50.001 Week 3 Slides

Notice

These slides are subject to changes for additional content and corrections of typos etc. Always refer to the online version for the latest copy.

Revise Constructors

Predict what you will see on the screen without executing the code.
Give me your prediction at b.socrative.com 593583
Note that primitive types are initialized with a default value,
But reference types are not and are assigned null.

```
public class Revise3 {

    public static void main(String[] args) {
        int k = 1;
        System.out.println(k);
        Apple apple1 = new Apple(3);
        apple1.seeValues();
        Apple apple2 = new Apple();
        apple2.seeValues();
    }
}

class Apple{

    private int a;
    private Double b;

    Apple(){
        System.out.println("A");
        this.b = Double.valueOf("1");
    }

    Apple(int value){
        System.out.println("B");
        this.a = value;
    } // more code below, see on your own computer

    public void seeValues(){
        System.out.println(a);
        System.out.println(b);
    }
}
```

Example 30 - Recall Instantiation

You create an instance of a class using the **new** keyword. This is called **instantiation**, and creates an object in memory, which has **attributes** (also called **instance variables**) and **methods**.

How you instantiate depends on the constructor(s) defined in the class. Once you have an object after instantiation, you can execute any methods that are defined in the class.

The keyword **static** allows a method to be executed *without* creating an instance of the class.

```
public class Orange {

    public static String getFact(){ return "Vitamin C";  }

    private String type;

    Orange(){ this.type = "Navel Orange"; }

    Orange(String type){ this.type = type; }

    public String getType(){
        return type;
    }
} // give me your answer in b.socrative.com 593583
```

```
public static void main(String[] args) {

    /** TODO 1. Write code to get the string " Vitamin C". */

    /** TODO 2. Write code to instantiate an Orange object,
     * and execute getType() to get the string "Navel Orange"  */

    /** TODO 3. Write code to instantiate an Orange object,
     *  and execute getType() to get the string "Tangerine"
     */
}
```

Recall Inheritance and Polymorphism

Inheritance. A child class can inherit non-private methods and instance variables from a parent class.

Overriding. A child class can provide a new definition of a method defined in the parent class. This is as long as the **method signatures** remain the same. The parent class method (of the same signature) can be invoked using the super keyword.

Subtype and Supertype. An inheritance relationship results in the two classes having a subtype and supertype relationship. In the declaration below, A is a **supertype** of B. B is a **subtype** of A. Thus a variable a declared as type A can refer to an object B, since B is a subclass of A.

```
A a = new B();
```

Dynamic binding In the declaration above, all methods available to variable **a** are based on what class **A** has, ("declared type"). However, remember that due to overriding, the same method can be defined in the subclasses.

Let's suppose both A and B have a method in their class definitions called **something().**

With the declaration above, if **`a.something()`** is executed, the JVM will begin looking for a **something()** method in the lowest level of the inheritance hierarchy i.e. in class B, before going up.

Example 31

```
public class TestA {

    public static void main(String[] args) {
        A a = new B();
        //what method can you execute? what will you see?
}
}

class A{

    public void something(){
        System.out.println("Anya in A");
    }

}

class B extends A{
    @Override
    public void something(){
        System.out.println("Bond In B");
    }

    public void nothing(){
        System.out.println("Nothing here");
    }
}
```

It is good practice for your class to override toString(), equals() and hashCode().

In Java, **`Object`** is the superclass of all classes. It has the following methods (and others) defined, which your class needs to override to provide the necessary functionality.

Method	**You should override this …**
String toString()	to provide an informative string representation of your class
boolean equals(Object obj)	If you need to define what makes two objects of your classes identical in contents Note that == checks whether two objects are aliased. See next section on String
int hashCode()	If you override equals(), override this to return a hashCode value, such that identical objects should return the same hashCode value

Example 32 – Overriding toString(), equals(), hashCode()

```
import java.util.Objects;

public class Coordinate {

    private int x;
    private int y;

    Coordinate(int x, int y){
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
    //TODO 1 override toString() to return an informative string

    //TODO 2 the implementation using Android studio's wizard
    // (or your own IDE's)
    // will be sufficient for equals() and hashCode()
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Coordinate)) return false;
        Coordinate that = (Coordinate) o;
        return getX() == that.getX() && getY() == that.getY();
    }

    @Override
    public int hashCode() {
        return Objects.hash(getX(), getY());
    }
}
```

```
public static void main(String[] args) {
    Coordinate c1 = new Coordinate(1, 3);
    Coordinate c2 = new Coordinate(1, 3 );
    System.out.println(c1.equals(c2)); // True
```

```
    System.out.println(c1);
}
```

Clicker Question

Which is the correct way to check that variable a has “Bocchi” as a value?

```
String a = "Bocchi";
```

```
A.a == "Bocchi";
B.a.equals("Bocchi");
C.a == new String("Bocchi");
D.both A & B are ok
E.A, B & C
```

Example 33 - String objects with identical contents are stored as a single instance in the same pool. Use equals() to compare their values, do not use == .

The JVM maintains a pool of String instances and strings declared using literals are **interned** in this pool.
What **interning** means is that there is only one instance of a string value in the pool, as long as it is declared using a literal. This helps to reduce memory usage.

Thus in the example below,

- **s1** and **s2** point to the same string in the String pool.
- **s3** is declared with the **new** keyword, it does not go in the pool.

Hence, even though the == operator checks for aliasing, it seems to work for comparing string values. However, this is a logic error if the aim of the code is to check if values are identical.

You should

- always use the **equals()** method to compare two String values (and **Integer**, **Double**, **BigDecimal** etc)
- avoid using the **new** keyword to instantiate a String, as it does not go into the string pool.

```
public static void main(String[] args) {
    String s1 = "hello";
    String s2 = "hello";
    String s3 = new String("hello"); //avoid doing this

    /** Don't do this */
    System.out.println(s1 == s2); // true
    System.out.println(s1 == s3); // false, since s3 is not interned
    System.out.println(s1 == "hello"); // true

    /** Always use the equals method */
    System.out.println(s1.equals(s3)); // true
    System.out.println(s1.equals("hello")); // true
```

```
}
```

Summary: == or .equals() ?

1. **For primitive types**, use the comparison operator to compare values.

```
int a = 10;
int b = 20;
System.out.println(a == b);
```

2. **To check if objects are aliased**, use the comparison operator.

```
BigDecimal a = new BigDecimal("12");
BigDecimal b = a;
System.out.println(a == b);
```

3. **To check if objects have equal contents or values**, use the .equals() method. This includes String objects.

```
BigDecimal a = new BigDecimal("12");
BigDecimal b = new BigDecimal("12");
System.out.println(a.equals(b));

String a = "apple";
System.out.println( a.equals("orange"));
```

Abstract Classes

You have learnt about inheritance. In the design of your class hierarchy, you may envision a situation where

- some methods have the **same implementation for all classes**
- other methods have to be **implemented in the subclasses**

Thus, it would not make sense to allow the top-most parent to be instantiated

In such a design situation, **abstract classes** can be used. They are defined like any other concrete class, but:

- the **`abstract`** keyword is used
- there may be zero or more **abstract methods** defined

abstract methods merely have the signature defined, but no body exists, and the **`abstract`** keyword is used.
➔ It will then be the responsibility of the subclasses to provide an implementation.

This is illustrated in the following example.

Example 34 - Given the following abstract class, write a concrete subclass Circle that implements the abstract methods

```
public abstract class GeometricObjectMod {
    private String colour;
    private boolean filled;

    GeometricObjectMod(){
        colour = "white";
        filled = false;
    }

    GeometricObjectMod(String colour, boolean filled){
        this.colour = colour;
        this.filled = filled;
    }
    // abstract methods, only the signature
    public abstract double getArea();

    public abstract double getPerimeter();

    public String getColour() {
        return colour;
    }

    public boolean isFilled() {
        return filled;
    }
}
```

Clicker Question (Look at Example 34A in my repo)

Given the following abstract class P, write a concrete class Q which is a subclass of P, b.socrative.com 593583 ---> 1608

- Q has two constructors like as P, both initializes name
- **displayAction()** displays *name* is running!

```
public abstract class P {

    private String name;

    public P(){
        name = "Pikachu";
    }

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

    public String getName() {
        return name;
    }

    public abstract void displayAction();
}
```

An Interface only has abstract methods defined.

Java has another kind of "class" definition called an **interface**.
An interface has only abstract methods (in Java 7).

By default,

- these abstract methods are **public**
- the **abstract** and **public** keywords are not necessary

An interface cannot be instantiated.

A concrete class implements the interface by using the **implements** keyword.

Let’s see how the syntax works in the next example.

Example 35 – Getting used to Interface

Write two classes that implements the following interface.

Write **System.out.println()** statements for each method.

You may use the following information, but feel free to use other information.

- Welsh Corgi – says “woof” but is tailess
- Siberian Husky – says “howl” and loves to wag tail

Each class has a private String variable **name** that is initialized through the constructor.

If there’s time, you can override **toString()** as you wish.

```
public interface Dog {

    void makeSound();

    void wagTail();

}
```

```
public class WelshCorgi implements Dog {
    @Override
    public void makeSound() {
        // Fill in what you like
    }

    @Override
    public void wagTail() {

    }
}
```

One reason for having interfaces is to provide a common behaviour for a group of classes, which makes code flexible.

If a concrete class A implements an interface B, this provides a **guarantee** that the abstract methods defined in B will be implemented in A, otherwise, this will generate a compile-time error. Also, an interface acts as a supertype to the concrete classes that implement it.

Using code in the previous example, you can declare:

```
Dog doggy = new WelshCorgi();
```

Since the methods available to **doggy** depends on the declared type (**Dog**), you have the guarantee that you can execute **doggy.makeSound()** and **doggy.wagTail().**

Furthermore, if you wish to change the implementation of **doggy**, simply change the declaration to:

```
Dog doggy = new SiberianHusky();
```

You STILL have the guarantee that you have the same behaviour – you still can execute **doggy.makeSound()** and **doggy.wagTail().**

Thus, using interfaces makes your program more flexible. This leads to our first OOP Design Principles.

OOP Design Principle
Single Responsibility Principle – Each class should have only one reason to change

This means that regardless of the type of class (concrete, abstract, interface), you should think very carefully about what it is meant to do and avoid specifying unrelated functionality in it. In this case, the Dog interface avoids having too many abstract methods and only specifies two behaviours that all Dog objects should have.

OOP Design Principle
Program to interfaces, not implementations

This means that you should declare a variable as an interface type. Then, you are free to use ANY concrete implementation of that interface, and you can CHANGE the concrete implementation without affecting the rest of the code.

Example 36 – Comparable

Java provides the **Comparable** interface (part of java.util library) for objects to be sorted in a natural ordering. It has one abstract method **`int compareTo( T o)`**. Recall your previous lesson on Generics: **T** is a **type parameter**, which enables different types to be specified in place of **T**.

```
public interface Comparable<T>{

    int compareTo( T o);
}
```

The method compareTo should, quoting the JavaDocs,
"Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object".

An ArrayList of objects can be sorted by Collections.sort() as long as they have the Comparable<T> interface implemented.

```
public class Coordinate implements Comparable<Coordinate> {

    private double x, y;

    Coordinate(){}

    Coordinate(double x, double y){
        this.x = x;
        this.y = y;
    }

    public double getDistance(){
        return Math.sqrt( x*x + y*y);
    }

    @Override
    public String toString() {
        String template = "(%f, %f)";
        return String.format( template, x, y);
    }

    @Override
    public int compareTo(Coordinate coordinate) {
```

```
        // Fill this in
    }
}
```

```
public static void main(String[] args) {

    Coordinate c1 = new Coordinate(5, 12);
    Coordinate c2 = new Coordinate(3, 4);
    Coordinate c3 = new Coordinate(7, 24);
    Coordinate c4 = new Coordinate();

    List<Coordinate> coordinateList = new ArrayList<>();
    coordinateList.add(c1);
    coordinateList.add(c2);
    coordinateList.add(c3);
    coordinateList.add(c4);

    // Pass coordinateList to Collections.sort()
    // then display the outcome

}
```

Example 37 – List Interface

Recall:

OOP Design Principle
Program to interfaces, not implementations

This means that you should declare a variable as an interface type. Then you are free to use ANY concrete implementation of that interface, and you can CHANGE the concrete implementation without affecting the rest of the code.

Another example of this is the **List<T>** Interface, of which you are familiar with **ArrayList<T>.** But there are other implementations, such as **LinkedList<T>** and **Vector<T>** , which you can choose depending on the performance requirements.

```
public class TestList {

    public static String ARRAYLIST = "ArrayList";
    public static String LINKEDLIST = "LinkedList";
    public static String VECTOR = "Vector";

    public static void main(String[] args) {

        String choice = "ArrayList";
        List<Integer> integerList;

        // Write if/else statements to instantiate integerList
based on choice

        System.out.println("List:"+(integerList instanceof List));
        System.out.println("ArrayList:" + (integerList instanceof
ArrayList) );

        System.out.println("List is empty: " + integerList);
        integerList.add( Integer.valueOf("123"));
        integerList.add( Integer.valueOf("456"));
        System.out.println(integerList);

    }
}
```

Example 38 – Stack

You will recall that a stack is a Last-In-First-Out linear data structure, with the following operations

- push – add an item to the top of the stack
- pop – remove the item on the top of the stack and return it
- peek – return the item on the top of the stack
- size – number of elements
- isEmpty – returns true if no elements and false otherwise

You can then define an interface as follows.

```
public interface CustomStack<T> {

    void push(T t);
    T pop();
    int size();
    T peek();
    boolean isEmpty();
}
```

To get your concrete stack class,

- make a decision on how you are going to store the elements within the class
- implement the methods according to this decision.

Interfaces allow changes in your code to be managed easily. Manage change by writing new code, not modifying existing code.

From the previous example, let's say you have decided on using **ArrayList** to store the stack elements and implemented a concrete class called **MyStack**.

However, you decide later that you would like another implementation due to performance issues and decide to use some other datatype.

Question: Do you modify **MyStack**? Or do you write another class?

OOP Design Principle
Open/Closed Principle – classes should be open to extension, and closed to modification.

This principle tells you that you should write your code in such a way to allow changes without having to modify existing code.

Interfaces already allow you to do that.

To change the way the stack elements are stored,

- leave the code for **MyStack** alone, and
- write a new class that also implements **CustomStack**.

You will still have the guarantee that code that worked with **MyStack** will also work with your new class. Explain how and why ☺