CH. 2
OBJECT-ORIENTED PROGRAMMING

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN JAVA, GOODRICH, TAMASSIA AND GOLDWASSER (WILEY 2016)
INTERFACES AND ABSTRACT CLASSES

• The main structural element in Java that enforces an application programming interface (API) is an interface.

• An interface is a collection of method declarations with no data and no bodies.

• Interfaces do not have constructors and they cannot be directly instantiated.
  • When a class implements an interface, it must implement all of the methods declared in the interface.

• An abstract class also cannot be instantiated, but it can define one or more common methods that all implementations of the abstraction will have.
INTERFACE EXAMPLE

1. public interface Robot {
2.   void sense(World w);
3.   void plan();
4.   void act(World w);
5. }

USE IMPLEMENTS TO ENFORCE THE INTERFACE

1. public class Roomba implements Robot {
2.   /* code specific to Roomba */
3.   public void sense(World w) { /* Roomba’s don’t sense */}
4.   public void plan() { /* code for Roomba's actions */}
5.   public void act(World w) { /* code to power motors */
6.     /* code specific to Roomba */
7.   }

In this context, enforce means required by the compiler. A good example is sorting algorithms. In order to sort two things must be comparable. So Java offers a comparable interface so that your own objects can fit into this framework.

Interfaces are also an example of inheritance. They are "weakly" inherited by implementing classes. So the rules of polymorphism also apply, i.e., an object can be converted to a variable of the interface:

```java
Robot r = new Roomba();
```
GENERIC PROGRAMMING

JAVA GENERICS
Generic Programming is a programming paradigm where the programmer programs interfaces and algorithms without a specific object hierarchy in mind. Rather the programmer only worries about operations (methods) needed to support an interface or perform an algorithm.
HOW DO WE PROGRAM GENERICALLY?

1. Convert things to raw memory, after all it is all 0's and 1's to the computer

2. Inheritance/polymorphism – Treat everything as an Object, or use very deep class hierarchies in combination with polymorphism

3. Generics/templates – A programming technique where we program without any specific type. Then when we instantiate a generic class/function the types becomes known

• (1) is outdated and for pure C programming (2) is done with polymorphism and (3) is done with Java Generics

• In all honesty though, we use (2) and (3) in combination.
SYNTAX FOR GENERIC OBJECTS

• Types can be declared using generic names:

1. `public class Array<E> {`
2. `private E e[];`
3. `/* Rest of class */`
4. `}`

• They are then instantiated using actual types:
  • `Array<String> arr = new Array<>();`

• There is not much to it actually, but it is a very strange thought process that you do not know what E is as you write it.
GENERIC OBJECTS

• You may have one or more generic types. This class, we will have at most two
  • public class Map<Key, Value> {
    }
  • Map<Integer, String>

• Generic types must be Java Objects, so you can use any class that inherits from Java Object, i.e., unfortunately you have to use Integer or Float instead of int or float.

• Many other quirks and oddities that will be experienced as we go!

• All code examples in the book and programming assignments involve this form of programming actually.
GENERIC FUNCTIONS

• Can also be used in functions:

1. `public static <T, S> String concat(T t, S s) {
2.    return t.toString() + s.toString();
3. }

• Used like:

1. `MyObject1 a;
2. `MyObject2 b;
3. `String c = concat(a, b);`
• We can specify constraints on the generic parameters through interfaces to only accept certain types in generic classes/functions

    public <T extends Comparable<T>> int compareTwo(T a, T b) {
        return a.compareTo(b); // Comparable<T> enforces // having the compareTo function
    }

• *extends* is always used, it generally refers to *extends* or *implements* in this context
EXERCISE

• With your team. Create a generic interface called Addable with one function add
• Create two classes that implement Addable: MyString and Point.
  • MyString owns a single string and uses concatenation for adding
  • Point should be a 2D point in the Cartesian plane. Adding will be the sum of components.
• Create a generic function sum that sums an array of Addable objects
• Test your function in main with both random MyStrings and Points