The Java Language

Coventry University

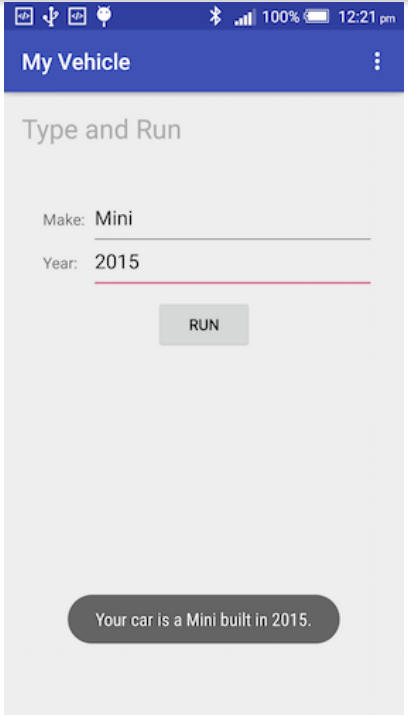
Last week?

- Tell me what you have learned so far? Anything new?

LAB 1

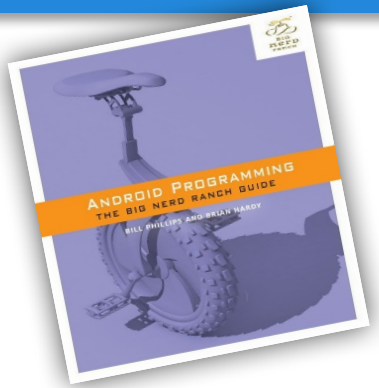
- Big Java
- Classes and objects
- Some keywords
- Relative layout and Toaster
- IntelliJ features

The app i.e. outcome



- Clear understanding of simple class structure
- Relative layout
- Toaster

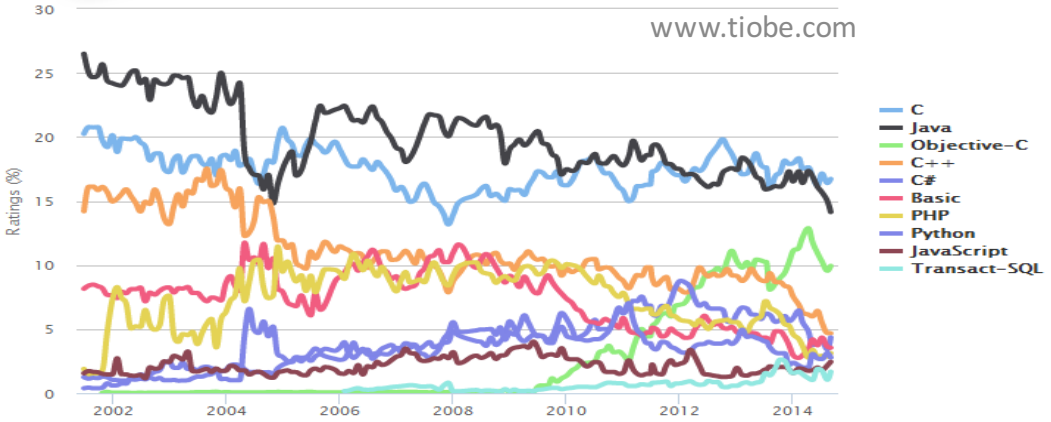
Big Java



"…you need to be familiar with Java, including ***classes and objects, interfaces, listeners, packages, inner classes, anonymous inner classes, and generic classes****.*

If these ideas do not ring a bell, you will be in the weeds by page 2…"

– Android Programming: The Big Nerd Ranch Guide



Classes and objects

Classes

A class is a template, blueprint, or contract that defines what an object's data fields and methods will be.

Objects

An object represents an entity in the real world that can be distinctly identified. An object is an instance of a class.

Class Name: Circle

Data Fields:
radius is _____

Methods:
getArea
getPerimeter
setRadius

Circle Object 1

Data Fields:
radius is 1

Circle Object 2

Data Fields:
radius is 25

```
public class Charge
{
   private final double rx, ry;
   private final double q;

   public Charge(double x0, double y0, double q0)
   {  rx = x0; ry = y0; q = q0;  }

   public double potentialAt(double x, double y)
   {
      double k = 8.99e09;
      double dx = x - rx;
      double dy = y - ry;
      return k * q / Math.sqrt(dx*dx + dy*dy);
   }

   public String toString()
   {  return q +" at " + "("+ rx + ", " + ry +")";  }

   public static void main(String[] args)
   {
      double x = Double.parseDouble(args[0]);
      double y = Double.parseDouble(args[1]);
      Charge c1 = new Charge(.51, .63, 21.3);
      Charge c2 = new Charge(.13, .94, 81.9);
      double v1 = c1.potentialAt(x, y);
      double v2 = c2.potentialAt(x, y);
      StdOut.printf("%.1e\n", (v1 + v2));
   }

}
```

class name

instance variables

constructor

instance variable names

instance methods

test client

create and initialize object

invoke constructor

object name

invoke method

Access modifiers

Visibility	Public	Protected	*Default*	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes, *through inheritance*	No	No
From any non-subclass class outside the package	Yes	No	No	No

Access modifiers determine whether other classes can use a particular field or invoke a particular method.

https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Method signature

```
public double calculateAnswer(double wingSpan, int numberOfEngines,
                              double length, double grossTons) {
    //do the calculation here
}
```

The only required elements of a method declaration are the method's return type, name, a pair of parentheses, `()`, and a body between braces, `{}`.

More generally, method declarations have six components, in order:

1. Modifiers—such as `public`, `private`, and others you will learn about later.
2. The return type—the data type of the value returned by the method, or `void` if the method does not return a value.
3. The method name—the rules for field names apply to method names as well, but the convention is a little different.
4. The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, `()`. If there are no parameters, you must use empty parentheses.
5. An exception list—to be discussed later.
6. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

https://docs.oracle.com/javase/tutorial/java/javaOO/methods.html

super keyword

Constructor with superclass initializer

Syntax

```
public ClassName(parameterType parameterName, . . .)
{
   super(arguments);
   . . .
}
```

```
public ChoiceQuestion(String questionText)
{
   super(questionText);
   choices = new ArrayList<String>;
}
```

The superclass constructor is called first.

The constructor body can contain additional statements.

If you omit the superclass constructor call, the superclass constructor with no arguments is invoked.

super keyword:

Invokes the superclass constructor or a superclass method

this keyword

this keyword

1. To assign a parameter variable to an instance variable of the current object
2. To call a constructor from another constructor in the same class
3. To pass a reference of the current object to another object

```
1. public class Curtain {
2.    Builder builder = new Builder();
3.    builder.setWallType(this);
4. }
5.  public class Builder {
6.    public void setWallType(Curtain c) {...}
7. }
```

```
1.  public class Curtain extends PrivacyWall {
2.   String color;
3.   public void setColor(String color) {
4.     this.color = color;
5.   }
6. }
```

```
1.  public class Curtain extends PrivacyWall {
2.   public Curtain(int length, int width) {}
3.   public Curtain() {
4.     this(10, 9);
5.   }
6. }
```

Three types of comments

1. Line comment
2. Block comment
3. Javadoc

The example source code

300CEM / Week_02_The_Java_language / MyVehicle / app / src / main / java / com / example / jianhuayang / myvehicle / Vehicle.java

jianhuayang lab2_a done c62991f a day ago

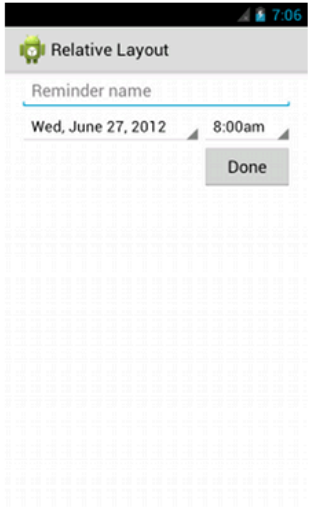
1 contributor

56 lines (47 sloc) | 1.13 KB Raw Blame History

```java
package com.example.jianhuayang.myvehicle;

/**
 * Created by jianhuayang on 08/10/2016.
 * @author jianhuayang
 * @version 1.1
 */

public class Vehicle {
    public static int counter = 0;
    private String make;
    private int year;
    private String message;

    // the default constructor
    public Vehicle() {
        this.make = "Volvo";
        this.year = 2012;
        this.message = "This is the default message.";
    }
```

Relative layout

```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/androi
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp" >
    <EditText
        android:id="@+id/name"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/reminder" />
    <Spinner
        android:id="@+id/dates"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_below="@id/name"
        android:layout_alignParentLeft="true"
        android:layout_toLeftOf="@+id/times" />
    <Spinner
        android:id="@id/times"
        android:layout_width="96dp"
        android:layout_height="wrap_content"
        android:layout_below="@id/name"
        android:layout_alignParentRight="true" />
    <Button
        android:layout_width="96dp"
        android:layout_height="wrap_content"
        android:layout_below="@id/times"
        android:layout_alignParentRight="true"
        android:text="@string/done" />
</RelativeLayout>
```

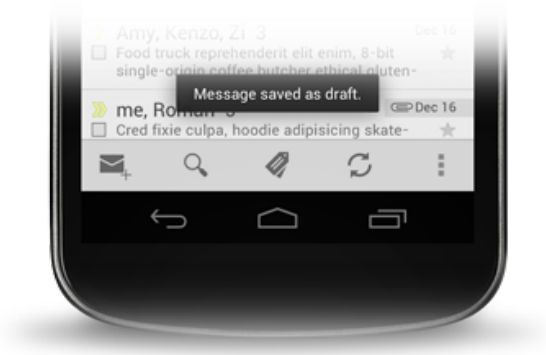


https://developer.android.com/guide/topics/ui/layout/relative.html

Toaster

```
Context context = getApplicationContext();
CharSequence text = "Hello toast!";
int duration = Toast.LENGTH_SHORT;

Toast toast = Toast.makeText(context, text, duration);
toast.show();
```

https://developer.android.com/guide/topics/ui/notifiers/toasts.html



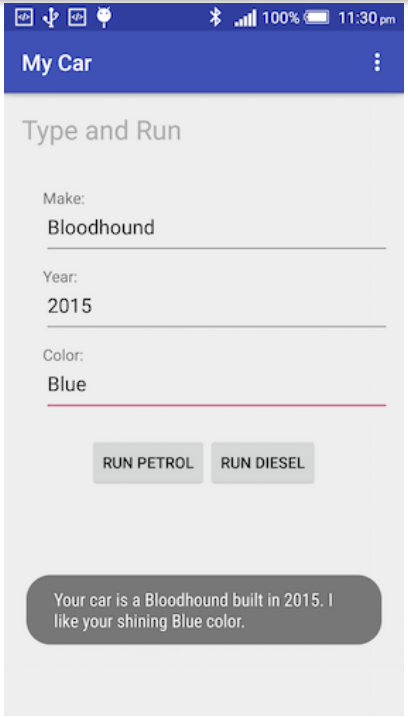
IntelliJ features

- Getter/Setter generator (right-click within class)
- Code reformat, rearrange (Preferences→Editor→Code Style)
- Live templates (‘cmd’ + j)

LAB 2

- Object-oriented programming (OOP) principles
- Interfaces (more keywords)
- Code example
- Linear layouts
- onClick method
- Packages
- Google Java style

The app i.e. outcome



- Class inheritance
- Interface
- Linear layout
- view.getId()

OOP principles

- **Encapsulation** is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.
- **Inheritance** is the process by which one object acquires the properties of another object.
- **Polymorphism** is a feature that allows one interface to be used for a general class of actions.

Encapsulation

Visibility	Public	Protected	*Default*	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes, *through inheritance*	No	No
From any non-subclass class outside the package	Yes	No	No	No

Access modifiers determine whether other classes can use a particular field or invoke a particular method.

abstract keyword

abstract classes

```
1.  public abstract class Alarm {
2.   public void reset() {...}
3.   public abstract void renderAlarm();
4. }
```

abstract methods

```
1.  public class DisplayAlarm extends Alarm {
2.   public void renderAlarm() {
3.     System.out.println("Active alarm.");
4.   }
5. }
```

- An abstract class is typically used as a base class and cannot be instantiated.
- An abstract class can contain abstract and nonabstract methods, and it can be a subclass of an abstract or a nonabstract class.
- An abstract method contains only the method declaration, which must be defined by any nonabstract class that inherits it

Overriding vs Overloading

Overriding

```
1.  public class CircleObject extends
    SimpleGeometricObject {
2. @Override
3.   public String toString() {
4.     return super.toString() +
    "\nradius is " + radius;
5.   }
6. }
```

Overloading

```
1.  public static double max(double num1, double
    num2) {
2.   if (num1 > num2)
3.     return num1;
4.   else
5.     return num2;
6. }
7.  public static double max(double num1, double
    num2, double num3) {
8.   return max(max(num1, num2), num3);
9. }
```

- Overridden methods are in different classes related by inheritance.
- Overloaded methods can be either in the same class or different classes related by inheritance.
- Overridden methods have the same signature and return type.
- Overloaded methods have the same name but a different parameter list.

Annotation

```
1.  public class CircleObject extends SimpleGeometricObject
    {
2. @Override
3.   public String toString() {
4.     return super.toString() + "\nradius is " + radius;
5.   }
6. }
```

- An *annotation* is an instance of an annotation type and associates metadata with an application element.
- It is expressed in source code by prefixing the type name with the @ symbol.
- @Override annotations are useful for expressing that a subclass method overrides a method in the superclass and doesn't overload that method instead.

final keyword

Prevent overriding

```
1.  class A {
2.   final void meth() {
3.     System.out.println("This is a final
   method.");
4.   }
5. }
6.  class B extends A {
7.   void meth() { // ERROR! Can't override.
8.     System.out.println("Illegal!");
9.   }
10.}
```

Prevent inheritance

```
1.  final class A {
2.   // ...
3. }
4. // The following class is illegal.
5.  class B extends A { // ERROR! Can't
   subclass A
6. // ...
7. }
```

`final` keyword:

A value that cannot be changed after it has been initialized, a method that cannot be overridden, or a class that cannot be extended

OOP – polymorphism

```
1.  public class PolymorphismDemo {
2.  public static void main(String[] args) {
3.     // Display circle and rectangle properties
4.     displayObject(new CircleFromSimpleGeometricObject(1, "red", false));
5.     displayObject(new RectangleFromSimpleGeometricObject(1, 1, "black",
6.         true));
7.   }
8.  public static void displayObject(SimpleGeometricObject object) {
9.     System.out.println("Created on " + object.getDateCreated()
10.        + ". Color is " + object.getColor());
11.  }
12.}
```

```
Created on Wed Sep 17 11:41:21 BST 2014. Color is red
Created on Wed Sep 17 11:41:21 BST 2014. Color is black
```

- The inheritance relationship enables a subclass to inherit features from its superclass with additional new features.
- A subclass is a specialization of its superclass; every instance of a subclass is also an instance of its superclass, but not vice versa.

Interfaces

Syntax

```
public interface InterfaceName
{
    method headers
}
```

```
public interface Measurable
{
    double getMeasure();
}
```

The methods of an interface are automatically public.

No implementation is provided.

Syntax

```
public class ClassName implements InterfaceName, InterfaceName, . . .
{
    instance variables
    methods
}
```

```
public class BankAccount implements Measurable
{
    . . .
    public double getMeasure()
    {
        return balance;
    }
    . . .
}
```

List all interface types that this class implements.

BankAccount instance variables

Other BankAccount methods

This method provides the implementation for the method declared in the interface.

Interfaces

	Abstract class	Interface
Variables	No restrictions	All variables must be `public static final`.
Constructors	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No constructors. An interface cannot be instantiated using the new operator.
Methods	No restrictions	All methods must be public abstract instance methods

Inner classes and event listeners

Inner classes

```
1.  public class MeasurerTester {
2.   public static void main(String[] args)
3.   {
4.     class AreaMeasurer implements Measurer
5.     {. . .}
6.     . . .
7.     Measurer areaMeas = new AreaMeasurer();
8.     double averageArea = Data.average(rects,
   areaMeas);
9.     . . .
10.  }
11.}
```

Event listeners

```
1.  import java.awt.event.ActionEvent;
2.  import java.awt.event.ActionListener;
3.  public class ClickListener implements
   ActionListener {
4.   public void actionPerformed(ActionEvent
   event) {
5.     System.out.println("I was clicked.");
6.   }
7. }
```

- Inner classes are commonly used for utility classes that should not be visible elsewhere in a program.
- An event listener belongs to a class that is provided by the application programmer. Its methods describe the actions to be taken when an event occurs.

Our example

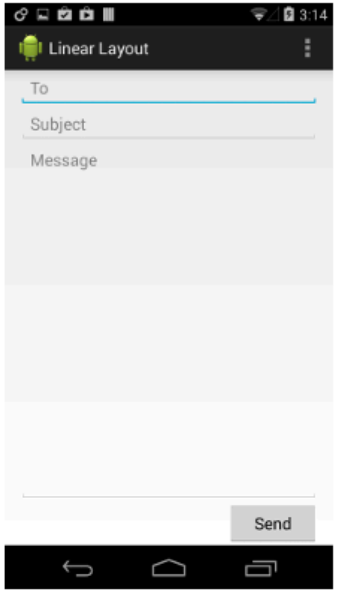
```
7   class Car extends Vehicle {
8       private String color;
9
10      public Car(String make, int year, String color) {
11          super(make, year);
12          this.color = color;
13          setMessage(getMessage() + " I like your shining " + color + " color.");
14      }
15  }
```

Our example

```
17  class Diesel extends Vehicle implements Vehicle.Controllable {
18
19      private String type;
20
21      public Diesel(String make, int year) {
22          super(make, year);
23          this.type = "Diesel";
24      }
25
26      @Override
27      public void control() {
28          setMessage(super.getMessage() + " Emission is under control.");
29      }
30
31      @Override
32      public String getMessage() {
33          control();
34          return super.getMessage();
35      }
36  }
```

Linear layout

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:orientation="vertical" >
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/to" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/subject" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:gravity="top"
        android:hint="@string/message" />
    <Button
        android:layout_width="100dp"
        android:layout_height="wrap_content"
        android:layout_gravity="right"
        android:text="@string/send" />
</LinearLayout>
```



https://developer.android.com/guide/topics/ui/layout/linear.html

Onclick listener

```
switch (view.getId()) {
    case R.id.buttonRunPetrol:
        vehicle = new Car(make, intYear, color);
        break;
    case R.id.buttonRunDiesel:
        vehicle = new Diesel(make, intYear);
        break;
    default:
        break;
}
```

- Single functions that fits all
- Switch statement depends on which button being clicked.
- View.getId() to get ID
- R.id.xxxx is and integer

Packages

- Java provides a method of organizing the code in your programming projects into logical modules, or collections of code, called Packages.
- Packages are declared at the top of each Java code module that utilizes their classes and methods contained within those packages.
- Packages are always declared using Java's package keyword.

```java
package com.example.mycontactlist;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteDatabase;

public class ContactDataSource {

    private SQLiteDatabase database;
    private ContactDBHelper dbHelper;

    public ContactDataSource(Context context) {
        dbHelper = new ContactDBHelper(context);
    }

    public void open() throws SQLException {
        database = dbHelper.getWritableDatabase();
    }

    public void close() {
        dbHelper.close();
    }

```

Google Java style

DON’Ts:

- Don't use more than one blank lines, don't use unnecessary white spaces.
- Don't declare more than one variables in a single line.
- Don't prefix your variable name.
- Don't import everything using things such as import java.util.*;
- Don't use meaningless names such as func1, temp, var2

Google Java style

DOs:

- Classes and interfaces
  - The first letter should be capitalized, and if several words are linked together to form the name, the first letter of the inner words should be uppercase (a format that's sometimes called "camelCase").
- Methods
  - The first letter should be lowercase, and then normal camelCase rules should be used. In addition, the names should typically be verb-noun pairs.
- Variables
  - Like methods, the camelCase format should be used, starting with a lowercase letter.
- Constants
  - Java constants are created by marking variables static and final. They should be named using uppercase letters with underscore characters as separators.