Introduction to Java

A prequel to Android development

Why Android?

- Worldwide, 72% of smartphones are Android-based.
- In the US, 50% of smartphones are Android-based.
- In the US, the average salary of an Android developer is 93K. In San Francisco, the average salary of an Android developer is 120K.
- Mobile development (smartphones and tablet) is an important focus for most product companies.
- Developing mobile apps is fun!

Who is this course for?

This is a course tailored for people who want to develop Android applications.

- No previous programming experience required.
- Java exercises will be similar to tasks needed when developing an Android application.
- This course will cover the minimum necessary to begin Android development.
- Next course: Introduction to Android, a project-based course.

Goal and scope

Goal

- To be comfortable with key programming fundamentals and the Java syntax required for Android development.

Scope

- Programming Fundamentals (Variables, Loops, Arrays, etc)
- Object Oriented Programming (Classes, Inheritance)

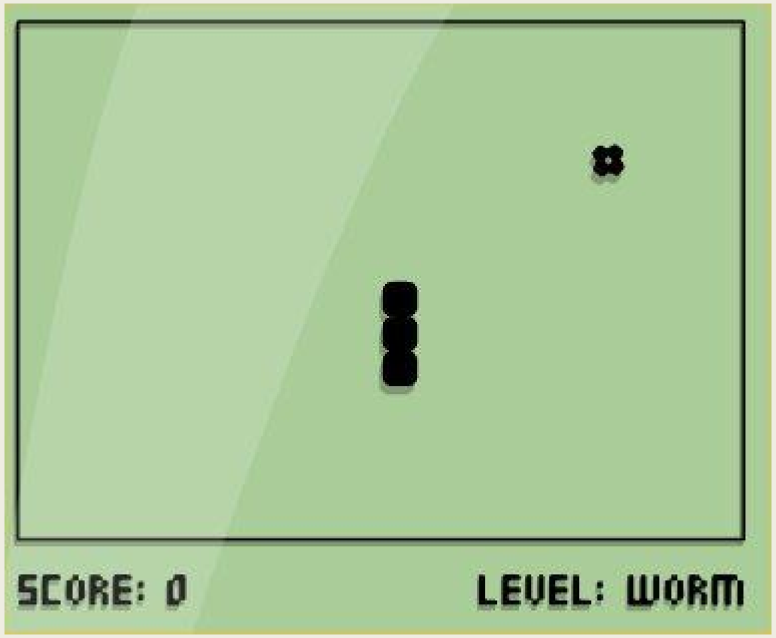
- **Next course**: Introduction to Android

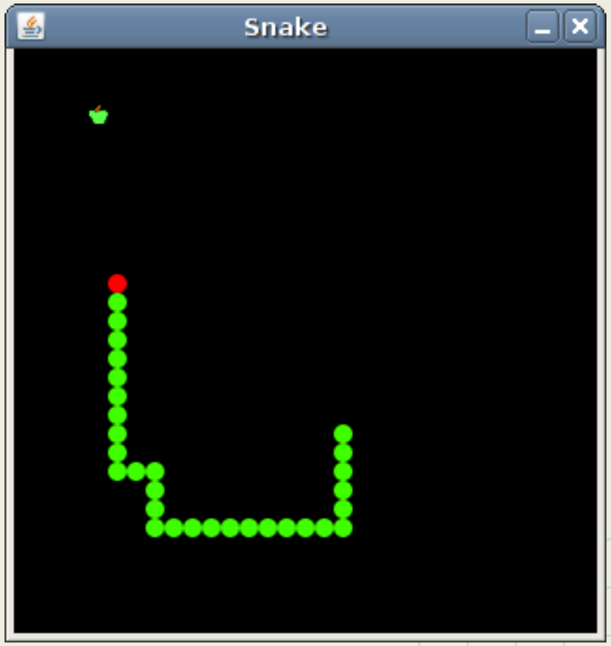
Class Structure

- Lesson Module (e.g., Variables or Loops)
  - Brief conceptual overview
  - Short hands-on exercises
  - Course project (building 2D snake game)
- We cover each lesson module in turn building on previous concepts.

Course Project

Over the course of the lessons, we will be building a 2D **Snake Game** in Java.





Setup Checklist

Let's make sure we have everything we need to get started.

- **Java and Eclipse IDE**
- **eGit Plugin for Eclipse**
- **Two Eclipse Projects (from Git):**
    - **Basic Exercises**
      git@github.com:thecodepath/intro_java_exercises.git
    - **Course Project (2D Game)**
      git@github.com:thecodepath/snake_exercises.git

Introduction

Understanding Programs

What is Programming?

- Computers are machines that can **run programs**.
- Programs describe **simple steps** for a computer to perform.
- Programs start from the **top** and follow the instructions on **each line** until a program is completed.

Types of Programs

- Programs that **you can see** and interact with (GUI)
- Programs that **you can't see** (Background services)

Development Platforms

- There are many programming **languages** available for use to develop software and many **frameworks** to develop different types of applications.

Java & Android

Objective-C & Cocoa

Ruby & Rails

Hello World

Understanding our first program

Hello World!

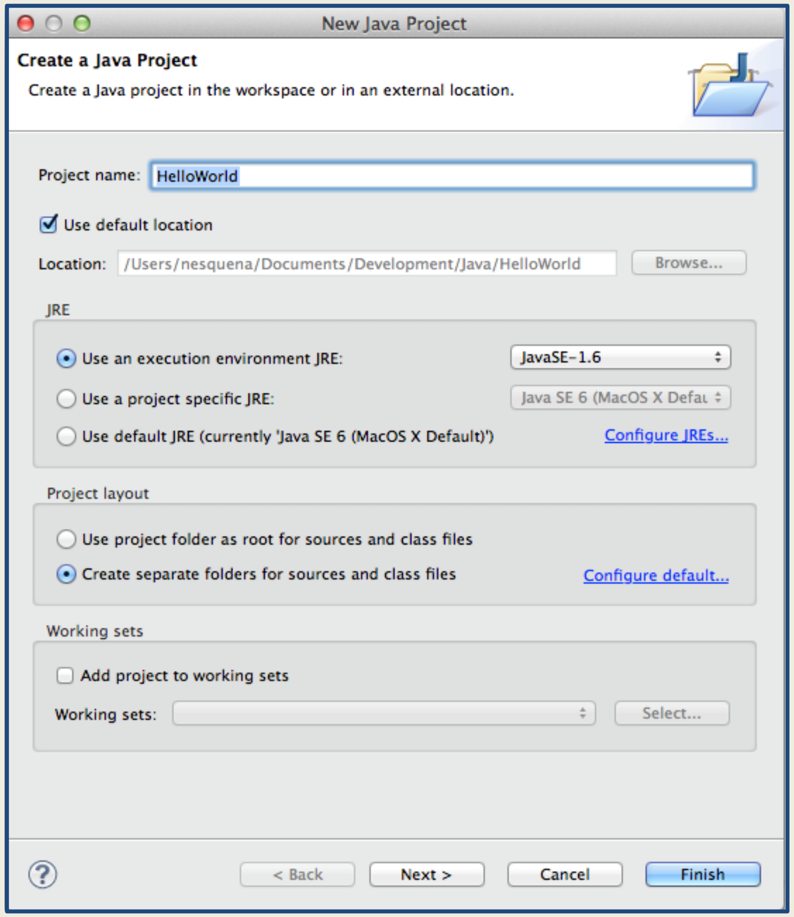
Let's **create** a simple Java program.

Select **File** → **New** to create new things.

- We are going to create a **New Project**
- ...and create a **New Class**.

Hello World!

- Let's **create** a simple Java program.



New Java Project

Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: HelloWorld

[✓] Use default location

Location: /Users/nesquena/Documents/Development/Java/HelloWorld   Browse...

JRE

(●) Use an execution environment JRE: JavaSE-1.6

( ) Use a project specific JRE: Java SE 6 (MacOS X Defau

( ) Use default JRE (currently 'Java SE 6 (MacOS X Default)')   Configure JREs...

Project layout

( ) Use project folder as root for sources and class files

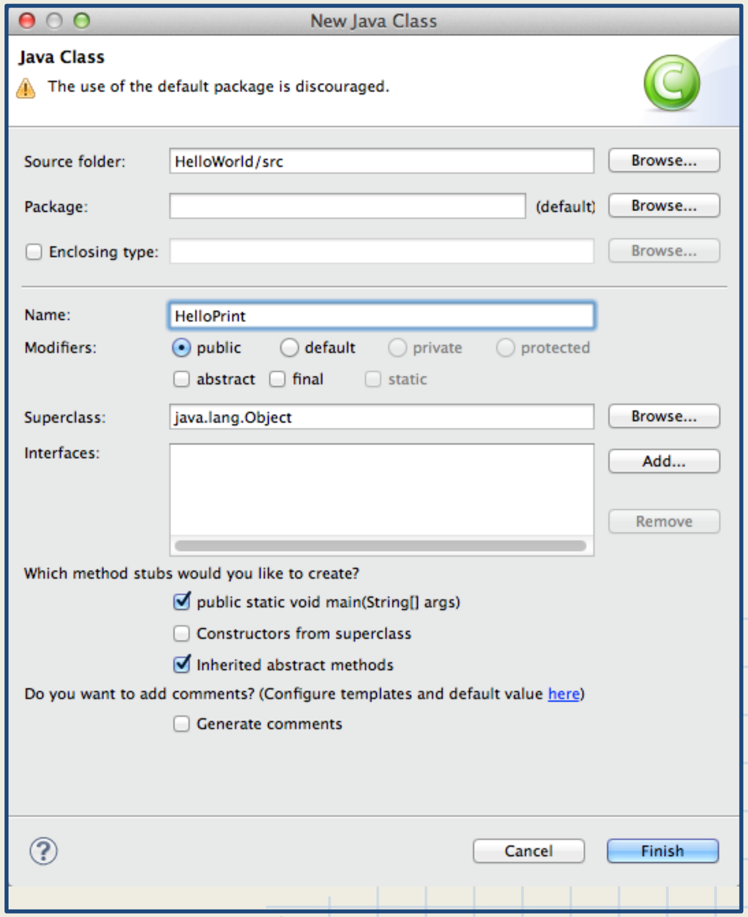
(●) Create separate folders for sources and class files   Configure default...

Working sets

[ ] Add project to working sets

Working sets: Select...

< Back   Next >   Cancel   Finish



New Java Class

Java Class

The use of the default package is discouraged.

Source folder: HelloWorld/src   Browse...

Package: (default)   Browse...

[ ] Enclosing type: Browse...

Name: HelloPrint

Modifiers: (●) public ( ) default ( ) private ( ) protected

[ ] abstract [ ] final [ ] static

Superclass: java.lang.Object   Browse...

Interfaces: Add...   Remove

Which method stubs would you like to create?

[✓] public static void main(String[] args)

[ ] Constructors from superclass

[✓] Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

[ ] Generate comments

Cancel   Finish

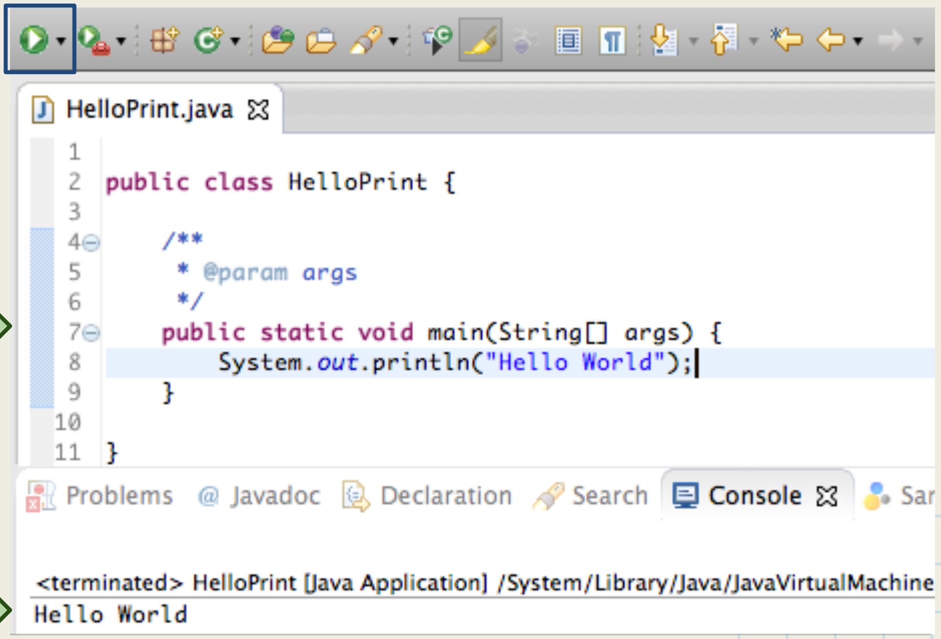
Hello World!

- Let's **run** the program in Eclipse.
- Click the Green **Run** Button.
- Check the **Console** for "Hello World"

Run Program

Edit Code

View Console



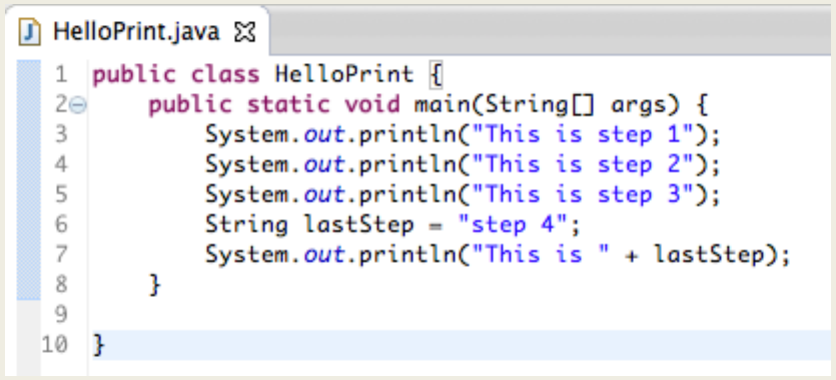
```
public class HelloPrint {

    /**
     * @param args
     */
    public static void main(String[] args) {
        System.out.println("Hello World");
    }

}
```

```
<terminated> HelloPrint [Java Application] /System/Library/Java/JavaVirtualMachine
Hello World
```

Hello World! Evolves

- Let's make a few changes to our program.
- Type the changes shown below.
- Be careful of **capitalization** and **semicolons**.

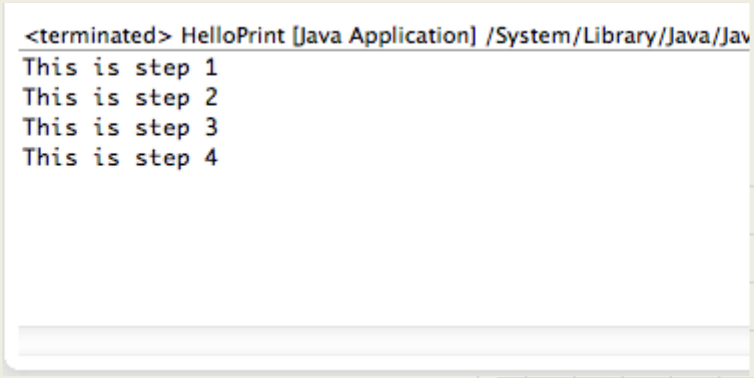
Code



HelloPrint.java

```
public class HelloPrint {
    public static void main(String[] args) {
        System.out.println("This is step 1");
        System.out.println("This is step 2");
        System.out.println("This is step 3");
        String lastStep = "step 4";
        System.out.println("This is " + lastStep);
    }

}
```

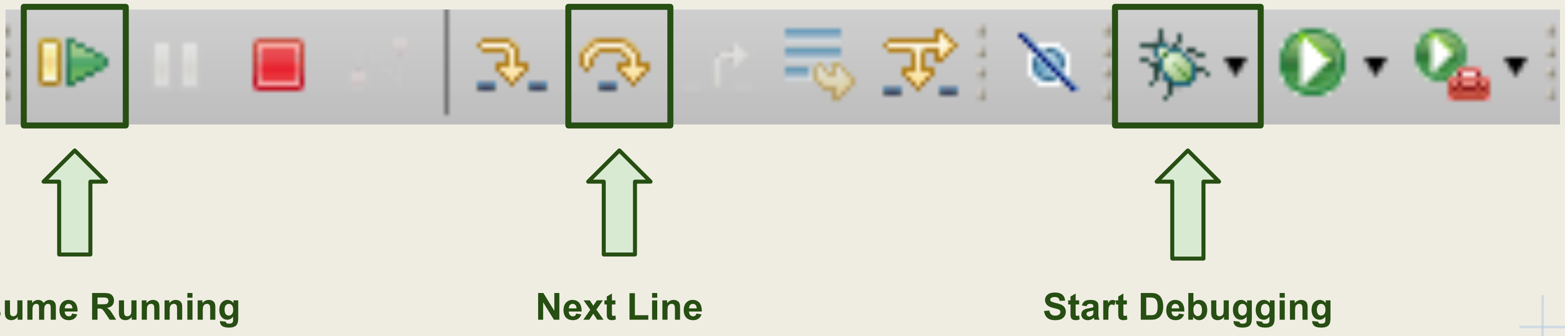
Output



```
<terminated> HelloPrint [Java Application] /System/Library/Java/Jav
This is step 1
This is step 2
This is step 3
This is step 4
```

Using the Debugger

- Let's **debug** (step through) our program.
- Debugging is about **watching** a program run
- We will be using this to understand our code.



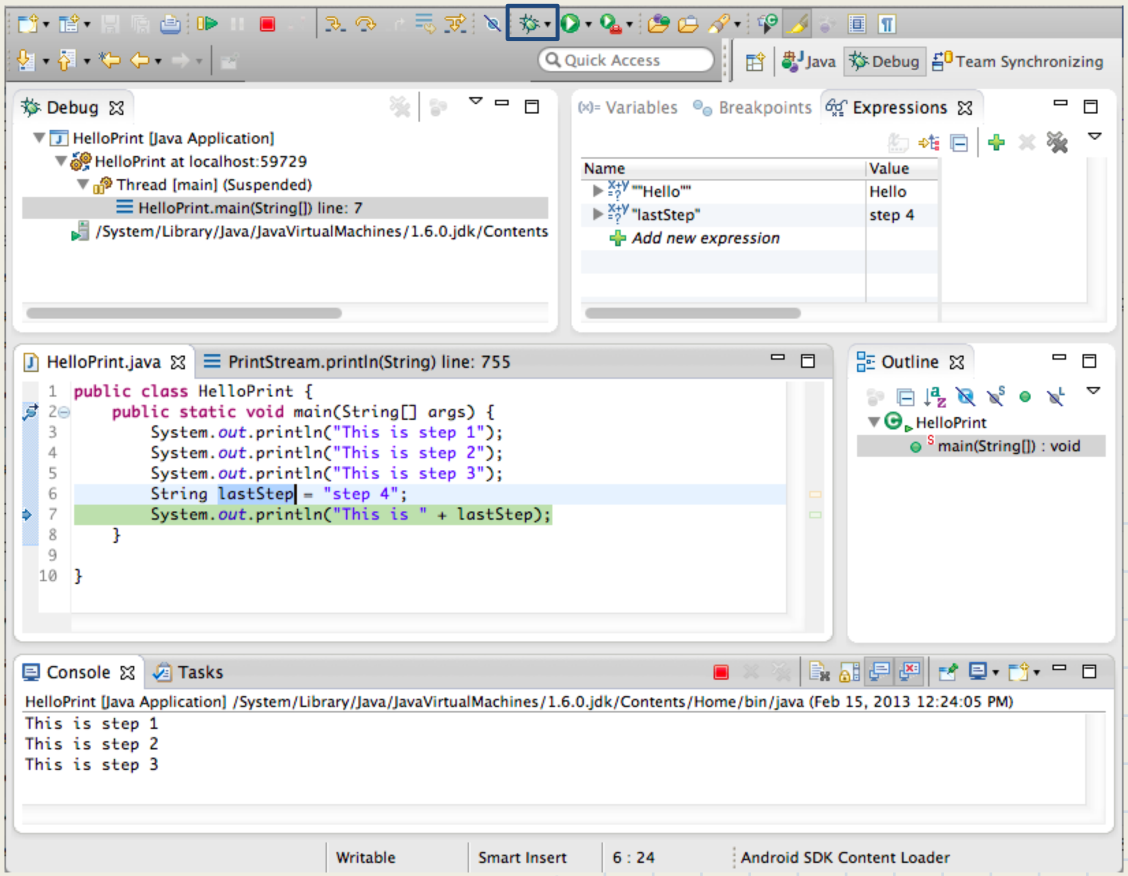
Using the Debugger

- Let's "debug" (step through) our program.

Current Line

Code View

Console View



```
HelloPrint [Java Application]
  HelloPrint at localhost:59729
    Thread [main] (Suspended)
      HelloPrint.main(String[]) line: 7
  /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents
```

Name	Value
""Hello""	Hello
"lastStep"	step 4
Add new expression	

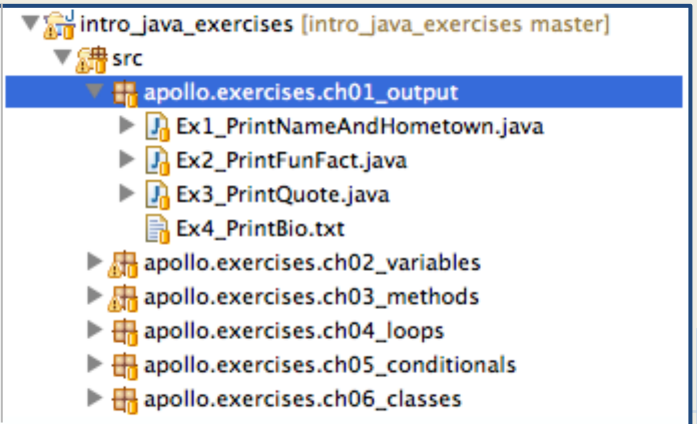
```
 1  public class HelloPrint {
 2      public static void main(String[] args) {
 3          System.out.println("This is step 1");
 4          System.out.println("This is step 2");
 5          System.out.println("This is step 3");
 6          String lastStep = "step 4";
 7          System.out.println("This is " + lastStep);
 8      }
 9
10  }
```

```
HelloPrint [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (Feb 15, 2013 12:24:05 PM)
This is step 1
This is step 2
This is step 3
```

Projects Overview

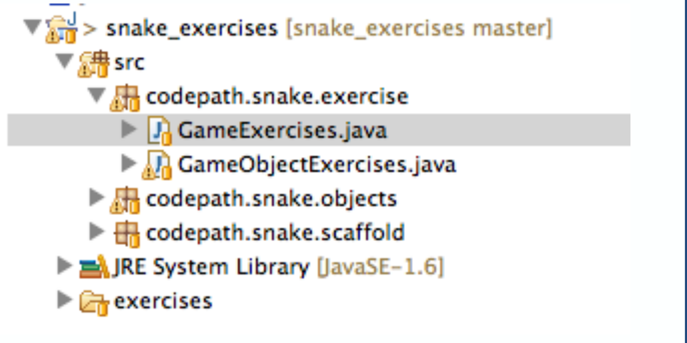
We will be working within two eclipse projects for these lessons:

intro_java_exercises are basic java exercises which we will complete after each concept is introduced.



```
▼ intro_java_exercises [intro_java_exercises master]
  ▼ src
    ▼ apollo.exercises.ch01_output
      ▶ Ex1_PrintNameAndHometown.java
      ▶ Ex2_PrintFunFact.java
      ▶ Ex3_PrintQuote.java
        Ex4_PrintBio.txt
    ▶ apollo.exercises.ch02_variables
    ▶ apollo.exercises.ch03_methods
    ▶ apollo.exercises.ch04_loops
    ▶ apollo.exercises.ch05_conditionals
    ▶ apollo.exercises.ch06_classes
```

snake_exercises is the initial project which we will develop into a 2D snake game during the lessons.



```
▼ > snake_exercises [snake_exercises master]
  ▼ src
    ▼ codepath.snake.exercise
      ▶ GameExercises.java
      ▶ GameObjectExercises.java
    ▶ codepath.snake.objects
    ▶ codepath.snake.scaffold
  ▶ JRE System Library [JavaSE-1.6]
  ▶ exercises
```

1. Output

How a program displays information

Output - Println

When writing programs, **displaying data** on-screen is called producing **output.**

- In Eclipse, the **console** is at the bottom and **contains** anything **printed** out in a program.
- To **print** out in the console, use the following command:

```
System.out.println("Hello World");
```

- We will be using this command to print out text as a part of our exercises.

Output - Main

```
public class MyFirstApp {

    public static void main(String[] args) {
        System.out.println("Hello World");
    }

}
```

Let's examine the most basic Java application:

- `MyFirstApp` is the thing that **holds all code** in this **file**
- **Curly braces** act as a **container** for lines of code
- The **`main`** method **runs** when you hit the **Run button**
- `System.out.println` is code that runs as part of the **`main` method**.

Output - Commands

- As shown before the following command **outputs** plain text to the **console**:

```
System.out.println("Hello World");
```

- But what do the parts of this command mean?

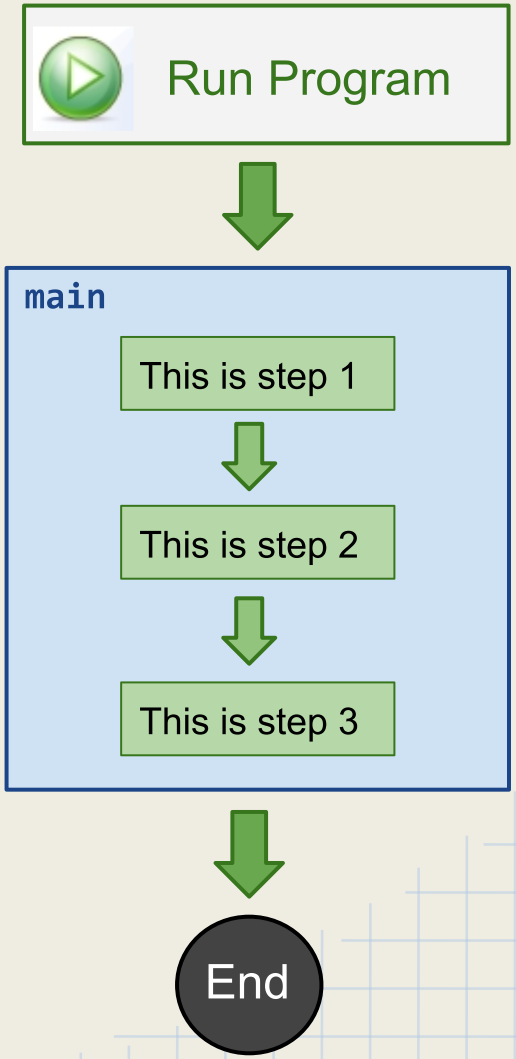
- There are **thousands of commands** in Java, one of which is `println`. Commands are organized into **categories**. Categories can have sub-categories.
- `System` is a category and `out` is a subcategory of commands. In Java, these are known as **packages**.

Output - Execution

Let's look at how a Java program **executes**...

```
public static void main(String args[]) {
   System.out.println("This is step 1");
   System.out.println("This is step 2");
   System.out.println("This is step 3");
}
```

=



Java programs are run **line by line** executing each **command** listed in `main`.

Output - Comments

- When writing programs, you can need a way to **document your code** and leave a summary explaining what certain parts do.
- To create a comment which is a summary of code that is **ignored by the computer**, simply put '//' at the beginning of a line:

```
// This is a comment, computer ignores these
```

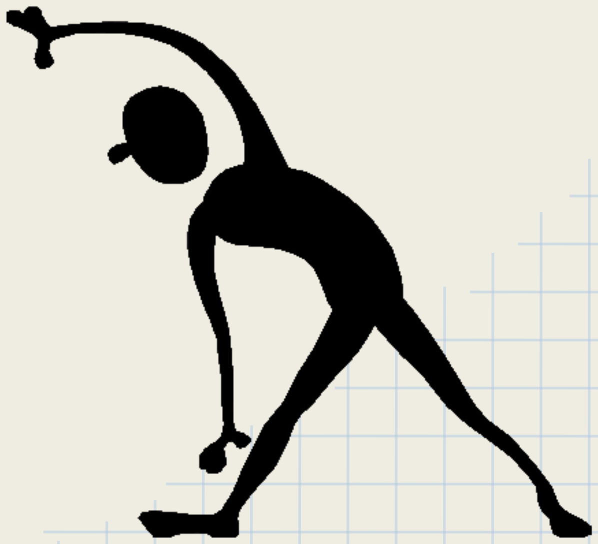
- Commenting is a good way to add descriptions or **summaries for other developers** looking through your program.

Output Examples

```java
System.out.println(5);
// 5

System.out.println("Hello World!");
// "Hello World!"

System.out.print("Hello ");
System.out.print("World!");
// "Hello World!"

// This comment is just for developers to read
// and is ignored by the computer when running the program

/* This is a block comment that can span multiple lines
   and will all be ignored when running the program until
   the terminating characters are found */
```

Output Exercises

- Open up **intro_java_exercises** in Eclipse
- Exercise 1 - **PrintNameAndHometown**
- Exercise 2 - **PrintFunFact**
- Exercise 3 - **PrintQuote**
- Exercise 4 - **PrintBio**



2. Variables and Types

How a program stores information

Variables and Types

Variables act as placeholders to **store** data **values** within a program.

- Computers work with data that **represents information** needed for a program to run.
- Programs store data into **variables** for later access
- Variables **can change** value as a program runs
- Think of a variable as a **pronoun** that represents a name. A women's name might be replaced with **she**.
- A variable is like a **placeholder** for some *real* data.

Types

Types are **different kinds of data** that a program needs to run. Variable types can include things such as names, prices, ages, songs or anything in your application.

There are a few **basic** (or primitive) types:

- **Numbers** (int and double)

```
55, 23.5, 1.45, 10000
```

- **Strings**

```
"Hello", "Bruce Wayne"
```

- **Boolean**

```
true, false
```

Complex Types

In addition to basic types, there are also **complex** types. These are many more types that represent more specific types of data.

Examples include:

- **Colors**

```
Color.RED, Color.GREEN
```

- **Date**

```
Date d = new Date();
```

- **Currency**

```
Currency dollars = new Currency("US");
```

Variables, cont'd.

Variables consist of a type, a name and a value.

- When we **declare** a new variable, we specify a **type** and a **name**.
- The name is **arbitrary**, cannot contain spaces or special characters but is completely up to the developer.

```
int someName;
```

This means I am **declaring** a variable (*placeholder*) named **someName** that can store only **int** types (*numbers*).

- We can then store any **value** of the **type** specified into the variable.

```
someName = 5;
```

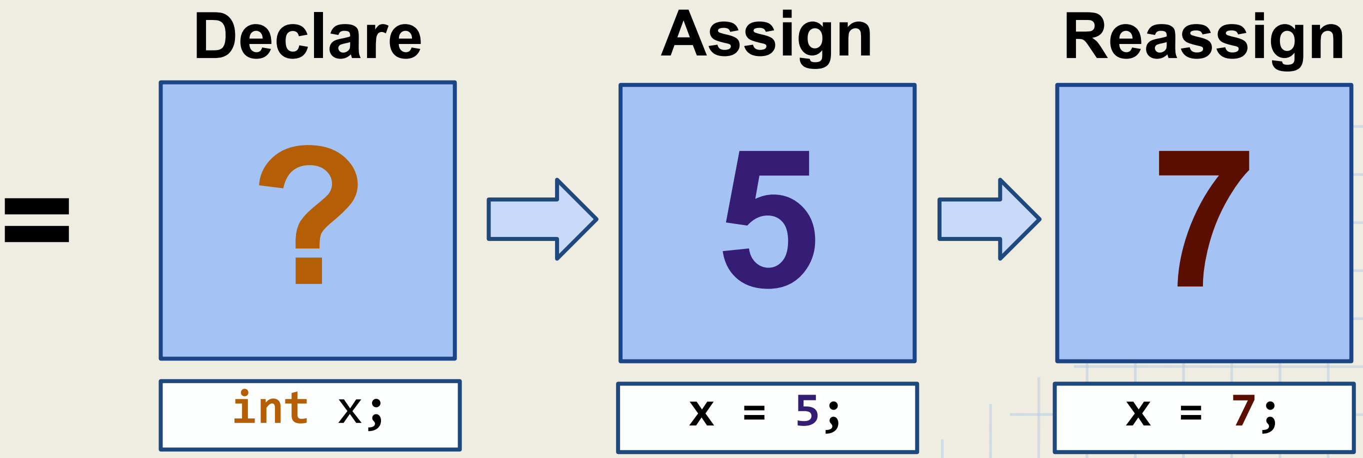
This means I am temporarily **assigning** the **someName** variable to the value of **5**.

Variables, cont'd.

Variables can be declared, assigned and reassigned.

- A variable is like a **box** or a **placeholder** that can **contain any value** that matches it's **type**.

```
int x;
x = 5;
x = 7;
```



Expressions

- Variables and types can be composed into **expressions**. A good example is performing arithmetic with numbers. For example:

```
int result = (2 + 3) x 6; // 30
```

- Expressions are like sentences and is ended with a semicolon.

- Expressions usually have a result:

```
boolean isGameOver = snakeHitWall || snakeHitTail;
// game is over if either is true (|| means "OR")
```

Variables Example

```
int x;      // declaring
x = 3;      // assigning
x = x + 3; // increment by 3
System.out.println(x);
// 6

int age = 19; // declaring and assigning
boolean canDrink = age > 21;
System.out.println(canDrink);
// false

String name = "Bruce";
name = name + " " + "Wayne";
System.out.println(name);
// "Bruce Wayne"

// Constants
final double PI = 3.1415;
```

Arrays

Arrays are data types that can contain **multiple values** within a **list** that are accessed sequentially.

- Variables and data types discussed so far only allow us to **contain a single value** at a given time.

- Often in programs, we need to be able to **build a list of items** that are all related. For example, a sequence of numbers for the lottery or a set of names that need to be invited to an event.

Arrays, cont'd

- When an array is defined you must specify **the type** and the **number of items**:

```
int[] numbers = new int[10]; // create array of 10 ints
// create array of 3 items
String[] names = { "Bob", "Suzie", "Emma" };
```

- Once an array is defined, the elements can be accessed by **specifying the position**:

```
System.out.println(numbers[0]); // prints first element
System.out.println(numbers[1]); // prints second element
String title = names[0] + " and " + names[2]; // Bob and Emma
```

Arrays, cont'd

Arrays are just **lists of items** stored sequentially in memory.

0	1	2	3	4	5	6	7
Bob	Suzie	Emma	John	Tim	Dan	Kelly	Joan

```
String[] names = new String[] {
 "Bob", "Suzie", "Emma", "John", "Tim", "Dan", "Kelly", "Joan"
};
names[0] = "Harold"; // reassign value
System.out.println(names[3]); // John
```

Arrays Example

```
// Arrays are accessed and modified using square brackets

int nums[]; // declaring
nums = new int[2]; // initializing
nums[0] = 5;
nums[1] = 6;
System.out.println(nums[1]);
// 6

String fullName;
String[] names = new String[2];
names[0] = "Bruce";
names[1] = "Wayne";
fullName = names[0] + " " + names[1];
System.out.println(fullName);
// "Bruce Wayne"
```

Variables Exercise

- Open up **intro_java_exercises** in Eclipse
- Exercise 1 - **Integers**
- Exercise 2 - **AgeComplimentor**
- Exercise 3 - **Siri**
- Exercise 4 - **BadDeclaration**
- Exercise 5 - **BasicMath**
- Exercise 4 - **BasicArrays**

3. Methods

How a program organizes steps

Defining Methods

- **Methods** (also known as functions) are small units of code that can be named and then run within a program.
- A method is a **simple way to group code** statements together.

- In essence, methods are a **name** (i.e `printInfo`) assigned to label the lines of code within **braces**:

```
public void printInfo() {
  System.out.println("My name is Joe");
  System.out.println("My favorite color is Blue");
}
// Calling a method from main
printInfo();
```

Defining Methods, cont'd

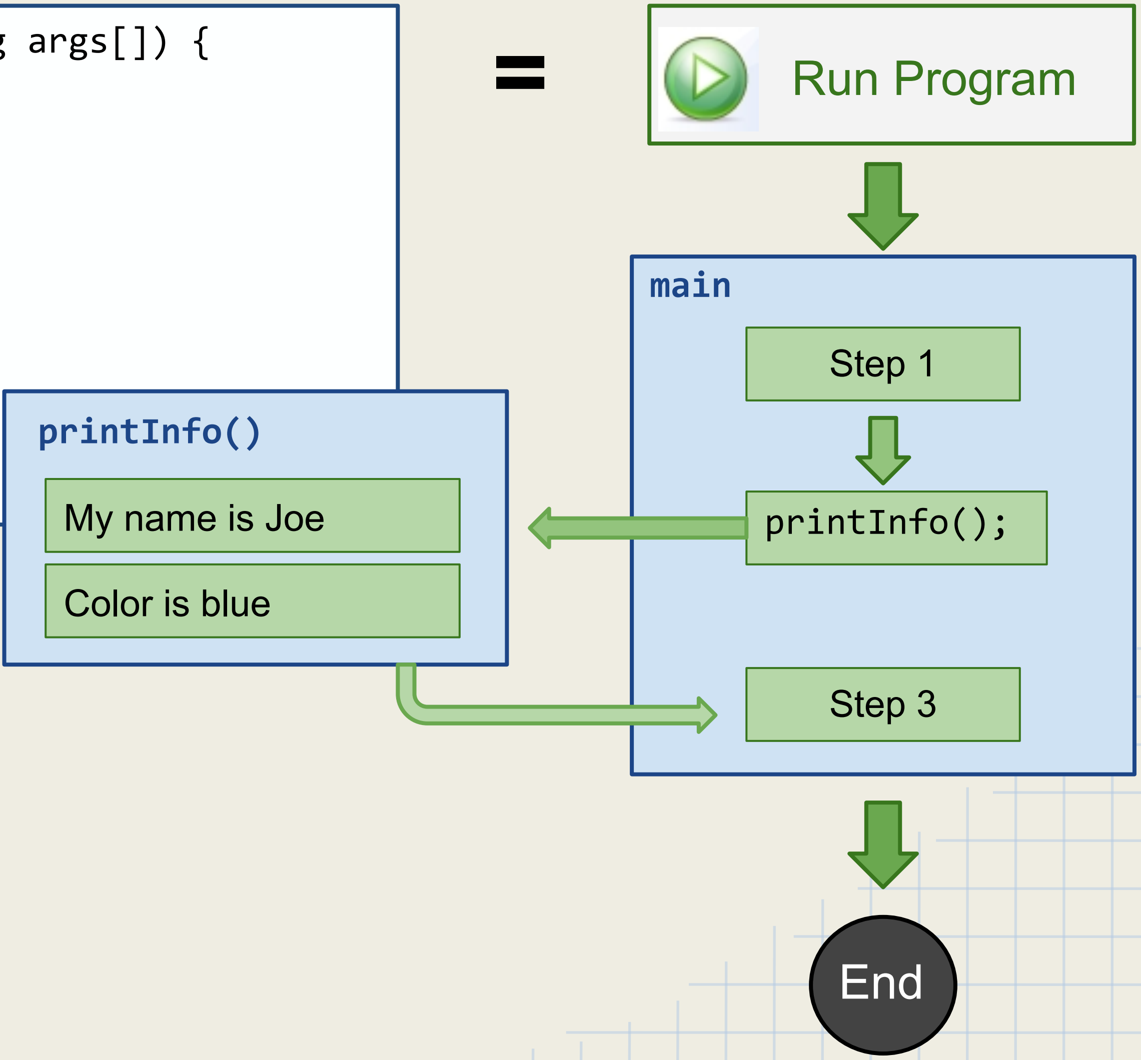
- The reason methods are important is because programming is often about **repeating similar code** instructions in different situations.

- Consider a game where there are 4 ways to lose and they all trigger the same "game over" message.

- Methods allow us to give a **set of code instructions** a name and then **repeat that set of steps** whenever we want by **calling** the method.

Method Execution

```
public static void main(String args[]) {
  // Calling a method
  println("Step 1");
  printInfo();
  println("Step 3");
}

public void printInfo() {
  println("My name is Joe");
  println("Color is Blue");
}
```



Defining vs Calling

- Let's understand the difference between **defining** a method and **calling** a method.
- Defining is giving a group of code statements a name but **does not** make the code run.
- Calling a method is when you run the code for a method by **referencing** the name.

```
// defining, does not run unless called in main
public void someMethodName() {
  int num = 3;
  System.out.println("The number is " + num);
}
```

```
public static void main(String args[]) {
  // running the code in a method
  someMethodName()
}
```

Defining Methods, cont'd

- Methods cannot be **defined** within other methods and are each created **separate** from one another.
- Methods are **always defined** within a **class** but **not defined** within other methods.

```
public class SomeClass {
    // methods cannot be defined within other methods
    public static void someMethodName() {
      int num = 3;
      System.out.println("The number is " + num);
    }

    // but methods can be called from other methods
    public static void someOtherMethod() {
       someMethodName(); // called first method
    }
}

// in main method
SomeClass.someOtherMethod(); // The number is 3
```

Method Parameters

Methods can **accept** values called **parameters** which can be accessed when they run.

- When calling a method, we can **pass along** variable values for the method **to use**.
- The **parameters** are values that the method needs in order to run successfully.

```
public void sayHello(String someName) {
    System.out.println("Hi " + someName);
}

sayHello("Bill"); // Hi Bill
```

Return Values

Methods can **return** back a **value** after completing a run.

- When calling a method, we can **return** values back to store for later use.
- The **return value** can be stored in a variable and used later in the program.

```
public int add(int n1, int n2) {
    return n1 + n2;
}

int sum = add(5, 6); // sum is assigned to return value
System.out.println(sum); // 11
```

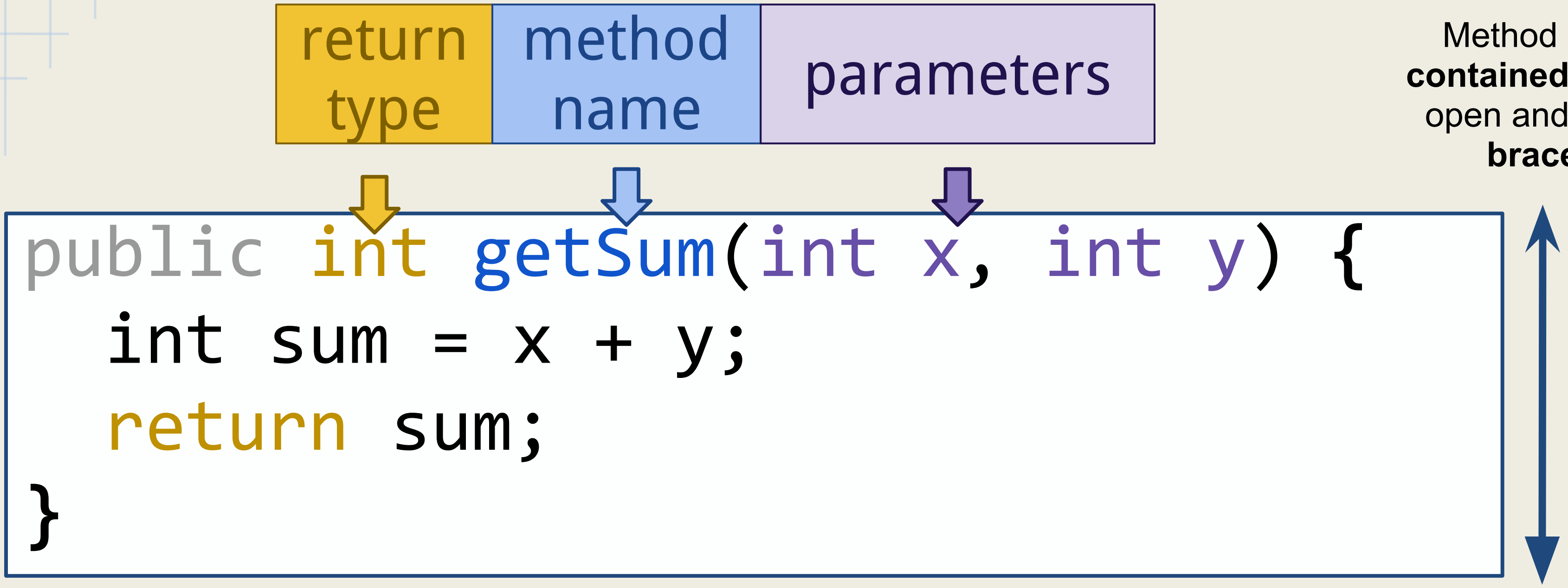
Parameters and Returning

- As an example, think of a mad libs program that adds two words into a sentence and returns the result. This can be done with **parameters** to the method:

```
public String madLibsSentence(String adj, String noun) {
  String sentence = "Oh no! Is that a " + adj + " " + noun + "??";
  return sentence;
}
System.out.println(madLibsSentence("mad", "cow"));
// Oh wow is that a mad cow??
```

- Using parameters and return values, you can create methods that work in many different areas of your code in varying ways.

Methods



```
public int getSum(int x, int y) {
   int sum = x + y;
   return sum;
}
```

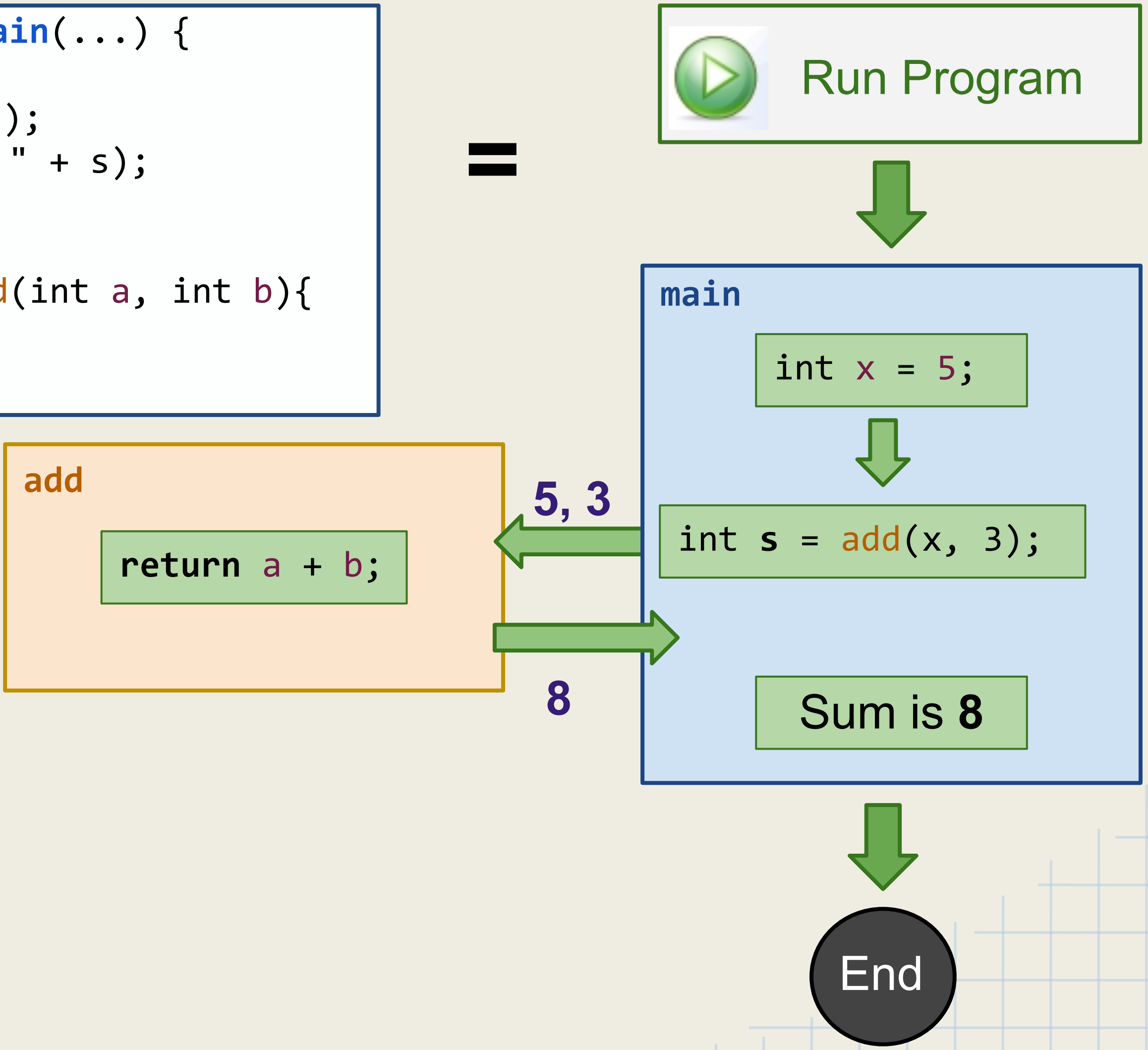
Method code **contained within** open and close **braces**

- **Return type** in method signature matches whatever type is returned with **return** keyword.
- **Method name** is a label for the code contained within curly braces.
- **Parameters** are values **passed into** a method.

Parameters and Returning

```
public static void main(...) {
     int x = 5;
     int s = add(x, 3);
     println("Sum is " + s);
}

public static int add(int a, int b){
     return a + b;
}
```

=



Printing vs Returning

- In programming, there is an important difference between **printing a string** and **returning a string**.
- Consider these two methods:

```
public String getHello(String name) {
  return "Hello " + name + "!";
}
String helloString = getHello("Billy");
System.out.println(helloString) + "!";
// Hello Billy!!

public void printHello(String name) {
  System.out.println("Hello " + name + "!");
}
printHello("Ted");
// Hello Ted!
```

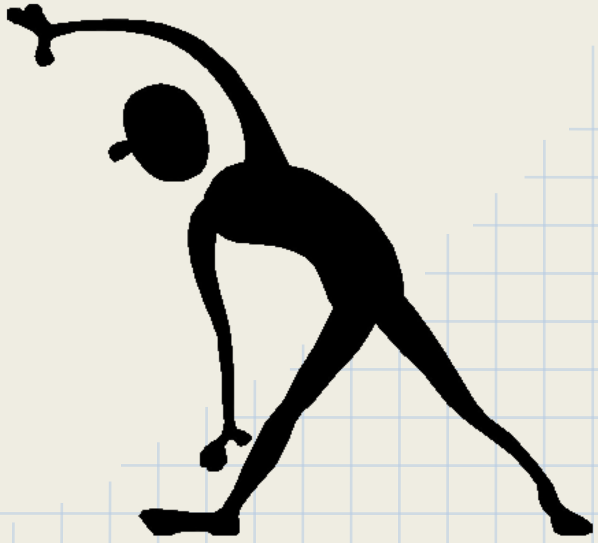
- The first 'returns' the value into a variable. The second prints the sentence directly. *Printing* does not *return.*

Methods examples

```
public void printFullName(String firstName, String lastName) {
  System.out.println("Nice to meet you " + firstName + " " + lastName);
}

printFullName("Bob", "Smith");
// "Nice to meet you Bob Smith"

public String askAboutJob(String name, String job) {
  return name + ", I see here you work as a " + job + "?";
}

String sentence = askAboutJob("Billy", "Doctor");
System.out.println(sentence);
// Billy, I see here you work as a Doctor?
sentence = askAboutJob("Joe", "Plumber");
System.out.println(sentence);
// Joe, I see here you work as a Plumber?
```

Methods exercise

- Open up **intro_java_exercises** in Eclipse
- Exercise 1 - **BasicMethod**
- Exercise 2 - **SayMyName**
- Exercise 3 - **PrintRandomNumber**



4. Loops

How a program repeats steps

Loops

Loops are used in cases where a task must be **repeated many times**.

- In programming, the same task must often be **repeated many times** or done to **many different** items.
- Loops help **reduce** the **amount of code** written. Often loops will be used **in conjunction** with **methods**.

    - Enumeration (i.e accessing arrays)
    - Animations (i.e moving drawn objects)
    - Displaying data (i.e printing tabular data)

More Loops

- The simplest type of loop is a **for-loop** which executes a group of code statements a specified number of times.
- As discussed earlier, loops help us to reduce duplication by repeating the same set of code statements over and over.
- In addition to the **for-loop**, there is also the **while-loop** and the **do-while-loop**.
- These **for**-loops help us to execute a group of code statements repeatedly until a particular condition is met.

For Loops

The syntax of a for loop:

Start i at 0 **While < 10** **Increment by 1**

```
for (int i = 0; i < 10; i=i+1) {
   sum = sum + nums[i];
}
```

In english this reads:

"Set a variable named i starting at 0. Run the loop while i is less than 10. Each time the loop runs, increment i by 1."

For Loops

```
for (int i = 0; i < 5; i=i+1) {
   System.out.println(i * 2);
}
```

Tracking the Loop Code Path

# Runs	Value of i	Output
1st	0	0
2nd	1	2
3rd	2	4
4th	3	6
5th	4	8

Code Being Executed

```
int i = 0;
if (i < 5) {
   System.out.println(i * 2);
   i = i + 1;
   // Run code again with new i
}
```

For Loops example

```
int[] nums = new int[] { 5, 6, 7, 8, 9, 10, 11, 12, 13 };
int sum = 0;
for (int i = 0; i < nums.length; i=i+1) {
     sum = sum + nums[i];
}
System.out.println(sum);
// 81

String[] names = new String[] { "Bruce", "Bill", "Tammy", "Joe" };
for (int i = 0; i < names.length; i=i+1) {
     System.out.println("Hey " + names[i] + "!");
}
// Hey Bruce!
// Hey Bill!
// Hey Tammy!
// Hey Joe!

// count up from 1 to 100
for (int i = 1; i <= 100; i=i+1) {
     System.out.println(i);
}
```

While Loops

While loops execute code **repeatedly** until a **specified condition** is met.

- While loop is simpler than a for-loop in many respects because the loop simply **repeats over and over until** a condition is **no longer true**.

- Used when we want to repeat code for as long as an arbitrary condition is true. A while loop does **not require** us to know the **bounds** ahead of time.

While Loops

While loops execute code **repeatedly** until a **specified condition** is met.

Repeat while count is less than 100

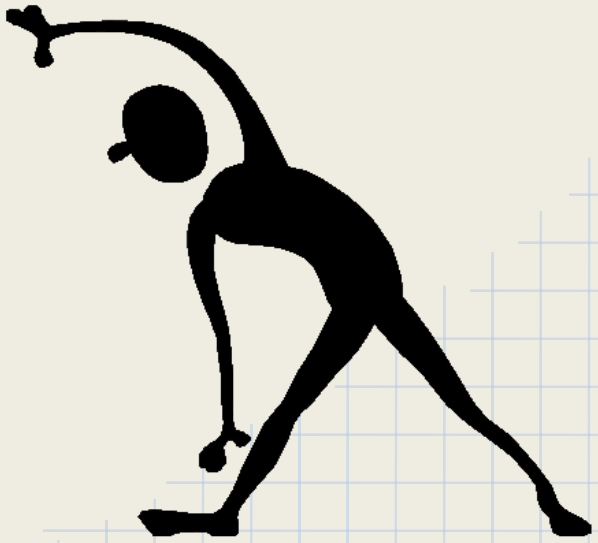
```
int count = 1;

while (count <= 100)
{
  System.out.println(count);
  count = count + 1;
}
```

While Loops example

```
int currentNum = 1;
int sum = 0;
while (currentNum <= 100) {
     sum = sum + currentNum;
     currentNum = currentNum + 1;
}
System.out.println("The sum of the numbers from 1 to 100 is " + sum + ".");

static int count2Div(int num) {
     int count = 0;   // count how many divisions we've done
     while (num >= 1) {
          num = num / 2;
          count++;
     }
     return count;
}
System.out.println(count2Div(40));
// 6
```

Loops exercise

- Open up **intro_java_exercises** in Eclipse
- Exercise 1 - **BasicLoop**
- Exercise 2 - **CountingBackwards**
- Exercise 3 - **PrintGrid**
- Exercise 4 - **CountByFive**



5. Conditional Execution

How a program decides between steps

Conditional Execution

If statements are about **checking conditions** and then **running** different **code** depending on **the result**.

```
int testscore = 76;
if (testscore > 50) {
  System.out.println("You passed");
}
```

- For instance, imagine that you are writing a pac-man game with basic game logic.
- Depending on what is happening (i.e ghost hits you, pac-man hits a wall) you need to have the **game react differently to different situations** (conditions).

Conditional Execution

- There are often cases where you need to check multiple conditions
- This is supported with an **if-else** as well as an **else** which executes if every other condition is false.

```
int testscore = 76;
if (testscore > 90) {
  System.out.println("You got an A!")
} else if (testscore > 50) {
  System.out.println("You passed");
} else {
  System.out.println("You failed!");
}
```

Conditional Execution

If statements are about **checking conditions** and then **running** different **code** depending on **the result**.

```
int score = 21;
if (score == 21) {
  System.out.println("Win");
} else {
  System.out.println("Hit");
}
```

First Condition

Else Condition

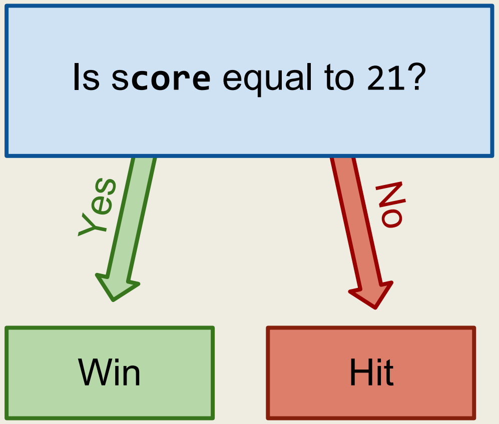
In English, this reads: **If** the value assigned to the variable **score** equals 21 then print a line of text reading "Win" as output **otherwise** print a line of text "Hit".

Conditional Execution

If statements are about **checking conditions** and then **running** different **code** depending on **the result**.

```
int score = 21;
if (score == 21) {
  System.out.println("Win");
} else {
  System.out.println("Hit");
}
```

=



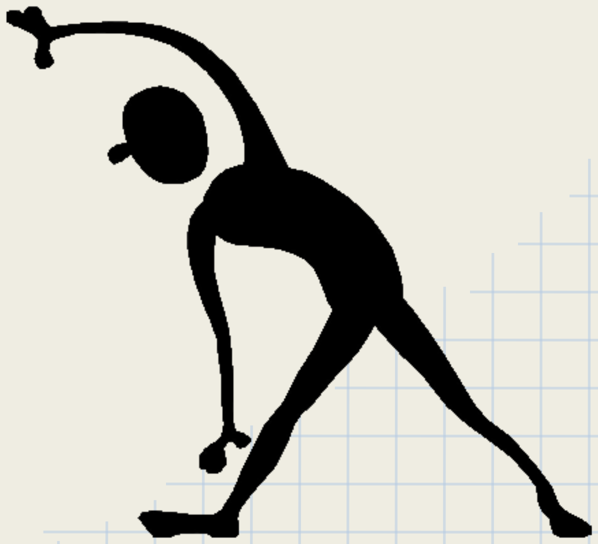
If allows us to **decide** what **action** to run based on the particular **situation**.

Conditionals example

```java
public void suggestGift(int age) {
     if (age < 3) {
          System.out.println("Toy Truck");
     } else if (age < 15) {
          System.out.println("Board Game");
     } else if (age < 25) {
          System.out.println("DVD");
     } else {
          System.out.println("Gift Card or Money");
     }
}

public void canDrink(int age) {
   if (age > 21) {
      System.out.println("Yes!");
   } else {
      System.out.println("Not Yet!");
   }
}
```

Conditionals exercise

- Open up **intro_java_exercises** in Eclipse
- Exercise 1 - **FizzBizz**
- Exercise 2 - **RomanNumerals**
- Exercise 3 - **BlackJack**
- Exercise 4 - **GCF**



6. Classes and Objects

How a program organizes code

Classes

- In programming, programs are often **organized** using methods, arrays and loops.
- In Java programming there is a **higher level of organization** of code into "**classes**".
- A **class** is like a **noun** (a person) and a **method** is like a **verb** (walks around).
- A **class** can contain many **actions** (methods) as well as different types of data.

Classes

Classes describe **complex types** and are **composed** of two aspects:

- **Fields** - adjective (descriptions)
- **Methods** - verb (actions)

A class defines a **type of an object** and is the basis of **object-oriented** programming.

Classes, examples

The following lists some examples of classes, with some of their fields and methods.

- **String**
  - Methods: length(), compareTo(string)
- **User**
  - Fields: name, birthday, tagline
  - Methods: followUser(user)
- **Book**
  - Fields: title, ISBN, numPages
  - Methods: exportPdf(filename)

Classes, cont'd.

The following code block defines a simple class for **User**:

```
public class User {
  // fields
  String name;
  String pictureUrl;
  String email;

  // methods
  public void printName() {
    System.out.println("My name is: " + name);
  }
}
```

Classes, cont'd.

Classes are a **definition** explaining to the program about the **attributes** and **actions** of a particular **noun**.

User has a

- name
- email
- pictureUrl

and can

- printName
- buyBook

Fields

Methods

Book has a

- title
- author
- numPages

and can

- printTitle
- setOwner

Instantiation

Instances, aka objects, are created via instantiation using the "**new**" keyword.

```
// Declaration
User u;

// Instantiation (allocates memory)
u = new User();

u.name = "John Smith";
u.printName();
// My name is John Smith
```

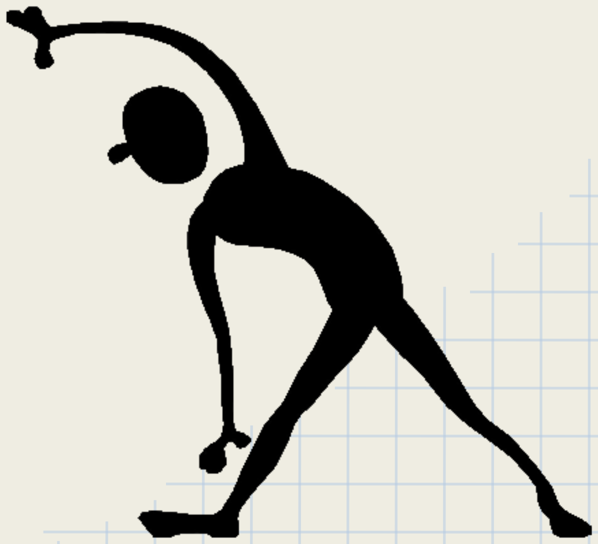
Objects

- A class can have many **instances**, which is to say that there can be **any number** of "versions" of a class. Each example is an **object**:

```
public class User {
  // fields and methods defining a user
}

User joe = new User();
joe.name = "Joe";
joe.printName();

User billy = new User();
billy.name = "Billy";
billy.printName();
```

Classes Exercise 1

- Open up **intro_java_exercises** in Eclipse
- Exercise 1 - Follow instructions in **Ex1_Book**



Constructors

Constructor methods are called when objects are created to initialize the object.

```
public class User {
  // Fields
  String name;

  // Constructor
  public User(String name) {
    this.name = name;
  }
}

// new user's name is initially set to "Billy"
User billy = new User("Billy");

// we can still manually reassign a user's name later...
billy.name = "William";
```

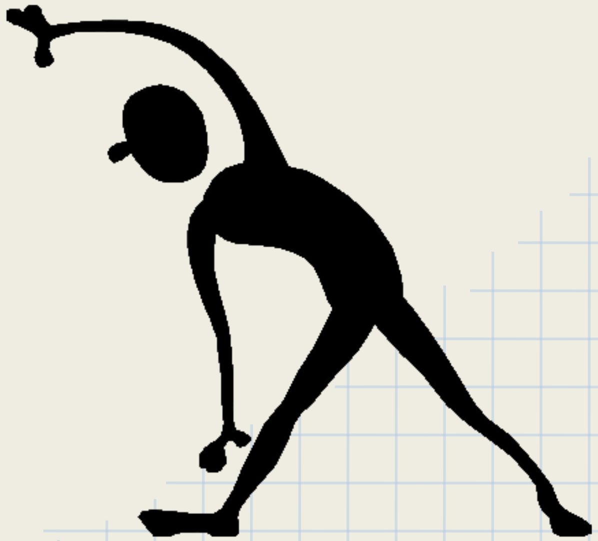
The "this" keyword

- Within an **instance method** or a **constructor**, "this" is a reference to the **current object** whose method or constructor is being called.
- You can refer to **any field or method** of the current object from within an instance method or a constructor by using the "**this"** keyword.

```
public class User {
    public String name;

    // constructor (notice this)
    public User(String name) {
        this.name = name;
    }
}
```

Classes Exercise 2

- Open up **intro_java_exercises** in Eclipse
- Follow the instructions in **Ex2_Constructors** and **Ex3_User**



Access Modifiers

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

- If a class, method or field is marked **public**, then that is **accessible** by **all classes** everywhere in an app.
- In contrast, if marked **private**, then the member is **accessible** only within **its own class**.

Access modifiers are important also to document which fields and methods are intended to act as a **public interface** when using this class.

Access Modifiers

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

```
public class Car {
    public String model;
    private int numCylinders;

    public void setSpeed(int speed) { ... }
}

car = new Car();
car.numCylinders = 6; // FAILS
car.model = "Toyota"; // WORKS
car.setSpeed(55); // WORKS
```

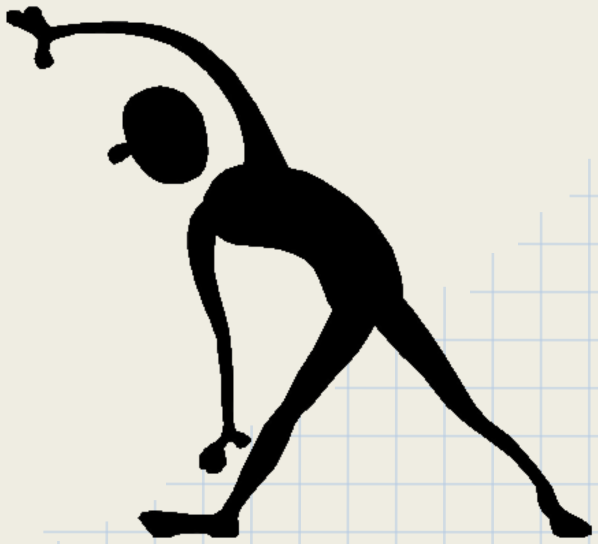
Getters and Setters

Often within classes, there is a common pattern using **private fields** along with a **public interface** for access.

```
public class User {
    private String firstName;
    private String lastName;
    public String getName() {
       return firstName + " " + lastName;
    }
    public void setName(String firstName, String lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
    }
}

User u = new User();
u.setName("Billy", "Joel");
System.out.println(u.getName()); // Billy Joel
```

Classes Exercise 3

- Open up **intro_java_exercises** in Eclipse
- Follow the instructions in **Ex4_Access**



Instance vs Class Members

Static members belong to the **class** rather than individual instances of the class.

- A field or method defined is **by default** known as **instance** level members.
- If you recall, each time I call "new", a **separate instance** of a class is created.
- Each instance has **separate fields** that contain **different information** (i.e users with different names).
- In some cases, you may want to **share fields or methods** across **all instances** of a class.

Instance vs Class Members

Static fields are called on the **class** name itself, **instance** fields are called on an individual **instance** of a class.

```
public class User {
    String name; // instance
    static int maxNameLength = 20;
    // constructor and other methods
}

System.out.println(User.maxNameLength); // 20
User billy = new User("Billy");
System.out.println(billy.name); // Billy
```

Instance vs Class Methods

Methods can be defined as **static** so that the method is **shared across all instances** and invoked on the **class**.

```
public class User {
    static void sayHello() {
      System.out.println("Hello!");
    }
    void sayName() {
       System.out.println("My name is" + name);
    }
}

User billy = new User("Billy");
User.sayHello(); // Hello!
billy.sayName(); // My name is Billy
```

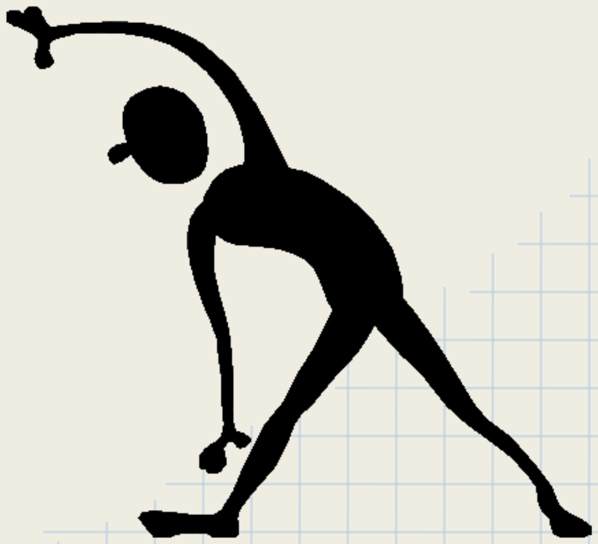
Instance vs Class Methods

Remember **static** fields or methods can **only call** other static members but **not instance** level members.

```
public class User {
    static String greetingPrefix() {
        return "My name is ";
    }
    void sayName() {
        System.out.println(User.greetingPrefix() + name);
    }
    static void sayHello() {
        System.out.println("Hello");
        sayName(); // FAILS!
    }
}
```

Classes Exercise 4

- Open up **intro_java_exercises** in Eclipse
- Follow the instructions in **Ex5_StaticMethods**



Classes example

```
public class iPhone {
    private String version;
    String driveSize;
    static int screenWidth = 320;

    public iPhone(String version, String driveSize) {
      this.version = version;
      this.driveSize = driveSize;
    }

    public String getVersion() {
        return version;
    }
}


iPhone billyiPhone = new iPhone("4G", "16GB");
iPhone sallyiPhone = new iPhone("5", "32GB");

System.out.print(iPhone.screenWidth); // 320
System.out.print(billyiPhone.getVersion()); // 4G
System.out.print(sallyiPhone.getVersion()); // 5
```

7. Inheritance

How related classes share behavior

Inheritance

Class inheritance shares common fields and methods across **related classes**.

- Classes can help us to organize code, data and methods into logical units which can manage their own **behaviors** within a program.
- Inheritance means that two related class definitions will **share** the same blueprint variables and methods.

Inheritance, cont'd.

- Consider the case of an application with **Customer** and **Seller** classes. Customer **purchase** items sold by the Seller.
- These two classes share many properties (name, email, etc) but have **different** roles in the system.

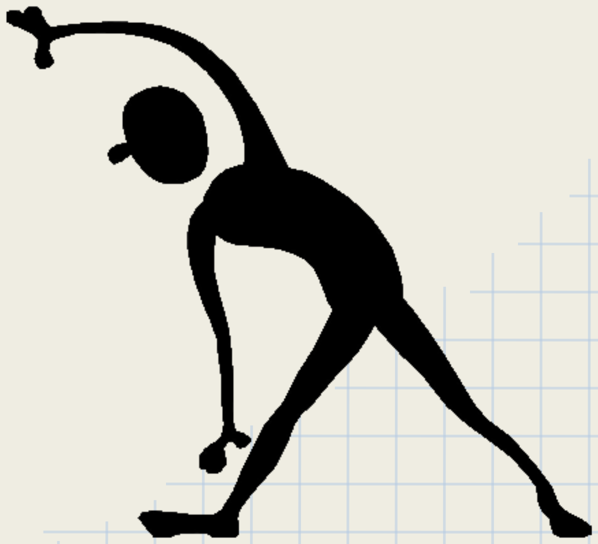
```
public class User {
  // attributes and methods defining a user
}
public class Customer extends User {
  // attributes and methods defining a customer
}
public class Seller extends User {
  // attributes and methods defining a seller
}
```

Inheritance example

```
public class IOSDevice {
  protected String version = "4G";
  protected String driveSize = "16GB";
  protected String screenSize;

  public String getVersion() {
    return version;
  }
}

public class IPhone extends iOSDevice {
   public IPhone() {
       this.screenSize = "320 x 480";
   }
}

public class IPad extends iOSDevice {
   public IPad() {
       this.screenSize = "768 x 1024";
   }
}
```

Inheritance Exercise 1

- Open up **intro_java_exercises** in Eclipse
- Follow Instructions in **Ex1_Animals**
- Follow Instructions in **Ex2_Constructors**



Overriding Methods

Overriding methods tweaks or **replaces** the **behavior** of a method within the parent class.

- When inheriting a parent class, certain **behavior** will **need to be modified** in the subclass.
- **Overriding** allows us to easily **redefine the methods** inherited from a **parent** class.
- This is especially useful for when a subclass has **similar but not identical** behavior from the parent.

Overriding Methods

Overriding methods augments or **replaces** the **behavior** of a method within the parent class.

```
public class iOSDevice {
    public void startDevice() {
        startBluetooth();
        startWiFi();
    }
}

public class iPhone extends iOSDevice {
     public void startDevice() {
         super.startDevice();
          connectCellTower();
     }
}
```

The "super" keyword

Use "**super**" to **reference** the **parent class** from within an inherited subclass.

- The **super** keyword is used to **access parent class** methods or constructors when **overriding** methods.
- Relevant when **augmenting** rather than replacing the **behavior** of a **parent class** constructor or method.

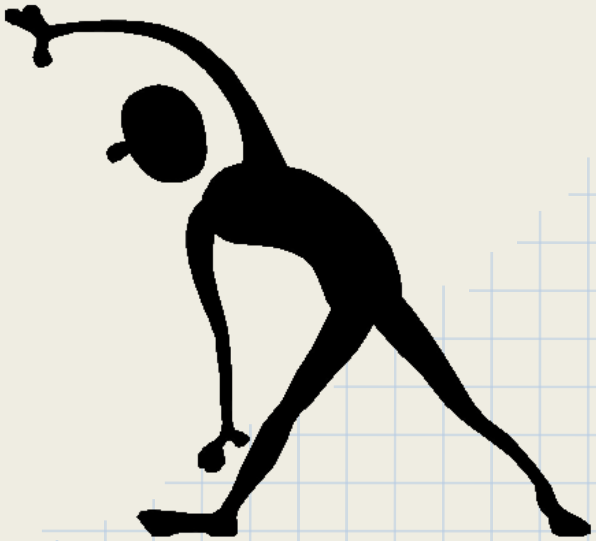
The "super" keyword, cont'd

Use "**super**" to **reference** the **parent class** from within an inherited subclass.

```
public class iOSDevice {
    public iOSDevice(String name) {
      this.name = name;
    }
    public void startDevice() {
       // ...start various things...
    }
}

public class iPhone extends iOSDevice {
     public iPhone(String name, String provider) {
       super(name); // run the parent constructor
       this.provider = provider;
     }
     public void startDevice() {
         super.startDevice(); // run the parent version
         connectCellTower();
     }
}
```

Inheritance Exercise 2

- Open up **intro_java_exercises** in Eclipse
- Follow Instructions in **Ex3_Overriding**



Overloading Methods

Overloading allows **multiple versions** of a method with **different** parameters.

- Java supports having **multiple** methods with the **same name** but **different parameters**.
- This is useful when we want to have a method **behave differently** depending on the **type** of the **arguments** passed.

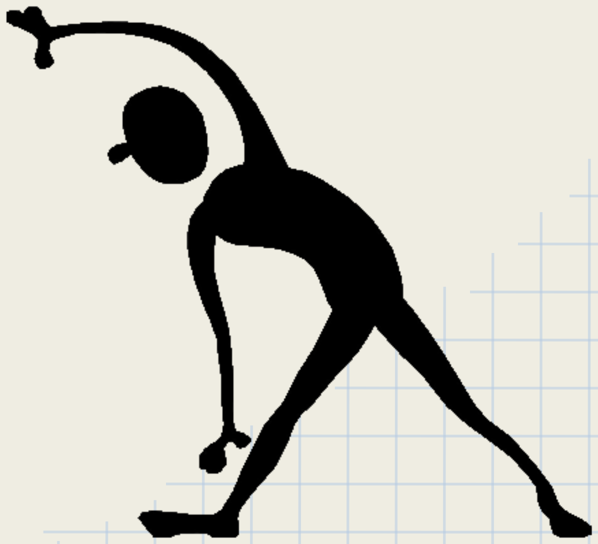
Overloading Methods, cont'd

Overloading allows **multiple versions** of a method with **different** parameters.

```
public class iPhone extends iOSDevice {
     public void setAlarm(String title, Time time, String note) {
       Alarm a = new Alarm();
       a.title = title;
       a.time = time;
       a.note = note;
     }

     public void setAlarm(String title, Time time) {
        setAlarm(title, time, "");
     }

     public void setAlarm(Time time) {
        setAlarm("Alarm", time);
     }
}

iPhone phone = new iPhone();
phone.setAlarm(new Time("7pm"));
```

Inheritance Exercise 3

- Open up **intro_java_exercises** in Eclipse
- Follow Instructions in **Ex4_Overloading**.txt



8. Collections

A better way to store lists

Collections

A **collection** is a *dynamic* list of data stored within your application.

Many **different** types such as:

- *lists* (ordered elements)
- *sets* (unordered unique non-duplicated elements)
- *map* (attaches an arbitrary key to an arbitrary value)

Different collections store lists of data in ways suited to various types of tasks.

Collections

A **collection** is a *dynamic* list of data stored within your application.

- Collections contain certain **shared methods** (through an interface) such as *add, remove,* and *size* which can be called on any collections available within Java.

- The two most important collections for us to understand are **ArrayList** (dynamic array) and **HashMap** (key-value store)

ArrayLists

ArrayList is a dynamically-sized Array of items that can have items **added** or **removed**.

- Basic arrays are useful but when we think of lists in the real world, lists are often **dynamic** with new items being **added** and then old items being **removed**.
- Basic arrays are a **fixed size** and adding or removing items is a very manual process which can be unnecessarily difficult.

ArrayLists, cont'd.

- An ArrayList is very much like an array but with more methods available and with no fixed size:

```
ArrayList<String> list = new ArrayList<String>();
```

- Adding a new item to this list is as easy as:

```
list.add("Tool");
```

- Removing an element is equally easy:

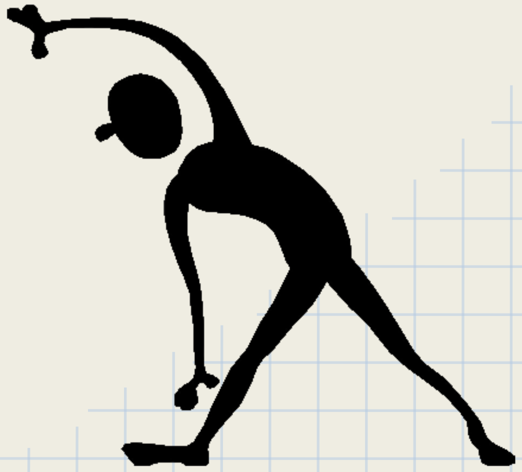
```
list.remove(1);
```

ArrayList example

```
ArrayList<String> stringList = new ArrayList<String>();

stringList.add("Item");

for (int i = 0; i < stringList.size(); i++)
   String item = stringList.get(i);
   System.out.println("Item " + i + " : " + item);
}

stringList.remove(0);

System.out.println("ArrayList is " +
    stringList.size() + " items long!");
```

ArrayList exercise

- Open up **intro_java_exercises** in Eclipse
- Follow Instructions in **Ex1_BasicArrayList**



Generics

Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods.

- When we define an ArrayList, we can specify the expected type of items using **`new ArrayList<String>`**.
- The pointy brackets are a signal that generics are being used to specify the type expected within the collection.

Generics, cont'd.

Generics allow us to parameterize the type to allow that type to be defined at runtime.

- Generics can be used to allow generic types in any class or method but are most frequently seen as a part of collections.
- In our usage of collections (i.e ArrayList) and throughout using Java, the concept of **generics** will appear from time to time.

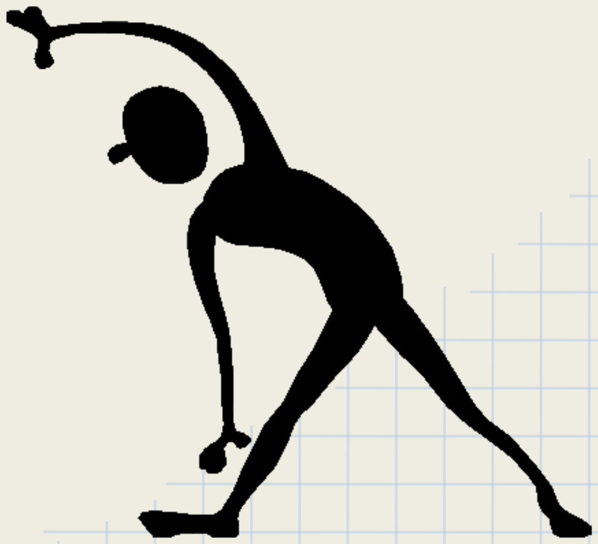
Defining Generics

Generics are defined using special uppercase generic type placeholders.

```
public class Calculator<T> {
    // T stands for "Type"
    // t is the name of the variable
    private T t;

    public T add(T t1, T t2) {
      return t1 + t2;
    }
    public T subtract(T t1, T t2) {
      return t1 - t2;
    }
}

// T becomes a placeholder for Integer
Calculator<Integer> calc = new Calculator<Integer>();
System.out.println(calc.add(5, 6)); // 11
```

Generics exercise

- Open up **intro_java_exercises** in Eclipse
- Follow Instructions in **Ex2_Generics**



HashMap

- The most common collections in Java are **ArrayList** and **HashMap**. We have already covered arrays, so let's learn about hashes.
- A HashMap is a collection of values which are not kept in order but are instead kept by "associating" each value to a 'key'.
- For example, imagine you want to determine a rating for a movie. You could attach the rating (value) to the movie title (key).

```
HashMap<String, Integer> ratingMap = new HashMap<String, Integer>();
ratingMap.put("Dark Knight", 5);
ratingMap.put("Pirahna 3D", 1);
```

HashMap, cont'd.

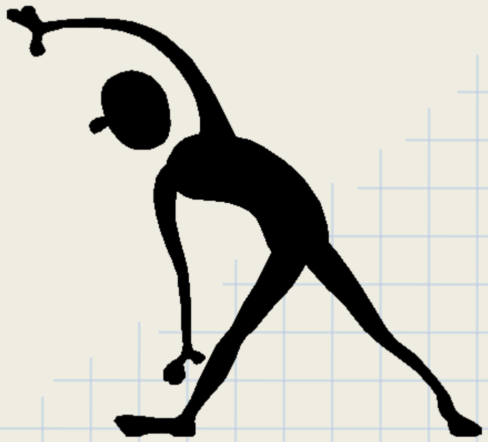
- HashMaps can serve a surprisingly large number of uses within programs. In particular, anytime you need to *associate* pairs of things together for each access by the 'key', you want a hash.
- The most important thing is to determine the correct 'key' and 'value'. The key should be the data you want to be able to access by and the value should be the resulting attached data. A good example is a phone directory (key is the name, value is the number).

HashMaps example

```
// Store ratings for movies
HashMap<String, Integer> ratingMap = new HashMap<String, Integer>();
ratingMap.put("Dark Knight", 5);
ratingMap.put("Spiderman 2", 3);
ratingMap.put("Pirahna 3D", 1);
System.out.println("Dark Knight is rated a " + ratingMap.get("Dark Knight"));

// or numbers for people
HashMap<String, String> personMap = new HashMap<String, String>();
ratingMap.put("Bob", "555-555-4532");
ratingMap.put("Bill", "555-123-4743");
ratingMap.put("Drake", "555-345-6753");
System.out.println("Bob can be reached at " + ratingMap.get("Bob"));
```

HashMaps exercise

- Open up **intro_java_exercises** in Eclipse
- Follow Instructions in **Ex3_NamesAndAddresses**
- Follow Instructions in **Ex4_RemoveOdd**



9. Advanced Classes

Diving deeper into designing classes

Interfaces

- When defining objects and classes, there is often situations where multiple classes and objects of unrelated type have certain shared expectations for certain methods.
- In cases where classes have the same expected methods but do not actually share much behavior, you should consider using **interfaces**.
- At the simplest level, an interface is a series of **methods** with **empty bodies** that can be 'implemented' within a class definition. Once **implemented**, that class has to then **define every method** that is contained within the **interface**.

Interfaces, cont'd.

- Interfaces **provides a contract** which guarantees **certain methods** will be defined within any class that **implements** it.
- This is helpful when you have disparate (unrelated) items or complex classes that should all have **certain methods** to expose the same **'interface'**.

```
public interface Friendly {
     public void sayHello();
}

public class Person implements Friendly {

     public void sayHello() {
          System.out.println("Hi!");
     }
}
```

Interfaces, cont'd.

- A class can actually implement **multiple interfaces** and follow all of their method requirements:

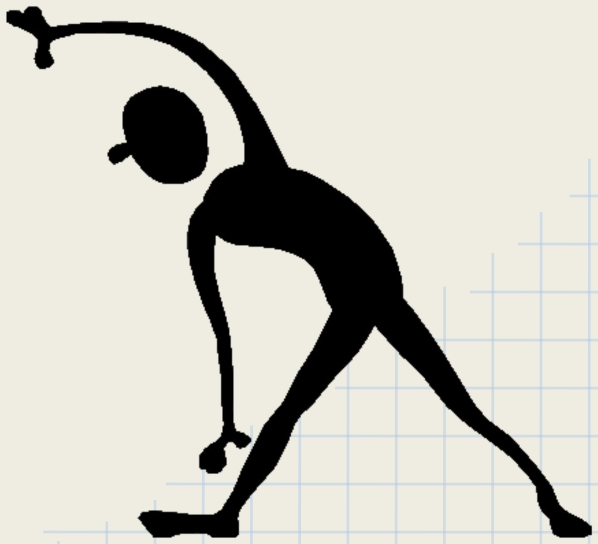
```java
public interface Friendly {
     public void sayHello();
}

public interface Athletic {
    public void jog(int miles);
}

public class Person implements Friendly, Athletic {

     public void sayHello() {
          System.out.println("Hi!");
     }

    public void jog(int miles) {
         System.out.println("I just jogged " + miles + " miles");
    }

}
```

Interfaces example

```
public interface MobileDevice {
  public String getScreenSize();
}

public class iOSDevice implements MobileDevice {
  private String version = "4G";
  private String driveSize = "16GB";
  private String screenSize;

  public String getScreenSize() {
    return screenSize;
  }
}

public class AndroidDevice implements MobileDevice {
  private String version = "4G";
  private String driveSize = "16GB";
  private String fullScreenSize;

  public String getScreenSize() {
    return fullScreenSize;
  }
}
```

Interfaces exercise

- Open up **intro_java_exercises** in Eclipse
- Follow Instructions in **Ex1_Alarm**



Abstract Classes

An **abstract class** is a class that cannot be directly instantiated (created) but can be inherited from by other classes.

- An abstract class is **like an interface** but can include certain methods that are implemented while others **need to be defined** by the subclass.
- An abstract class can contain **abstract methods** which are methods with name and arguments but **no** implementation.

Abstract Classes, cont'd.

Abstract classes are like regular classes but they can include **abstract methods** (with no implementation) and **cannot be created** directly (only subclassed)

```
public abstract class iOSDevice {
   // declare fields
   // declare non-abstract methods
   // declare abstract methods (must be defined by subclass)
   abstract void shutDown(int delay);
   abstract int getBatteryLife();
}

public class iPhone extends iOSDevice {
  void shutDown(int delay) {
    System.out.println("Shutting Down");
  }

  int getBatteryLife() {
     // ...
  }
}
```

Abstract vs Interface

What is the difference between an abstract class and an interface?

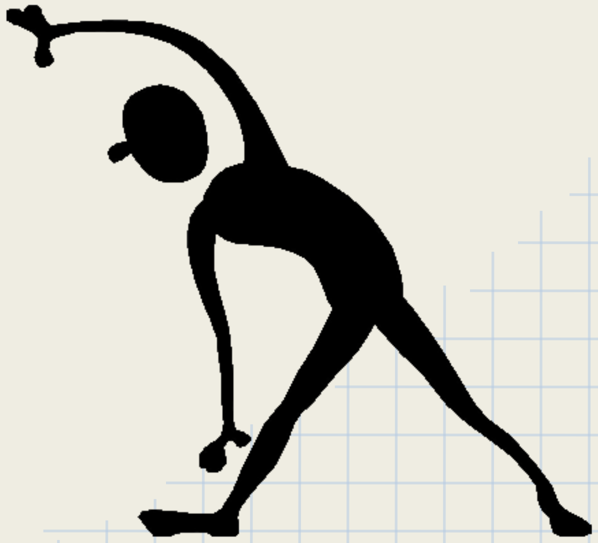
- Both of them allow us to **list method signatures** that must be defined in a class.
- Abstract classes can **contain fields** and **fully implemented methods** along with abstract methods.
- If an abstract class contains **only abstract** method declarations, it should be **declared** as an **interface** instead.

Abstract Classes example

```
public abstract class MobileDevice {
  // fields
  String name;
  String screenSize;
  // implemented methods
  public String getScreenSize() {
    return screenSize;
  }
  // and abstract
  public void startUp();
  public void shutDown(int delay);
}

public class iOSDevice extends MobileDevice {
  public String startUp() {
     // ... implement startUp
  }

  public String shutDown(int delay) {
     // ... implement shutDown
  }
}
```

Abstract Classes exercise

- Open up **intro_java_exercises** in Eclipse
- Follow Instructions in **Ex2_Abstract**



Inner Classes

An **inner class** is a class that is **defined** within another **class**.

- An inner class is a part of the enclosing class and has access to other fields or methods of the parent class.
- Inner classes are used as a way to organize and group classes together if they are only ever used together.
- Small "helper" classes that are used only to make a parent class work can be hidden from public access.

Inner Classes, cont'd.

An **inner class** is a class that is **defined** within another **class**.

```
public class MainGraphicalWindow() {
   // fields
   public int width = 500;
   // methods
   public void drawScreen(Graphics g) {
      MenuSection menu = new MenuSection();
      menu.drawMenu(g);
   }
   // inner class definition
   class MenuSection {
      public void drawMenu(Graphics g) {
         // ...
         g.drawRect(0, 0, width, 100);
      }
   }
}
```

Inner Classes

An **inner class** only **exists** within the parent class instance.

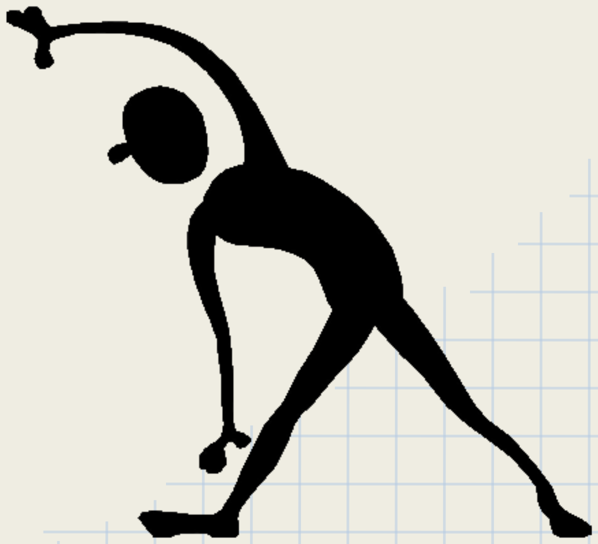
- Inner classes have **direct access** to the parent object's methods and fields.
- Inner classes **cannot exist** independently outside the outer class instance.
- **Static** inner classes are called **nested classes** and behave differently (no access to instance variables or methods).

Inner Class Example

```
public class MobileDevice {
  private int width = 500;
  private int height = 300;

  public MobileDevice() {
    ScreenDrawer drawer = new ScreenDrawer();
    drawer.drawScreen();
  }

  // defines inner class
  class ScreenDrawer {
     void drawScreen() {
        // draw screen
        drawBox(width, height);
     }
     void drawBox(int w, int h) { ... };
  }
}
```

Inner Classes exercise

- Open up **intro_java_exercises** in Eclipse
- Follow Instructions in **Ex3_Inner**



Anonymous Classes

An **anonymous class** is a class created on the fly **without** a specified name.

- **Anonymous classes** are **created inline** without a name and can be **passed as parameters** into methods.
- These are often used in Android when you want to **define complex behavior** as a **parameter** for a **method**.

Anonymous Classes

Anonymous classes are **inline classes** defined as part of a larger block of code.

In this example, we are defining the behavior of a button when clicked

```
Button aButton = getMyButton();
aButton.setOnClickListener(new OnClickListener() {

    public void onClick(View v) {
        // User clicked my button, do something here!
        System.out.println("clicked!");
    }

});
```

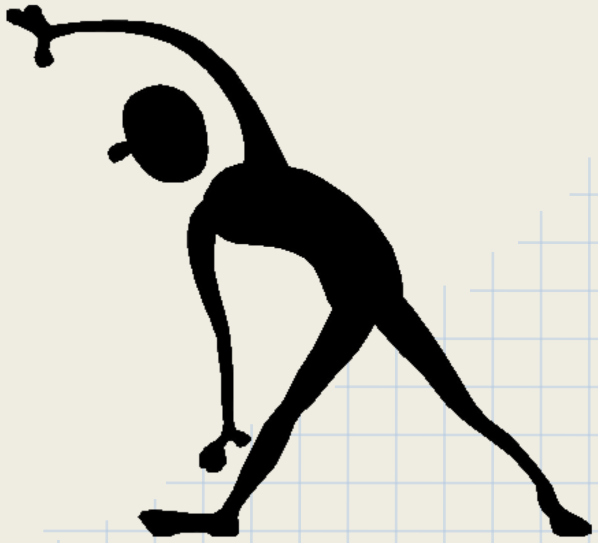
Anonymous Class Example

Anonymous classes can be created based on any **existing class or interface**

```
public class MobileDevice {

  // define click handler using anonymous class
  public MobileDevice() {
    // setup click handling in constructor
    Button b = getButton();
    b.setOnClickListener(new OnClickListener() {
        public void onClick(View v) {
            System.out.println("clicked!");
        }
    });
  }

}
```

Anonymous Class Exercise

- Open up **intro_java_exercises** in Eclipse
- Follow Instructions in **Ex4_Anonymous**



10. Addendum

Additional concepts for Android development

XML Documents

An **XML Document** is a file that contains a description of some data.

- **XML** has markup tags similar to HTML that contain content. XML is a **node**-based document.
- XML starts with a **root** node that contains all other content in the document.
- Each node can contain **child nodes** as well as **node attributes**.

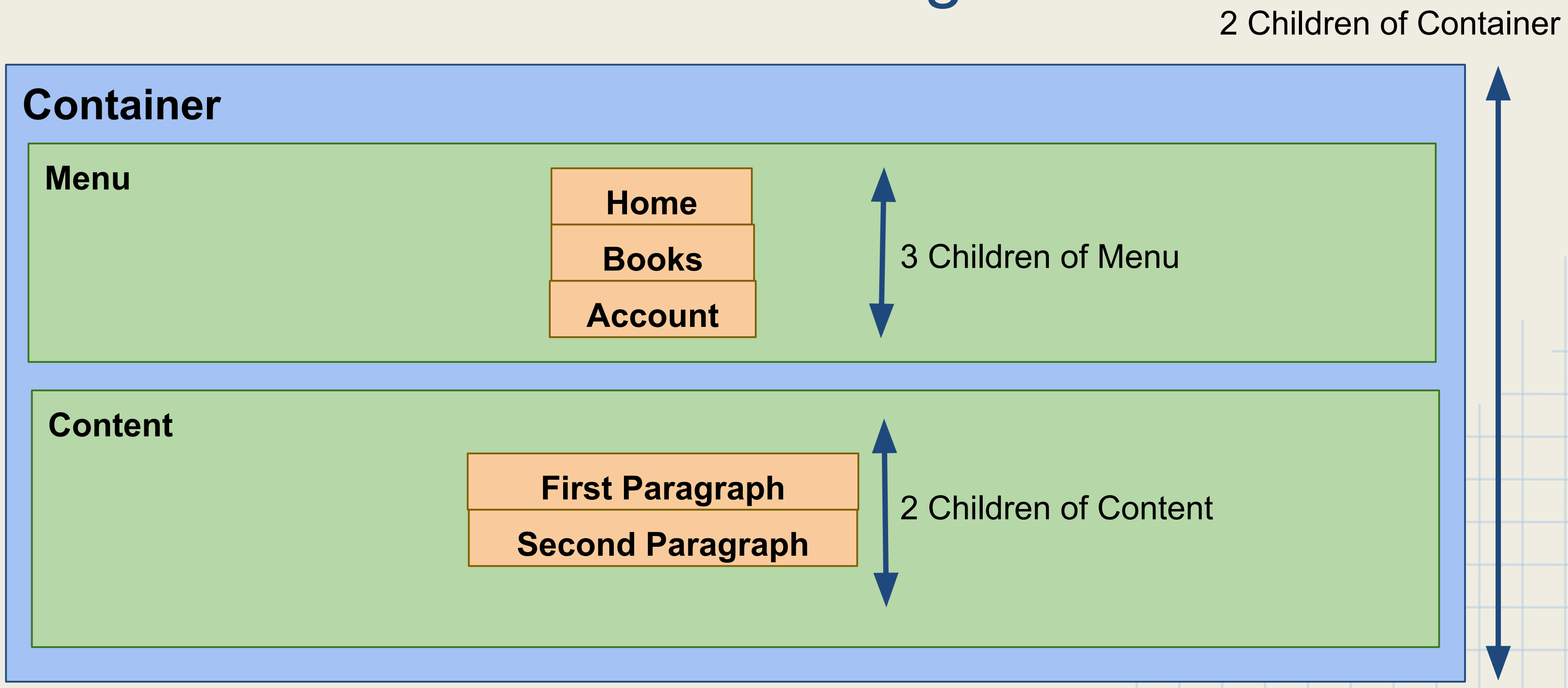
XML Example

Anonymous classes can be created based on any **existing class or interface**

```
<container>
  <menu name="top-navigation">
    <item title="Home"/>
    <item title="Books" />
    <item title="Account" />
  </menu>
  <content backgroundcolor="red">
    <paragraph>First paragraph</paragraph>
    <paragraph>Second paragraph</paragraph>
  </content>
</container>
```

XML Documents

An **XML Document** contains a series of nested nodes that describe something.



XML User Interfaces

XML is often used to describe user interfaces in applications.

- In **Web** development, a form of XML called HTML is used to define a website's interface.
- In **Android** development, mobile graphical interfaces are built using an XML file.
- There are at least two types of nodes: **containers** (contains widgets) and **widgets** (paragraphs, buttons, textboxes).

Exception Handling

An **exception** is an object that represents an unexpected event within our application.

- When an **error occurs** within a method, the method creates an **object** which contains information about the error, including its **type** and other details.

- Creating an exception object and having it passed to the Java runtime is called **throwing an exception**.

Exception Handling

Exceptions are used to handle when problems arise in an application.

- When an exception is *thrown* (because an error occurred), an exception can be **handled** by code that is triggered. This exception handler chosen is said to **catch the exception**.

- If nothing handles the exception event in the code, then the **application stops running** and the **exception is displayed** in the console.

Exceptions Example

Anonymous classes can be created based on any **existing class or interface**

```
PrintWriter out = null;

try {
    System.out.println("Entering" + " try statement");
    out = new PrintWriter(new FileWriter("OutFile.txt"));
    out.println("Hello World");
} catch (IOException e) {
    System.err.println("Caught Error: " + e.getMessage());
} finally {
    System.out.println("Closing PrintWriter");
    out.close();
}
```

Wrapping Up

Summary and next steps

Concept Review

- Variables
- Loops
- Conditionals
- Arrays
- Classes
- Inheritance
- Interfaces
- Collections
- Generics

Keep Learning

- Form a small group or schedule a recurring meetup.
- Learn to read blog tutorials and stackoverflow.com
- Build more 2D games and apps.
- Take next course, **Introduction to Android**.

Next Steps

We have covered many important parts of the **Java** programming language.

All of these concepts we learned are critical parts of almost any type of software development and are **directly applicable** to **Android** mobile applications.

To learn **Android development**, we must build on all these concepts and learn several new ones as well.

Introduction to Android

Introduction to Android is a 12 week course covering the basics of developing Android apps.

- Android fundamentals
- User Interfaces and Views
- User Interactions
- Creating Menus and Actions
- Application Flows
- Networking and Building Client Apps
- Communicating with other Apps

Comments? Questions?

We love feedback