value) pair for lookup later.
- Use `HashSet` if you have a collection of elements with no duplicates and order is not important.

- Use `ArrayList` if you have a collection of elements with possible duplicates and order is important, and retrieving a specific location is more important than removing elements from the list.
- Use `LinkedList` if you have a collection of elements with possibly duplicates and order is important, retriving a specific location is less important than removing elements from the list.

You should understand the reasons above after CS2040.

Further, if you want to check if a given object is contained in the list, then `ArrayList` and `LinkedList` are not good candidates. `HashSet`, on the other hand, can quickly check if an item is already contained in the set. There is, unfortunately, no standard collection class that supports fast `contain` and allow duplicates.

Exercise

1. Consider the following code snippet:

```
interface I {
  void f();
  default void g() {
  }
}

abstract class A implements I {
  abstract void h();
  abstract void h(int x);
  void j() {
  }
}

class B extends A {
  :
}
```

   Class `B` is a concrete class. What are the methods that `B` needs to implement?

2. For each of the statement below, indicate if it is a valid statement (no compilation error). Explain why in one sentence.
   - `List<?> list = new ArrayList<String>();`
   - `List<? super Integer> list = new List<Object>();`
   - `List<? extends Object> list = new LinkedList<Object>();`
   - `List<? super Integer> list = new LinkedList<>();`

3. Consider a generic class `A<T>` with a type parameter `T` and a constructor with no argument. Which of the following statements are valid (i.e., no compilation error) ways of creating a new object of type `A`? We still consider the statement as valid if the Java compiler produces a warning.
   - `new A<int>();`
   - `new A<>();`
   - `new A();`

4. In this question, we will explore the more strange behavior of `Integer` class.

   Remember that `==` compares only references: whether the two references are pointing to the same object or not. The `equals` method has been overridden to compare if the values are the same or not. Try the following in `jshell`:

```
Integer x = 1;
Integer y = 1;
x == y;
```

   What do you expect the comparison `==` to return? What did it return?

   Now try:

```
Integer x = 1000;
Integer y = 1000;
x == y;
```

   What do you expect it to return? What did it return?

   Look up on the Internet (e.g., StackOverflow) on why this happens. (Hint: `Integer` caching)

   The moral of the story here is to always use `equals` to compare two reference variables.

5. Consider a method declared as `int foo(double x)` Which of the following statements will NOT result in a compilation error:
   - `int cs = foo(2030);`
   - `double cs = foo(2.030);`
   - `int cs = foo(new Double(2.030));`
   - `Integer cs = foo(new Double(2.030));`

6. In this question, we will explore the behavior of `ArrayList` class and autoboxing/unboxing. Will the following code compile? If so, what will be printed?

   (a)

```
List<Integer> list = new ArrayList<>();
int one = 1;
Integer two = 2;

list.add(one);
list.add(two);
list.add(3);

for (Integer num : list) {
  System.out.println(num);
}
```

   (b)

```
List<Integer> list = new ArrayList<>();
int one = 1;
Integer two = 2;

list.add(one);
list.add(two);
list.add(3);
```

```

for (int num : list) { // Integer -> int
  System.out.println(num);
}
```

©

```
List<Integer> list = Arrays.asList(1, 2, 3);

for (Double num : list) {
  System.out.println(num);
}
```

(d)

```
List<Integer> list = Arrays.asList(1, 2, 3);

for (double num : list) {
  System.out.println(num);
}
```

(e)

```
List<Integer> list = new LinkedList<>();
list.add(5);
list.add(4);
list.add(3);
list.add(2);
list.add(1);

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

7. Here is another set of questions to explore about the behavior of autoboxing/unboxing and primitive type conversion.

(a)

```
double d = 5;
int i = 2.5;

System.out.println(d);
System.out.println(i);
```

(b)

```
double d = (int) 5;
int i = (double) 2.5;

System.out.println(d);
System.out.println(i);
```

©

```
double d = (int) 5.5;
int i = (int) 2.5;

System.out.println(d);
System.out.println(i);
```

(d)

```
Double d = 5;
Integer i = 2.5;

System.out.println(d);
System.out.println(i);
```

(e)

```
Double d = (double) 5;
Integer i = (int) 2.5;

System.out.println(d);
System.out.println(i);
```

(f)

```
double d = (Integer) 5;
int i = (Integer) 2;

System.out.println(d);
System.out.println(i);
```

(g)

```
double d = (Double) 5;
int i = (Integer) 2;

System.out.println(d);
System.out.println(i);
```

---

1. This is widely considered as a bad practice (or *anti-pattern*), called the *constant interface anti-pattern*.
2. Look up bridge methods if you want to know the gory details.
3. If you want to do so, however, it is likely more useful to inherit from the abstract class `AbstractCollection<E>` (which implements most of the basic methods of the interface) rather than implementing the interface `Collection<E>` directly.
4. Later in CS2030, you will see how we significantly reduce the verbosity of this code! But let's do it the hard way first.