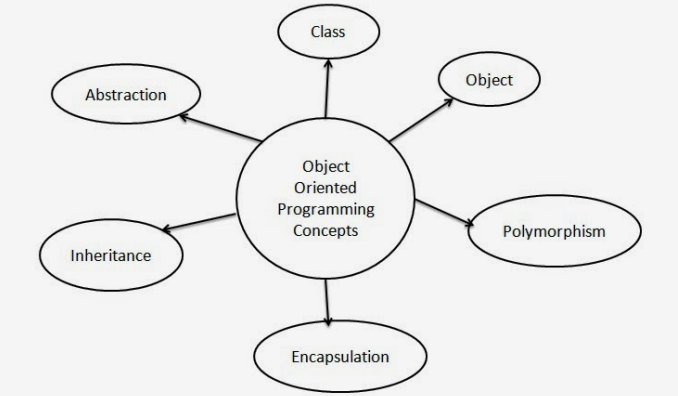
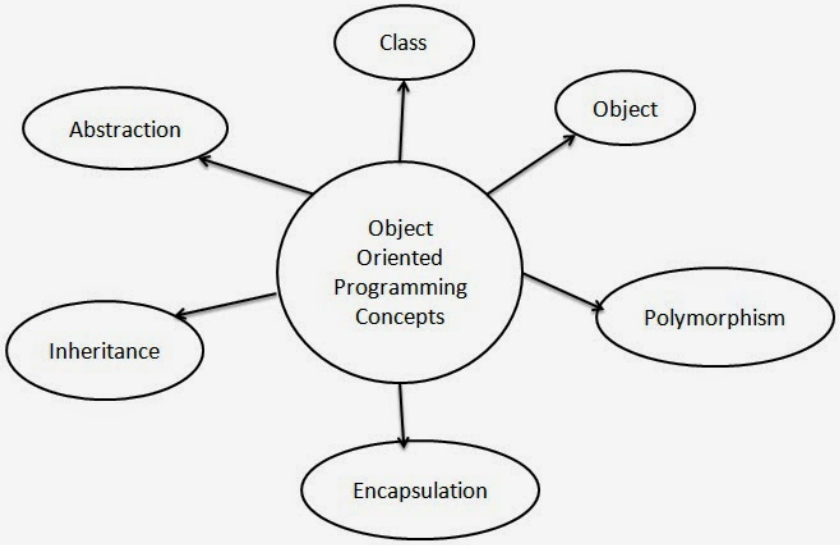


OOP concepts

Object-oriented Programming

- Fundamentals
  - Classes & Objects
  - Methods & Constructors
  - Encapsulation
  - Inheritance
- Libraries & Clients

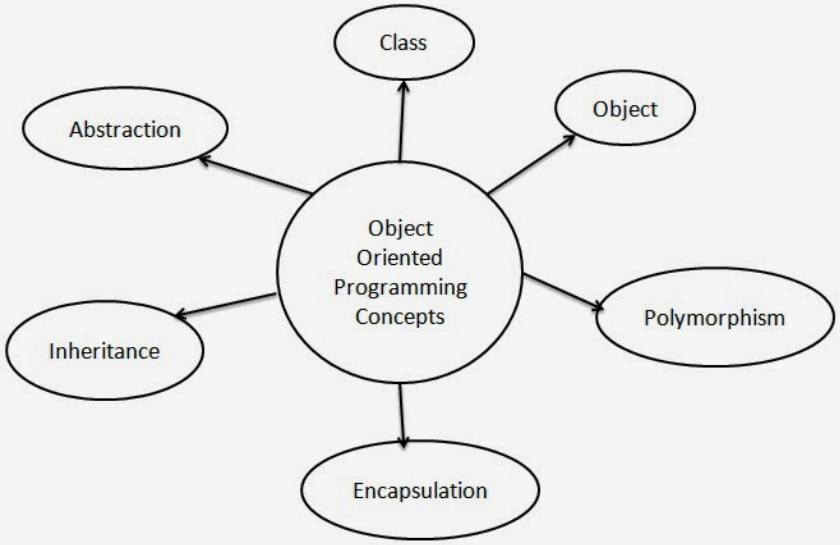


Object-oriented Programming

- Fundamentals
  - Classes & Objects
  - Methods & Constructors
  - Encapsulation
  - Inheritance
- Libraries & Clients

Java programming language was originally developed by **Sun Microsystems** which was initiated by James Gosling and **released in 1995** as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE]).

- **Object:** is the **basic unit** of object oriented programming. That is **both data and function that operate on data** are bundled as a unit called as object.
- **Class:** When you define a class, **you define a blueprint for an object**. This doesn't actually define any data, but it does define **what the class name** means, **what data (properties) an object of the class will consist of** and **what operations (functions) can be performed** on such an object.
- **Encapsulation:** is binding the class's data and behaviours together in a single unit. Also it is a language mechanism for **restricting access to some components** (achieved by access modifiers like **private**, **public**, **protected** etc.).
- **Inheritance:** One of the **most useful aspects of object-oriented programming** is **code reusability**. As the name suggests Inheritance is **the process of forming a new class** (called as derived class or subclass) **from an existing class** (called as base class or superclass or parent class). This feature **helps to reduce the code size**.



Object-oriented Programming

- Fundamentals
  - Classes & Objects
  - Methods & Constructors
  - Encapsulation
  - Inheritance
- Libraries & Clients

- **Abstraction:** is a process where you **show only "relevant" data and "hide" unnecessary details of an object** from the user. Consider your mobile phone, you just need to know what buttons are to be pressed to send a message or make a call, what happens when you press a button, how your messages are sent, how your calls are connected is all abstracted away from the user.

- **Polymorphism:** the ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.

- **Overloading:** the concept of overloading is also **a branch of polymorphism**. When the exiting operator or function is made to **operate on new data type**, it is said to be overloaded.

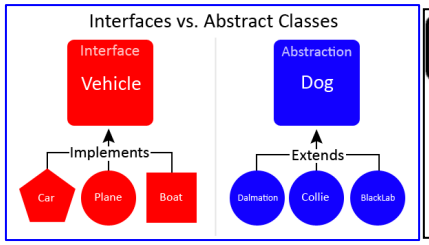
- **Override:** to override the **functionality of an existing method**. If a class inherits a method from its **superclass**, then there is a chance to override the method provided that it is not marked final. The benefit of overriding is: ability to define a behaviour that's specific to the **subclass type**, which means a subclass can implement a parent class method based on its requirement.

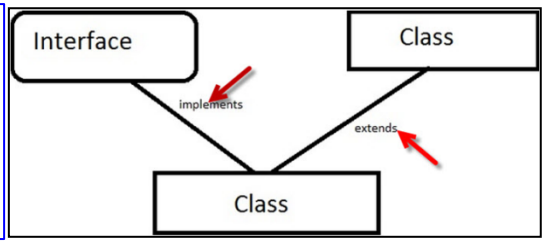
```
package dang.daniel.project.ames.ac.nz.lab02_1_randomnumbergenerator;

import android.graphics.Color;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.RelativeLayout;
import android.widget.TextView;
import java.util.Random;
```

Import classes

```
public class MainActivity extends AppCompatActivity implements View.OnClickListener {
    ////////////////////////////////////////////////////////////////////////////////////////////////////
    //1) Declare instance variables
    private int operand1 = 0;
    private int operand2 = 0;

    //Define minimum and maximum numbers for each operand by using 2D-array
    //Example: operand 1: the minimum=10, the maximum=20 => 10 <= operand1 <= 20
    //         operand 2: the minimum=80, the maximum=80 => 80 <= operand1 <= 90
    private int[][] operandRange = {{10, 20}, {80, 90}};

    //Declare an instance variable for a random number generator
    private Random random;

    //Declare instance variables for the user interface elements that we defined in the layout
    private TextView operand1Txt, operand2Txt;
    private TextView sum;
    private TextView title;
    private Button generate;
    private RelativeLayout mainRelativeLayout;

    ////////////////////////////////////////////////////////////////////////////////////////////////////
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main_screen);

        ////////////////////////////////////////////////////////////////////////////////////////////////////
        //2) Find Widget elements and do casting
        operand1Txt = (TextView) findViewById(R.id.operand1);
        operand2Txt = (TextView) findViewById(R.id.operand2);
        generate = (Button) findViewById(R.id.generate);

        sum = (TextView) findViewById(R.id.sum);
        title = (TextView) findViewById(R.id.gameTitle);
        mainRelativeLayout = (RelativeLayout) findViewById(R.id.mainRelativeLayout);

        ////////////////////////////////////////////////////////////////////////////////////////////////////
        //3)Set Listeners for these Widget control elements
        generate.setOnClickListener(this);

        //Initialise a random number generator
        random = new Random();

        //Set two operand1 and operand2 "0"
        operand1Txt.setText("0");
        operand2Txt.setText("0");
    }

    ////////////////////////////////////////////////////////////////////////////////////////////////////
    //Override the onClick() method
    @Override
    public void onClick(View myView) {
        ////////////////////////////////////////////////////////////////////////////////////////////////////
        //4)Check which Widget has been clicked and perform responses
        switch (myView.getId()) {
            case R.id.generate:
                //"generate" button has been clicked
                //Generate the operand 1 randomly in between: [10;20]
                operand1 = random.nextInt(operandRange[0][1] - operandRange[0][0]) + operandRange[0][0];
                //Display operand1 to TextView
                operand1Txt.setText(operand1 + "");

                //Generate the operand 1 randomly in between: [80;90]
                operand2 = random.nextInt(operandRange[1][1] - operandRange[1][0]) + operandRange[1][0];
                //Display operand1 to TextView
                operand2Txt.setText(operand2 + "");

                break;

            default:
                //Errors or Exceptions
                //Do nothing
                break;
        }
    }
}
```

Class/Object/Inheritance?
Interface?
Encapsulation?
Override?





What Is an Interface?

An interface is a collection of abstract methods (it means all methods are only declared in an Interface). A class implements an interface, thereby inheriting the abstract methods of the interface. And that class implements interface then you need to defined all abstract function which is present in an Interface.

An interface is not a class. Writing an interface is similar to writing a class, but they are two different concepts. A class describes the attributes and behaviors of an object. An interface contains behaviors that a class implements.

Constructors

- Constructors are special methods
- *They are* called only once when the class is being instantiated:
  Tax t = new Tax(40000, "CA",4);
- They must have the same name as the class.
- They can't return a value and you don't use void as a return type.

```
public class Tax {
    // class variables / fields
    private double grossIncome;
    private String state;
    private int dependents;

    // Constructor
    public Tax (double grossIncome, String state, int depen){
        // class variable initialization
        this.grossIncome = grossIncome;
        this.state = state;
        this.dependents = dependents;
    }
}
```

Method Overloading

- *Method overloading* means having a class with more than one method having the same name but different argument lists.

```
class LoanShark{
    int calcLoanPayment(int amount, int numberOfMonths){
        // by default, calculate for New York state
        calcLoanPayment(amount, numberOfMonths, "NY");
    }

    int calcLoanPayment(int amount, int numberOfMonths, String state){
        // Your code for calculating loan payments goes here
    }
}
```

A method can be overloaded not only in the same class but in a descendant too.

Method Overriding

- If a subclass has the method with the same name and argument list, it will *override* (suppress) the corresponding method of its ancestor.
- Method overriding comes handy in the following situations:
- The source code of the super class is not available, but you still need to change its functionality
- The original version of the method is still valid in some cases, and you want to keep it as is

Data Types

- A data type is a set of values and operations on those values.
  - **String** for text processing
  - **double, int** for mathematical calculation
  - **boolean** for decision making
- In Java, you must:
  - Declare type of values.
  - Convert between types when necessary
- Why do we need types?
  - Type conversion must be done at some level.
  - Compiler can help do it correctly.
  - Example: in 1996, Ariane 5 rocket exploded after takeoff because of bad type conversion.

String Data Type

- Useful for program input and output.

Data Type Attributes	
Values	sequence of characters
Typical literals	"Hello", "1 ", "*"
Operation	Concatenate
Operator	+

String Data Type

"1234" + " " + "99"

character

operator operator

Expression	Value
"Hi, " + "Bob"	"Hi, Bob"
"1" + "2" + "1"	"121"
"1234" + "+" + "99"	"1234+99"
"1234" + "99"	"123499"

Type Conversion and Casting

- Explicit conversion
  - Two types are incompatible
  - Syntax for conversion : (target-type) value;

double → float → long → int → short → byte

Example :

To convert `double` expressions to `int` requires a *typecasting* operation and *truncation* will occur

```
i = (int) (10.3 * x)
```

Creating 1-D Array

- *type arrayname [] = new type [size];*
- Example

double stockPrices [] = new double[10];

Classes

- Java is an object-oriented language
- Its constructs represent concepts from the real world.
- Each Java program has at least one class that knows how to do certain actions or has properties
- Classes in Java may have methods and properties (a.k.a. attributes or fields)

Objects

- Objects are created using Classes as a blueprint
- Each object is an instance of a Class
- Objects must be **instantiated** before they can be accessed or used in a program
- Objects are instantiated using the **new** keyword

Car Class

```
class Car{
   String color;
   int numberOfDoors;

   void startEngine() {
     // Some code goes here
   }
   void stopEngine () {
      int tempCounter=0;
    // Some code goes here
   }
}
```

Fields represent some attributes (properties) of a car – number of doors or color.

numberOfDoors is a variable of type `int` – to store integers; color can hold a string of characters (text)

Local variables are declared inside methods, Fields (a.k.a. member variables) - outside

Single-line comments start with //
Multi-line comments go between /* and */

Methods describe what our car can do: stop and start the engine.

This class has no main() method. What does it mean?

Creating Car Objects

- These two Car instances are created with the new operator:

```
Car car1 = new Car();
Car car2 = new Car();
```

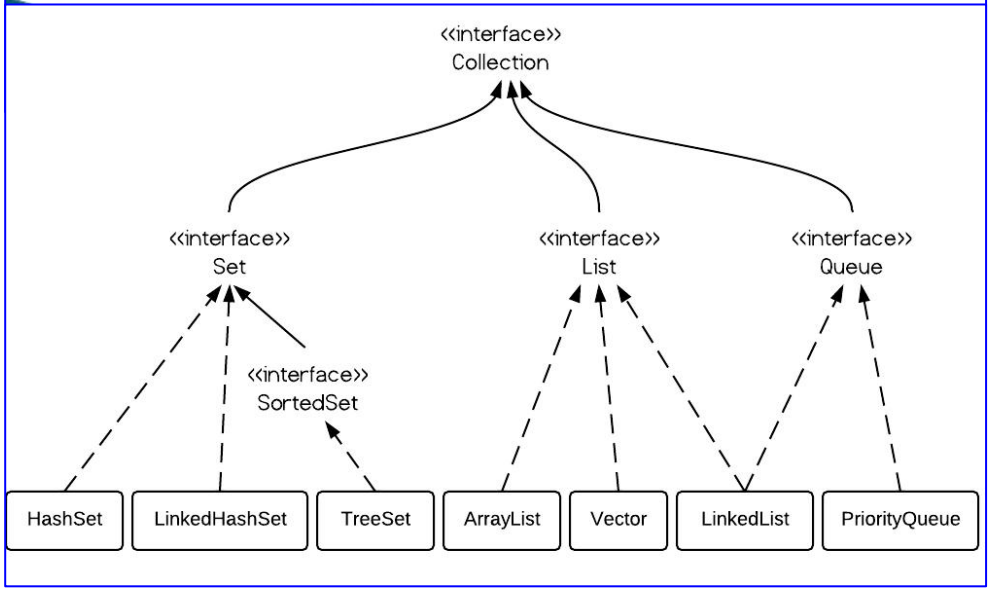
- Now the variables car1 and car2 represent new instances of Car:

```
car1.color="blue";
car2.color="red";
```

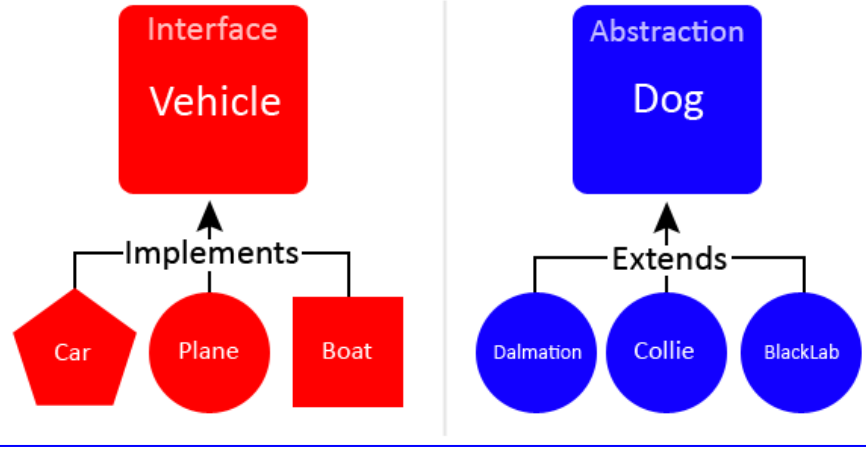
What Is an Interface?

An interface is a collection of abstract methods (it means all methods are only declared in an Interface). A class implements an interface, thereby inheriting the abstract methods of the interface. And that class implements interface then you need to defined all abstract function which is present in an Interface.

An interface is not a class. Writing an interface is similar to writing a class, but they are two different concepts. A class describes the attributes and behaviors of an object. An interface contains behaviors that a class implements.

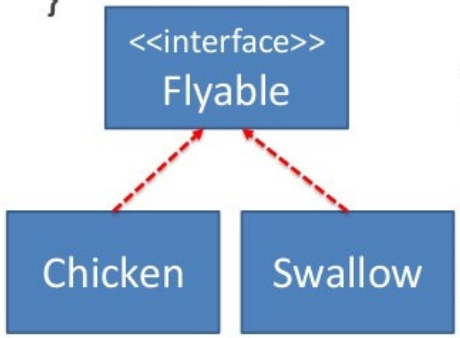


Interfaces vs. Abstract Classes



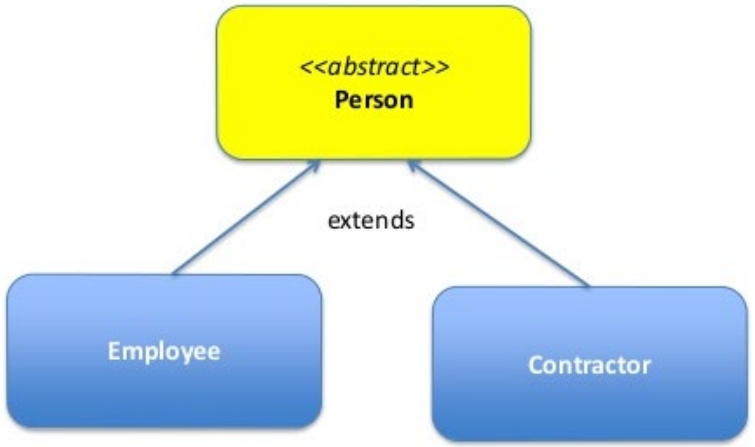
Interface

```
public interface Flyable {
    public String fly();
}
```




```
class Chicken implements Flyable{
    @Override
    public String fly() {
        return "Low and near";
    }
}
class Swallow implements Flyable{
    @Override
    public String fly() {
        return "High and far";
    }
}
```

Class Inheritance



Inheritance

- Ability to define a new class based on an existing one.
- E.g. lets create a James Bond Car from our existing Car class

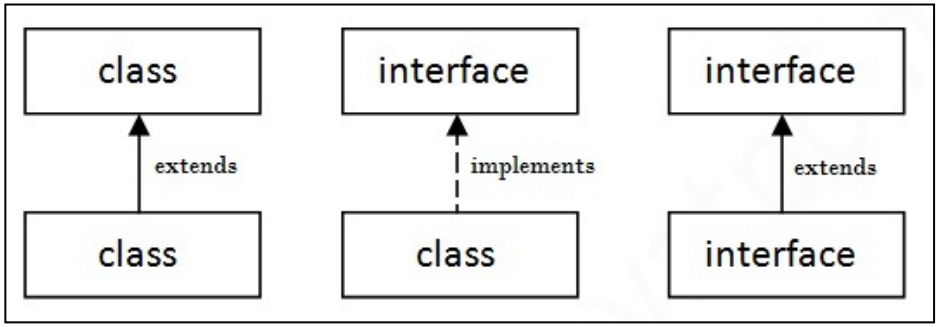
```
class JamesBondCar extends Car{

   int currentSubmergeDepth;
   boolean isGunOnBoard=true;
   final String MANUFACTURER;

   void submerge() {
      currentSubmergeDepth = 50;
     // Some code goes here
   }

   void surface() {
     // Some code goes here
   }
}
```

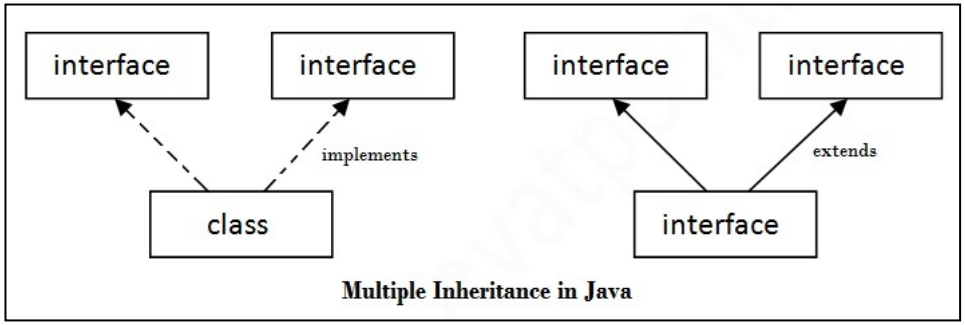
```
class JamesBondCar extends Car{

// ...

}
```

The class JamesBondCar has everything that the class Car has *plus something else*.

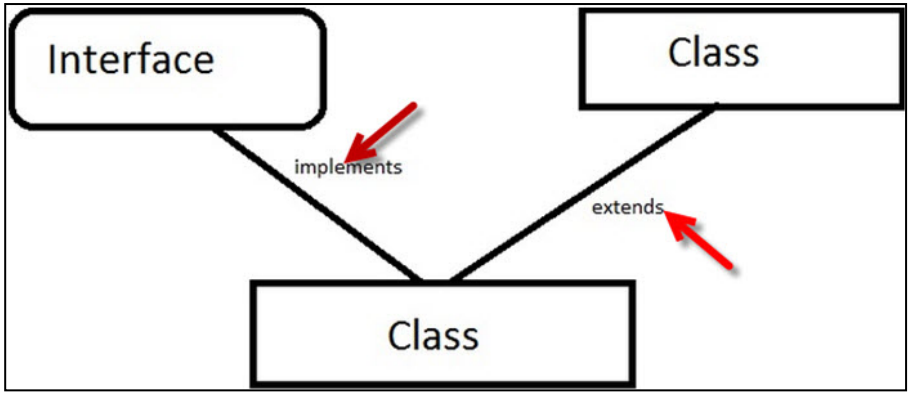
In this example, it defines:
- three more attributes

- two more methods.

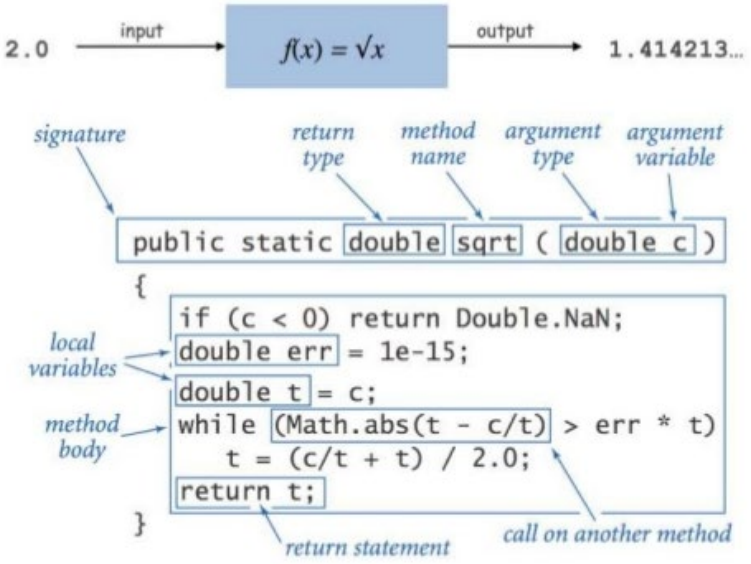




Multiple Inheritance in Java

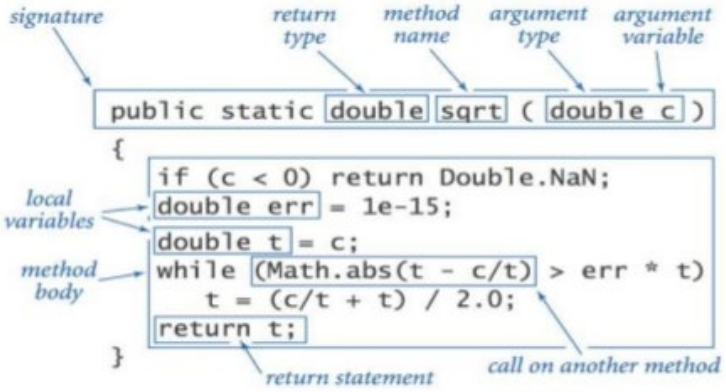


Anatomy of a Java Method



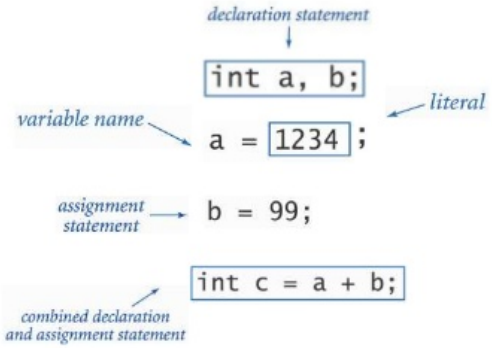
Scope

- The block of code that can refer to that named variable
- E.g. A variable's scope is code following in the block.
- Best practice: **declare variables to limit their scope**



Basic Definitions

- **Variable** - a name that refers to a value.
- **Assignment statement** - associates a value with a variable.



Type Conversion

- Convert from one type of data to another.
- Implicit
  - no loss of precision
  - with strings
- Explicit:
  - cast
  - method.

Constructors

- Constructors are special methods
- *They are* called only once when the class is being instantiated:

  Tax t = new Tax(40000, "CA",4);
- They must have the same name as the class.
- They can't return a value and you don't use void as a return type.

```
public class Tax {
    // class variables / fields
    private double grossIncome;
    private String state;
    private int dependents;

    // Constructor
    public Tax (double grossIncome, String state, int depen){
        // class variable initialization
        this.grossIncome = grossIncome;
        this.state = state;
        this.dependents = dependents;
    }
}
```

Method Overloading

- *Method overloading* means having a class with more than one method having the same name but different argument lists.

```
class LoanShark{
    int calcLoanPayment(int amount, int numberOfMonths){
        // by default, calculate for New York state
        calcLoanPayment(amount, numberOfMonths, "NY");
    }

    int calcLoanPayment(int amount, int numberOfMonths, String state){
        // Your code for calculating loan payments goes here
    }
}
```

A method can be overloaded not only in the same class but in a descendant too.

Method Overriding

- If a subclass has the method with the same name and argument list, it will *override* (suppress) the corresponding method of its ancestor.
- Method overriding comes handy in the following situations:
- The source code of the super class is not available, but you still need to change its functionality
- The original version of the method is still valid in some cases, and you want to keep it as is

Method Overriding :

Method overriding is nothing but the method in the child class should have the same name, same signature and parameters as the one in its parent class and also have the same return type.

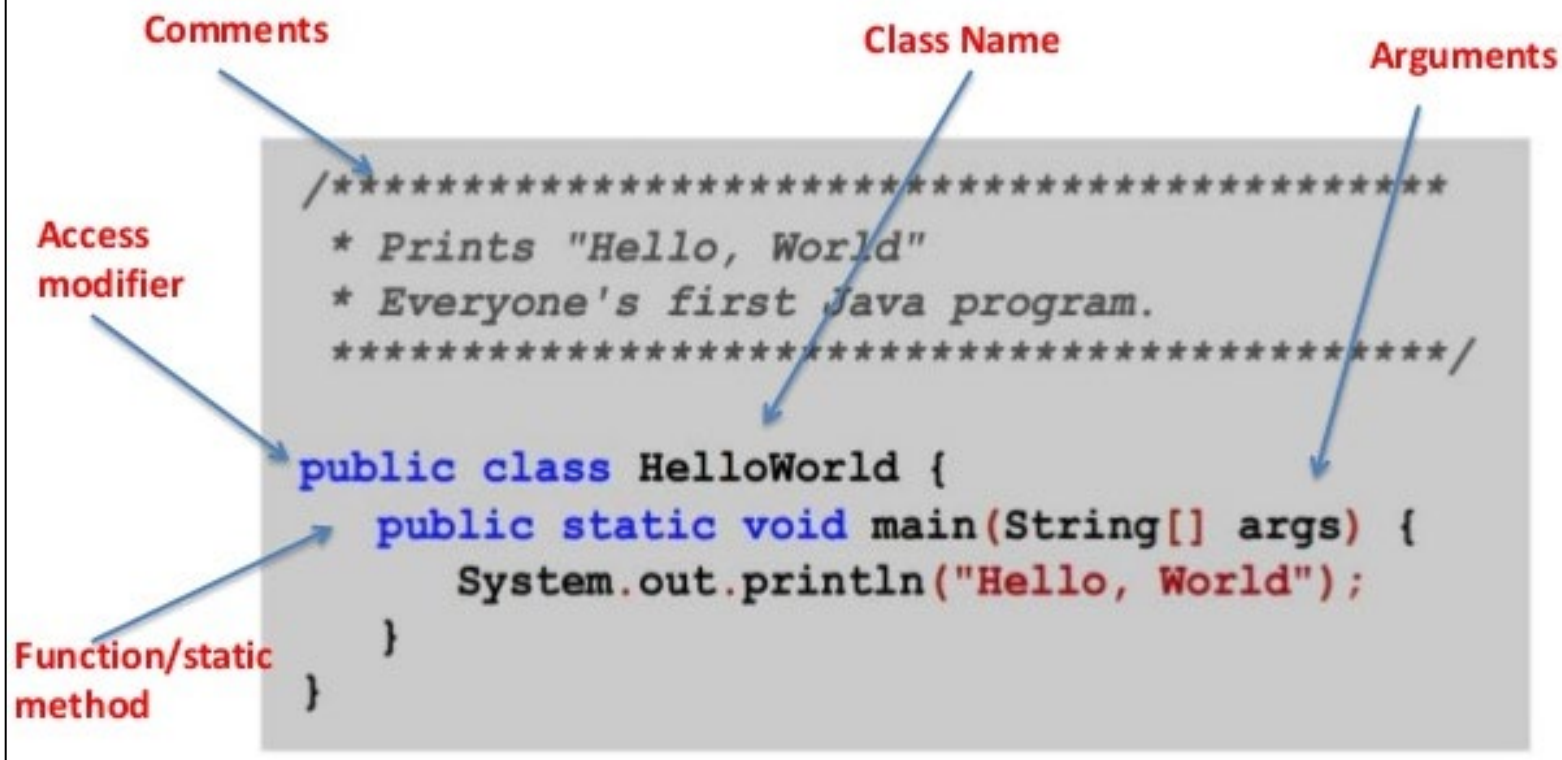
- If a method declared final then it cannot be overridden.
- If a method declared static then it cannot be overridden but it can be re-declared.
- If a method cannot be inherited, then it cannot be overridden. It is used for runtime polymorphism. It must be is-a relationship.
- Polymorphism is applied on method overriding. It is a run-time concept.
- Abstract methods must be overridden.
- Constructors cannot be overridden.
- Dynamic binding is used for method overriding.
- Private and final method cannot be overridden.

Method Overloading : It is nothing but in the same class, if name of the method remains same but the number and type of arguments or parameters are different, then it is called as method overloading.

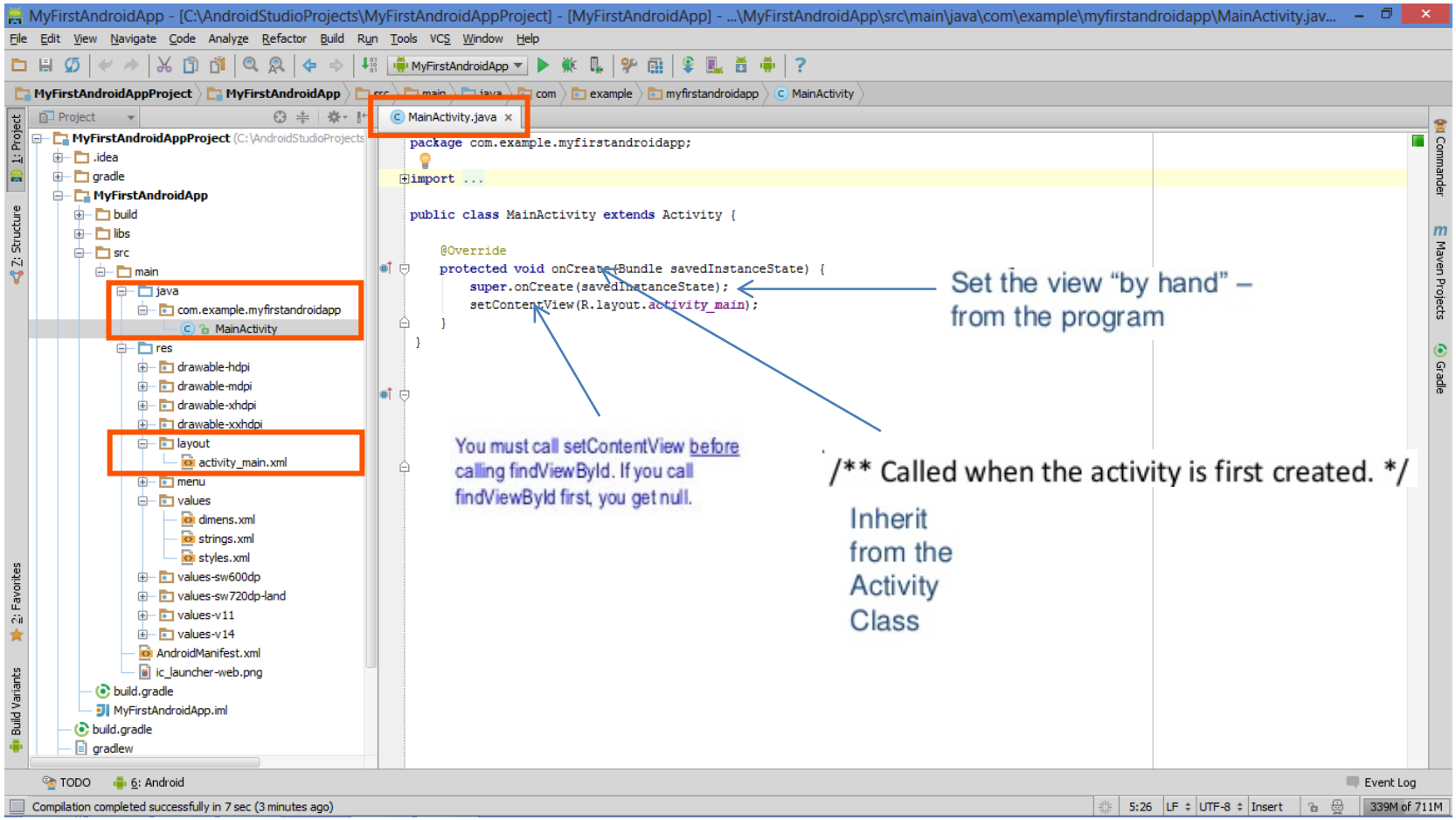
- This concept is used for compile-time. Present in the same class. And can have different return types.
- It helps in maintain consistency in method naming, doing same task with different parameter.
- It helps to reduce overhead.
- It is also known as static polymorphism.
- Static method can be overloaded. Static binding is used for method overloading. It gives better performance than method overriding.
- Private and final methods can be overloaded.
- In method overloading, return type should be same as the other methods of the same name.
- In method overloading, argument list should be different.

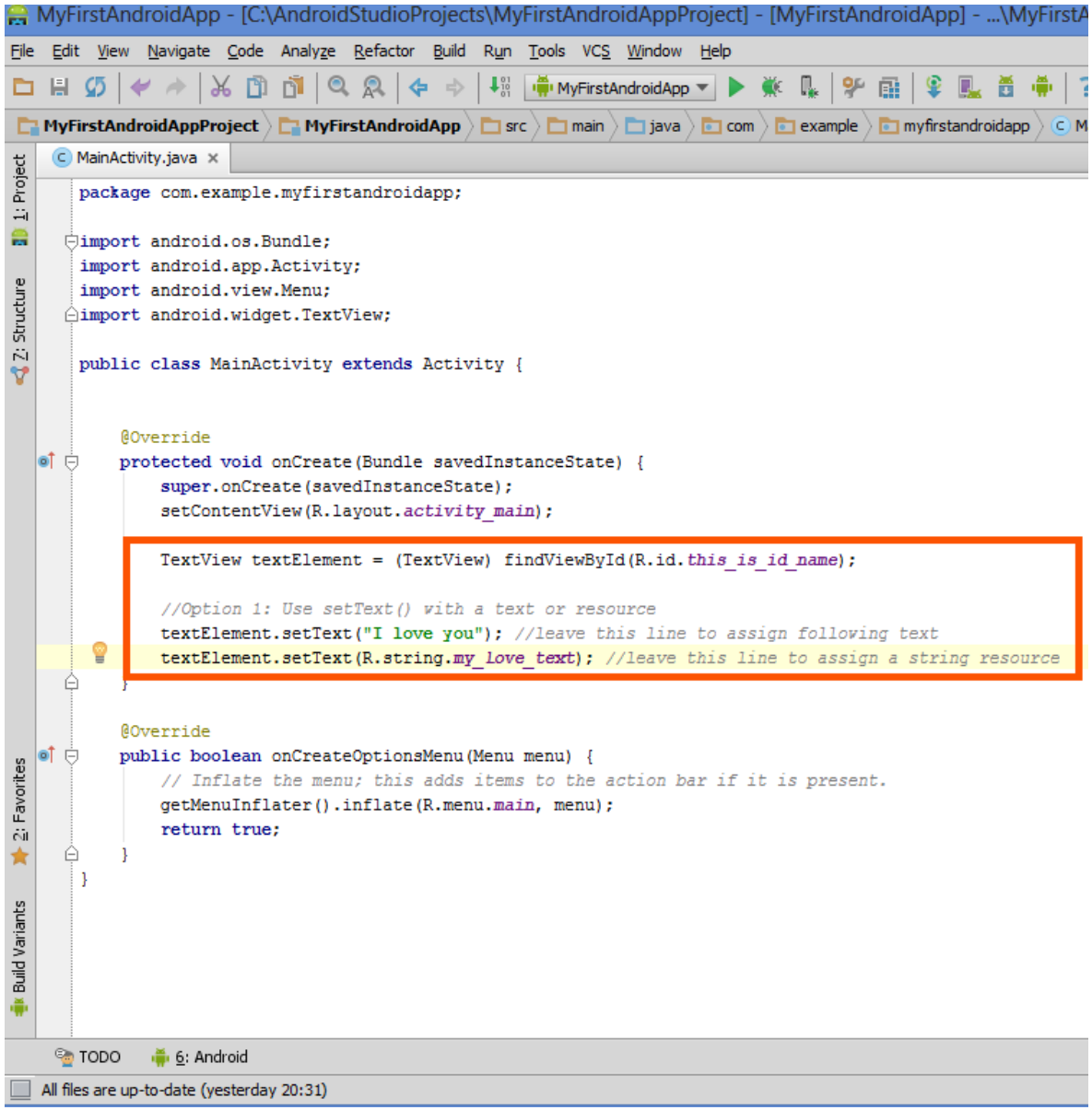
Java Programming

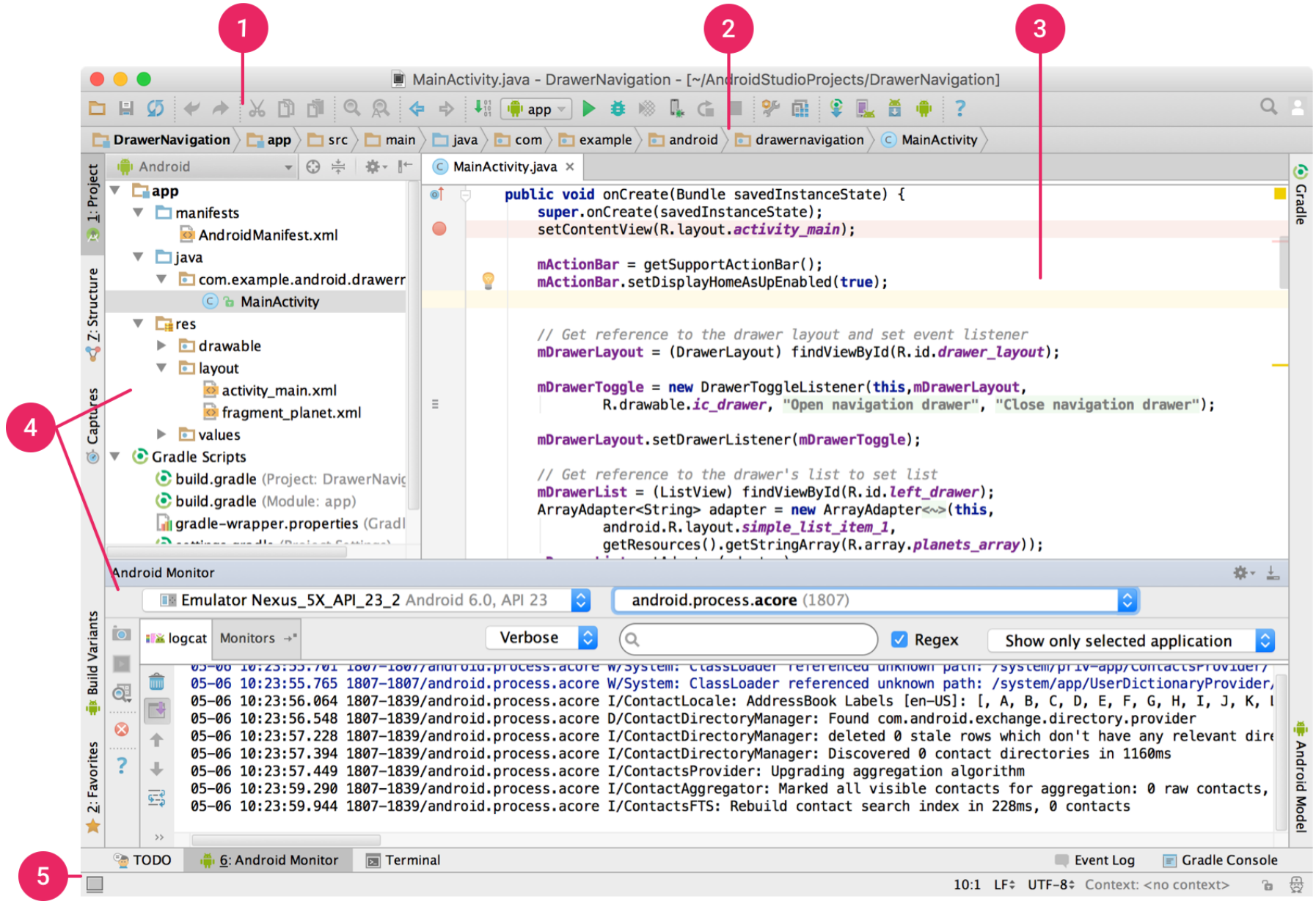
Anatomy of a Java Application



HelloWorld.java


MyFirstAndroidApp - [C:\AndroidStudioProjects\MyFirstAndroidAppProject] - [MyFirstAndroidApp] - ...\MyFirstAndroidApp\src\main\java\com\example\myfirstandroidapp\MainActivity.jav...
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
MyFirstAndroidAppProject
MyFirstAndroidApp
src
main
java
com
example
myfirstandroidapp
MainActivity
Project
MyFirstAndroidAppProject (C:\AndroidStudioProjects
.idea
gradle
MyFirstAndroidApp
build
libs
src
main
java
com.example.myfirstandroidapp
MainActivity
res
drawable-hdpi
drawable-mdpi
drawable-xhdpi
drawable-xxhdpi
layout
activity_main.xml
menu
values
dimens.xml
strings.xml
styles.xml
values-sw600dp
values-sw720dp-land
values-v11
values-v14
AndroidManifest.xml
ic_launcher-web.png
build.gradle
MyFirstAndroidApp.iml
build.gradle
gradlew
MainActivity.java
package com.example.myfirstandroidapp;
import ...
public class MainActivity extends Activity {
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
}
}
Set the view "by hand" – from the program
You must call setContentView before calling findViewById. If you call findViewById first, you get null.
/** Called when the activity is first created. */
Inherit from the Activity Class
1: Project
2: Favorites
Build Variants
Z: Structure
Commander
Maven Projects
Gradle
TODO
6: Android
Event Log
Compilation completed successfully in 7 sec (3 minutes ago)
5:26
LF
UTF-8
Insert
339M of 711M


MyFirstAndroidApp - [C:\AndroidStudioProjects\MyFirstAndroidAppProject] - [MyFirstAndroidApp] - ...\MyFirstA
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
MyFirstAndroidApp
MyFirstAndroidAppProject MyFirstAndroidApp src main java com example myfirstandroidapp
MainActivity.java
1: Project
7: Structure
2: Favorites
Build Variants
package com.example.myfirstandroidapp;
import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.widget.TextView;
public class MainActivity extends Activity {
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
TextView textElement = (TextView) findViewById(R.id.this_is_id_name);
//Option 1: Use setText() with a text or resource
textElement.setText("I love you"); //leave this line to assign following text
textElement.setText(R.string.my_love_text); //leave this line to assign a string resource
}
@Override
public boolean onCreateOptionsMenu(Menu menu) {
// Inflate the menu; this adds items to the action bar if it is present.
getMenuInflater().inflate(R.menu.main, menu);
return true;
}
}
TODO
6: Android
All files are up-to-date (yesterday 20:31)


1
2
3
4
5
MainActivity.java - DrawerNavigation - [~/AndroidStudioProjects/DrawerNavigation]
app
DrawerNavigation
app
src
main
java
com
example
android
drawernavigation
MainActivity
Android
MainActivity.java
app
manifests
AndroidManifest.xml
java
com.example.android.drawerr
MainActivity
res
drawable
layout
activity_main.xml
fragment_planet.xml
values
Gradle Scripts
build.gradle (Project: DrawerNavig
build.gradle (Module: app)
gradle-wrapper.properties (Gradl
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
mActionBar = getSupportActionBar();
mActionBar.setDisplayHomeAsUpEnabled(true);
// Get reference to the drawer layout and set event listener
mDrawerLayout = (DrawerLayout) findViewById(R.id.drawer_layout);
mDrawerToggle = new DrawerToggleListener(this,mDrawerLayout,
R.drawable.ic_drawer, "Open navigation drawer", "Close navigation drawer");
mDrawerLayout.setDrawerListener(mDrawerToggle);
// Get reference to the drawer's list to set list
mDrawerList = (ListView) findViewById(R.id.left_drawer);
ArrayAdapter<String> adapter = new ArrayAdapter<>(this,
android.R.layout.simple_list_item_1,
getResources().getStringArray(R.array.planets_array));
Android Monitor
Emulator Nexus_5X_API_23_2 Android 6.0, API 23
android.process.acore (1807)
logcat
Monitors
Verbose
Regex
Show only selected application
05-06 10:23:55.765 1807-1807/android.process.acore W/System: ClassLoader referenced unknown path: /system/app/UserDictionaryProvider,
05-06 10:23:56.064 1807-1839/android.process.acore I/ContactLocale: AddressBook Labels [en-US]: [, A, B, C, D, E, F, G, H, I, J, K,
05-06 10:23:56.548 1807-1839/android.process.acore D/ContactDirectoryManager: Found com.android.exchange.directory.provider
05-06 10:23:57.228 1807-1839/android.process.acore I/ContactDirectoryManager: deleted 0 stale rows which don't have any relevant dire
05-06 10:23:57.394 1807-1839/android.process.acore I/ContactDirectoryManager: Discovered 0 contact directories in 1160ms
05-06 10:23:57.449 1807-1839/android.process.acore I/ContactsProvider: Upgrading aggregation algorithm
05-06 10:23:59.290 1807-1839/android.process.acore I/ContactAggregator: Marked all visible contacts for aggregation: 0 raw contacts,
05-06 10:23:59.944 1807-1839/android.process.acore I/ContactsFTS: Rebuild contact search index in 228ms, 0 contacts
TODO
6: Android Monitor
Terminal
Event Log
Gradle Console
10:1
LF
UTF-8
Context: <no context>
1: Project
Z: Structure
Captures
Build Variants
2: Favorites
Gradle
Android Model

Intro to Object-oriented Programming with Java

Java Libraries

Definitions

- **Library** - a module whose methods are primarily intended for use by many other programs.
- **Client** - a program that calls a library.
- **API** - the contract between client and implementation.
- **Implementation** - a program that implements the methods in an API.

- Why use libraries?
  - Makes code easier to understand.
  - Makes code easier to debug.
  - Makes code easier to maintain and improve.
  - Makes code easier to reuse.

Standard Random API

public class StdRandom

int	uniform(int N)	*integer between 0 and N-1*
double	uniform(double lo, double hi)	*real between lo and hi*
boolean	bernoulli(double p)	true *with probability p*

Standard Random Implementation

```
public class StdRandom {
   //between 0 and N-1
   public static int uniform (int N){
      return (int)(Math.random() * N);
   }

   // between lo and hi
   public static double uniform (double lo, double hi){
      return lo + (int)(Math.random() * (hi-lo));
   }

   // truth with probability
   public static boolean bernoulli(double p){
      return Math.random() < p;
   }
}
```

Exercise: Random Number Generator

- Write a java class named **RandomInt** to generate a pseudo-random number between **0** and **N-1** where N is a number passed as an argument to the program

Solution: Random Number Generator

```
public class RandomInt {
   public static void main(String[] args) {
      int N = Integer.parseInt(args[0]);
      double r = Math.random();
      int n = (int) (r * N);

      System.out.println("random integer is " + n);
   }
}
```

String to int (method)

double between 0.0 and 1.0

double to int (cast)

int to double (automatic)

int to String (automatic)

```
% java RandomInt 6
random integer is 3
% java RandomInt 6
random integer is 0
% java RandomInt 10000
random integer is 3184
```

Language Features

Built-In Types	
int	double
long	String
char	boolean

System
System.out.println()
System.out.print()
System.out.printf()

Math Library	
Math.sin()	Math.cos()
Math.log()	Math.exp()
Math.sqrt()	Math.pow()
Math.min()	Math.max()
Math.abs()	Math.PI

Flow Control	
if	else
for	while

Parsing
Integer.parseInt()
Double.parseDouble()

Primitive Numeric Types		
+	-	*
/	%	++
--	>	<
<=	>=	==
!=		

Boolean	
true	false
||	&&
!	

Punctuation	
{	}
(	)
,	;

Assignment
=

Objects	
class	static
public	private
final	toString()
new	main()

String	
+	""
length()	compareTo()
charAt()	matches()

Arrays
a[i]
new
a.length

Data Types

- A data type is a set of values and operations on those values.
  - **String** for text processing
  - **double, int** for mathematical calculation
  - **boolean** for decision making
- In Java, you must:
  - Declare type of values.
  - Convert between types when necessary
- Why do we need types?
  - Type conversion must be done at some level.
  - Compiler can help do it correctly.
  - Example: in 1996, Ariane 5 rocket exploded after takeoff because of bad type conversion.

String Data Type

- Useful for program input and output.

Data Type Attributes	
Values	sequence of characters
Typical literals	"Hello", "1 ", "*"
Operation	Concatenate
Operator	+

String Data Type

```
"1234" + " + " + "99"
```

operator, character, operator

Expression	Value
"Hi, " + "Bob"	"Hi, Bob"
"1" + " 2 " + " 1"	" 1 2 1"
"1234" + " + " + "99"	"1234 + 99"
"1234" + "99"	"123499"

Boolean Data Type

Useful to control logic and flow of a program.

Data Type Attributes	
Values	true or false
Typical literals	true false
Operation	and or not
Operator	&& || !

Truth-table of Boolean Operations

a	!a	a	b	a && b	a || b
true	false	false	false	false	false
false	true	false	true	false	true
		true	false	false	true
		true	true	true	true

Boolean Comparisons

Take operands of one type and produce an operand of type **boolean**.

operation	meaning	true	false
==	equals	2 == 2	2 == 3
!=	Not equals	3 != 2	2 != 2
<	Less than	2 < 13	2 < 2
<=	Less than or equal	2 <= 2	3 <= 2
>	Greater than	13 > 2	2 > 13
>=	Greater than or equal	3 >= 2	2 >= 3

Integer Data Type

- Useful for expressing algorithms.

Data Type Attributes	
Values	Integers between -2E31 to +2E31-1
Typical literals	1234, -99 , 99, 0, 1000000
Operation	Add subtract multiply divide remainder
Operator	+ - * / %

Expression	Value	Comment
5 + 3	8	
5 – 3	2	
5 * 3	15	
5 / 3	1	no fractional part
5 % 3	2	remainder
1 / 0		run-time error
3 * 5 - 2	13	* has precedence
3 + 5 / 2	5	/ has precedence
3 – 5 - 2	-4	left associative
(3-5) - 2	-4	better style
3 – (5-2)	0	unambiguous

Double Data Type

Useful in scientific applications and floating- point arithmetic

Data Type Attributes	
Values	Real numbers specified by the IEEE 754 standard
Typical literals	3.14159 6.022e23 -3.0 2.0 1.41421356237209
Operation	Add subtract multiply divide
Operator	+ - * /

Expression	Value
3.141 + 0.03	3.171
3.141 – 0.03	3.111
6.02e23 / 2	3.01e23
5.0 / 2.0	1.6666666666667
10.0 % 3.141	0.577
1.0 / 0.0	Infinity
Math.sqrt(2.0)	1.4142135623730951

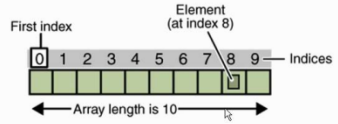
What is an array?

Dimensions	Example	Terminology
1	0 1 2	Vector
2	0 1 2 3 4 5 6 7 8	Matrix
3	0 1 2 3 4 5 6 7 8	3D Array (3rd order Tensor)
N	...	ND Array

Creating 1-D Array

- ***type arrayname [] = new type [size];***
- Example

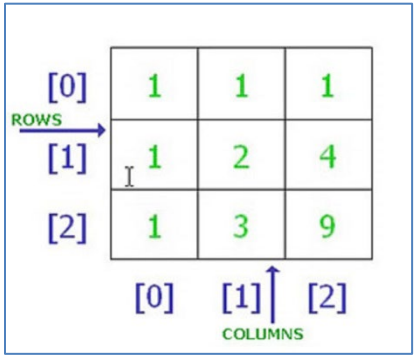
  double stockPrices [] = new double[10];




```
int resultArray[] = new int[6];
```

Type — Array Name — Array Size

Two-Dimensional Arrays



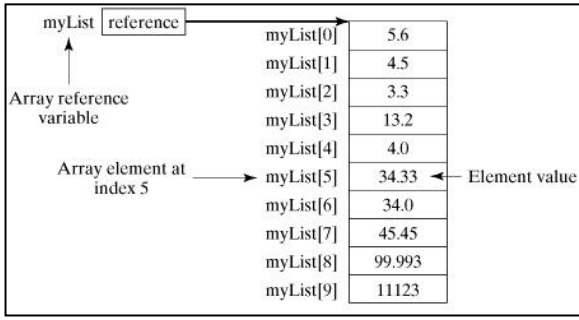

```
int M = 10;
int N = 3;
double[][] a = new double[M][N];
```

The Arrays Class

- Defined in the `java.util` package.
- Contains static methods for manipulating arrays:
  - `binarySearch`: search for a value.
  - `equals`: compare the contents of two arrays.
  - `fill`: fill an array with a particular value.
  - `sort`: sort the contents of an array.

Difference between array and an Arraylist

- An array is of fixed size
- An ArrayList can grow and reduce in size
- Any collection can grow and reduce in size but arrays cannot
- i.e. ArrayList list = new ArrayList();
  list.add(new Integer()); //is allowed
- But this is not allowed:
  Integer[] intArr = new Integer[3];
  intArr.add(new Integer());



```java
// Demonstrate a two-dimensional array.
class TwoD
{
  public static void main(String[] args)
  {
    int t, i;
    int table[][] = new int[3][4];

    for(t=0; t < 3; ++t)
    {
      for(i=0; i < 4; ++i)
      {
        table[t][i] = (t*4)+i+1;
        System.out.print(table[t][i] + " ");
      }
      System.out.println();
    }
  }
}
```

```
C:\Windows\system32\cmd.exe
1 2 3 4
5 6 7 8
9 10 11 12
```

- Array Access.
  - Use **a[i][j]** to access element in row **I** and column **j**.
  - Zero-based indexing.
  - Row and column indices start at **0**.

```java
int M = 10;
int N = 3;
double[][] a = new double[M][N];
for (int i = 0; i < M; i++) {
   for (int j = 0; j < N; j++) {
      a[i][j] = 0.0;
   }
}
```

a[][]		
a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]
a[2][0]	a[2][1]	a[2][2]
a[3][0]	a[3][1]	a[3][2]
a[4][0]	a[4][1]	a[4][2]
a[5][0]	a[5][1]	a[5][2]
a[6][0]	a[6][1]	a[6][2]
a[7][0]	a[7][1]	a[7][2]
a[8][0]	a[8][1]	a[8][2]
a[9][0]	a[9][1]	a[9][2]

A 10-by-3 array

```java
public class TwoDimensionalArrayDemo{

    public static void main(String args[]) {

        // let's create board of 4x4
        int[][] board = new int[4][4];

        // let's loop through array to populate board
        for (int row = 0; row < board.length; row++) {
            for (int col = 0; col < board[row].length; col++) {
                board[row][col] = row * col;
            }
        }

        // let's loop through array to print each row and column
        for (int row = 0; row < board.length; row++) {
            for (int col = 0; col < board[row].length; col++) {
                board[row][col] = row * col;
                System.out.print(board[row][col] + "\t");
            }
            System.out.println();
        }
    }

}
```

```
0   0   0   0
0   1   2   3
0   2   4   6
0   3   6   9
```

create an array with random values	`double[] a = new double[n];`
	`for (int i = 0; i < n; i++)`
	`  a[i] = Math.random();`

print the array values, one per line	`for (int i = 0; i < n; i++)` `  System.out.println(a[i]);`

find the maximum of the array values	`double max = Double.NEGATIVE_INFINITY;`
	`for (int i = 0; i < n; i++)`
	`  if (a[i] > max) max = a[i];`

compute the average of the array values	`double sum = 0.0;`
	`for (int i = 0; i < n; i++)`
	`  sum += a[i];`
	`double average = sum / n;`

reverse the values within an array	`for (int i = 0; i < n/2; i++)`
	`{`
	`  double temp = a[i];`
	`  a[i] = a[n-1-i];`
	`  a[n-i-1] = temp;`
	`}`

copy sequence of values to another array	`double[] b = new double[n];`
	`for (int i = 0; i < n; i++)`
	`  b[i] = a[i];`

```java
import java.util.Arrays;

/**
 * JUnit test cases for program to find maximum contiguous sum in array of positive
 * or negative integers.
 *
 * @author WINDOWS 8 *
 */
public class ArrayToList {


    public static void main(String args[]){

        List<String> movies = Arrays.asList("Captain America", "Avtaar", "Harry Potter");
        String[] arrayOfMovies = new String[movies.size()];
        movies.toArray(arrayOfMovies);

        System.out.println("list of String : " + movies);
        System.out.println("array of String : " + Arrays.toString(arrayOfMovies));
    }
}
```

Type Conversion

- Convert from one type of data to another.
- Implicit
  - no loss of precision
  - with strings
- Explicit:
  - cast
  - method.

double → float → long → int → short → byte

Narrowing

byte → short → int → long → float → double

widening

Type Conversion and Casting

- Explicit conversion
  - Two types are incompatible
  - Syntax for conversion : (target-type) value;

  double → float → long → int → short → byte

  Example :

  To convert `double` expressions to `int` requires a *typecasting* operation and *truncation* will occur

```
i = (int) (10.3 * x)
```

Type Conversion Examples

expression	Expression type	Expression value
"1234" + 99	String	"123499"
Integer.parseInt("123")	int	123
(int) 2.71828	int	2
Math.round(2.71828)	long	3
(int) Math.round(2.71828)	int	3
(int) Math.round(3.14159)	int	3
11 * 0.3	double	3.3
(int) 11 * 0.3	double	3.3
11 * (int) 0.3	int	0
(int) (11 * 0.3)	int	3

```
XDocument contactFile = XDocument.Load(Server.MapPath("Contacts.xml"));

var contacts = from c in contactFile.Descendants("Contact")
               select new {
                  Name = (string) c.Element("Name"),
                  Title = (string) c.Element("Title"),
                  Email = (string) c.Element("Email"),
                  YearsAtCompany = (int?) c.Element("YearsAtCompany")
               };

GridView1.DataSource = contacts;
GridView1.DataBind();
```

```java
public class Promote {
    public static void main(String[] args) {
        byte b = 1; // promoted to float
        char c = 'a'; // promoted to integer
        short s = 1024; // promoted to double
        int i = 1000000; // promoted to float
        float f = 5.67f; // promoted to double
        double d = 1.100023005;
        System.out.println("The result of b*f: " + (
        f * b));
        System.out.println("The result of i/c: " + (
        i / c));
        System.out.println("The result of d*s: " + (
        d * s));
        double result = ( (f * b) + (i / c) - (d *
        s) );
        System.out.println( (f * b) + " + " + (i /
        c) + " - " + (d * s) );
        System.out.println("result = " + result);
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
The result of b*f: 5.67
The result of i/c: 10309
The result of d*s: 1126.42355712
5.67 + 10309 - 1126.42355712
result = 9188.246364755
```

Control Flow

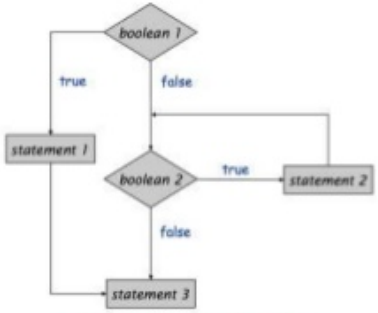
- Sequence of statements that are actually executed in a program.
- Conditionals and loops enable us to choreograph the control flow.

Control flow	Description	Example
Straight line programs	all statements are executed in the order given	
Conditionals	certain statements are executed depending on the values of certain variables	If If-else
Loops	certain statements are executed repeatedly until certain conditions are met	while for do-while

Conditionals and Loops

- Sequence of statements that are actually executed in a program.
- Enable us to choreograph control flow.

straight-line control flow



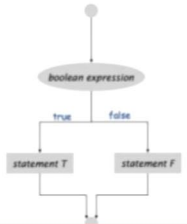
control flow with conditionals and loops

Conditionals

- The **if** statement is a common branching structure.
  - Evaluate a **boolean** expression.
    - If **true**, execute some statements.
    - If **false**, execute other statements.

```
if (boolean expression) {
   statement T;
}
else {
   statement F;
}
```

can be any sequence of statements



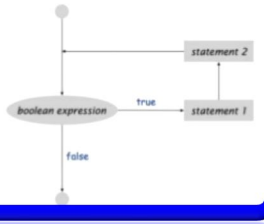
While Loop

- A common repetition structure.
  - Evaluate a **boolean** expression.
  - If **true**, execute some statements.
  - Repeat.

loop continuation condition

```
while (boolean expression) {
   statement 1;
   statement 2;
}
```

loop body



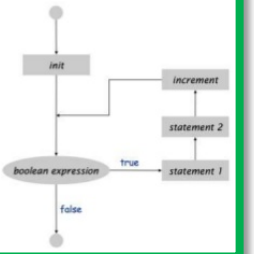
For Loop

- Another common repetition structure.
  - Execute initialization statement.
  - Evaluate a **boolean** expression.
    - If **true**, execute some statements.
  - And then the increment statement.
  - Repeat.

loop continuation condition

```
for (init; boolean expression; increment) {
   statement 1;
   statement 2;
}
```

body

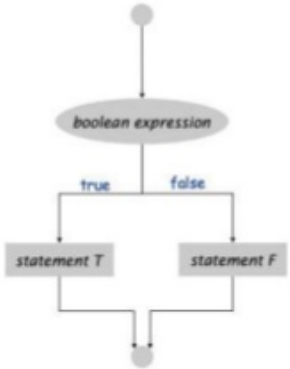


Conditionals

- The **if** statement is a common branching structure.
  - Evaluate a **boolean** expression.
    - If **true**, execute some statements.
    - If **false**, execute other statements.

```
if (boolean expression) {
   statement T;
}
else {
   statement F;
}
```

can be any sequence of statements



More If Statement Examples

absolute value	`if (x < 0) x = -x;`

put x and y into sorted order	`if (x > y)`
	`{`
	`  int t = x;`
	`  x = y;`
	`  y = t;`
	`}`

maximum of x and y	`if (x > y) max = x;` `else  max = y;`
error check for division operation	`if (den == 0) System.out.println("Division by zero");` `else  System.out.println("Quotient = " + num/den);`

error check for quadratic formula	`double discriminant = b*b - 4.0*c;`
	`if (discriminant < 0.0)`
	`{`
	`  System.out.println("No real roots");`
	`}`
	`else`
	`{`
	`  System.out.println((-b + Math.sqrt(discriminant))/2.0);`
	`  System.out.println((-b - Math.sqrt(discriminant))/2.0);`
	`}`

Loop Examples

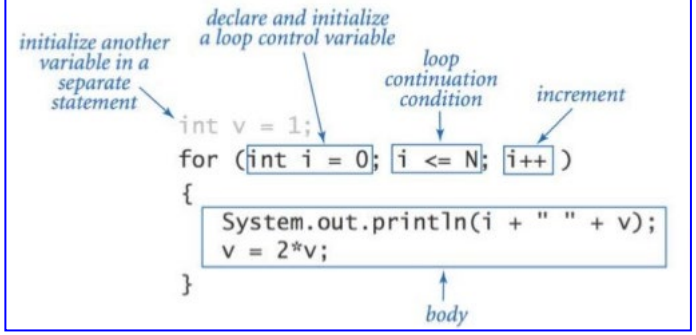
print largest power of two less than or equal to N	`int v = 1;`
	`while (v <= N/2)`
	`  v = 2*v;`
	`System.out.println(v);`

compute a finite sum $(1 + 2 + \ldots + N)$	`int sum = 0;`
	`for (int i = 1; i <= N; i++)`
	`  sum += i;`
	`System.out.println(sum);`

compute a finite product $(N! = 1 \times 2 \times \ldots \times N)$	`int product = 1;`
	`for (int i = 1; i <= N; i++)`
	`  product *= i;`
	`System.out.println(product);`

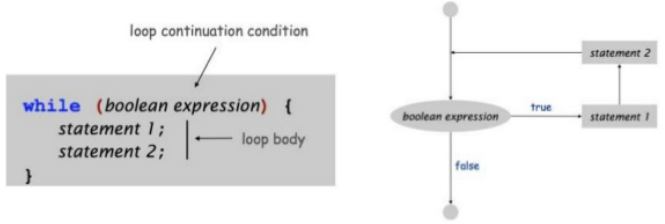
print a table of function values	`for (int i = 0; i <= N; i++)` `  System.out.println(i + " " + 2*Math.PI*i/N);`

Anatomy of a For Loop



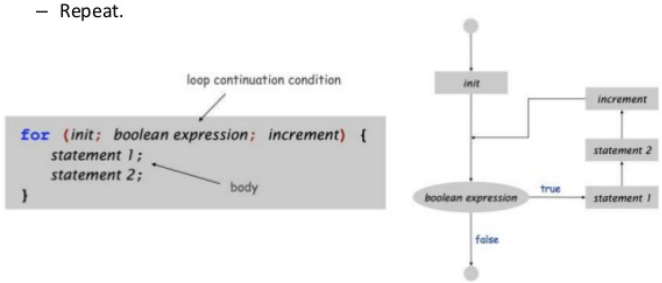
While Loop

- A common repetition structure.
  - Evaluate a **boolean** expression.
  - If **true**, execute some statements.
  - Repeat.



For Loop

- Another common repetition structure.
  - Execute initialization statement.
  - Evaluate a **boolean** expression.
    - If **true**, execute some statements.
  - And then the increment statement.
  - Repeat.



Java programming language was originally developed by **Sun Microsystems** which was initiated by James Gosling and **released in 1995** as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE]).

- **Object:** is the **basic unit** of object oriented programming. That is **both data and function that operate on data** are bundled as a unit called as object.
- **Class:** When you define a class, **you define a blueprint for an object**. This doesn't actually define any data, but it does define **what the class name** means, **what data (properties) an object of the class will consist of** and **what operations (functions) can be performed** on such an object.
- **Encapsulation:** is binding the class's data and behaviours together in a single unit. Also it is a language mechanism for **restricting access to some components** (achieved by access modifiers like **private**, **public**, **protected** etc.).
- **Inheritance:** One of the **most useful aspects of object-oriented programming** is **code reusability**. As the name suggests Inheritance is **the process of forming a new class** (called as derived class or subclass) **from an existing class** (called as base class or superclass or parent class). This feature **helps to reduce the code size**.

- **Abstraction:** is a process where you **show only "relevant" data and "hide" unnecessary details of an object** from the user. Consider your mobile phone, you just need to know what buttons are to be pressed to send a message or make a call, what happens when you press a button, how your messages are sent, how your calls are connected is all abstracted away from the user.
- **Polymorphism:** the ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.
- **Overloading:** the concept of overloading is also **a branch of polymorphism**. When the exiting operator or function is made to **operate on new data type**, it is said to be overloaded.
- **Override:** to override the **functionality of an existing method**. If a class inherits a method from its **superclass**, then there is a chance to override the method provided that it is not marked final. The benefit of overriding is: ability to define a behaviour that's specific to the **subclass type**, which means a subclass can implement a parent class method based on its requirement.