Week 2: Building Bigger Programs

CS 151

Important Dates

Midterm - March 6 (6:00 - 7:20 PM)

Final - May 15th (5:30 - 7:30 PM)

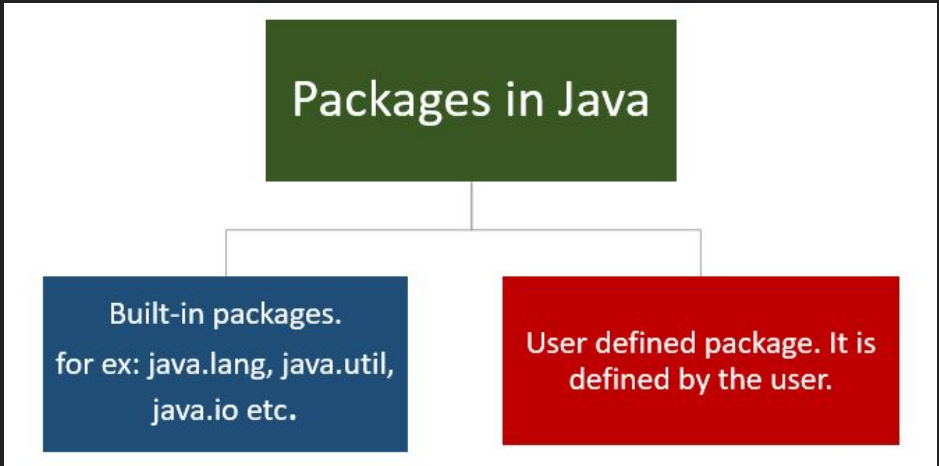
Office Hours: Friday 1 - 2 PM

https://drive.google.com/drive/folders/1ZtqGCz7HwUFirrzZRa7e7o0uGP7PtDZ5?usp=drive_link

Accessible by SJSU email

Java - Packages

- Packages are ways to group related classes/code
- A larger software system is split into packages
  - your coursework may be organized by multiple folders in Google Drive
- Benefits:
  - Makes large systems easier to understand
  - Prevent naming conflicts
  - Provides granular access control



Java - Packages

```
import java.util.Date; // Import the Date class from the java.util package

public class DateExample {
    public static void main(String[] args) {
        // Create a Date object to get the current date and time
        Date today = new Date();

        System.out.println("Today's date is: " + today.toString()); // Display the
current date and time
    }
}
```

Java - Packages

```java
package greetings;

public class Greeting {
    public static void sayHello(String name) {
        System.out.println("Hello, " + name + "!");
    }
}
```

```java
package test;
import greetings.Greeting;

public class TestGreeting {
    public static void main(String[] args) {
        Greeting.sayHello("Alice");
    }
}
```

- To make a package, group all classes into a single directory and add a package statement (lowercase) to the beginning of each class file
  - The start of every file should be a package statement, other than comments and blank lines
  - Following that should be a list of imports

Java - Packages

```java
package greetings;

public class Greeting {
    public static void sayHello(String name) {
        System.out.println("Hello, " + name + "!");
    }
}
```

```java
package test;
import greetings.Greeting;

public class TestGreeting {
    public static void main(String[] args) {
        Greeting.sayHello("Alice");
    }
}
```

- The file name must match the public class/interface name in the file
- Example: If a file defines `public class Greeting`, the file name must be Greeting.java
- A single Java file can contain multiple classes, but only one can be public
- The “entrypoint” of your program (over multiple packages) remains Main.java

Java - Packages

- Single class import:
  - `import java.util.Date;`
  - will import just one class
- Wildcard import
  - `import java.util.*;`
  - imports all classes/interfaces in the package
- The downside of importing * is reduced code clarity, slight overhead, potential naming conflicts

```
import greetings.Greeting;
// import greetings.*


public class TestGreeting {
    public static void main(String[] args) {
        Greeting.sayHello("Alice");
    }
}
```

Java - Scanner

- Java includes a class for keyboard input called `Scanner`
- Importing scanner will tell Java to
  - Make Scanner class and its methods available to the program
  - Find the Scanner class in a library of classes (ie, package) named java.util
- Your terminal will look like:

```
$ javac SimpleScanner.java
$ java SimpleScanner
Enter something: Hello, world!
You entered: Hello, world!
```

```java
import java.util.Scanner;

public class SimpleScanner {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter something: ");
        String input = scanner.nextLine();
        System.out.println("You entered: " + input);
        scanner.close();
    }
}
```

Java - Scanner

- Common methods:
  - nextLine()
    - reads entire next line, including spaces
  - next()
    - reads the next word from the input, separated by whitespace
  - What's the difference?
    - nextLine() is more practical and captures entire lines of text. next() is useful for capturing individual tokens
  - nextInt() and nextDouble()
    - read specific types of tokens from input
- In a real-world application, you might echo the input back to the user to confirm

```java
import java.util.Scanner;

public class SimpleScanner {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter something: ");
        String input = scanner.nextLine();
        System.out.println("You entered: " + input);
        scanner.close();
    }
}
```

```java
import java.util.Scanner;

public class ScannerExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Using nextLine() to read a full line of input
        System.out.print("Enter a sentence: ");
        String sentence = scanner.nextLine(); // Reads the entire line
        System.out.println("Full sentence: " + sentence);

        // Using next() to read individual words
        System.out.print("Enter another sentence: ");
        String firstWord = scanner.next(); // Reads only the first word
        System.out.println("First word: " + firstWord);

        // Demonstrating multiple calls to next()
        System.out.print("Enter yet another sentence: ");
        String word1 = scanner.next(); // Reads the first word
        String word2 = scanner.next(); // Reads the second word
        String word3 = scanner.next(); // Reads the third word
        System.out.println("Words: " + word1 + " " + word2 + " " + word3);

        scanner.close(); // Closing the scanner
    }
}
```

```
$ javac ScannerExample.java
$ java ScannerExample

Enter a sentence: Hello world
from Java
Full sentence: Hello world from
Java

Enter another sentence: Learning
Java is fun
First word: Learning

Enter yet another sentence: Java
programming is interesting


Words: Java programming is
```

Java - Math

- The Math class provides standard mathematical methods
  - This is a Class and not a package, so
  - 1) no need to import (comes with java.lang)
  - 2) all methods are Static and require invoking with the class name Math instead of a calling object
- Math class comes with predefined constants
  - e, base of natural logs
  - pi, or as close as we can fit into a double

```java
import java.util.Scanner;

public class Example {
    public static void main(String[] args) {
        // Scanner class (requires creating an object)
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a number: ");
        int number = scanner.nextInt();
        System.out.println("You entered: " + number);

        // Using Math class (no need to create an object)
        double squareRoot = Math.sqrt(number);
        System.out.println("Square root: " + squareRoot);
    }
}
```

Java - Math

- And several more advanced ones regarding trigonometry and calculus

```
public class MathExample {
    public static void main(String[] args) {
        // Finding the square root
        double squareRoot = Math.sqrt(25.0); // Expected output: 5.0

        // Finding the power
        double power = Math.pow(2.0, 3.0); // Expected output: 8.0

        // Finding the maximum of two numbers
        double max = Math.max(5.0, 10.0); // Expected output: 10.0

        // Finding the minimum of two numbers
        double min = Math.min(5.0, 10.0); // Expected output: 5.0

        // Rounding a number
        long rounded = Math.round(9.75); // Expected output: 10

        // Finding the absolute value
        double absolute = Math.abs(-20.5); // Expected output: 20.5

        // Generating a random number
        double random = Math.random(); // Expected output: A random
number between 0.0 and 1.0
    }
}
```

Object Oriented Programming

- In Java, almost everything revolves around Classes and Objects.
- Objects:
  - Often, real world entities
  - If your car were a Java Object
    - `car.accelerate()`
    - `car.brake()`
    - `car.openDriverDoor()`
- Classes:
  - template or blueprint that defines the attributes and behaviors of objects
  - allows creation of multiple objects with similar properties but different attributes
  - class Car

```
public class CarTest {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota", "Corolla", "Blue");
        System.out.println("Created a " +
myCar.getDetails());
    }
}

class Car {
    private String brand;
    private String model;
    private String color;

    public Car(String brand, String model, String color) {
        this.brand = brand;
        this.model = model;
        this.color = color;
    }

    public String getDetails() {
        return brand + " " + model + " in " + color;
    }
}
```

Java - Objects

- We can create objects that belong to a Class, just like we create primitives that belong to primitive types
  - object creation uses the keyword new
- Objects
  - Represent instances/implementation of classes
  - Consist of attributes (fields) and behaviors (methods)
- Objects invoke methods using object.methodName()
  - dot notation

```
public class CarTest {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota", "Corolla", "Blue");
        System.out.println("Created a " +
myCar.getDetails());
    }
}

class Car {
    private String brand;
    private String model;
    private String color;

    public Car(String brand, String model, String color) {
        this.brand = brand;
        this.model = model;
        this.color = color;
    }

    public String getDetails() {
        return brand + " " + model + " in " + color;
    }
}
```

Java - Constructors

- Objects are created from a class using a **constructor**
  - Constructors must have the same name as the ClassName
  - don’t return anything, not even void
  - when you create an object with “new” you’re calling the constructor
- Constructor is a function that describes how to create an object
- When you create an object, you should always create a constructor
  - Java will create one for you if you don’t but it won’t accept any parameters and is considered bad practice

```
public class CarTest {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota", "Corolla", "Blue");
        System.out.println("Created a " +
myCar.getDetails());
    }
}

class Car {
    private String brand;
    private String model;
    private String color;

    public Car(String brand, String model, String color) {
        this.brand = brand;
        this.model = model;
        this.color = color;
    }

    public String getDetails() {
        return brand + " " + model + " in " + color;
    }
}
```

Java - Constructors

- a simple Car class could contain instance variables describing its basic properties
- In order to exist, a Car must have a brand, model, and color. Therefore, the constructor should include these arguments
- Often, constructors have simple validation logic
    - example: might disallow a Honda Camry to be created

```
public class CarTest {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota", "Corolla", "Blue");
        System.out.println("Created a " +
myCar.getDetails());
    }
}

class Car {
    private String brand;
    private String model;
    private String color;

    public Car(String brand, String model, String color) {
        this.brand = brand;
        this.model = model;
        this.color = color;
    }

    public String getDetails() {
        return brand + " " + model + " in " + color;
    }
}
```

Java - Classes

- The “this” keyword references the object itself in all methods or constructors
- Example:
  - this.brand means you are setting the “brand” field of the object, to the argument “brand” of the constructor
- This allows a method to differentiate between the attribute and the parameter, which often have the same name

```
public void setColor(String color) {
    this.color = color;
}
```

```
public class CarTest {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota", "Corolla", "Blue");
        System.out.println("Created a " +
myCar.getDetails());
    }
}

class Car {
    private String brand;
    private String model;
    private String color;

    public Car(String brand, String model, String color) {
        this.brand = brand;
        this.model = model;
        this.color = color;
    }

    public String getDetails() {
        return brand + " " + model + " in " + color;
    }
}
```

Java - Static Methods

- Most of the time, an object is needed to perform a method conceptually
  - `car.Drive()`
- Static means the method belongs to the class, rather than instances of the class
  - often used for constants or values shared across all instances
  - can be called without instantiating an instance of the class

```
public static String getBrandInfo(String brand) {
    return switch (brand.toLowerCase()) {
        case "toyota" -> "Toyota: Reliable and fuel-efficient.";
        case "honda" -> "Honda: Japanese automobiles and
motorcycles.";
        case "ford" -> "Ford: American automaker.";
        case "bmw" -> "BMW: German luxury vehicles.";
        default -> "Information not available.";
    };
}

System.out.println("Brand Info: " + Car.getBrandInfo("Toyota"));
```

Java - Static Attributes

- Variables can also be static. These variables belong to the Class as a whole and not any particular object
  - only one copy per class
  - unlike instance variables, where each object has its own copy
  - often some constant or shared concept
  - to make it unchangeable, add the modifier `private static final int CAR_COUNT = 0`
- All objects of a Class can read/write to a static variable
- only static methods can access static variables

```
class Car {
    private String brand;
    private String model;
    private String color;
    private static int carCount = 0;
}


public Car(String brand, String model, String color) {
    this.brand = brand;
    this.model = model;
    this.color = color;
    carCount++; // Increment the car count whenever a new
    Car object is created
}

    public static int getCarCount() {
    return carCount; // total cars created
}
```

Java - null constant

- `null` is a special constant that can be assigned to any reference type
- not an object, but rather a placeholder to indicate there is no real value
  - often used in constructors to initialize Class variables with no starting value
- can be assigned to all Objects but not primitives

```
String nullString = null;
Car nullCar = null;

System.out.println(nullString.length());
// Expected error: NullPointerException
System.out.println(nullCar.getDetails());
// Expected error: NullPointerException
```

Java - Methods

- Methods for Objects are still functions and must follow rules of functions
- Some methods compute and return a value
  - `public int add(num1, num2) {...}`
- Some methods perform an action
  - If it doesn't return anything, it's a `void` method
  - Invoking a void method is simply a statement and not an expression
  - `objectName.staticMethodName()`
- Some methods return null (NOT the same as void)
  - resp = func return null; → assigns null value
  - resp = void function; → compilation error

```
class Account {
    private double balance;

    public Account(double initialBalance) {
        this.balance = initialBalance;
    }

    public double withdraw(double amount) {
        ...
    }

    public void deposit(double amount) {
        ...
    }

    public String closeAccount() {
        ...
        return;
    }

    public class Main {
        public static void main(String[] args) {
            Account acc = new Account(1000);
            acc.deposit(500);
            System.out.println("Withdrawn: $" + acc.withdraw(300));
            String closeResponse = acc.closeAccount();
            // What will closeResponse be?
        }
    }
}
```

Java - Methods

- Variables declared within a function are *local variables*
- Within a method, all method parameters are local variables
  - Two different methods can each have a local variable of the same name
- Java does not have true global variables, the highest would be class level aka static variables
  - defined within a class but outside any method

```
int globalVar = 10;

public class VariableScopeDemo {

    public static void main(String[] args) {
        VariableScopeDemo demo = new VariableScopeDemo();
        demo.demonstrateLocalVariable();
        demo.demonstrateScope();
    }

    public void demonstrateLocalVariable() {
        int localVar = 20;
        System.out.println("Local variable: " + localVar);
        System.out.println("Global variable accessed inside
function: " + globalVar);
    }

    public void demonstrateScope() {
        // Uncommenting will cause a compile-time error
        // System.out.println("Trying to access localVar: " +
localVar);
    }
}
```

Getters and Setters

- Encapsulation
  - ensures that an object's internal state is protected from unauthorized operations
  - getters and setters allow devs to control what operations others can do, and will sometimes include some validation logic
- When do we need getters and setters?
  - For all private fields to expose them
  - When we need to validate the data first
  - When we want some fields to be read-only or write-only, which private/public cannot support alone

```
class Car {
    private String brand;
    private String model;
    private String color;

    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }
}
```

Getters and Setters

- Let’s say another team is working on a CarMechanic Class
- What if `public int age`?
  - public fields can be read and modified by all
  - anyone could write `person.age = -5`
- What if `private int age` without getters/setters?
  - no external class or code will be able to view or access age at all
  - it would still be accessible/modifiable within the class, but it will lose its functional purpose for others to see and use

```
class Car {
    private String brand;
    private String model;
    private String color;

    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }
}
```

Getters and Setters

- Benefits of getters/setters:
  - Validation logic
  - Only one way to set/get means modifying that one way can enforce behavior across different usages
  - Some fields can be read/write only
  - Abstraction of implementation details

```
class Car {
    private String brand;
    private String model;
    private String color;

    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }
}
```

Java - Access Modifiers

- Access modifier: public vs private
  - public means other packages can see it
  - if we used private static void main, the runtime would not be able to find and execute main
- Why is this important? To dictate how other developers can interact
  - Car class might internally have `car.leftWheelForward()` or `car.rightWheelBackwards()`
  - But we would only expose `car.turnLeft()` and `car.turnRight()`

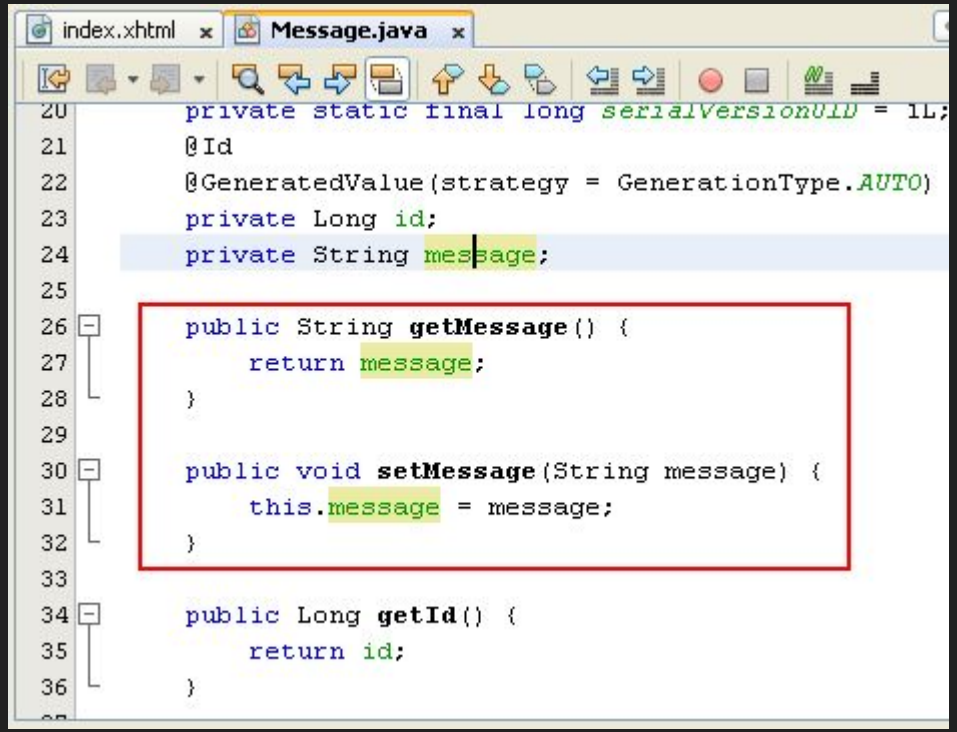
Java - Access Modifiers

- Default
  - Accessible from the same package
- Public
  - Accessible from anywhere
  - anyone with a Car object can run `car.start()`
- Private
  - Accessible only from within the class
  - Only the Car class can access the engine, safety checks, electricity, etc. that is required to start
  - if `internalSafetyCheck()` was public, a user might break the class by calling it in a place it wasn't meant to be called
- Protected
  - Accessible in same package and subclass, including subclasses in different packages

	Private	Public
Same class	Yes	Yes
Same package subclass	No	Yes
Same package non-subclass	No	Yes
Different package subclass	No	Yes
Different package non-subclass	No	Yes

Java - Access Modifiers

- Abstraction
  - It is considered good practice to make instance attributes private
  - Most methods are public and provide controlled access to the Object
  - Methods can be private when used as internal implementation details
- Why make an attribute private when we just provide a public method to get it?
  - Protect the internal state and allow changes only through controlled methods
  - Ensures modifications to attributes can be validated before being applied
  - Can set some attributes to be read-only
  - Can change internal implementation without affecting how users set/get



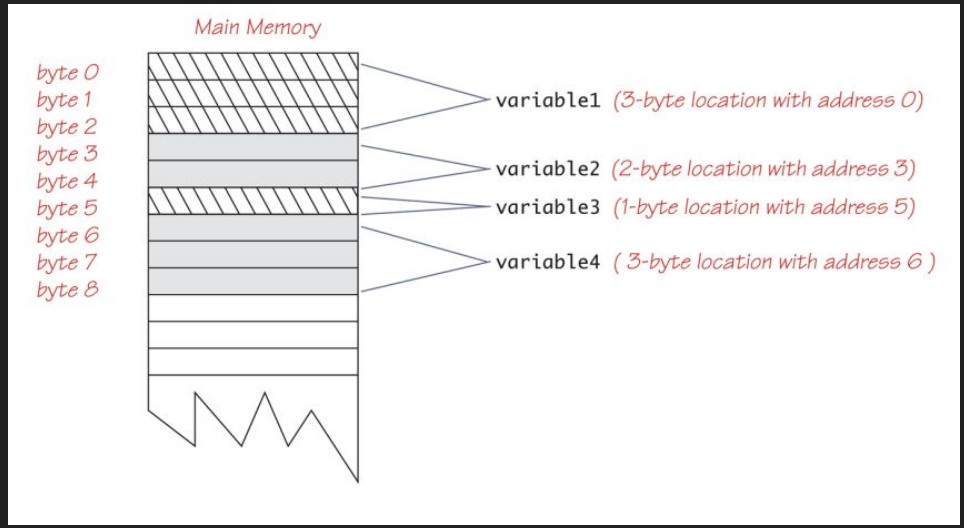
Java - Classes and Objects

```
class Book {
    private String title;
    private String author;
    private int year;
    private Person checkedOutTo;

    public Book(String title, String author, int year) {
        this.title = title;
        this.author = author;
        this.year = year;
        this.checkedOutTo = null;
        // Value when initialized
    }

    public void checkOutTo(Person person) {
        this.checkedOutTo = person;
    }

    public void returnBook() {
        this.checkedOutTo = null;
    }
}
```

```
class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
}

public class Library {
    public static void main(String[] args) {
        Person alice = new Person("Alice");
        Person bob = new Person("Bob");

        Book book1 = new Book("To Kill a Mockingbird",
"Harper Lee", 1960);

        book1.checkOutTo(alice);
        book1.returnBook();
        book1.checkOutTo(bob);

    }
}
```

Java - Memory

- Most computers have 2 types of memory
  - Secondary memory (for permanent storage)
  - Main memory (for running programs)
- Main memory is seen as a long list of numbered locations (bytes)
- The number that identifies a byte is called its address
  - 8GB RAM = 8 x 2^30 usable bytes
- Larger data items can require more than one byte of data to hold its data
  - The address of the first byte is used as the address for this whole data item



Java - Memory

- When objects are created using new we assign some memory to hold that info
- Java abstracts away much of this from devs compared to C++
  - memory allocation and garbage collection (freeing unused memory) is automatic
- When we reference an Object, we're actually referencing its memory address

```
import java.util.ArrayList;

public class MemoryComparison {
    public static void main(String[] args) {
        ArrayList<String> list1 = new ArrayList<>();
        list1.add("Hello");
        ArrayList<String> list2 = new ArrayList<>();
        list2.add("Hello");

        System.out.println(list1 == list2);
        // false
        System.out.println(list1.equals(list2));
        // true


System.out.println(System.identityHashCode(list1));
// Example: 366712642

System.out.println(System.identityHashCode(list2));
// Example: 1829164700
    }
}
```

Java - Memory

- With primitives, the VALUE of each argument is passed into the function
  - call-by-value
  - each primitive type always requires the same amount of memory to store its values (see week1 slides)
- With Objects, the actual object is passed in since we're passing the memory location of the actual object
  - more efficient than passing along a huge object for each function call
  - call-by-reference
  - valid code: `variable2 = variable1;`
  - changing one will change the other

```java
public class PrimitiveModification {
    public static void main(String[] args) {
        int number = 10;
        System.out.println("Original value: " + number);

        modifyPrimitive(number);

        System.out.println("Value after function call: "
+ number);
    }

    public static void modifyPrimitive(int num) {
        num *= 2; // Attempt to double the value
        System.out.println("Value inside function: " +
num);
    }
}

// Original value: 10
// Value inside function: 20
// Value after function call: 10
```

Java - Memory

```java
public class ArrayModification {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};
        System.out.println("Original array: " + java.util.Arrays.toString(numbers));

        modifyArray(numbers);

        System.out.println("Modified array: " + java.util.Arrays.toString(numbers));
    }

    public static void modifyArray(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            arr[i] *= 2; // Double each element
        }
    }
}

// Original array: [1, 2, 3, 4, 5]
// Modified array: [2, 4, 6, 8, 10]
```

Object vs Primitives

- If an object doesn't come with methods to change it, it is considered *immutable*
  - Safe to return such objects
  - `String` class is immutable
- Not safe:
  - returning person.birthDate directly
- Safe:
  - returning new Date(person.birthDate)
  - returning some immutable object
- Deep copy
  - a copy that has no references to original
- Shallow copy
  - any copy that isn't a deep copy
  - more dangerous, can cause privacy leaks

```java
import java.util.Date;

public class Person {
    private Date birthDate;

    public Person(Date birthDate) {
        this.birthDate = new Date(birthDate.getTime()); // Deep copy
    }

    public Date getBirthDateShallow() {
        return birthDate; // Shallow copy, dangerous
    }

    public Date getBirthDateDeep() {
        return new Date(birthDate.getTime()); // Deep copy, safe
    }

    public static void main(String[] args) {
        Person person = new Person(new Date());
        person.getBirthDateShallow().setTime(0); // Modifies original date
        System.out.println("Shallow Copy: " + person.getBirthDateShallow());
        // Shows modified date
        person.getBirthDateDeep().setTime(0); // Doesn't modify original date
        System.out.println("Deep Copy: " + person.getBirthDateDeep());
        // Shows original date
    }
}
```

Object vs Primitives Comparison

```java
public static void main(String[] args) {
    // Comparison of primitive types
    int num1 = 100;
    int num2 = 100;
    System.out.println("Comparing two ints with '==': " + (num1 == num2));
    // Output: true

    // Comparison of object types (Strings)
    String str1 = new String("Hello");
    String str2 = new String("Hello");
    System.out.println("Comparing two Strings with '==': " + (str1 == str2));
    // Output: false
    System.out.println("Comparing two Strings with 'equals()': " + str1.equals(str2));
    // Output: true

    // Using compareTo() for strings, which normally determines alphabetic ordering.
    System.out.println("Comparing two Strings with 'compareTo()': " + (str1.compareTo(str2) == 0));
    // Output: true
}
```

Luckily, Java requires most Classes to implement methods like `equals` and `toString` to make comparison easier

Java - Wrappers

- Wrapper classes provide a class type corresponding to each primitive types
  - in other words, we can treat primitives like classes and access useful methods
- `Integer integerObject = new Integer(42);`
- Wrapper classes include useful constants such as:
  - Integer.MAX_VALUE = 2147483647
  - Double.MAX_VALUE = 1.7976931348623157e+308
  - Boolean.TRUE
- And methods, such as converting between strings and numbers in a standardized way
  - Double.parseDouble(String s)
- Collections (ArrayList, HashMap) can only store objects

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

HW 2: Classes in practice

- Program 1: Create BankAccount Class
- Requirements:
  - BankAccount Class:
    - private attributes accountNumber, balance, and Customer
      - a Customer object should be required in the constructor
    - implement public methods: deposit, withdraw, getBalance, toString
      - most Classes implement toString. This should return a string of all the details of a Bank Account including its customer and balance.
  - Customer Class:
    - private attributes firstName, lastName, dateOfBirth
    - implement public methods: getFullName, getDateOfBirth
    - bonus: can you implement dateOfBirth using the Date class? (slide 4)

HW 2: Classes in practice

- Program 2: Students and Courses
- Requirements:
  - Course Class:
    - private attributes courseName, list of students.
      - the constructor should require courseName and initialize an empty list of students
    - public methods: addStudent(), removeStudent(), toString()
      - toString() should return the string name and a list of all students
  - Student Class:
    - private attributes firstName, lastName, studentId
    - public methods getFullName(), getStudentId()