CISC124 Quiz 3 Notes

JavaDoc

- Javadoc.exe is a program that comes with the JDK, its NOT included in the JRE only
- If I have written the class "MyClass.java", that contains properly formatted comments, then running "javadoc MyClass.java" **generates a file "MyClass.html"**
- The html file contains external documentation generated from the formatted comments in the source code
- Normal block comments are /* */ JAVADOC comments are: /** */
- General form of Javadoc tag is @tagName comment
- The tags you use depend on what you are describing (class, method, attribute)
- In the car of methods, you can have a tag for each parameter, return value, and a tag for each thrown exception
- Typically you will only write javadoc comments for public attributes and methods

- Html tags can also be added to the comments to add more formatting to the resulting document:
  - for emphasis
  - <code> for code font
  - <img> for images
  -   for bulleted lists

- The output from Javadoc looks exactly like the API documentation
- The advantage is that when source code is changed, the Javadoc comments can be changed in the source at the same time
- The external documentation is than easily re-generated
- Javadoc also provides a consistent look and feel to these API documents

- Most modern IDE's allow you to run Javadoc from within the environment, and provide tools and wizards to help you create comments

Debugging

- With the debugger, you can run to a breakpoint, stop your program and then execute one line at a time while watching the call stack, variables, and custom expressions

Take a look at the UseDebugger.java file

Testing

- JUnit is a framework designed for this kind of work and it is very easy to use

JUnit Testing:

-

Assertions

Use (expected, actual) or (String, expected, actual):

- assetArrayEqual()
- assertEquals()
- assertFalse()
- assertNotEquals()
- assertNotNull()
- assertNotSame() //for objects

- assertSame() //for objects
- assertNull()
- assertTrue()

assertThat()
- Takes a Matcher<T> object for its second (or third) parameter
- Note that the order is different, the *actual* comes first followed by the *expected*
- You can still have a String message as the first parameter
- You can build more sophisticated assertions with this method
- (tests to see if exceptions are thrown by using: @test (expected=IllegaHalloweenException.class)

No try/catch
- If you are testing code that has a throws declaration, then just have your unit test method throw Exception to satisfy the compiler
- If an exception is thrown unexpectedly (a run-time error) you will get a different kind of failure notice in the report. (NOTE: the testing DOES NOT stop)

Test Suites
- You can combine separate JUnit testing classes into a single suite and run them all at once

Advantages
- Tests are all separate from the code being harnessed
- Very easy to add tests
- Running tests is a very simple and fast
- Results are nicely summarized and you can zoom in on failed tests easily and get some decent diagnostics

Ex. Test can be run on remote classes on a server

Designing and Running Unit Tests
- Start by testing the most concrete independent methods first
- Then test the methods that depend on these ones
- Then test classes
- Then test systems

- You will need to write stub and driver code as required in the testing classes
- Stub code **substitutes** something “fake” to a method that depends upon it
- Driver code **simulates** code that uses the method being tested
- Keep the stub and driver code SIMPLE so that error can only come from the harnessed code, not the testing code

- Start with tests that easily fall into the normal, expected range of inputs. Few or many? Use test generator tools?
- Create more tests for inputs that you suspect are related to each other
- Choose tests for inputs that are just barely legal
- Choose tests for inputs that are way out there!
- Make sure you exercise all the code being harnessed

- Keep all tests unless you have deleted some of the methods/classes that you were testing. You can even leave these in with a “fail” call

- Fixes applied to some methods can invoke a ripple effect, causing previously passed tests to fail
- In a team effort always run (and pass) all of your unit tests before committing your code to the repository

Unit Testing
- You can never test all possible input conditions
- Unit testing doesn't prove that your code is without errors, but you can end up feeling pretty good about your code if it passes all tests

Test Driven Development (TDD)
- How much code can you "implement" without testing it?
- You should NOT separate the process of implementation and testing
- Not surprising — this is the basic tenant of Agile Development
- Is all about writing tests while you are writing, or decomposing your code
- In fact the testing DRIVES and LEADS the coding
- We need a systematic technique to create tests that ensures coverage:

The Three Laws of TDD
1. You may not write production code until you have written a failing unit test
2. You may not write more of a unit test than what is sufficient to cause a failure, and not compiling is a failure
3. You may not write more production code than what is sufficient to pass the current failing test

- So tests should be written at the same time as the production code. You are always one test ahead of what the production code can pass.
- As the amount of production code grows so do the number of tests
- Testing code should be in separate files from production code
- It really helps to have some kind of framework to help organize and run these tests

Packages
- "package" is a Java keyword
- It provides a means of grouping classes together into a package
- All classes in a package share some common theme
- It is used as in: package packageName;
- This line at the top of a class definition, before the public class

- Eclipse prefers you to create classes in a package
- When you create a new Class, specify what package you want it to belong to. This will automatically add the package packageName; line to the top of the class, the proper folder is created in src, and the new class is saved in the folder
- Eventually you will create a *jar (or *.zip) file with all the *.class files in your package
- Put this user library somewhere in or below your **class path** and then another class will be able to import it

- The structure is:
  classpath\folder\packagename
- "folder" can be a series of folders
- The import statement looks like:
  import folder.packagename.*;
- The ".*" says to import all classes in the package, or you can import specific classes

- We have been importing existing packages already
  import java.util.Scanner;
  import java.io.*;
- NOTE: Java automatically imports the java.lang.* package for you
- This package contains many classes fundamental to the Java language:
  - String
  - Math
  - all Wrapper classes (Boolean, Character, Integer, Long, Float, and Double)
  - System, and a few others…

static Import
- Used to import all static methods in a class, so that they can be used as if they were declared within the class that uses the,

Ex. import static java.lang.Math.*;

Protected Access
- So far we have used public and private access modifiers
- Protected is what you get if you don't specify an access modifier
- This means "public inside the package, private outside the package"
- Used in the API
- Use sparingly!

Enumerated Types
- enum is a keyword that is new since Java 5.0 but C has had enum's for a while

Ex. enum IceCream {CHOCOLATE, VANILLA, STRAWBERRY, GOLD_MEDAL_RIBBION, WOLF_PAWS, BUBBLE_GUM};

IceCream favourite = IceCream.GOLD_MEDAL_RIBBON;
IceCream leastFavourite = IceCream.BUBBLE_GUM;

System.out.println("Favourite is " + favourite);
System.out.println("Least Favourite is " + leaveFavourite);
DISPLAYS: Favourite is GOLD_MEDAL_RIBBON
Least Favourite is BUBBLE_GUM

- The contents of IceCream are NOT Strings, but they are a kind of Object
- IceCream is also an Object
- They behave as named constants, so by convention, they should be in upper case
- Once you have specified the possible values for IceCream in the enum declaration, you cannot change them
- A variable of type IceCream can only be assigned to one of the possible types

- The Objects in an enum Object inherit a few methods including:
  - equals(), toString(), compareTo(), ordinal(), values()
- you can use equals() or == to test for equality
  Ex. System.out.println(leastFavourite == IceCream.BUBBLE_GUM);
  prints out true to the console
- compareTo() compares the members on the basis of their position in the enum collection

ordinal() Method
- Returns the numeric location of the value in the list (numbering starts from zero)

Ex.
System.out.println("Location = " + favourite.ordinal());
DISPLAYS: Location = 3

values() Method
- Returns an array of the enum's Objects in exactly the same order

Ex. IceCream[ ] flavours = IceCream.values();
System.out.println(flavours[3]);
DISPLAYS: GOLD_MEDAL_RIBBON

Enumerated Types Summary
What's the point of these things?
- enums are used when you need a short, simple, list of things, or "named values"
- The list has a finite size, will not grow and will not (and cannot) change, unlike arrays
- You want to know that you can access ONLY the defined members of the list and no others
- Modern IDE's will provide a list of all the values, when you type a period after the names of the enum

Yes there are other ways to do this, but an enum is pretty easy to use!

Inner Classes
- Simply defined as a class defined within a class
- The inner class can easily get at all the attributes and methods of the outer class
- The outer class can also access the attributes and methods of the **private inner class** after it **instantiates** the inner class
- However, all private classes, methods, and attributes are hidden outside the outer class

Inner Classes: Why Private?
- If you are declaring a non-static object inside your class, you are only doing so because you want to hide it away
- The object can only be used inside the outer class and is hidden everywhere else

So, whats the point of private inner classes?
(Used quite often with GUI coding)
One Reason, when you wish to use a small class (in terms of the size of its definition), but don't want to bother creating a separate class definition file

- An inner class can be a "hidden" object!
- This is often used with Linked List definitions to define the node object
- Yes, you can have an inner class inside an inner class

public static Inner Classes

What is the point of declaring an inner class public static?

- It would allow you to "categorize" methods into topical groups. Invoke them like:

First:

```
TestClass tc = new TestClass();
```

Then, invoking methods from public static inner classes:

```
tc.Group1.method();
tc.Group2.anotherMethod();
```

Anonymous Classes

- Are a kind of inner class
- Defined by putting the code for the class right in the instantiation statement for the object

```
public class AnonymousClassDemo {
    public static void main(String[] args) {
        MessageSender ms = new MessageSender() {
            public void sendGreeting(String name) {
                System.out.println("Hello " + name" + "!");
            }//end sendGreeting
        }; //end ms, NOTE ";"!
        ms.sendGreeting("Alan");
    }//end main
}//end AnonymousClassDemo
```

- MessageSender is an interface, not an Object
- So you have not actually named the Object that contains the definition of the method sendGreeting()
- ms is an Anonymous Object

- Some coders will tell you that anonymous classes are just classes written by lazy coders
- This is TRUE, but you will see these little beasties used with GUI coding
- Java 9 now has a very tidy solution to using messy anonymous classes in a GUI program — using Lambda Functions

abstract Classes

- It is not unusual to declare a class in the root of a hierarchy to be abstract:

```
public abstract MyClass…
```

- Any class declared this way CANNOT be instantiated
- It can only be extended
- Unlike an interface, an abstract class can also contain concrete method definitions and any kind of attribute
- If a class has one or more abstract methods, the class must be declared abstract as well
- abstract methods have no code in them

Ex. public abstract String getListing();

- A class that extends an abstract class MUST override all the abstract methods in the class, unless it wants to be abstract too
- Unlike the interface, you should write public abstract for each method signature

Why Bother?
- An abstract class forces sub-classes to define certain methods. The shells ensure that the hierarchy has a consistent design
- Also, when declaring a method in a very abstract class, then you don't have to worry about what to do in the method body, especially if it must return a value
- Provides the mechanism for polymorphism!

Interfaces
- Interfaces contain constant attributes and/or abstract methods ONLY
- Abstract methods consist of just the method header — there is a semi-colon instead of { }
- (Abstract classes can contain both concrete and abstract methods, along with any kind of attribute)

- An interface is a design specification, that lists a set of expectations for classes that implement the interface
- Interfaces DO NOT extend Object
- Interfaces can extend multiple interfaces (but not other classes)
- Classes can implement one or many interfaces:
  public class Test implements interface1, interface2, interface3, … { }
- An interface cannot be instantiated, but you can use the interface type as if it is a class
- The interface acts as a "stand-in" for the concrete object that will replace it when the program is running
- The interface "guarantees" the required behaviour of the object it is standing in for

- Attributes in interfaces can only be public final static, you CANNOT use private. You don't even have to specify public final static as it is ASSUMED. Constants must be initialized
- Method signatures must also be public and you don't have to state it explicitly. They are also "abstract", but you do not use the abstract keyword in an interface (considered tacky)
- A class that implements an interface must have a concrete implementation of every method signature in the interface

When designing an interface, you must be careful to use unique method and constant names, in case your interface becomes part of a multiple implementation

Viewing Java Source Code …

The Comparable Interface in Java
Without all the javadoc comments:

```
package java.lang;
import java.util.*;
public interface Comparable<T> {
        public int compareTo( T o);
}
```

- Thats it!, This is a GENERIC interface now
- A class that implements Comparable can be sorted with:
  Array.sort() or Collections.sort() or ArrayList.sort()
- The first two methods are already contained in java (package java.util) and use very fast merge sort or Quicksort algorithms

Why is this implementation necessary?
Implementation in Arrays.mergesort()
The line code from mergeSort() method in java.util.Arrays class:
if (((Comparable) src[mid-1]).compareTo(src[mid]) <= 0)
- This method sorts src[ ] which is of TYPE Object[ ]
- So provided your Objects implements Comparable, this method can sort it!

Interfaces…
- While interfaces are not proper objects in Java, you can "pretend" that they are in some circumstances

Ex. A class can use the interface as an object type, and write code as if you are invoking a method declared in the interface. You can pretend that the specification is an objet
- Part of polymorphism
- An instance of a class that implements an interface can be cast to that interface type

Inheritance
- An OOP design technique that allows you to add to, modify, replace, or simply adopt the methods and attributes in an existing class
- The class that you are modifying is called the **parent** or **super** class, and the resulting class is called the **child** or **sub** class
- The child or sub-class **inherits** all the PUBLIC attributes and methods of the parent or superclass
- The child class **extends** the parent class
- The sub-class is always **more concrete** or **less abstract** than the super class

Why Bother?
- One reason is **code re-use**
- It is also a design tool that allows the coder to more easily model a real-life structure
- You can also use inheritance to enforce code structure on child objects (using interfaces and abstract classes)
- And it is part of the polymorphism mechanism

- Child Objects always have an "is a " relationship with the Parent Object
- In the Person class you would code all the attributes (age, gender, hair colour etc) that describe a person
- The sub classes only need to hold those attributes that are specific to them (such as numExamsToCreate in the Professor Object)
- The sub-classes do not have to repeat any of the code in the super-class

Some Goals of Hierarchy Design
- All attributes in an instance should be defined with meaningful values
- Minimize code repetition
- The same attribute should not be declared in more than one class

- A child class should be more concrete than its parent class
- Design for polymorphism:
    - Use abstraction (abstract classes and interfaces) to ensure common behaviour and to allow for polymorphism
- Control and minimize the public interface of all classes
- Do not have any “pass through” classes that do not contribute to anything
- Make sure the structure is extensible
- Use a consistent variable and method naming scheme throughout the hierarchy
- Follow the “real world” hierarchy as closely as possible

The Object class
- So we are inheriting all the public members of this class — whether we want this stuff or not

extends Keyword
- Creating the Person class:
    public class Person { … }
- Which is the same as:
    public class Person extends Object { … }
- But the “extends Object” is always there, so you don’t have to write it
- Creating the Professor and Student classes:
    public class Professor extends Person { … }
    public class Student extends Person { … }

super Keyword
- Provides a reference to the immediate parent class
- NOTE: the compiler will force you to invoke super in a child class’ constructor and it must be the first line in the constructor

Polymorphism

Is when a pointer of a parent class ends up pointing to a different child class objects at runtime. Also called “DYNAMIC BINDING” the process also must satisfy early binding
Early Binding: is satisfied when the parent class also owns the method that will end up being invoked from the morphed child class objects. The use of interfaces and abstract classes can make it easier to code for early binding

- Pointers of type Person are pointing to objects of type Student and Professor
- At run time the pointers will morph into their actual object types — this is late binding or dynamic binding

Methods in Sub-Classes
- You have 5 ways to have methods in sub-class:

1. Write a new method (concrete or abstract)
2. **Inherit** the method — it is available to use inside the sub-class or it can be invoked from an instance of the sub-class
3. **Override** the method — You must declare the method with exactly the same method declaration line, the same **signature**, as was used in the super-class
4. **Overload** the method in the super-class
5. **Refine** the method by invoking the super class’s method in an overridden method

Inheriting Methods

- As we have discussed, all public methods from the parent class are inherited by the child
- Of course, the parent class may have inherited a bunch of methods from its parent class, so every public method in the hierarchy above the current class is available
- They behave just as if they were written in the child class
- Note that an abstract sub-class can simply inherit an abstract method from an abstract super-class or an interface — it does not have to implement the method. In this case, the sub-class will have to be abstract as well
- suppose you don't want all these methods — How can you avoid inheriting some but not others?

Method Overriding & Refining

- When a sub-class overrides a method of the super-class, then it is the sub-class version that is invoked when called from an instance of the sub-class
- Inside the sub-class, if you wish to invoke the super class version of the overridden method, use the super word
- If the overriding sub-class method invokes the super-class version of the same method, then it has **refined** the super-class method, as well as overriding it

the Use of the final Modifier

- If "final" is included in a class header line, as in:
  public final class ClassName { … }
- Then the class cannot be extended
- If you add "final" to a method definition as in:
  public final void methodName() { … }
- The method cannot be overridden

the Effect of "private"

- private methods and attributes are not inherited by the sub-class
- But, you do inherit public methods and attributes
- You can invoke public methods inherited from the parent class even when they themselves refer to attributes and methods that are private in the parent class

Overloading Methods

- If you use the same method name and return type as in the parent class, but a different parameter list in the sub-class then you are just overloading the inherited method

Multiple Inheritance

- This is when a sub-class extends more than one super-class
- In our diagrams, a sub-class would point to more than one super-class
- Java does not support multiple inheritance (C++ allows it)
- Designers of Java felt that multiple inheritance would lead to structures that are way too complex
- Java designers have supplied the use of interfaces as a way of getting around this restriction
- A sneaky way to get around the lack of multiple inheritance in Java is to combine two classes into one, by making one of them an Inner Class, and then you extend the outer class

Polymorphism, Second Pass
- Where one type can appear as itself, but ends up being used as another type
- Java is a very strongly typed language! Even so, you can allow one object to appear to be something else
- Polymorphism myst be constructed through object extension or interface implementation

Polymorphism: Late and Early Binding
- Your program must always satisfy early binding for it to compile
- Late Binding occurs when the program is running, and only occurs when a variable of a parent-type object is pointing to a child object
- The compiler does not (and cannot) check the late binding to see if it works. A failure of late binding results in a runtime error, not a compiler error

Visualization of Hierarchies
- The easiest way is to view the structure is as a Unified Modelling Language Class Diagram
- This is what we have been doing, but normally more detail is shown

The ArrayList<T> Class
- This is a **generic** data structure class
- The ArrayList<T> class resides in the java.util package. Stores and returns Objects of type T
- You can use the class without specifying a size

Syntax:

```
ArrayList<type> listName = new ArrayList<type>();
```

- type must be an Object, it cannot be a primitive type but can be an array type
- You can supply an initial size to the constructor, if you wish
- In newer versions of Java you do not have to specify the type twice

```
ArrayList<type> listName = new ArrayList< > ();
```

- The empty < > is called the “diamond”
- This is an aspect of what is called **“type inference”**

- Add elements to the collection: myList.add(new Double(456.78)); OR
  myList.add(456.78); //Automatic Boxing
- To get the size of the collection, invoke the size() method:

```
int myListSize = myList.size();
```

- The get(index) method returns the Object at the given index. Note that index positions are numbered from zero
- The set(position, newValue) method changes the Object at the given position
- To insert an element, invoke add(position, newValue). All elements are moved up, and the new value is added at the given position
- The method, remove(position) removes the element at the provided position

- If you are done adding elements to the ArrayList, invoke **trimToSize()** to remove empty element locations
- Also **clone()** DOES NOT WORK properly because it does not return an independent, un-aliased copy. The objects in the collected will still be aliased.

- toArray(tArray) returns the underlying array of type T[ ]. Supply tArray which is an instantiated array of type T. The size will be increased if necessary, so just use a size of zero

Ex. If T is String then invoke as in:

```
arrayList.toArray(new String[0])
```

- This method returns the array String[ ] of the exact size needed to hold all the element in the arrayList

- Once you have accessed an element of an ArrayList<T>, you do not have to cast it back to type T, it is already of the correct type

Ex. To get the number back out of the first element in myList:

```
double aVal = myList.get(0);
```

- Uses **automatic boxing** as well
- This is an advantage over using a collection like Object[ ] (for example)

Your Own Generic Classes

- Also called **"Parameterized Classes"** — kind of like having a parameter in the class header, except it only specifies a type and does not pass a reference, like a parameter does

Ex.

```
public class Sample<T> {
        private T data;
        public void setData(T newData) {
                data = newData.clone();
        }
        public T getData() {
                return data;
        }
}//end sample<T>
```

(Review examples in notes)

Generic Classes: Limitations for T

- You can use an array type, so "<int[ ]>" would be legal
- You cannot specify primitive types, so "<int>" would not be legal
- Neither can you use Exception classes
- You CAN use interface and abstract types

Your Own Generic Classes: Constructor

A construct for Sample<T> class:

```
public Sample (T newData) {
        data = newData.clone();
}
```

NOTE: that there is not use of <T> in the constructor

Using the constructor and automatic boxing:

```
Sample<Double> num3 = new Sample< >(56.7);
System.out.println(num3.getData());
```

- You can specify any number of type parameters:
  public class Sample2<T1, T2, T3> {…
- You can also "bound" the parameters by specifying a root class:
  public class Sample3<T extends RootClass>
- So only classes that extend RootClass can be used for T (in this example)

Generics and Backwards Compatibility
- Generic code can make for very versatile programs that can be applied to a range of objects — saving you a great deal of boring coding work
- But generic code will not work easily with pre Java 5.0 Code
- and of course, the Java 7 type inferences stunts will not work with pre-Java 7 code
- Generics are a bit like templates in C++ and C#, but are implemented in a different way

Wildcards
- The ?? within < >
- A wildcard can provide a super type generic class for other generic classes
- You can bind a wildcard to classes that extend other classes using the extends keyword

Ex.

  ArrayList<? extends Person? db
- db can be assigned to an object of type ArrayList<Person>, ArrayList<Student> or ArrayList<Professor>
- However, a generic typed with a wildcard is not mutable
- You can get around this using generic methods without the wildcard BUT:
  - This is more restrictive and potentially "brittle"
  - Avoid casting using the generic type
  - This may lead to "Unchecked casts" which reduces the type safety of the generic method
  - Use of Class<T> may be a way around this problem
  - ArrayList<?>[ ] is the only possible type that allows the creation of an array of generic classes
  - But why would you want to? —> easier to create an ArrayList of ArrayList for example

The Class<T> Object
- You can pass a type as an argument into a method using a Class<T> object
- The Object class contains a method .getClass() that returns a Class<T> object. Or you can just use the .class literal on a type

Ex. String aString = "hello!";
System.out.println(aString.getClass());
System.out.println(String.class);
SHOWS: class java.lang.String
  class java.lang.String

- Also has a method called **.newInstance()** that returns an instance of type T
- This method can only invoke the default constructor of the type T. So you must use mutators to set attributes. But now you have an instance to work with!
- In a generic method, the type "T" cannot be instantiated

A Lambda Function

- Kind of like an "anonymous method"

Syntax

- parameters for the method first, no parameters, use { }, one parameter by itself — no brackets required, more than one use (a, b, etc) No types required
- Then the fancy -> "arrow"
- Then the method body. Put more than one statement inside { }

- Can even define an inner interface

Ex.

```
public class LambdaDemo {
    interface MessageSender {
        bond sendGreeting(String aName);
    }//end MessageSender Interface
    public static void main(String[ ] args) {
        MessageSender ms = name -> System.out.println("Hello " + name);
        ms.sendGreeting("Steve");
    }//end main
}//end LambdaDemo
```

- NOTE: how abstract method in interface determines the structure of the lambda function — the parameter and parameter type, the method name and the lack of a return statement
- These functions could be useful (especially when attaching events to GUI components)
- Only certain interface structures can be used
- Suppose you have a method the only displays certain members of a collection, depending on a criteria that is specified outside the method and provided as an argument
- You don't want to hard code the criteria in the display method

- FIRST technique: Supply an object implementing an interface that has a "filter" method that returns a true or false.
- SECOND technique: Use an anonymous class instead
- THIRD technique: Use a Lambda function

But how does it work?

The structure of the method implemented by the lambda function is specified by the interface type used in the displaySome method. The signature is:

```
boolean check(Person)
```

- The compiler knows from this that the type to the left of the -> must be a Person and the expression to the right of the -> must return a boolean

- Filter is an example of a **functional interface**
- These interfaces can only contain a single abstract method
- Lambdas can only be created using Functional interfaces
- You can use the @FunctionalInterface annotation to make sure your interface is OK

- You can have multiple lines of code in a lambda, but don't make them too long
- You can use as many lambdas as you wish when invoking a function, as long as they each match up to some functional interface
- In GUI programming the most common event handler interface, EventHandler<ActionEvent> is a functional interface so lambdas can be used to attach event code to handlers

Pre-Defined Functional Interfaces

- Turns out java.util function package has many pre-defined generic functional interfaces
- The one that matches our check function signature is called Predicate<T>. It has the abstract method signature: boolean test(T)

Method References

- ArrayList.sort() accepts a Comparator<T> object that can specify the desired algorithm for comparison of objects of type T
- Turns out Comparator is a FUNCTIONAL INTERFACE, so you can build a lambda function for a comparator

- But suppose our Person class has a method that matches the Comparator interface:

```
public static int comapreByAge(Person p1, Person p2) {
        return p1.age - p2.age;
}
```

- in this case, you can supply a METHOD REFERENCE instead of building a lambda function

```
db.sort(Person: : compareByAge);
```

- The method reference must match the functional interface's specification
- You can use non-static methods or methods from an instance. If you wish to supply a constructor, use:

```
ClassName: : new
```

- and you can use existing API methods