StringBuffer and StringBuilder

Since Strings are immutable, it becomes inefficient to use them if we are making many new Strings constantly, for example if we are generating new Strings in a `for` or `while` loop. Instead, we can use the `StringBuilder` and `StringBuffer` classes, both of which are **mutable**. `StringBuilder` contains helpful methods like `.append()` and `.insert()` which mutate its internal sequence of characters. `StringBuffer` is like `StringBuilder` but is synchronized, which is useful for multi-threaded applications.

Class	**Immutable?**	**Thread-safe?**	**Speed**
String	Y	Y	Slowest
StringBuilder	N	N	Fastest
StringBuffer	N	Y	Fast

`StringTokenizer` is a related class which can parse a String and splits it based on a delimiter.

Arrays, Var-args, for-each loops

Arrays

An array is a contiguous block of memory storing a group of sequentially stored elements of the same type. Arrays in Java are of a fixed size and cannot be resized after declaration. Arrays are declared with square brackets after the type of the array like so:

```
int[] myInts = new int[]{1, 2, 3, 4};
String languages[] = {"Java", "JavaScript", "SQL"};
```

Items in an array are referenced via their index in square bracket notation, which begins with `0` for the first element. Arrays also have a `length` property specifying the length of the array. This is helpful when iterating over arrays with a `for` loop:

```
String[] myArr = {"first", "second", "third"};
for (int i = 0; i < myArr.length; i++) {
  System.out.println(myArr[i]);
}
```

Varargs

Instead of writing our `main` method the standard way, we can use an alternative notation:

```
public static void main(String... args) { }
```

Here we are using the `varargs` construct `...` which replaces the array notation. `varargs` stands for "variable arguments", and allows us to set an argument to a method whose size is determined at runtime. Java will create an array under the hood to fit the arguments provided. You can only ever have 1 varargs parameter in a method, and it **MUST** be the last parameter defined (otherwise, how would the JVM know the difference between the last value in varargs and the next parameter of the method?). You can omit the vararg value when invoking the method and Java creates an array of size 0.

```
public class VarargsExample {
  public static void someMethod(int a, int... manyInts) {
    System.out.println("First argument: " + a);
        System.out.println("Next argument: ");
        for (int i = 0; i < manyInts.length; i++) {
          System.out.println(manyInts[i]);
        }
  }

  public static void main(String[] args) {
    VarargsExample.someMethod(1, 3, 4, 5, 6);
        // First argument: 1
        // Next argument:
        // 3
        // 4
        // 5
        // 6
  }
}
```

forEach() Method

We can use the `.forEach` method of the `Iterable` interface, which accepts a lambda expression as its argument:

```
List<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");
names.add("Charlie");
names.forEach(str -> System.out.println(str));
```

This will print out the names just as if we had used a `for` loop. The lambda syntax could also be done with an explicit type declaration for the parameter, but the compiler can infer the type from the value used. For multiple parameters, parentheses are required around them. Also, curly braces are optional for single statements but required for multiple. Finally, the `return` keyword is also optional for a single expression because the value will be returned by default.

`.forEach()` method

The `forEach()` method actually accepts what is called a functional interface as its parameter (specifically a `Consumer`), which the lambda expression then implements at runtime. The `forEach()` method then loops through `names` and passes each element to the lambda expression to be "consumed".

Annotations

Java annotations are special constructs you may see throughout Java code, which use the `@` symbol followed by the name of the annotation. These annotations provide metadata about the source code to the compiler and the JVM. They can be placed on classes, methods, interfaces, and other constructs - however some annotations are restricted to only being placed on certain types or class members.

Annotations can be used to enforce rules in the code, or to abstract some functionality provided by a library or framework. Java frameworks and libraries often process annotations using the Reflection API (covered in another module) to dynamically provide functionality to developers.

Java has a few built-in annotations you should be familiar with:

- `@Override` - declares the method must override an inherited method (otherwise, a compilation error occurs)
- `@Deprecated` - marks a method as obsolete (compilation warning if used anywhere)
- `@SuppressWarnings` - instructs compiler to suppress compilation warnings
- `@FunctionalInterface` - designates an interface to be a functional interface (covered in another module)

Wrapper Classes

Wrapper classes are classes that let you treat primitives as Objects. This is necessary - for example - for certain methods which only accept objects and not primitives. **Boxing** is the process of converting a primitive to its wrapper class. Java has a feature called **autoboxing** which will automatically convert primitives to wrapper classes implicitly. **Unboxing** is the reverse - converting a wrapper class to its primitive. Below the wrapper classes are listed:

Primitive	Wrapper Class
boolean	Boolean
byte	Byte
short	Short
char	Character
int	Integer
long	Long
float	Float
double	Double

Wrapper classes have static helper methods like `.parseX()` and `.valueOf()` for explicit primitive conversion.

```
public class AutoboxingExample {

  public static void main(String[] args) {
    int n = 5;
    someMethod(n); // autoboxing is done here!
        // 8
  }

  public static void someMethod(Integer i) {
    System.out.println(i + 3);
  }
}
```

Object class

We also saw earlier in the garbage collection example about an `Object` - this is a special class in Java which is the root class from which all other classes inherit, either directly or indirectly. Thus, all objects have at least the methods defined in the `Object` class:

1. `Object clone()`
2. `boolean equals(Object o)`
3. `void finalize()`
4. `Class<?> getClass()`

5. `int hashCode()`
6. `void notify()`
7. `void notifyAll()`
8. `String toString()`
9. `void wait()`
10. `void wait(long timeout)`
11. `void wait(long timeout, int nanos)`

Object class methods

The `toString()` method is automatically called if you print an Object. Usually, this is overridden to provide human-readable output. Otherwise, you will print out `fully.qualified.ClassName@memoryAddress`

The `equals(Object o)` method compares two Objects. The `==` operator also compares objects, but only the memory address (i.e. will return `true` if and only if the variables refer to the exact same object in memory). By default, and unless you explicitly override it, the `equals` method simply calls the `==` operator.

The `hashCode()` method returns a hash code - a number that puts instances of a class into a finite number of categories. There are a few rules that the method follows:

- You are expected to override `hashCode()` if you override `equals()`
- The result of `hashCode()` should not change in a program
- if `.equals()` returns true, the hash codes should be equal
- if `.equals()` returns false, the hash codes do not have to be distinct. However, doing so will help the performance of hash tables.

Finally, the `.finalize()` method is called by the garbage collector when it determines there are no more references to the object. It can be overridden to perform cleanup activities before garbage collection, although it has been deprecated in newer versions of Java.

Abstract Classes

- An abstract class is a class that is declared `abstract` —it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be sub classed.
- An abstract class can have 0 or more abstract methods, but if a class has at least one abstract method then the whole class has to be abstract.
- An abstract class can have implemented methods as well.
- Use the `extends` keyword to extend an abstract class.

```
public abstract class GraphicObject {

   // declare fields

   // declare nonabstract methods
   abstract void draw();
}

class Circle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}

class Rectangle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```

Difference between interface and Abstract classes

- Conceptually, interfaces define behaviors and abstract classes are for concepts and inheritance.
- You can implement multiple interfaces, but you can extend only one class.

Interfaces

An interface acts as a contract for behaviors that a class can implement.

```
public interface InterfaceA {
 public void methodName(); //You don't implement the method!
}
```

Interfaces have implicit modifiers on methods and variables.

- Methods are 'public' and 'abstract'
- Variables are 'public', 'static', and 'final.' To inherit interfaces, a class must *implement* them and they are REQUIRED to implement all methods, unless the class is abstract.

Intro to TDD and Unit Testing

TDD

When developing software, it is important to ensure that most if not all of the code being written is tested to verify the functionality of the code. One way to ensure this is to follow a process called **test-driven development**, or TDD.

TDD Process

The TDD process consists of writing unit tests first, **before** the application code has been written. Then, code can be written to make the test pass and the process can be completed for each piece of functionality required. Thus, the process is:

1. Write a unit test
2. Run the test => test will fail
3. Fix the test by writing application code
4. Retest until the test passes
5. Repeat

Following the TDD process can be useful for ensuring that a valid unit tests always exists for any class or method that is written. Later, when refactoring code, the unit tests give us confidence that we can change the source code without breaking existing functionality. If we mess up somewhere, when the unit tests are run we can pinpoint exactly where the problem lies. This makes debugging much easier.

Unit Testing

Unit testing is the testing of individual software components in isolation from the rest of the system. This is done by writing unit tests which execute the code we want to inspect. When the code under test deviates from an expected outcome or behavior, the test will fail. If a test passes, it means the application performs as expected (unless there is a problem with the test itself). In Java, the most common unit testing framework is called JUnit.

JUnit Annotations

JUnit is a Java framework for unit testing. JUnit has several annotations within the `org.junit` package that developers can use to create tests and setup test environments:

- `@Test` - declares a method as a test method
- `@BeforeClass` - declares a setup method that runs once, before all other methods in the class
- `@Before` - declares a setup method that runs before each test method
- `@After` - declares a tear-down method that runs before each test method
- `@AfterClass` - declares a tear-down method that runs once, after all other methods in the class

When writing a test method, JUnit helps us check the functionality of the code being tested by providing static `Assert` methods, of which there are many. A few include:

- `assertTrue()`
- `assertFalse()`
- `assertEquals()`
- `assertNotEquals()`
- `assertThat()`

Assertions verify that the state of the application meets what is expected. For example, to test a simple addition method:

```
@Test
public void additionTest() {
  Assert.assertEquals(4, Calculator.addNumbers(2,2));
}
```

If the `.addNumbers()` method returns anything other than `4`, the test will fail. This will alert us that something is wrong in the logic of the method and we can then debug the issue. When we think we've fixed the problem, just run the test again and check that it passes.

Note: to avoid needing to reference `Assert` every time, we can use a `static import org.junit.Assert.*;` statement to import all `static` members of the `Assert` class.

Ignoring Tests

To prevent a test from running, use the `@Ignore` annotation. Use this sparingly, however, because ignoring valid tests only means that you are pretending a problem does not exist. If tests are constantly ignored, you will have no guarantee that the application functionality has not regressed.

Testing Best Practices

When it comes to testing code, a few best practices to follow include:

- Utilize dependency injection
- Write testable code
- Use a mocking library like Mockito for dependencies
- Measure your code coverage with a tool like Jacoco
- Externalize test data when possible (i.e. read in the test data from an external file or generate it dynamically)
- Generally, you want to use **only 1 assert statement per test** - this ensures you can pinpoint the defect when debugging
- Write deterministic tests (they shouldn't fail sometimes and pass other times - otherwise known as "flaky" tests)

Assert Methods

When writing a test method, JUnit helps us check the functionality of the code being tested by providing static `Assert` methods, of which there are many. A few include:

- `assertTrue()`
- `assertFalse()`
- `assertEquals()`
- `assertNotEquals()`
- `assertThat()`

Assertions verify that the state of the application meets what is expected. For example, to test a simple addition method:

```
@Test
public void additionTest() {
  Assert.assertEquals(4, Calculator.addNumbers(2,2));
}
```

If the `.addNumbers()` method returns anything other than `4`, the test will fail. This will alert us that something is wrong in the logic of the method and we can then debug the issue. When we think we've fixed the problem, just run the test again and check that it passes.

Note: to avoid needing to reference `Assert` every time, we can use a `static import org.junit.Assert.*;` statement to import all `static` members of the `Assert` class.