Java Fundamentals

Course Overview

The Introduction to Java training course provides students with a foundational knowledge of the Java platform and Java language required to build stand-alone Java applications. This course assumes students have a background in another programming language. The course begins with defining and introducing the Java programming language. The course then dives into defining basic Java syntax and creating a stand-alone Java application. The course also covers some commonly used Java Collection classes, and Unit Testing.

Topics covered

- What is Java?
- Basic Java Syntax
  - Class Structure.
  - Data types.
  - Operators.
  - Control Flow.
  - Arrays.
  - Variable argument methods.
  - Packages and access modifiers.

Topics covered

- Object Oriented Programming in Java
  - Introduction to OO concepts.
  - Subclassing and inheritance
  - Interfaces
  - Default methods
  - Abstract classes
- Functional interfaces
  - Lambdas
- Enumerations

Topics covered

- Static Class Design
  - Static variables.
  - Static methods.
  - Static initializers.
- Exceptions in Java
  - What is an Exception
  - Runtime vs. Checked Exceptions.
  - Throwing and catching Exceptions.
  - Creating custom Exceptions.

Topics covered

- Java Collections
  - Understanding the collection types.
  - Choosing between List, Set or Map.
  - Creating type safe collections with Generics.
  - Searching and converting collections.
- Unit Testing
  - Testing with JUnit 5.

What is Java?

- Created in 1995
  - Originally called Oak.
- Type safe and interpreted.
  - Compiles down to an intermediate representation called **Byte Code.**
  - A **Java Virtual Machine (JVM)** is required to interpret and run the byte code.
- Upside is that Java programs can run on any platform for which someone has written a virtual machine.
  - “Write once, run anywhere” ™.
- Downside is it that interpreted languages can be slower than compiled languages.
  - Enter the **Just in time compiler (JIT)**

What is Java?

- Downside is it that interpreted languages can be slower than compiled languages.
  - Enter the **Just in time compiler (JIT)**
  - The JVM can compile the bytecode down to native code as it is interpreting it.
  - This allows it to concentrate on compiling those parts of your code where your application spends a lot of time – i.e. those parts which are *hot.*
    - The oracle JVM is sometimes called **Hotspot.**
- Many other languages can also be compiled down to byte code and run on a VM:
  - Scala, Kotlin, Python, Ruby etc. etc.
- So what is Java? Is it the source language we are going to get introduced to in this class, or is it the byte code?

Java Nuts and Bolts - Types

- Strongly typed language
  - All variables have a type which has to be declared before they can be used.
- All code has to be in some Class.
- 8 **built in** or **primitive** types, all with defined sizes
  - boolean, byte – 1 byte
  - char (for character data), short – 2 bytes
  - int, float – 4 bytes
  - long, double – 8 bytes
- Can also have arrays of them.
- Every other type is known as a **reference** type. References point at objects you create on the **heap** using the **new** operator.
- More on that when we get to talking about Classes.

Operators

- Operators
  - The usual bag of suspects
    - +, -, %, / etc.
    - ++, –– (prefix and postfix)
  - bitwise operators
    - <<, >>, >>>
    - &, |, ^
  - logical operators
    - &&
    - ||
  - comparision
    - ==
    - !=
  - assignment
    - =
    - +=, -= etc.
      - a += 2 is the same as a = a + 2
  - ternery
    - ? :

Variables

- All variables have a type, which ***has*** to be declared before the variable can be used
  - int someVar = 10;
- Literals also have a type
  - 10 – int
  - 10L – long
  - 10.5 – double
  - 10.5F – float
  - ‘c’ – char literal
  - “This is a String Literal” – Strings are ***not*** primitive types. More on this after we have discussed classes.

Variables contd

- Strings look like they are a primitive type because we can initialize them with
  - String message = “hello”;
- But appearances can be deceiving.
- More on this when we get to classes.
- For now, it is important to remember that when comparing Strings for equality, you have to use an **equals** method
  - String x = “hello”;
  - String y = “hello”;
  - if(x.equals(y)) { … }
    - if(x == y) //Bad, wrong, do **NOT** do this

Flow Control

- if(...) {...} else if(...) {...} else {}
- while(something is true) { ... }
- do {...} while(something is true);
- for(initialization; test; change conditions) { ... }
- for(somevar : someCollection) { ... }
- switch(var) {

  case ‘x’: ...

  case ‘y’: ...

  }
- break and continue for controlling loop behaviour

Arrays

- Arrays are our first introduction to **reference** types.
- Two parts to a reference
  - The variable of reference type that you declare in your program
    - int [] iarr, or int iarr []
    - And the actual object that the reference is referring to
  - Very good to always remember that the reference and the object **are NOT the same thing.**
- Objects are created using the **new** operator
  - iarr = **new** int[10];
  - have to provide a size when you create an array.

Arrays contd.

- All arrays have a length property
  - for(int i = 0; i < iarr.length; i++) {iarr[i] = I;}
- Arrays can be initialized when created. Useful if you have a small set of values that you know about at compile time.
  - int [] iarr = {0, 10, 25 };
  - String [] messages = { “Yes”, “No”, “Maybe” };
- The String case is interesting because Strings are also reference types. We will dig into it a bit more when we look at classes

Methods

- All Java code has to be in some method, which in turn has to be in some Class.
- Method declarations
  - return_type method_name(zero or more arguments) { … }
  - int someMethod(int value, int upperLimit) {

    if(value > upperLimit) return upperLimit else return value

    }
- Void return type means the method returns nothing
  - void methodWithSideEffects() { … }
- Methods taking and/or returning arrays
  - String [] methodReturningStringArray(int [] iarr) { … }
- Methods can have visibility modifiers. For now, we just use **public.** More on this later.
- Methods can optionally be **static.** For now, all our methods will be **static.** More on this later too.

Methods contd.

- Methods can have a variable number of arguments.
  - void methodWithVarArgs(int size, **String ...messages**) {…}
  - varargs argument has to be the last argument in the method.
  - Can be called in various ways
    - methodWithVarArgs(10, “OK”, “NOT OK”);
    - methodWithVarArgs(10, “OK”);
    - methodWithVarARgs(10);
  - Accessed inside the method as an array
    - int numMessages = messages.length;
    - Array will have zero length if no varargs are passed

Methods

- Methods can be **overloaded**.
  - Two methods can have the same name as long as they differ in the number and/or type of arguments.
  - int doSomething()
  - int doSomething(int x)
  - double doSomething() – **NOT** a valid overload, will be a compile error – the return type is not used to determine validity.

Classes and Objects

- Object Oriented languages allow us to model concepts and things in the problem space your are addressing
- A **Class** is a description of a particular **type** (i.e *class*) of things that you want to model
  - class Student { … }
  - class Customer { … }
  - class Order { … }
  - class LineItem { … }
- You create an **instance** of a class by using the **new** operator
  - Student s = new Student(…);
- The new instance is also called an **object.**

Classes and Objects

```
public class SimpleCircle {

   int posX, posY;
   int radius;

   double getArea() {
      return Math.PI * radius * radius;
   }

   double getPerimeter() {
      return 2 * Math.PI * radius;
   }
}

class ShapeApp {
   public static void main(String [] args){
      Circle c = new Circle();
      c.posX = 10;
      c.posY = 11;
      c.radius = 25;

      double area = c.getArea();
   }
}
```

Class declaration

Properties or variables or instance variables

Methods or instance methods or behaviors

Create instance (object) with **new** operator

Set properties of instance

Call methods on the object, or invoke behaviors

Classes - Encapsulation

- **Encapsulation** is a very common OO technique.
  - It turns out that it's useful to hide the insides of a class from the outside world.
  - Only allow outside code to access specific parts of your class
    - Usually through methods, though not always.
  - Allows you to change the way our class is implemented without affecting outside code.
  - Also allows you to have control over how the state of your objects will change. Users have to come through your code to change the object, allowing you to do validation.
  - You can create **immutable** objects by not providing any public way to change the state of an object.
  - Java visibility levels – **public**, **protected,** package visible, **private**

```
public class EncapCircle {

   private int posX, posY;
   private int radius;

   public int getPosX() {
      return posX;
   }

   public void setPosX(int posX) {
      this.posX = posX;
   }

   //other get/set methods
}

class EncapApp {
   public static void main(String [] args) {
      EncapCircle c = new EncapCircle();
      //c.centerX = 10;
      c.setPosX(10);
      c.setPosY(11);
      c.setRadius(25);

      double area = c.getArea();
      System.out.println("Area: " + area);
   }
}
```

Encapsulated properties

Compile Error

Use setters to set properties

Classes - Constructors

- **Constructors**
  - Making your users call several set methods to initialize an object is inconvenient at best.
  - But a worse possibility is that they forget to set a necessary property. Now you have an object in a bad state. Not good.
  - Constructors to the rescue.
  - Special methods used to initialize objects. Two important rules for being a constructor
    - **The name of the method has to be exactly the same as the name of the class** and
    - **No return type**
  - Users have to call the constructors based on your arguments, and so can’t just forget to pass in required variables.
  - You can do validation on all the data before initializing the object.

```java
public class ConstructCircle {

    private int centerX, centerY;
    private int radius;

    public ConstructCircle(int x, int y, int r) {
        centerX = x;
        centerY = y;
        radius = r;
    }

    //Other methods

}

class ConstructApp {
    public static void main(String [] args) {
//        ConstructCircle c = new ConstructCircle();

        ConstructCircle c = new ConstructCircle(10, 11, 25);

        double area = c.getArea();
        System.out.println("Area: " + area);
    }
}
```

Same name as the class and no return type makes this method a Constructor.

Compile Error. Because of your custom constructor, compiler does not make a zero argument constructor.

Constructor call. Have to pass in all data that the class requires to be properly initialized.

Classes - Constructors

- Constructors can be overloaded
  - Compiler will choose the constructor to call based on the arguments supplied.
- Often useful to call another constructor of the same class, to do all the initialization in one spot.
  - use the **this** keyword to call another constructor of the same class.
  - **this** call has to be first. Other code can follow.

```
public class TwoConstructorsCircle {

   private int centerX, centerY;
   private int radius;

   //create a Unit circle
   public TwoConstructorsCircle(int x, int y) {
      //call 3 argument constructor
      this(x, y, 1);
      //maybe other code
   }

   public TwoConstructorsCircle(int x, int y, int r) {
      centerX = x;
      centerY = y;
      radius = r;
   }

   //other methods
}

class TwoConstructorApp {
   public static void main(String[] args) {
//       ConstructCircle c = new ConstructCircle();
      TwoConstructorsCircle c = new TwoConstructorsCircle(10, 11);
      //c.centerX = 10;

      double area = c.getArea();
      System.out.println("Area: " + area);
   }
}
```

Overloaded constructor. Will call the other constructor to create a circle of radius 1. **this** call has to be the first thing in the constructor.

Call 2 argument constructor to create a circle with radius 1.

Classes – Object Initialization

- 3 steps to initialization of variables in an object
  - set to default values for the type
    - 0 for integral types
    - null for references
    - false for boolean
  - Explicit initialization when declaring the instance variable
    - int x = 10
  - Lastly, initialization in a constructor

Classes and Objects - this

- ***this*** is an implicit reference that is passed in to all instance methods.
  - You never declare an argument called *this*, but you have access to it in all instance methods and constructors.
- Instance methods have to be called on behalf of some instance.
  - MyClass mc = new MyClass();

    mc.doSomething()
- The *doSomething()* method above is called on behalf of the *mc* object.
  - In the method, the **this** reference will be available and point at the same object that mc is pointing at
    - i.e. this == mc for this particular call

Classes and Objects

- The **this** reference is not always needed.
- You will typically see **this** when using the same names for method arguments and instance variables.

```
public class TwoConstructorsCircle {

    private int posX, posY;
    private int radius;

    public TwoConstructorsCircle(int posX, int posY, int radius) {
        this.posX = posX;
        this.posY = posY;
        this.radius = radius;
    }
```

Argument names are same as class variable names. Useful as documentation, but now we have to disambiguate the instance variables from the arguments.

Use **this** to disambiguate.
this points at the object being constructed, so the variables on the LHS are the instance variables

Classes - static

- It can be useful to have properties and methods associated with the **Class** as a whole, rather than with each instance.
  - e.g. to do operations that apply to all instances, or to no instances.
  - Sometimes you want to have some code which is totally *stateless* i.e. it gets everything it needs to do it's work as arguments, making instances will be useless. Look, for example, at **java.lang.Math**
- For such situations, **static** is your friend.
  - double s = **Math.sin(25)**
    - *sin* is a static method in the Math class – no need to make an instance of Math to call it. Would be a useless and wasteful thing to do.
- So **static** things are accessed through the class itself, rather than a particular instance.
- **static** variables can be initialized in a **static block** which is called when the class is loaded in, possibly much before any instances are created.

```
public class CountCircle {

    private int centerX, centerY;
    private int radius;
    private static int instancesCreated;

    public CountCircle(int centerX, int centerY, int radius) {
      this.centerX = centerX;
      this.centerY = centerY;
      this.radius = radius;

      instancesCreated++;
    }

    public static int getInstancesCreated() {
       //int x = this.centerX;
       return instancesCreated;
    }
    //other stuff
}

class CountCircleApp {
    public static void main(String [] args) {
       int firstCount = CountCircle.getInstancesCreated();
       CountCircle c = new CountCircle(10, 11, 25);

       int secondCount = CountCircle.getInstancesCreated();
       System.out.println("fc: " + firstCount + ", sc: " + secondCount);
    }
}
```

static property

static method.
Compile error on first line because no **this** pointer in static methods, because no instance is involved.

Called through the class. Can be called before any instances are created.

Classes and Objects - Arrays

- Arrays of reference types have to be handled differently from primitive arrays.
- The elements of such arrays are themselves **references**.
- You have to also make those references point at new or existing objects.

```
class CountCircleArrayApp {
   public static void main(String [] args) {
      CountCircle [] carr = new CountCircle[10];
      //int x1 = carr[0].getCenterX();  NullPointerException

      for(int i = 0, x = 10, y = 10; i < carr.length; i++, x+=5, y+=5) {
         carr[i] = new CountCircle(x, y, 2);
      }
      //No NPE here
      int x1 = carr[0].getCenterX();
   }
}
```

An array of 10 CountCircle references. The references are initially **null.**

You have to make sure the references are pointing at objects before you use them.

Detour - packages

- All java classes should be put into a package.
- **package** declaration should be the first line in the class.
- Two reasons to use packages
  - As a **name spacing** mechanism.
  - To help organize your code.
- The first reason is the more important one
  - The **fully qualified name** of a class is the package name followed by the simple class name
  - Which makes it possible to have two classes with the same name as long as they are in different packages.
- Packages also have an effect on the directory structure of your application.
  - directory hierarchy has to follow the dot separated package names
  - e.g org.xyz.javacourse.Student.class should be in org/xyz/javacourse/Student.class

Classes and Objects - Inheritance

- In the real world, different **types** of objects have common properties and behaviors
  - Checking Accounts and Savings Accounts are both different **types** of Accounts. They share attributes like balance and behaviors like deposit and withdraw
  - Circles and Squares and Triangles are all different **types** of Shapes. They share common attributes like color, line thickness, position etc.
- **Inheritance** is a technique used in many OO languages to allow you to express such **is-a** relationships in code.
- When a class **extends** from another class, it inherits (acquires) all the non private properties and behaviors of that class
  - the class that extends is a **sub** class, the class that gets extended is the **super** class
  - Other languages refer to them as the derived class and the base class.

Classes and Objects - Inheritance

- Sub classes can either use inherited behavior directly, or they can selectively **override** behaviors of the super class.
- You think of the sub class as a **specialization** of the super class.
- Examples in code.
- Inheritance provides us with a feature called **polymorphism**
  - Many forms
  - A sub class object has several types
    - It’s own type
    - And the types of all of it’s super classes.
    - It can be used when any of it’s types is required.
- Code reuse is fine, but Polymorphism is the more interesting consequences of inheritance.

Inheritance – Object methods

- All classes eventually inherit from **Object**
- 3 Object methods that are often overridden
  - **toString** – provide a String representation of an object
  - **equals** and **hashcode.** Used for various equality tests. If you implement one, you should implement the other.
  - Look at the documentation for constraints on implementation.

Classes and Objects – Abstract Class

- Sometimes you would like to extract common properties and behaviors into a super class for the convenience of having a base type.
- But the class with the extracted properties does not really exist in the real world
  - e.g the class Shape does not represent any particular shape in the real world. Those are all Circles and Squares and Triangles etc.
  - If you were to draw an instance of Shape, what would it look like?
- Shape is an *abstraction*. The way to express that in Java is to make it an **abstract** class. Then you can't make instances of Shape any more.
- An abstract class can have implementations for methods where a reasonable default implementation is possible. Other methods are declared as **abstract**, which means they *have* to be implemented by sub classes.
  - abstract public double getPerimeter();

Classes and Objects – Interfaces

- **Interface**s carry the idea of abstraction one step further.
- They are generally used to express a *contract* for the behaviors an instance is expected to have.
  - The **List** interface specifies the behaviors that anything that says it is a List **has** to have.
  - The org.ttl.javafundas.interfaces.Shape interface specifies all the behaviors that all Shapes **have** to have.
- A class extends other classes, but **implements** interfaces.
- Interfaces can have
  - **abstract** methods – no implementation
  - **default** methods – Java 8+. Default methods **must** have an implementation.
  - **static** methods – Same as for classes.
- Difference between Interfaces and Abstract classes is that Interfaces *cannot declare state.* No instance variables.

Implementing Interfaces

- 3 ways to implement interfaces
  - As a full class implementation
  - As an anonymous inner class
    - Requires slightly less syntax
    - Useful for one-off implementations
    - You declare the implementing class and the interface in one fell swoop.
  - As of Java 8, using Lambdas
    - Lambdas can be only be used to implement interfaces that have **just one abstract method.**
    - **Functional Interface** – new name for any interface which has just one abstract method.
      - **@FunctionInterface** – Can use this annotation both as documentation and for compiler enforcement of 1 method rule.
  - Lambdas are most useful for small implementations.

Interfaces – Full class implementation

```
interface Checker {
    public boolean check(Shape shape);
}

/**
 * As a full fledged class
 */
class FullClassAreaChecker implements Checker {
    @Override
    public boolean check(Shape shape) {
        return shape.getArea() > 100;
    }
}


class InterfaceApp {
    public static void main(String [] args) {

        Circle circle = new Circle(20, 20, 20);
        //Use the full class implementation
        Checker fullChecker = new FullClassAreaChecker();
         boolean r1 = fullChecker.check(circle);
        System.out.println("Circle1 area > 100: " + r1);
    ...
}
```

Interfaces – Anonymous class

```
interface Checker {
    public boolean check(Shape shape);
}

class InterfaceApp {
    public static void main(String [] args) {

        Circle circle = new Circle(20, 20, 20);

        //Implement Interface as anonymous class
        Checker anonymousChecker = new Checker() {
            @Override
            public boolean check(Shape shape) {
                return shape.getArea() > 100;
            }
         };

        boolean r2 = anonymousChecker.check(circle);
        System.out.println("Circle2 area > 100: " + r2);
```

Interfaces – Lambda

```
@FunctionalInterface
interface Checker {
    public boolean check(Shape shape);
}

class InterfaceApp {
    public static void main(String [] args) {

        Circle circle = new Circle(20, 20, 20);
        //Lambdas can only be used to implement
        //interfaces with 1 abstract method.
        //Lambdas give you the most concise
        //syntax.  Most useful for small implementations

        Checker lambdaChecker = (Shape s) -> { return s.getArea() > 100; };
        Checker lambdaCheckerToo = (s) -> s.getArea() > 100;

        boolean r3 = lambdaChecker.check(circle);
        System.out.println("Circle3 area > 100: " + r3);

    ...
}
```

Optional, but makes compiler enforce “One Function” rule. And serves as documentation.

Basic syntax for lambda is:
(arg1, …) → { code ; }

Syntax cleanup
1) Argument types can often be omitted. Compiler can infer them from context.
2) If only 1 expression in the lambda, then squiggly braces and *return* statement can be omitted. The result of the expression is the result (and type) of the lambda.

Enums

- Type safe Enumerations
- Useful if you have a variable that should only take on one of a restricted set of values.
  - Days of Week
  - Customer Statuses
- Create a new *type*, just like a class.
  - But the instances of the class are restricted to the set of enum members.
- Some useful enum methods
  - **toString** – get String representation of Enum
  - **valueOf** – convert String into Enum instance
  - **values** – returns an array of all the Enum values

Enums

```
public enum ColorEnum {
    GREEN,
    RED,
    BLUE,
    BLACK;
```

Enum values

Overridden **toString** method to get nicer String representation.

```
    public String toString() {
        String str = name();
        return str.substring(0, 1).toUpperCase() + str.substring(1).toLowerCase();
    }
}
```

Use **valueOf** method to convert String into Enum.

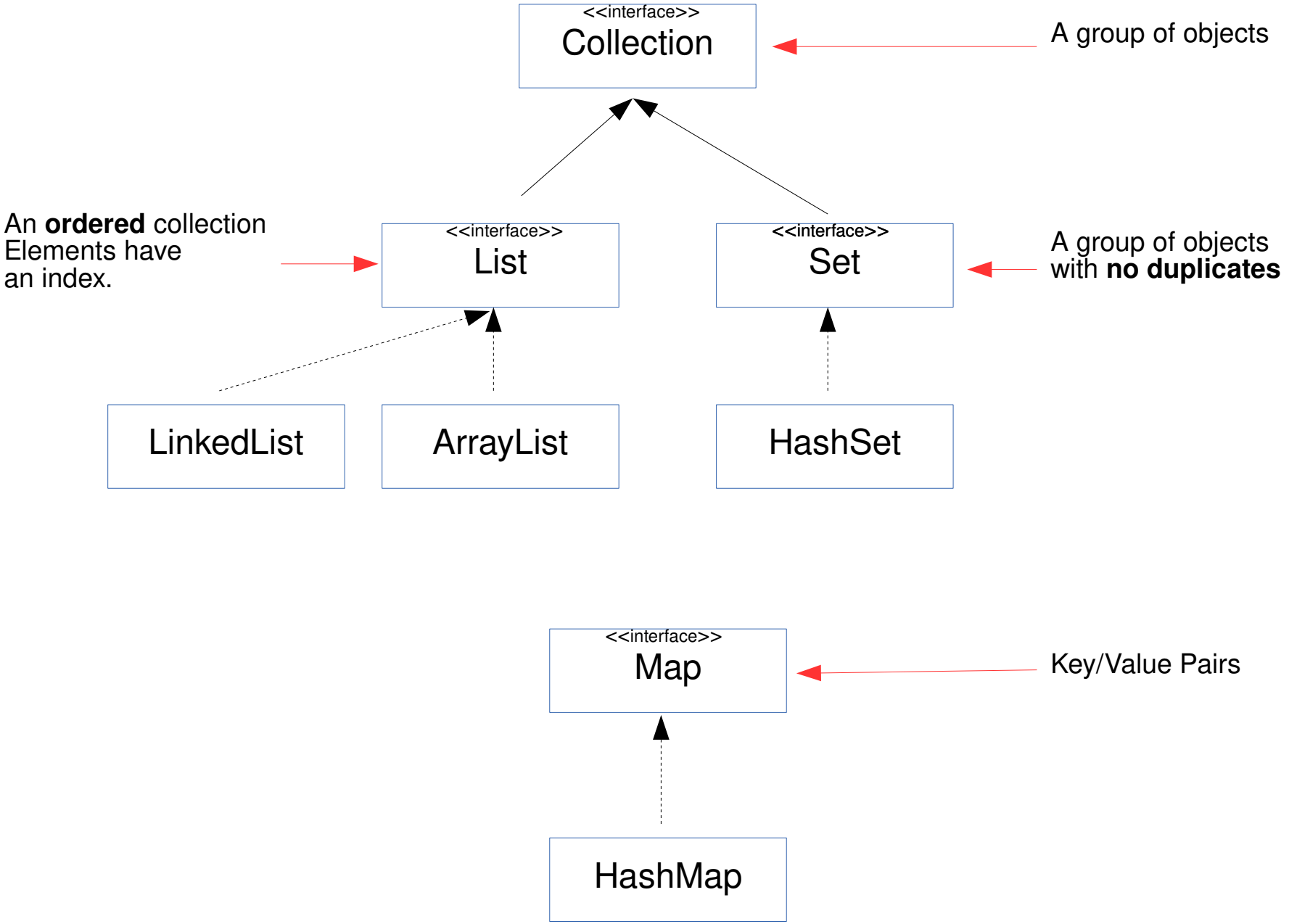
```
@Test
public void testColorEnumsValueOf() {
    String strColor = "Green";

    ColorEnum color = ColorEnum.valueOf(strColor.toUpperCase());

    assertEquals(ColorEnum.GREEN, color);
}
```

Collections

- Arrays have some limitations
  - Have to declare the size in advance
  - Have to keep track of how much of the array you are using currently.
  - Have to create a new array and copy when you run out of space.
- Collection classes take over these responsibilities.
  - No need to declare size on creation (though you can).
  - They keeps track of current number of elements.
  - Resize automatically if necessary.
  - Offer different interaction patterns.

Collections

- Base collection code is in **java.util**
- Implemented using Inheritance hierarchies
- All rooted in Interfaces which define the contracts that implementations will have to satisfy



Collections and Generics

- **Generics** are used to enforce **compile time type safety** in Java.
- Can be used everywhere, but become specially important and necessary when working with Collections.
- General rule of thumb – **never** use a Genericized class without supplying the appropriate type parameter
  - List<**String**> list
  - Map<**LocalDate, Integer**> dobMap
  - Map<**Integer**, **Set<String**>> uniqueFileSizes
- **Don't do this:**
  - List list
  - Map dobMap

Exceptions

- Exceptions are a way of signaling error conditions
- Two types of Exceptions
  - **checked** – The compiler *checks* to make sure you are handling checked Exceptions
    - surround in try/catch block
    - or declare in a method Exception specification
  - **unchecked** or **Runtime –** The compiler does not check.
    - if a Runtime Exception is thrown and the code is not dealing with it, the program dies.
    - You can throw Runtime Exceptions from anywhere without having to declare them.

Exceptions

- Basic elements
  - **try** – you surround code that might throw Exceptions in a try block
  - **catch –** you can have one or more catch blocks after a try block to handle the Exceptions that may get thrown
  - **throw –** use this to throw an Exception. The JVM will start to unwind the call stack, looking for a catch block that can handle the Exception.
    - If one is found, it will be used to handle the Exception.
    - If not, the call stack will unwind all the way to *main*, and your program will end.
  - **throws –** method Exception specification. Required for **checked** exceptions that are not caught in a method
  - **finally** – A block of code that will be run no matter how you exit the try block. Useful for doing resource clean up if necessary.
  - **try with resources** – an alternate syntax for the try block which can automatically close resources for you.

JUnit

- Library to create Unit tests.
- JUnit 5 is current version
  - https://junit.org/junit5/
- Uses annotations to configure test.
  - @Test
  - @BeforeEach/@AfterEach
  - @BeforeAll/@AfterAll
- Uses assertion to specify the expected outcome of a test
  - assertEquals(…)
  - assertTrue(...) / assertFalse(...)
  - assertNull()... / assertNotNull(…)
  - assertThrows(Exception, Executable)

The End