Lesson 1: Generics

Professor Abdul-Quader

Data Structures

We will be exploring abstract data types: lists, stacks, queues, etc.

Problem: Do we need to re-write our stack code to support `ListOfIntegers` vs `ListOfDoubles` vs `ListOfStrings`?

```
public class MyList<T> {
   public void add(T newElement) {
      // add newElement to the list
   }

   public T get(int index) {
      // get the element at that position of the list
   }
}
```

T is the **type parameter** or **generic**. MyList is a "generic type" (or "parameterized type").

Autoboxing / unboxing, diamonds

Technically, the type parameter needs to be instantiated by an actual class:

```
// MyList<int> l = new MyList<>(); this does not compile!
MyList<Integer> l = new MyList<>();
```

- Use the wrapper classes to instantiate: Integer for int, Double for double, etc.
- **autoboxing**: When we add to the collection, Java automatically changes ints to Integers, so we don't need to worry about it: `l.add(3);` instead of `l.add(new Integer(3));`.
- **unboxing**: When we get from the collection, we can store it in a regular int: `int num = l.get(3);`

Exercise

Implement a simple generic class which supports two operations: read and write. A skeleton is posted on Moodle under the name "ReadWriteCell.java". Fill in the code for that

Implement a main method which instantiates this class and test out that it works. What happens if you do the following:

```
ReadWriteCell<String> r = new ReadWriteCell<>();
r.write(32);
```

Wildcards

Generics are not **covariant**: Suppose `Apple` extends from class `Fruit`. We might hope that `Collection<Apple>` can be used whenever `Collection<Fruit>` is asked for (in a parameter). This is not the case!

```
// cannot call printFruit() and pass a Collection<Apple>!
public void printFruit(Collection<Fruit> c) {
   for (Fruit f : c) {
      System.out.println(f);
   }
}

// use wildcards instead!
public void printFruit(Collection<? extends Fruit> c) {
   // .. same code as before
```

Can also use `<? super Fruit>` to mean a superclass instead of a subclass.

Generic Static Methods

Quick exercise: try to write a static method which searches a generic array for a value. It should return true if the value is in the array, false otherwise. Where do you declare the type parameter?

Generic Static Methods

Quick exercise: try to write a static method which searches a generic array for a value. It should return true if the value is in the array, false otherwise. Where do you declare the type parameter?

```
public static <T> boolean contains(T[] array, T value) {
    ...
}
```

Type Bounds

Quick exercise 2: try to update the `SelectionExercise` to work for a generic type. How would we compare generic objects, without knowing that they are integers? Use the `Comparable<T>` type!

Type Bounds

Quick exercise 2: try to update the `SelectionExercise` to work for a generic type. How would we compare generic objects, without knowing that they are integers? Use the `Comparable<T>` type!

```
public class SelectionExercise<T extends Comparable<T>> {
   // ...
   // only change is compareTo instead of <
```

What if `Fruit` implements `Comparable<Fruit>`? Will `SelectionExercise<Apple>` work? (That is, does "Apple" extend `Comparable<Apple>`?)

Solution:
`SelectionExercise<T extends Comparable<? super T>>`.

Exercise

Implement a generic, static method which finds the minimum element in an array of (generic) objects. Assume these objects implement the Comparable interface.

Syntactic Sugar

```
String[] list = new String[5];
Object[] o = list;
o[0] = new Integer(30);
```

Does this compile? Does it run?

Generics were created to fix this: now run-time errors become compile-time errors. But the fix, in Java, was a half-measure: generics are only "syntactic sugar". When you compile a class `MyList<T>`, it creates just a single **raw type**, rather than one for each possible type parameter T. That is: it just becomes MyList (the parameter T gets replaced by its "bound", Object in this case).

Type Erasure

Type Erasure: after the class is compiled to bytecode, the "JVM"-version of the class is the raw type (with no generic). This means:

- Primitive types cannot be used as type parameters, because `int` does not inherit from `Object` (or from any other class).
- Casts can mess you up:

```
ReadWriteCell<String> rws = new ReadWriteCell<>();
rws.write("Hello");
ReadWriteCell<Integer> rwi = (ReadWriteCell<Integer>)
    rws; // this works!
int x = rwi.read(); // this is bad
```

- Cannot instantiate a generic type (what constructor would **new** call?)

More Restrictions

- Cannot create an array of generic objects.
  (`T[] array = new T[5];`)
- Cannot createa an array of parameterized types:

```
ReadWriteCell<String>[] arr1 = new ReadWriteCell<>[5];
ReadWriteCell<Double> c = new ReadWriteCell<>();
c.write(0.01);
Object[] badGuy = arr1;
badGuy[0] = c;
String s = arr1[0].read();
```

We will start this next class.

- **Abstract Data Type**: an abstraction of a data type. Set of obejects with some defined set of operations. Does not talk about the specific implementation of the type!
- Often we use interfaces to specify the operations.
- First examples: Lists, Stacks, and Queues

```
public interface List<T> {
   void insert(T object, int position);
   void remove(int position);
   void printList();
   T get(int i);

   // maybe others
}
```