Interfaces and Generics

Ch 5.7 & 10.5
Highlights

- Implementing interfaces

```java
public class MyInt implements Comparable{
```

- Passing types

```java
public class GenericClassExample <MyType> {
```
Abstract classes

Remember that **abstract classes** are a way to define a blueprint

An **abstract class** simply has **abstract** in front of a method to mean: “I will define this method later (i.e. my children will)”

This is useful if you want all of your children to have a method (e.g. a Mammal class has a `reproduce()` method)
Interfaces

Abstract classes can contain both real methods and abstract methods.

You can think of an interface as an extreme abstract class in that:
- Interfaces contain only public abstract methods
- Interfaces can only have constant variables
Interface example

Suppose you wanted to make some methods to compare numbers and words

- Both have an ordering (a < z, 2 < 4)
- However they are different types

To be able to compare both of these, you would need to overload:

```java
public static boolean lessThan(int left, int right)
public static boolean lessThan(char left, char right)
```

(See: ComparingIntChar.java)
Interface example

It is silly to have to define these methods that do the exact same thing multiple times.

We want to somehow use late binding to find the appropriate types.

The solution to this is to use an interface:
- Provides abstract methods for late binding.
- All types that use interface must define those methods.
Interface example

This interface already exists and is called “Comparable” (more on this in Ch 14)

The comparable interface has one method:
public int compareTo(Object o)
  - returns -1: if this < o
  - returns 0: if this == o
  - returns 1: if this > o

(See: CompareExample.java)
Implementing an interface

The reason both Integer and Character have the compareTo method is that both use the interface Comparable.

In order to do this, you have to add the this to the class (and override compareTo):

```java
public class MyInt implements Comparable {

(See: MyInt.java)
```
Implementing an interface

For inheritance (extends) you can only have one parent

For interfaces (implements) you can use as many interfaces as you like (comma separated)

```java
public class MyIntGet
    extends MyInt
    implements Comparable, Serializable {
```

(See: MyIntGet.java)
Serializable

We have already used interfaces when doing file input/output

Remember that if you wanted to write a class to a binary file you needed this line:

```java
public class MyDog implements Serializable{
```

However, you did not need to override any methods when implementing Serializable...
Interface is a type

You can use interfaces as types (much like abstract classes)

The reason to implement Serializable (but not override any methods) is so your class can be used as the Serializable type

This allows more generalizable methods (allows you to reuse other parts of code) (See: InterfaceAsType.java)
did you know?

The bushes in Super Mario Bros. were just recolored clouds.
Making an interface

You can make your own interfaces in normal .java files much like classes, but you write:

```java
public interface SampleInterface {
}
```

instead of (at the top):

```java
public class SampleClass {
}
```

Interfaces can also extend other interfaces: (much like classes can extend other classes)

```java
public interface ChildInterface extends Comparable{
}```
Making an interface

As we said at the start, interfaces have the following restrictions:

- Methods can only be abstract and public (cannot be static)
- Variables can only be constant (public static final)

However if you leave out any of these words in blue, they will still be implied (See: SampleInterface.java)
One reason Java only allows one parent is to prevent inconsistencies.

Since you can implement multiple interfaces, you can get inconsistencies (with methods and constants), and will cause errors.

(See: InterfaceError.java)
Generics
Generics allow you to “pass” types along with passing arguments like we always have.

If you want to pass a class a type, you write:

```java
public class GenericClassExample <MyType> {

    MyType is a label
    (much like a variable name)

    After this, you can use MyType similar to a normal type inside the class
    (See: GenericClassExample)
```
Generics have a couple shortcomings:
- They can only hold references (no primitive types like int, double, char)
- Since they are general, you can only do very general operations on them

You might want to add numbers or concatenate Strings, but addition does not make sense for all classes (like Scanner + Scanner = ????)

Restricting generics

You can put restrictions on what types you can pass, allowing you to do more operations

To specify what types you can pass, you write:

```java
public class NumberTypes <MyNumber extends Number> {
```

This means MyNumber has to be typecast-able to the thing right of `extends`:
- (for classes) Being a child of the class
- (for interfaces) Implements the interface

(See: NumberTypes.java, Sortable.java)
Generics in book

In addition to what we have covered, generics can be used to:

- Pass multiple types
- Have multiple restrictions
- Pass types to methods
- (also generics are not really a “type”)

The book describes all of these (or you can search the web), but we will not discuss them