

# Welcome to Java!

---



# Topics

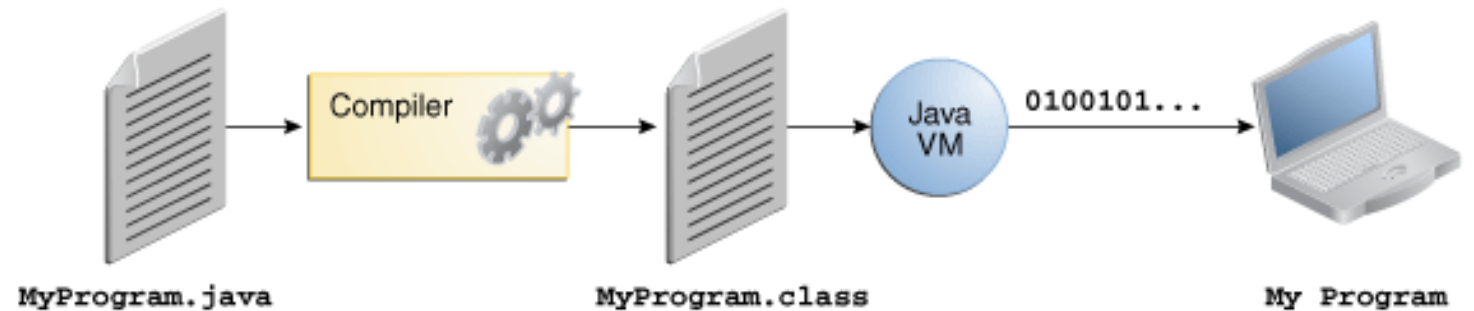
---

- 1) Why Java?
- 2) Classes and Objects
- 3) Hello World Application
- 4) Primitive Data Types
- 5) Arrays
- 6) Stack VS Heap
- 7) Strings
- 8) Type Conversion and Casting
- 9) Operators

# Why Java?

---

- Portable
- Simple syntax
  - Based on C
- Rich API
- Java is FREE
- Support from Oracle Corporation
- Easy to learn
  - Automatic memory management
  - No pointers!

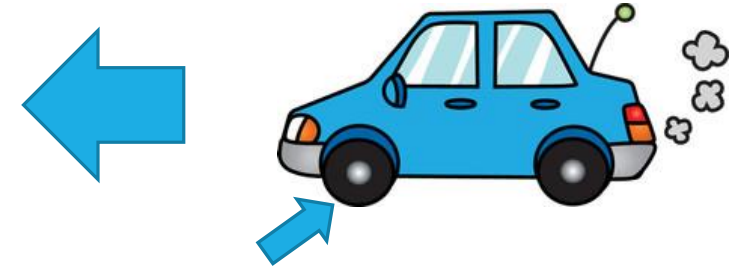


- Object-oriented
  - Polymorphism
  - Inheritance
  - Abstraction
  - Encapsulation

# Classes and Objects

- **Class** is a blueprint that defines:
  - State (variables)
    - Car has wheels.
  - Behavior (methods)
    - Car can drive.
  - State and behavior are called “class members”
- **Object** is an instance of a class
  - Objects are created from class blueprints
  - 2005 Honda Civic is a Car

```
MyClass ref = new MyClass();
```



# Basic Class Structure

---

```
class MyClassName {  
    int variable;  
    String word;  
  
    void myMethod(){  
        //method implementation  
    }  
}
```

# Hello World!

---

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



# Primitive Data Types

---

<b>boolean</b>	N/A	True or False
<b>byte</b>	1 byte	-128 to 127
<b>char</b>	2 bytes	0 to 65536
<b>double</b>	8 bytes	1.7e-308 to 1.7e+308
<b>int</b>	4 bytes	-2,147,483,648 to 2,147,483,647
<b>long</b>	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
<b>float</b>	4 bytes	3.4e-308 to 3.4e+308
<b>short</b>	2 bytes	-32,768 to 32,767

# Arrays

---

- Array is a group of variables
- Elements of an array must be the same type

```
items = new int[5];
```

**OR**

```
int[ ] items = {1, 2, 3, 4, 5};
```

- Elements are accessed by an index

```
items[0] = 1;
```

```
items[3] = 4;
```

- Find an array's size with `arrayName.length`  
`items.length`



# 2-D Arrays

---

```
int variable[][] = new int[2][3];
```

Index Locations in a 2-D Array

<b>[0][0]</b>	<b>[0][1]</b>	<b>[0][2]</b>
<b>[1][0]</b>	<b>[1][1]</b>	<b>[1][2]</b>

```
variable[0][2] = 5;
```

```
variable[1][1] = 10;
```

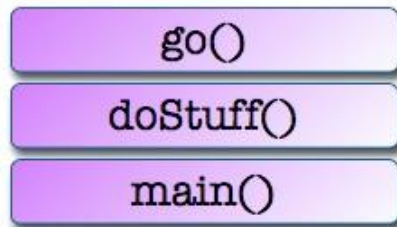
		<b>5</b>
	<b>10</b>	

# Stack VS Heap

---

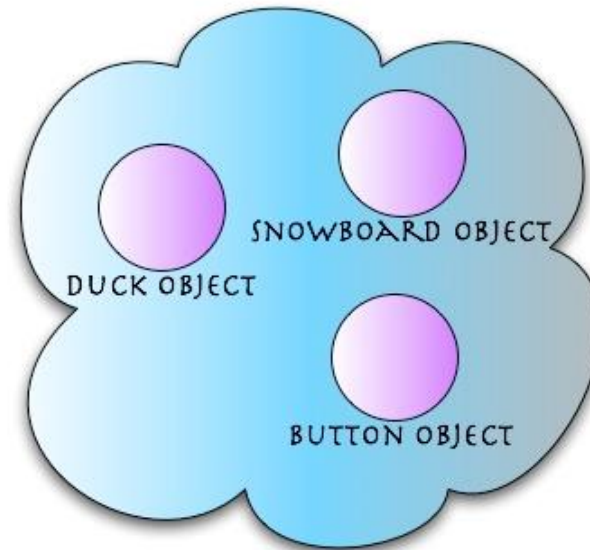
## The Stack

Where method invocations  
and local variables live



## The Heap

Where ALL objects live



# String

- String is a special Java class

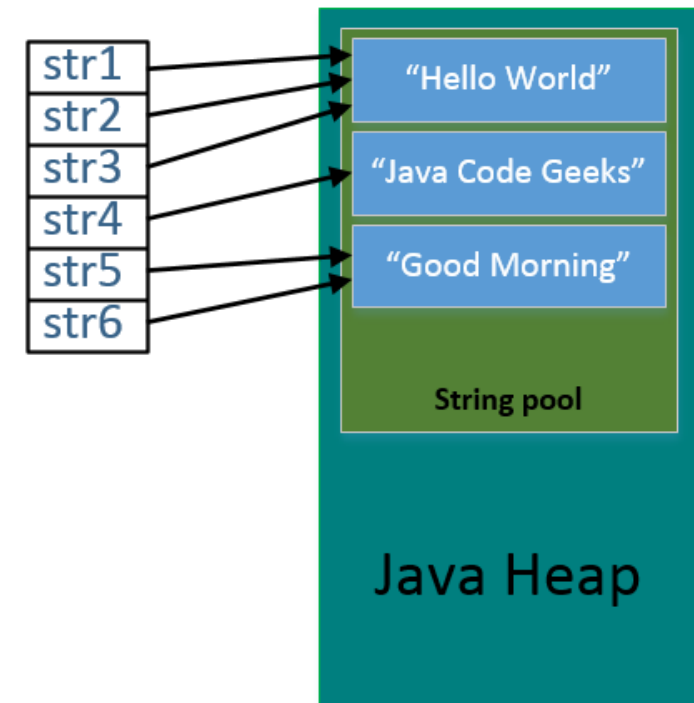
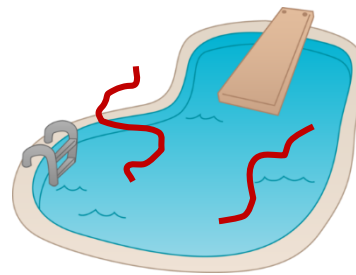
```
String words = new String("Hello World");
```

**OR**

```
String words = "Hello World";
```

- Strings are immutable
- Strings go to the String Pool

```
String str1 = "Hello World";  
String str2 = "Hello World";  
String str3 = "Hello World";  
String str4 = "Java Code Geeks";  
String str5 = "Good Morning";  
String str6 = "Good Morning";
```



# Type Conversion and Casting

---

- Convert a variable from one type to another (explicit or implicit)
- Explicit conversion (must be done with casting):

```
int variable = 10;
```

```
byte item = (byte) variable;
```

- Implicit conversion:

```
byte item = 10;
```

```
int variable = item;
```

- You can cast objects, too!

```
Dog dog = new Dog( );
```

```
dog = (Animal) dog;
```

# Operators

---

+	ADDITION
-	SUBTRACTION
*	MULTIPLICATION
/	DIVISION
%	MODULUS
++	INCREMENT
--	DECREMENT
+=	ADDITIONASSIGNMENT
-=	SUBTRACTION ASSIGNMENT
%=	MODULUSASSIGNMENT

# Relational Operators

---

== EQUAL TO

!= NOT EQUAL TO

> GREATER THAN

< LESS THAN

>= GREATER THAN OR EQUAL TO

<= LESS THAN OR EQUAL TO

# Boolean Logical Operators

---

&	BITWISE AND
	BITWISE OR
&&	SHORT-CIRCUITAND (LOGICAL)
	SHORT-CIRCUITOR (LOGICAL)
!	LOGICALUNARY NOT
^	XOR
>>	RIGHT SHIFT
<<	LEFT SHIFT

# Ternary Operator

---

?      TERNARY OPERATOR

➤ A quick “if-else” expression

```
int k = 10;
```

```
int m = k > 6 ? 1 : 0;
```

➤ k greater than 6? Then m = 1. Else m = 0.



# Review

---

- 1) Why Java?
- 2) Classes and Objects
- 3) Hello World Application
- 4) Primitive Data Types
- 5) Arrays
- 6) Stack VS Heap
- 7) Strings
- 8) Type Conversion and Casting
- 9) Operators

# Assignment

---

- Write a program to add, subtract, multiply and divide two non zero hard-coded numbers.
- Write a program to search for the greatest of three numbers using Short-circuit Operators and print the result.
- Write a program –declare two variables a and b and initialize them to true and false respectively. Get the output of the following computations:

`!a`

`a | b`

`(!a & b) | (a & !b)`

- Never delete your practice projects. You never know when you need to reference that code!

# Intro to Java Syntax

---



# Topics

---

## 1) Control Statements

- I. **if**
- II. **switch-case**
- III. **for**
- IV. **while**
- V. **do-while**
- VI. **break**
- VII. **continue**

## 2) More on Java classes and objects

## 3) Stack and Heap

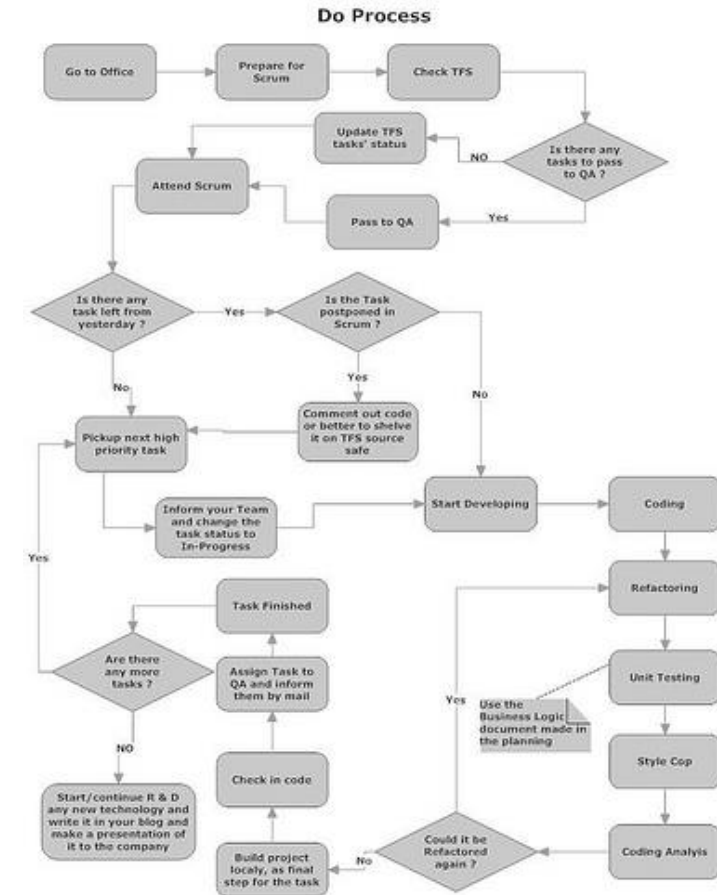
## 4) Method calls

## 5) Methods with parameters

## 6) Parameters vs Arguments

# Control Statements

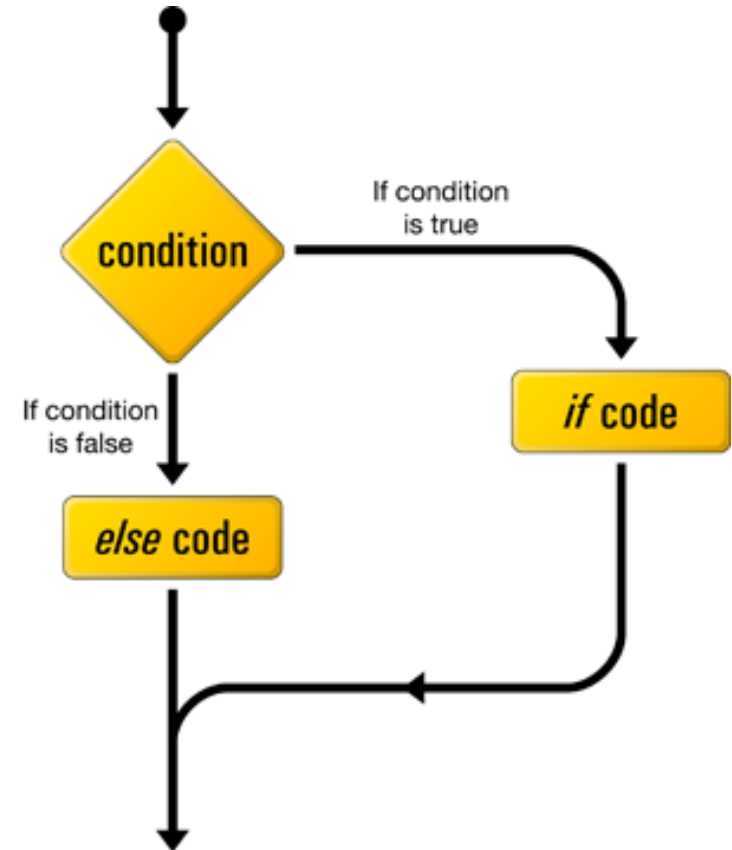
- if
- switch-case
- for
- while
- do-while
- break
- continue



# If/Else

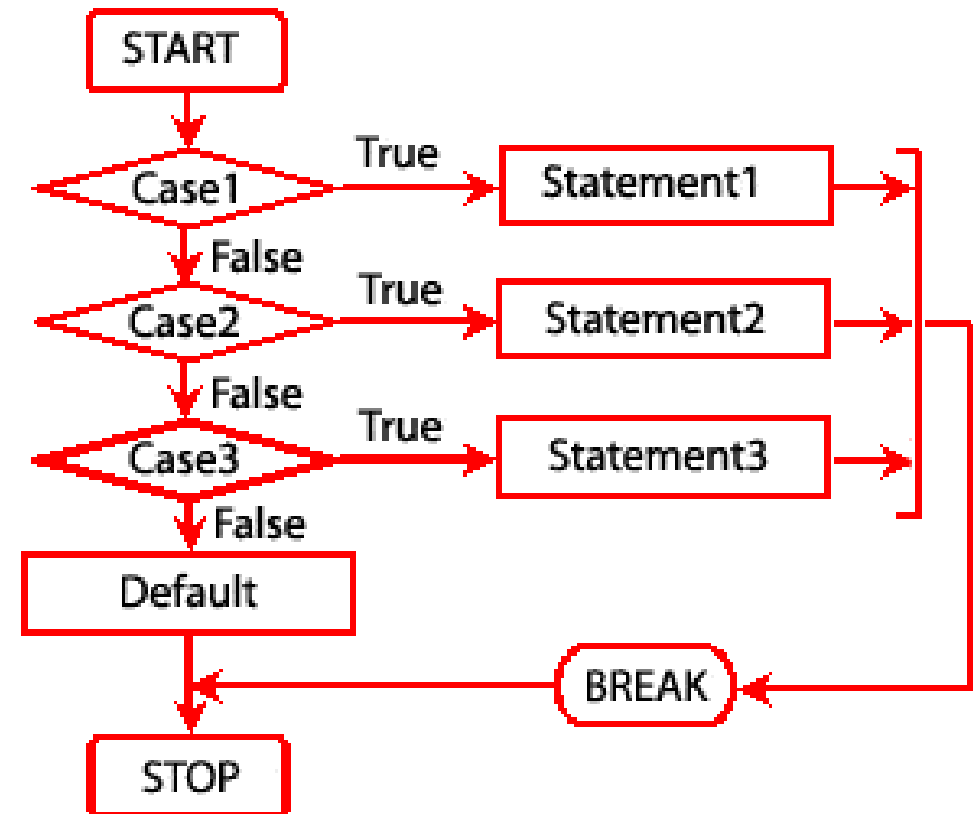
---

```
int a = 0;  
if (a > 5){  
    System.out.println("greater than 5");  
}  
else{  
    System.out.println("less than 5");  
}
```



# Switch/Case

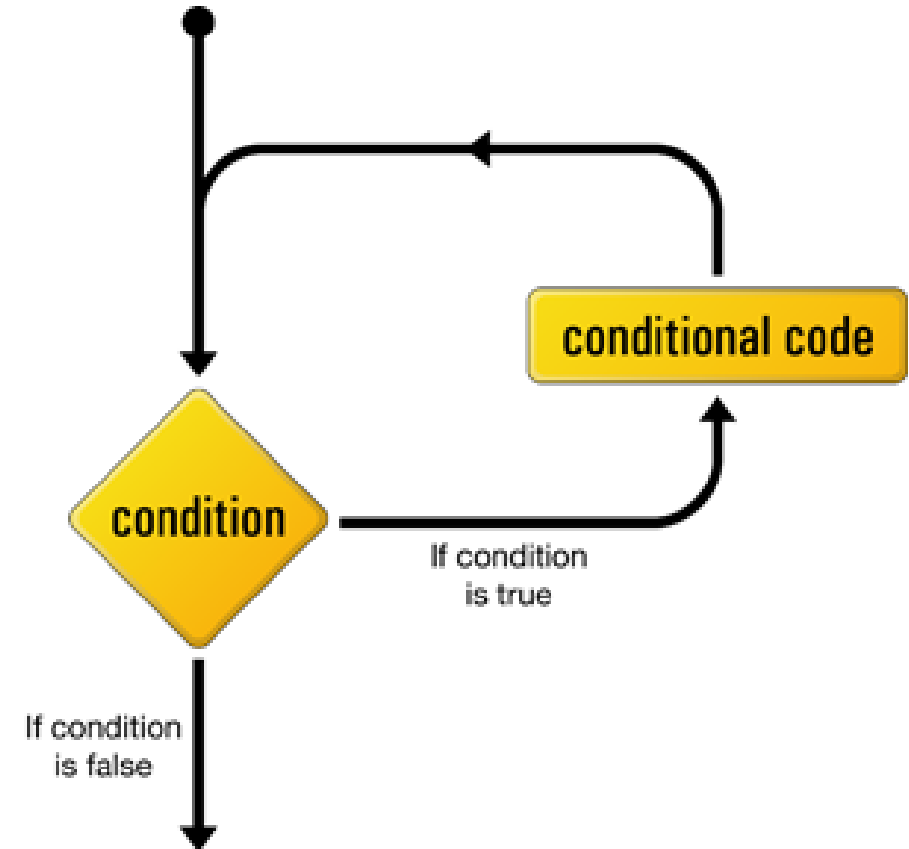
```
int i = 2;  
switch(i) {  
    case 0:  
        System.out.println("Zero");  
        break;  
    case 1:  
        System.out.println("One");  
        break;  
    case 2:  
        System.out.println("Two");  
        break;  
    default:  
        System.out.println("greater than 2");  
}
```



# While Loop

---

```
int num = 1;  
while (num < 10) {  
    System.out.println(num);  
    num++;  
}
```

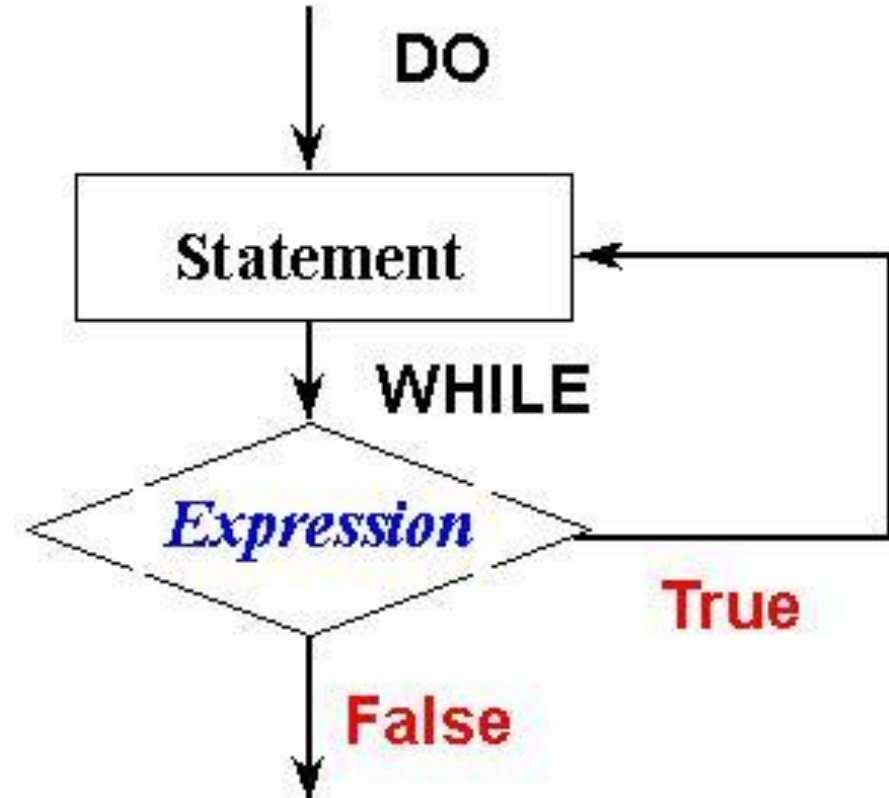




# Do/While Loop

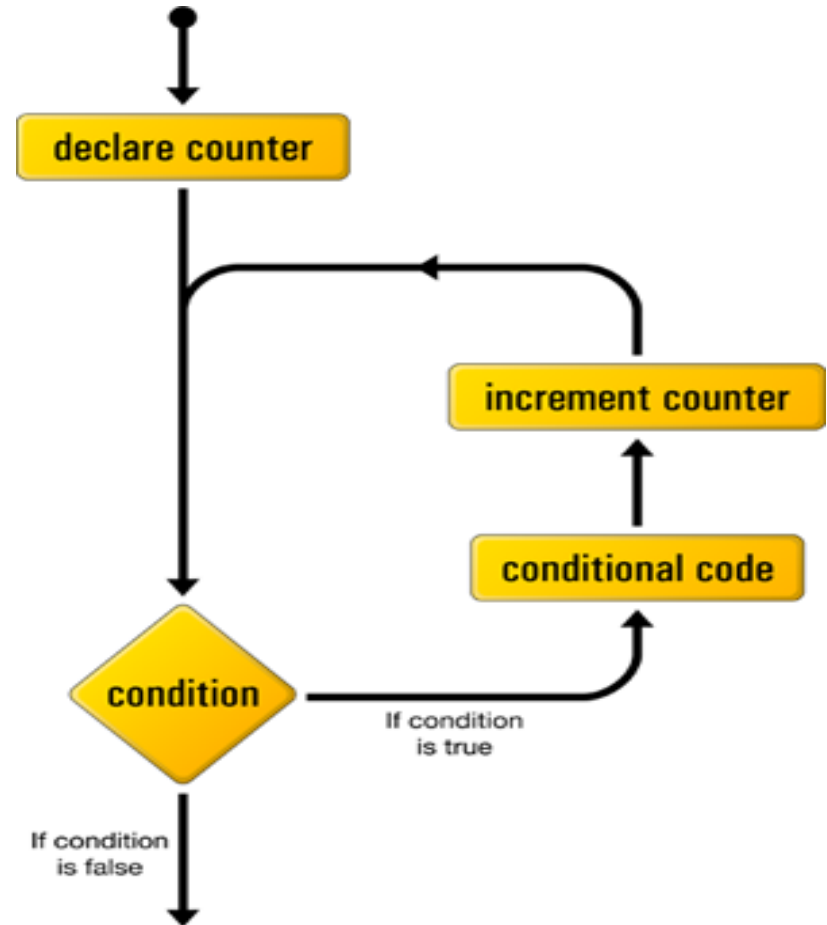
---

```
int num = 1;  
do {  
    System.out.println(num);  
    num++;  
} while(num < 10);
```



# For Loop

```
for (int num = 0; num < 10 ; num++)  
{  
    System.out.println(num);  
}
```



# Enhanced For Loop

---

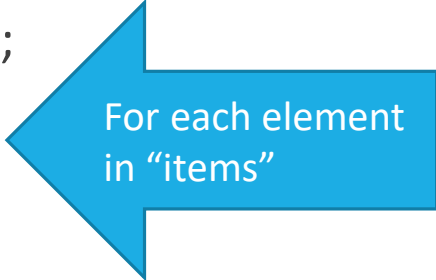
```
int[ ] items = {1, 2, 3, 4, 5};
```

```
for (int element : items)
```

```
{
```

```
    System.out.println(element);
```

```
}
```

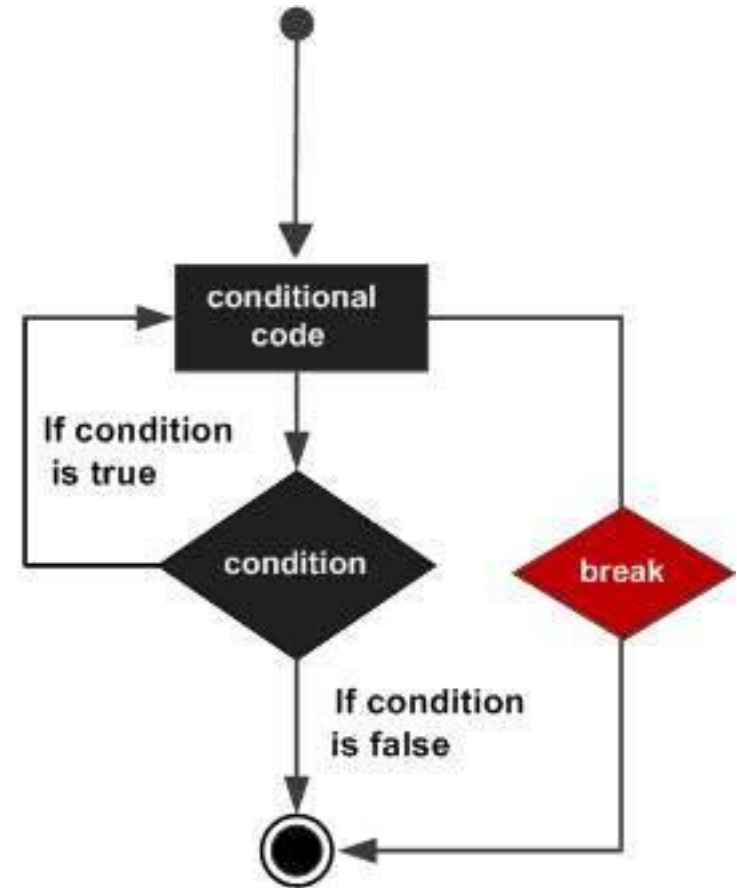


For each element  
in "items"

- "int element" declares a local variable
- At the end of the block:
  - Loop moves to the next element in the array

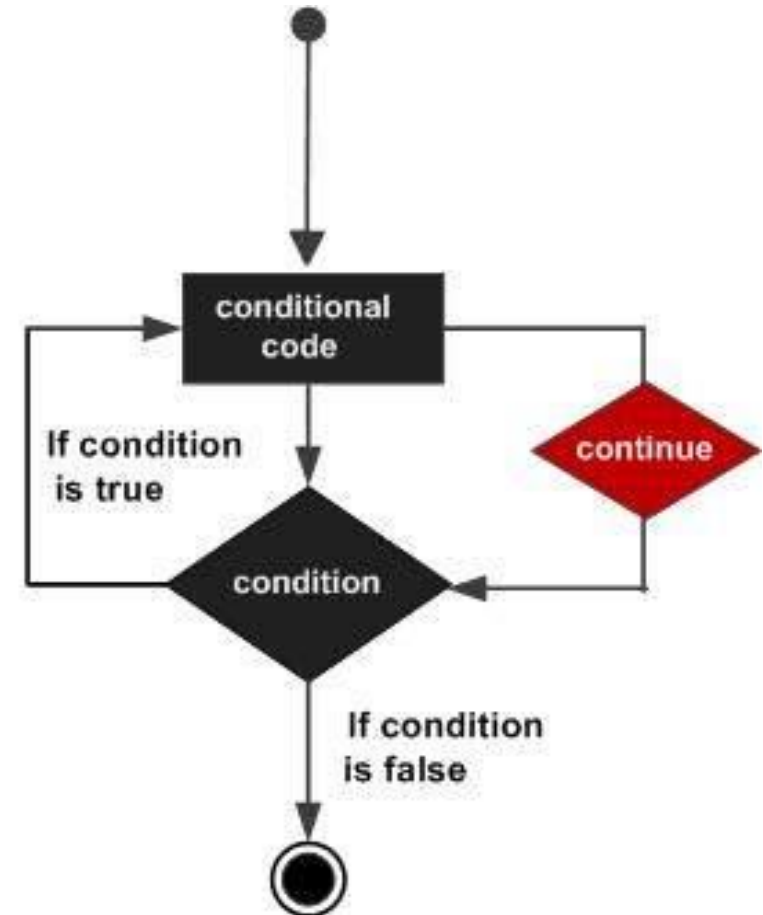
# Break

```
for (int a = 0; a<10; a++){  
    if (a == 5) break;  
    System.out.println(a);  
}  
  
System.out.println("Loop complete!");
```



# Continue

```
for (int i = 0; i < 10; i++){  
    if (i == 5) continue;  
    System.out.println(i);  
}
```



# Class

---

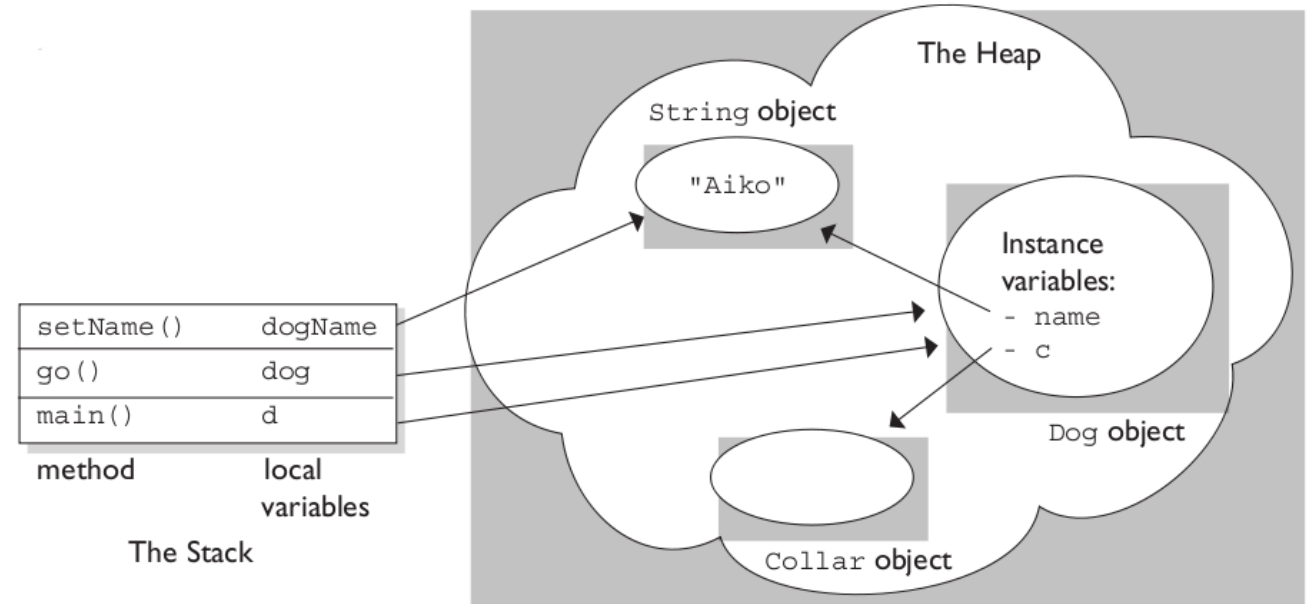
```
class MyClass{  
    // instance variables  
    int variable;  
  
    // methods  
    void myMethod(){  
  
    }  
}
```

Circle
-radius:double = 1.0 -color:string = "red"
+Circle(radius:double,color:string) +getRadius():double +setRadius(radius:double):void +getColor():string +setColor(color:string):void +getArea():double

# Objects and Methods

- Create an object of a class:
  - `MyClass ref = new MyClass( );`
- Objects let you call methods
  - `ref.myMethod( );`
- Every method has a return type
  - `String`
  - `int`
  - `An Object`
- If not, return type is `void`

```
void myMethod( ){  
    }  
}
```



# Return

---

- In a method use:
  - **return** someVariable;
- Return Statement will:
  - Terminate the local method
  - Pass the value to where the method was called
- Example:
  - You can use return multiple times
  - The first one that is reached is used

```
int myReturn(){  
    int variable = 1;  
    return variable;  
}
```

```
void myMethod(){  
    int number;  
    Test ref = new Test();  
    number = ref.myReturn();  
}
```




# Parameterized Methods

---


- Methods may take parameters of any data type

```
int myMethod(int x, int y){  
    return x + y;  
}
```



- Methods may take any number of parameters
- Values passed into parameters are called arguments

```
void add(){  
    Test ref = new Test();  
    int sum = ref.myMethod(5, 2);  
    System.out.println(sum);  
}
```



# Review

---

## 1) Control Statements

- I. **if**
- II. **switch-case**
- III. **for**
- IV. **while**
- V. **do-while**
- VI. **break**
- VII. **continue**

## 2) More on Java classes and objects

## 3) Stack and Heap

## 4) Method calls

## 5) Methods with parameters

## 6) Parameters vs Arguments

# Assignment

---

- Write a program to add, subtract, multiply and divide two numbers using methods with parameters using only one class.
- In one project, create two classes. One class should contain only methods (add, subtract, multiply and divide). The other class should contain only the main() method which calls each of the methods from the previous class.
- Create an example for each control statement.

# Members, Modifiers, and More

---



# Topics

---

- 1) Access Modifiers
- 2) Constructors
- 3) Overloaded Constructors
- 4) Overloaded Methods
- 5) Static Members
- 6) Final
- 7) String API
- 8) Command Line Arguments

# Access Modifiers

---

➤ **public**

➤ **protected**

➤ **default** (no modifier)

➤ **private**

Modifier	Class	Package	Subclass	World
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
<i>no modifier*</i>	✓	✓	✗	✗
private	✓	✗	✗	✗

# Applying Access Modifiers

---

➤ Class:

```
public class MyClass {  
  
}
```

➤ Method:

```
protected void myMethod(){  
  
}
```

➤ Variable:

```
private String password;
```

➤ Constructor:

```
public MyClass(){  
  
}
```

# Constructors

---

- The constructor is called using the **new** keyword
- A constructor initializes an object immediately
- It has the same name as the class
- It looks like a method
- It has no return type
- It can accept parameters
- The JVM creates a “default constructor” if you do not provide one

```
MyClass ref = new MyClass();
```



```
public class MyClass {  
  
    public MyClass(){  
        // initialization code  
    }  
}
```





# Overloaded Constructors

---

- A class may have more than one constructor
- Each constructor must take in different arguments
- Arguments can be of any data type or object
- The order of the arguments matters!
- A constructor without arguments is generally called:
  - “No-arg” constructor
  - A no-arg constructor is NOT the default constructor

```
MyClass ref1 = new MyClass(5, "Hello");  
MyClass ref2 = new MyClass();  
MyClass ref3 = new MyClass(5, 2);  
MyClass ref4 = new MyClass("World", 3);  
MyClass ref5 = new MyClass(10);
```

```
public class MyClass {  
    public MyClass(){  
    }  
    public MyClass(int x){  
    }  
    public MyClass(int x, int y){  
    }  
    public MyClass(String str, int x){  
    }  
    public MyClass(int x, String str){  
    }  
}
```

# Overloaded Methods

---

- Overloaded methods have:
  - The same signature
  - Different parameters
- Arguments can be of any data type or object
- The order of the arguments matters
- Changing the return type alone will not work!
  - The method call would be ambiguous

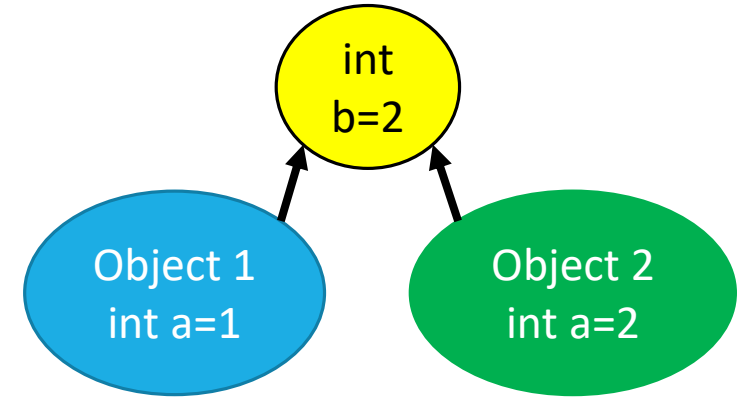
```
public String myMethod(){  
  
}
```

```
public class MyClass {  
  
    public void myMethod(){  
  
    }  
  
    public void myMethod(int x, String str){  
  
    }  
  
    public void myMethod(String str, int x){  
  
    }  
  
}
```

# Static Members

---

- Static variables
  - Shared by all instances of that class
  - Changing value in one object is seen by all others
- Static methods
  - Method calls do not require an object
  - `MyClass.myStaticMethod( );`
- Static block
  - Also known as the “static initializer”
  - Block of code run by the JVM at Class Loading
  - Used to initialize static variables
- **Non-static variables cannot be referenced from a static context**



# Static Examples

---

```
public class MyClass {
```

1 → `static int counter;`

2 → `static{`  
    `counter = 0;`  
    `}`

3 → `public MyClass(){`  
    `counter++;`  
    `MyClass.myMethod();`  
    `}`

4 → `public static void myMethod(){`  
    `}`

```
}
```

# Final

---

- Final variables
  - Cannot be changed once initialized
  - Constant value
  - Typically written in all uppercase
- Final methods
  - Cannot be overridden
- Final classes
  - Cannot be extended
  - The last leaf on the “Animal” Inheritance tree

```
private static final long serialVersionUID = 3839097431617638416L;
```

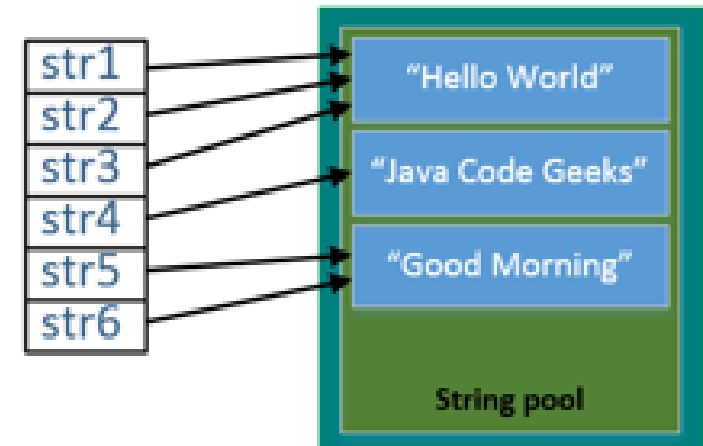
```
public final void myMethod(){  
    // My algorithm is boss... Don't change it!!!  
}
```

```
public final class UrsusAmericanus {  
  
    /*  
     * Type of Bear cannot get any  
     * more specific, dude...  
     */  
  
}
```



# String API

- The String class is final
- String objects are immutable
  - Remember the String pool?
- String objects are constructed as:
  - A **byte**[ ] array
  - A **char**[ ] array
- String class has many String manipulating methods:
  - equals(Object ref)
  - charAt(int index)
  - toUpperCase( )
  - toLowerCase( )
- Find a String's length with the "length" variable:  
myString.length



# Command Line Arguments

---

- Command line arguments
  - Pass as arguments into main( ) method
  - Values are passed as Strings
- Entered through the command line:
  - Windows Command Prompt
  - Linux/UNIX shell
  - Macintosh Terminal
- Entered through an IDE:
  - Type a space between each argument

```
public static void main(String[] args){  
    for (int i = 0; i < args.length; i++) {  
        System.out.println(args[i]);  
    }  
}
```

1. Click on **Run -> Run Configurations**
2. Click on **Arguments** tab
3. In **Program Arguments** section , Enter your arguments.
4. Click **Apply**

# Review

---

- 1) Access Modifiers
- 2) Constructors
- 3) Overloaded Constructors
- 4) Overloaded Methods
- 5) Static Members
- 6) Final
- 7) String API
- 8) Command Line Arguments



# Assignment

---

- Write a program that prints all Strings passed through the command line.
  - (Hint: loop through `main(String[ ] args)`)
- Create a new Java project
  - Create a Customer class with:
    - At least 2 constructors
    - At least 2 overloaded methods
    - At least 1 static variable
    - At least 1 final variable
  - Create a main method in a new class that:
    - Creates 2 customers
    - Uses each of the Customer class members

# Intermediate Concepts

---



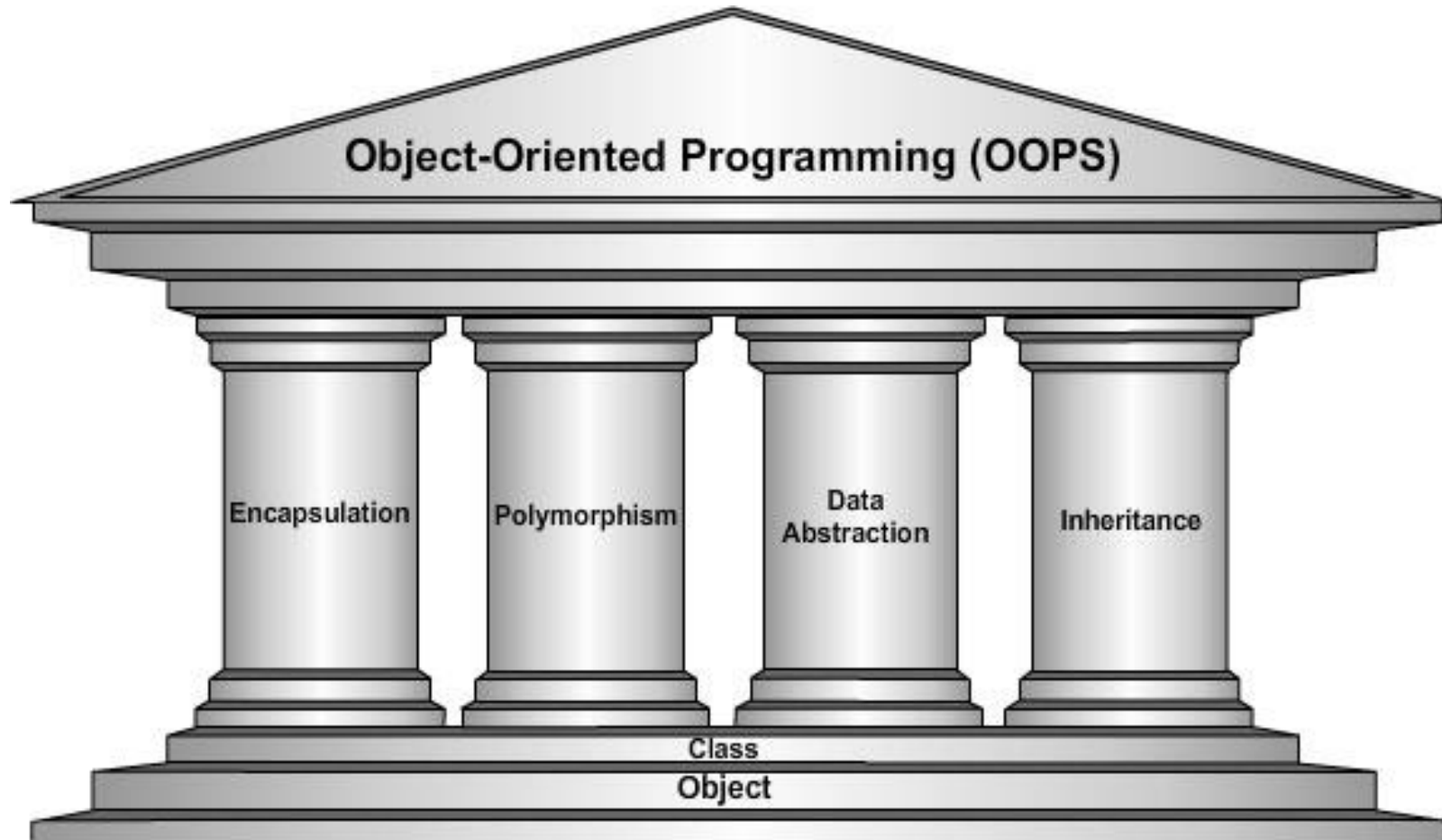
# Topics

---

- 1) Abstraction (Abstract classes and methods)
- 2) Encapsulation
- 3) Inheritance
- 4) Polymorphism
- 5) Method Overriding
- 6) Constructor execution sequence
- 7) Packages
- 8) Import Statements
- 9) Introduction to Interfaces

# Four Pillars of OOP

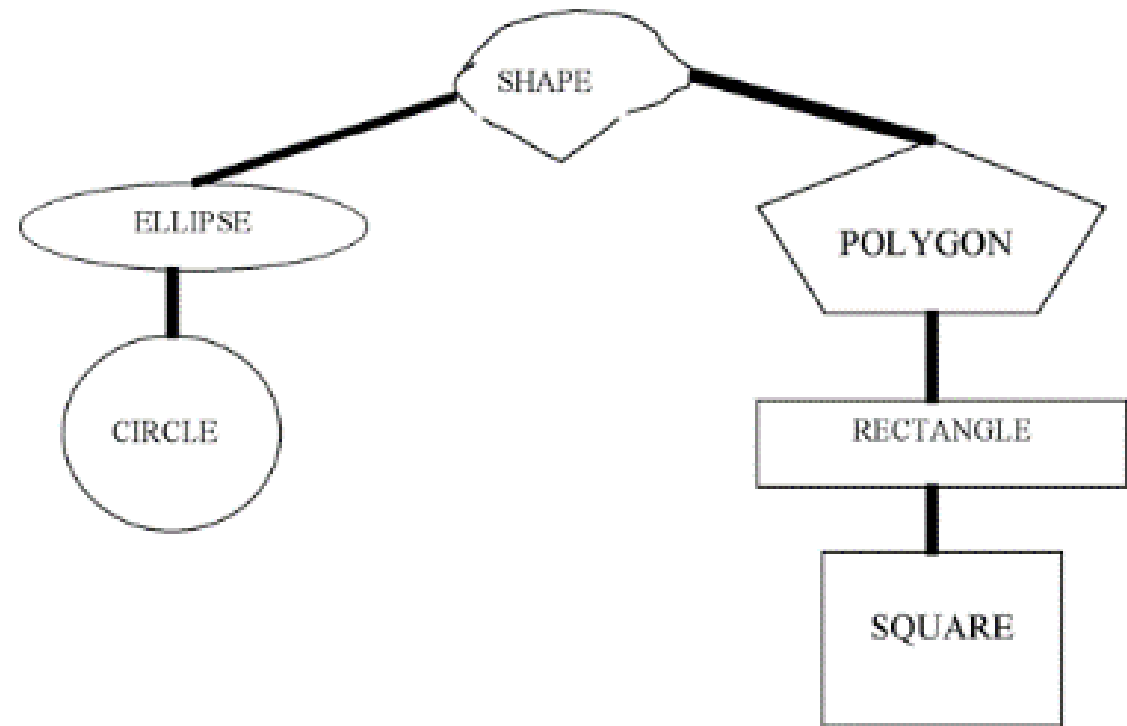
---



# Abstraction

---

- Centralize common characteristics
- Reduce complexity
- Increase coding efficiency
- Eliminate duplicate code
- Generalize behavior into conceptual classes
- Provide a template for other classes



# Abstract Keyword

---

- Abstract classes and methods use the **abstract** modifier
- Abstract classes cannot be instantiated!
- An abstract method does not have a body
  - No braces or implementation
  - Terminated with semicolon
- Abstract class can have both:
  - Abstract methods
  - Concrete methods

```
public abstract class Shape {  
  
    private String color;  
  
    public abstract void drawShape();  
  
    public void setColor(String str){  
        this.color = str;  
        System.out.println(color);  
    }  
}
```

# Encapsulation

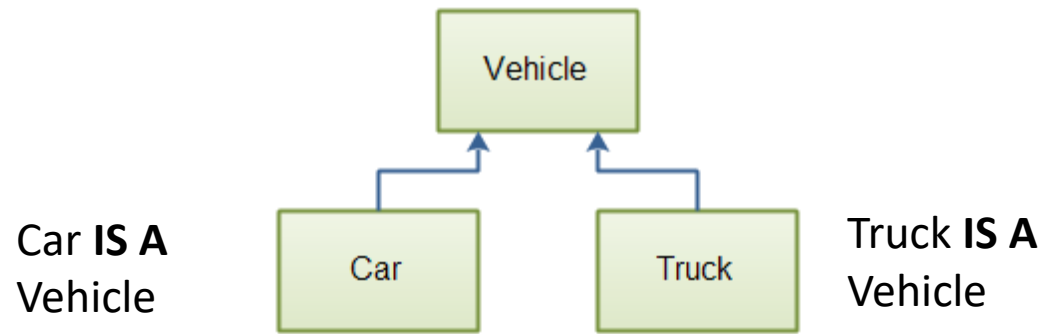
---

- Restriction of access to class members
- Class members packaged into one component
- Variables declared **private** to hide data
- External classes retrieve data through **public** methods
- Maintains boundaries between components
- Separation of concerns

```
public class Employee {  
  
    private String name;  
    private int SSN;  
    private String position;  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public int getSSN() {  
        return SSN;  
    }  
    public void setSSN(int sSN) {  
        SSN = sSN;  
    }  
    public String getPosition() {  
        return position;  
    }  
    public void setPosition(String position) {  
        this.position = position;  
    }  
}
```

# Inheritance

- A class can inherit members from another class
- A class can extend another class's functionality
- The class that **extends** another is called a sub-class
  - Also called child class
- The class that is extended is the super-class
  - Also called parent class



```
public class Rectangle extends Polygon{  
  
    public double sideLength;  
    public double topLength;  
    public double angle;  
  
    public Rectangle(){  
        this.numberOfSides = 4;  
        angle = 90;  
    }  
}
```



# Polymorphism/Method Overriding

---

- From Greek:
  - πολύς, polys, "many, much"
  - and μορφή, morphē, "form, shape"
- A class can take many forms
- Polymorphism is achieved through:
  - Subclasses
  - Method overriding
- Methods in a subclass override a parent:
  - When the method signature is identical
  - When the method has the same parameters

```
public class Canine {  
    public void bark(){  
        System.out.println("Bark!");  
    }  
}  
  
public class Papillon extends Canine{  
    public void bark(){  
        System.out.println("Tiny, yappy bark!");  
    }  
}
```

# Polymorphism in Practice

```
public class Canine {  
    public void bark(){  
        System.out.println("Bark!");  
    }  
}
```

```
public class Papillon extends Canine{  
    public void bark(){  
        System.out.println("Tiny, yappy bark!");  
    }  
}
```

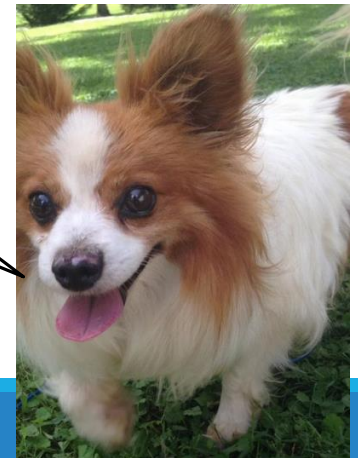
```
public class Bulldog extends Canine {  
    public void bark(){  
        System.out.println("Deep, intimidating bark!");  
    }  
}
```

```
public class MyClass {  
    public static void main(String[] args) {  
        Canine nugget = new Papillon();  
        Canine frances = new Bulldog();  
  
        nugget.bark();  
        frances.bark();  
    }  
}
```

Deep,  
intimidating  
bark!

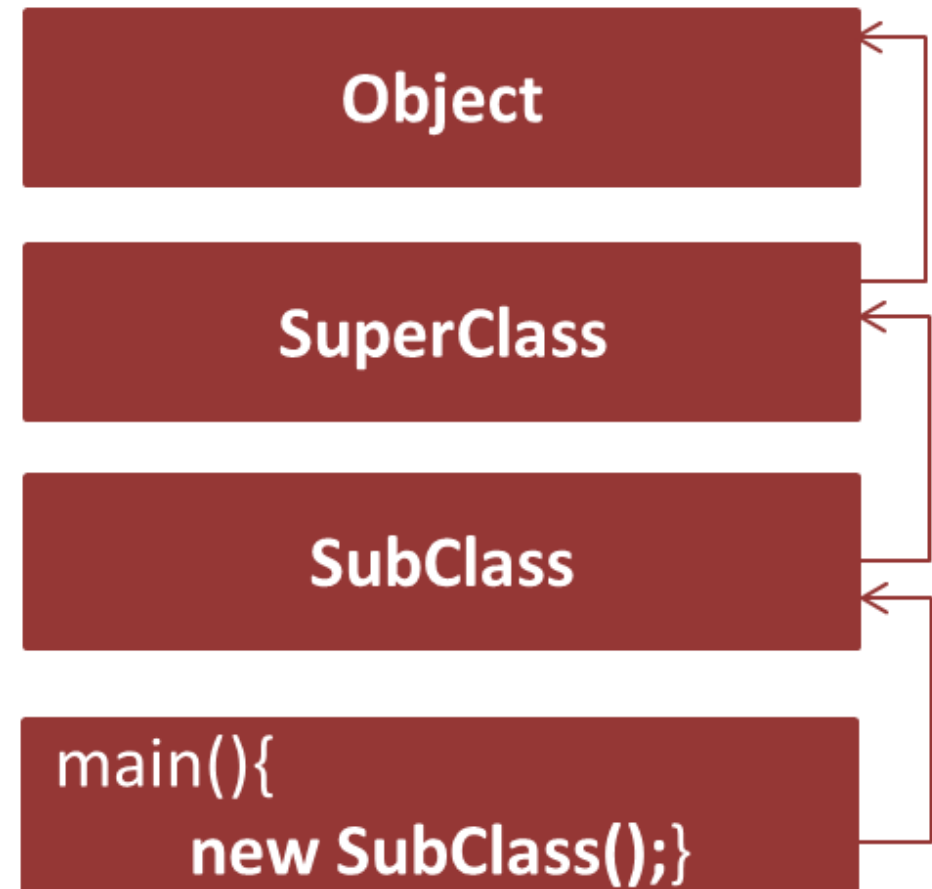


Tiny,  
yappy  
bark!



# Constructor Execution

- When an object is constructed, the JVM calls:  
    `super( );`
- Calls the no-arg constructor of its parent
- This continues to chain up the tree
- When the Object constructor is called...



# Packages

---

- A package is a folder to organize Java class files
- The package statement declares where your class resides
- The package statement is Line 1 of every class
- A best practice is to use “reverse domain naming”

```
package day1;  
  
public class HelloWorld {
```

# Imports

---

- Used when code is outside the current package
- Imports utilities from the Java API
- Imports classes from various frameworks
- Necessary step to using tools in your arsenal



```
package com.revature;

import fruits.Apple;

class Trainees{

    public void takeQuiz(){
        Apple fruits = new Apple();
        Apple.start();
    }

}
```

# Interfaces

---

- Interfaces are blank templates
- Used to define class behavior
- Enforce behavior from subclasses
- Hide implementation details
- Act as a contract for others to use your code
- A class **implements** an interface

```
public interface EmployeeInterface{  
    public void hire(Employee emp);  
    public void terminate(Employee emp);  
    public void setPay(Employee emp);  
}
```

```
public class Employee implements EmployeeInterface{  
    private String name;  
    public String getName() {  
        return name;  
    }  
    public void hire(Employee emp) {  
        System.out.println(this.getName() + " is hired!");  
    }  
    public void terminate(Employee emp) {  
        System.out.println(this.getName() + " is fired!");  
    }  
    public void setPay(Employee emp) {  
        System.out.println(this.getName() + " gettin paid!");  
    }  
}
```

# Interfaces

---

➤ Anyone can use your code without understanding the complexity!

➤ Why? Polymorphism!

➤ Interfaces are used widely in:

➤ Spring

➤ Hibernate

➤ EJB

➤ WebServices

➤ And many more frameworks!

```
public class Trainee extends Employee{  
  
    public void hireTrainee(){  
        EmployeeInterface ref = new Employee();  
        ref.hire(this);  
    }  
}
```

# Review

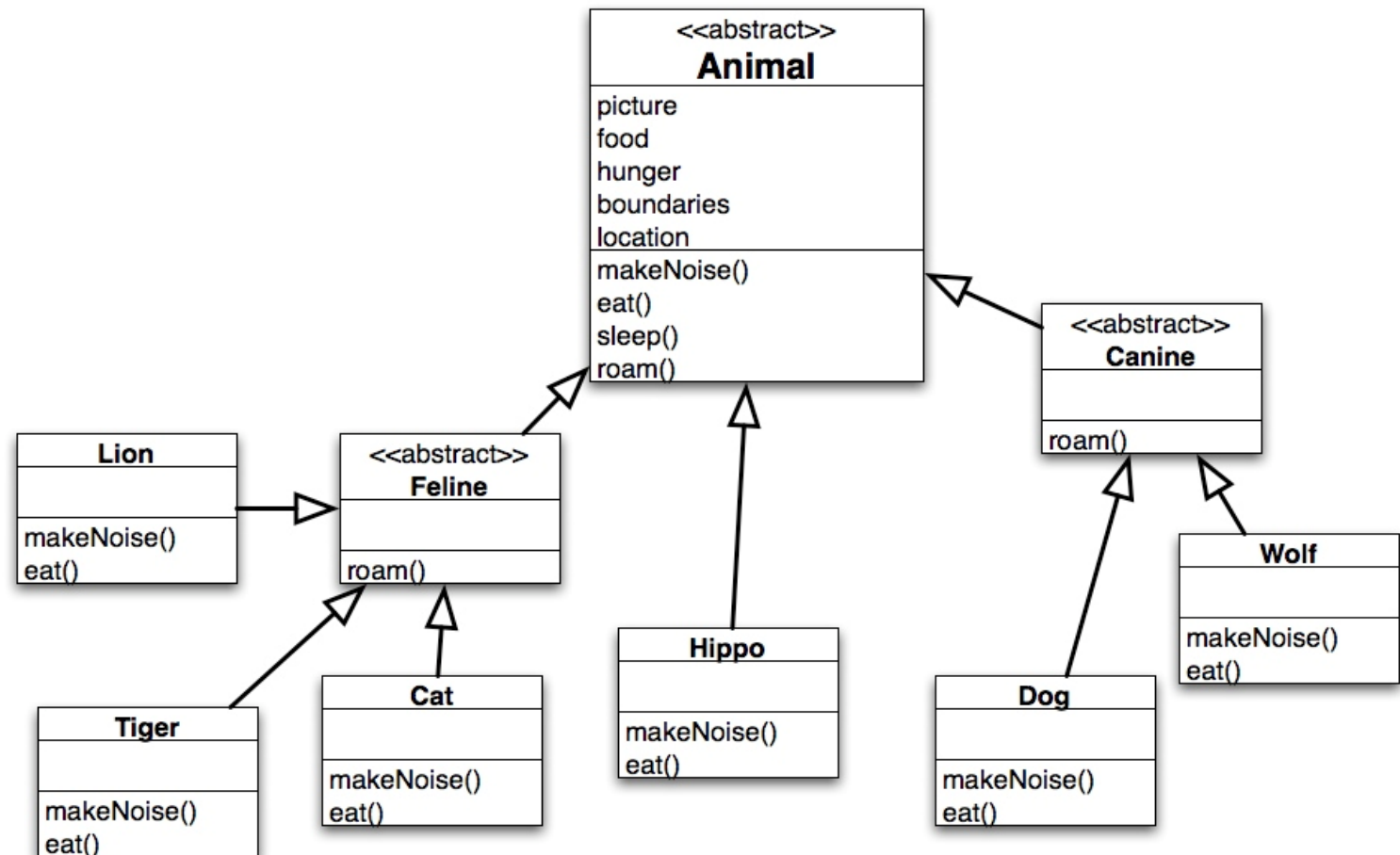
---

- 1) Abstraction (Abstract classes and methods)
- 2) Encapsulation
- 3) Inheritance
- 4) Polymorphism
- 5) Method Overriding
- 6) Constructor execution sequence
- 7) Packages
- 8) Import Statements
- 9) Introduction to Interfaces



# Assignment

- Create the Animal interface, Feline and Canine abstract classes, and the concrete classes.
- Each should have a variable and a method.
- Create an object of each, call all of the methods for each object, and observe the effects.



# Exception Handling

---



# Topics

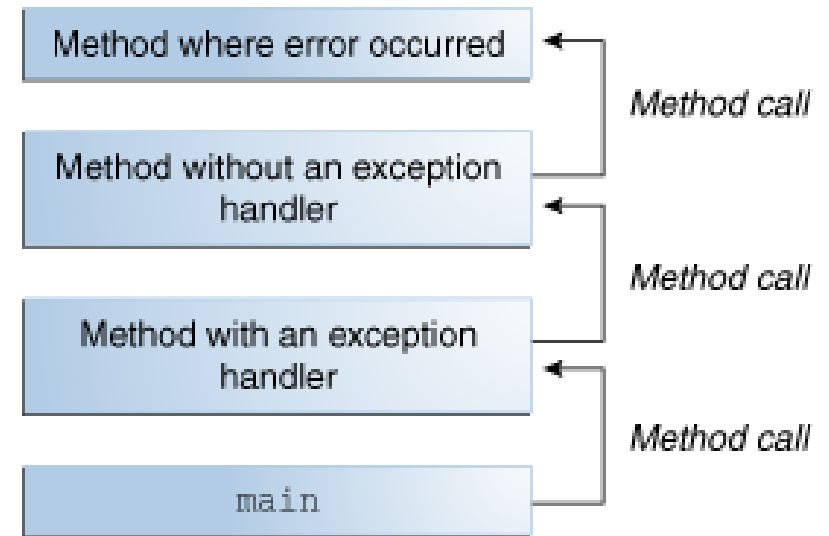
---

- 1) Intro to Exception Handling
- 2) Try/Catch Blocks
- 3) Exception Messages
- 4) The Stack Trace
- 5) Handling Multiple Exceptions
- 6) Throws Clause
- 7) Throw Statement
- 8) Custom Exceptions
- 9) Finally Block

# Intro to Exceptions

---

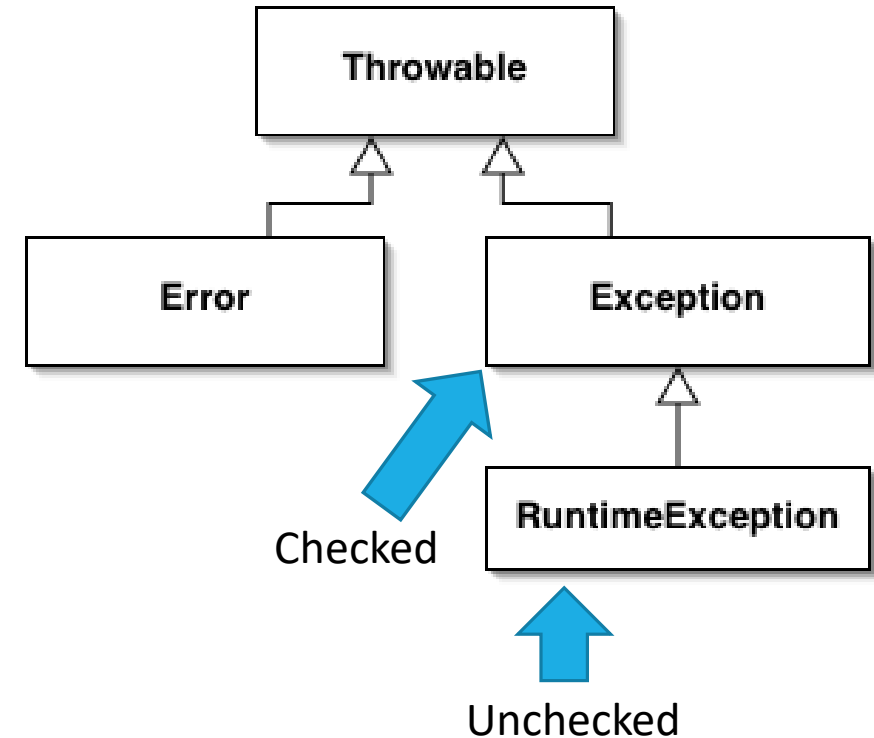
- Exceptions are “exceptional events”
- Exceptions disrupt normal execution flow
- Methods create Exception objects for the JRE
- Exception objects contain important information:
  - Exception type
  - State
  - Stack trace
- JRE searches down the stack for an appropriate handler
- If one is not found, the application terminates



The call stack.

# Exception Hierarchy

- All Exceptions extend the Throwable class
- Applications can recover from Exceptions
- The compiler can anticipate Checked Exceptions
  - FileNotFoundException, IOException, SQLException
- The compiler cannot anticipate Unchecked Exceptions
  - These occur at runtime
  - ClassCastException
  - ArrayIndexOutOfBoundsException
- Applications cannot reasonably recover from Errors
  - OutOfMemoryError
  - StackOverflowError



# Try/Catch Blocks

---

- Risky behavior is enclosed in a **try** block
- Recovery procedures are in a **catch** block
- If an exception is thrown:
  - **Try** block execution halts
  - JRE moves to the **catch** block
  - JRE must find a handler in the call stack
  - No handler = application termination

```
public void baseJump(){  
  
    try{  
        this.leapOffBuilding();  
        this.pullParachuteCord();  
    }  
  
    catch(Exception e){  
        this.pullEmergencyCord();  
    }  
  
}
```



# Exception Messages

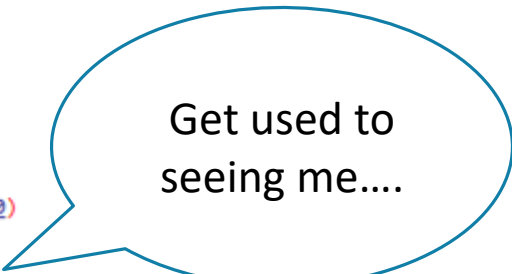
---

- Exception objects have many uses
- Print a Stack Trace to the console
- Get the Exception's message
- Get a Localized message
  - Used for internationalization

```
public void divide(){  
  
    try{  
        int x = 12;  
        int quotient = x / 0;  
        System.out.println(quotient);  
    }  
  
    catch(ArithmeticException e){  
        e.printStackTrace();  
        System.out.println(e.getMessage());  
        System.out.println(e.getLocalizedMessage());  
    }  
  
}
```

# The Stack Trace

```
org.jboss.resteasy.spi.UnhandledException: java.lang.NullPointerException
    at org.jboss.resteasy.core.SynchronousDispatcher.handleApplicationException(SynchronousDispatcher.java:323)
    at org.jboss.resteasy.core.SynchronousDispatcher.handleException(SynchronousDispatcher.java:199)
    at org.jboss.resteasy.core.SynchronousDispatcher.handleInvokerException(SynchronousDispatcher.java:175)
    at org.jboss.resteasy.core.SynchronousDispatcher.getResponse(SynchronousDispatcher.java:529)
    at org.jboss.resteasy.core.SynchronousDispatcher.invoke(SynchronousDispatcher.java:491)
    at org.jboss.resteasy.core.SynchronousDispatcher.invoke(SynchronousDispatcher.java:120)
    at org.jboss.resteasy.plugins.server.servlet.ServletContainerDispatcher.service(ServletContainerDispatcher.java:200)
    at org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher.service(HttpServletDispatcher.java:48)
    at org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher.service(HttpServletDispatcher.java:43)
    at javax.servlet.http.HttpServlet.service(HttpServlet.java:722)
    at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:304)
    at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:210)
    at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:224)
    at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:169)
    at org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:472)
    at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:168)
    at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:100)
    at org.apache.catalina.valves.AccessLogValve.invoke(AccessLogValve.java:929)
    at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:118)
    at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:405)
    at org.apache.coyote.http11.AbstractHttp11Processor.process(AbstractHttp11Processor.java:964)
    at org.apache.coyote.AbstractProtocol$AbstractConnectionHandler.process(AbstractProtocol.java:515)
    at org.apache.tomcat.util.net.JIoEndpoint$SocketProcessor.run(JIoEndpoint.java:302)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1110)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:603)
    at java.lang.Thread.run(Thread.java:722)
Caused by: java.lang.NullPointerException
    at com.filter.rest.guimpl.FilterRestServiceImpl.listNodesXML(FilterRestServiceImpl.java:37)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:601)
    at org.jboss.resteasy.core.MethodInjectorImpl.invoke(MethodInjectorImpl.java:140)
    at org.jboss.resteasy.core.ResourceMethod.invokeOnTarget(ResourceMethod.java:252)
```



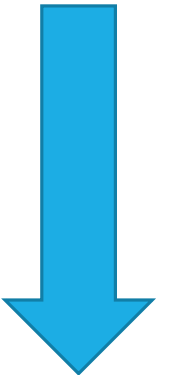
Get used to  
seeing me....



# Handling Multiple Exceptions

- Each **try** block can have multiple **catch** blocks
- You can have as many as you want
- The JRE checks from top to bottom
- It will pick the first one that can catch the Exception
- Always use specific Exception subclasses first!
  - RuntimeException
  - Exception

```
public void divideAndStore(){  
    try{  
        int x = 12;  
        int quotient = x / 2;  
        System.out.println(quotient);  
  
        int[] array = new int[3];  
        array[100] = quotient;  
    }  
    catch(ArithmeticException e){  
        e.printStackTrace();  
    }  
    catch(RuntimeException e){  
        e.printStackTrace();  
    }  
    catch(Exception e){  
        e.printStackTrace();  
    }  
}
```



# Throws Clause

---

- Two choices when an exception is thrown:

**Catch it**

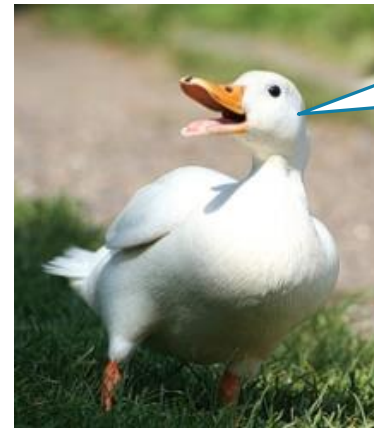
Got it,  
bruh.



```
try{  
  
}catch(Exception e){  
  
}
```

**Duck it**

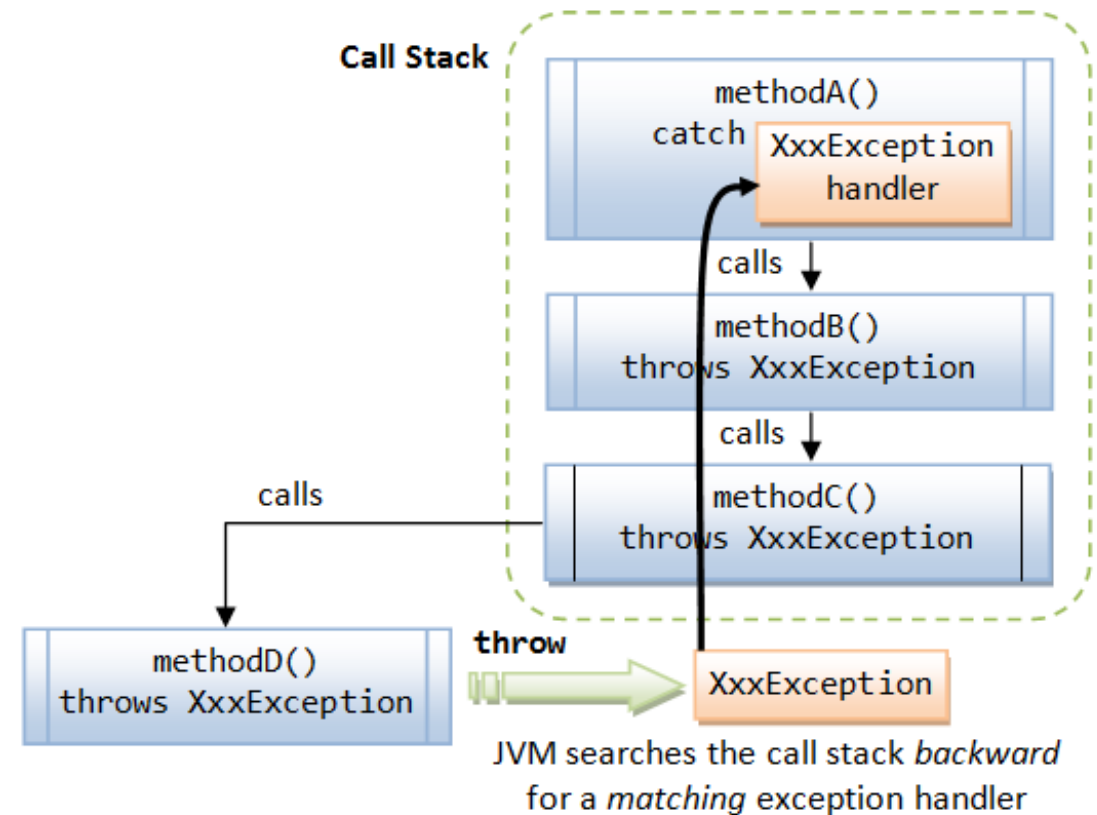
Let  
someone  
else handle  
it.



```
public void method() throws Exception{  
  
}
```

# Ducking Exceptions

- When a method ducks an exception:
  - JRE moves down the call stack
  - Searches for an appropriate handler
- You can duck an exception many times
- A handler must be reached somewhere, or:
  - The application terminates

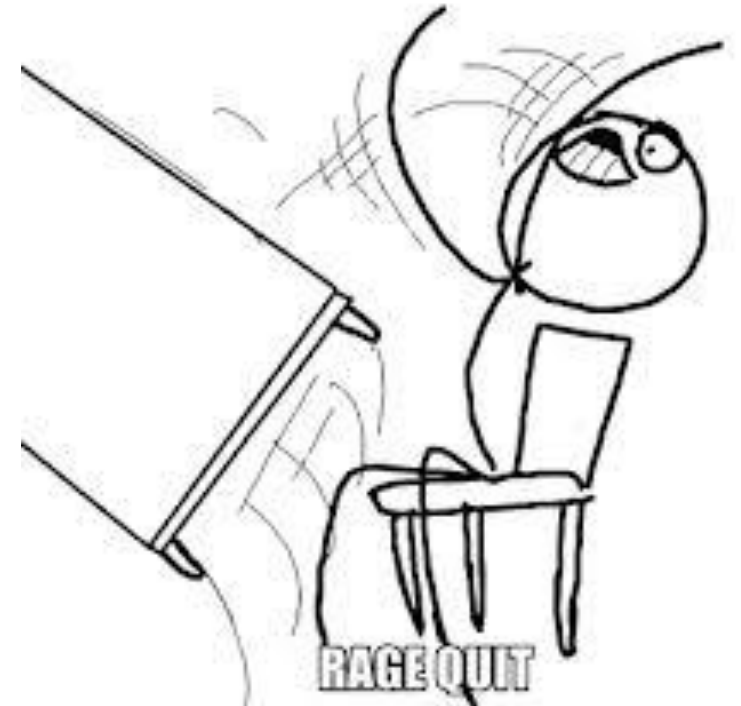


# Throw Statement

---

- To manually **throw** an Exception
- Creates an Exception-type object
- Passes the object to the JRE
- Is treated like any other exception occurrence
- Useful when testing your code
- Needed to **throw** custom exceptions

```
public void quitMath(){  
    // Math is too tough...  
    throw new ArithmeticException();  
}
```



# Custom Exceptions

---

- Tailored to your application
- Custom Exceptions are useful when:
  - The Exception type is not in Java API
  - Differentiation is necessary
  - Code throws too many related exceptions
  - Give users access to exceptions
- Custom Exceptions are easy!
  - Create a class that extends any Exception type
  - Override any Throwable methods you want
  - **Throw** the exception in your code

```
public class FallingException extends Exception{  
    public String getMessage(){  
        return "WARNING: You are free-falling!";  
    }  
}  
  
public void baseJump(){  
    try{  
        this.leapOffBuilding();  
        this.pullParachuteCord();  
    }  
    catch(FallingException e){  
        e.getMessage();  
        this.pullEmergencyCord();  
    }  
}  
  
public void pullParachuteCord() throws FallingException{  
    if(parachuteCord.isEmpty()){  
        throw new FallingException();  
    }  
}
```

# Finally Block

---

- A **finally** block follows a **try** or **catch** block
- Executes if an exception is thrown
- Executes if an exception is NOT thrown
- Used to close resources gracefully
  - Files
  - Databases
  - Network Connections
- Optional block
- But... a try block must have either a catch and/or a finally block

```
public void baseJump(){  
  
    try{  
        this.leapOffBuilding();  
        this.pullParachuteCord();  
    }  
    catch(FallingException e){  
        this.pullEmergencyCord();  
    }finally{  
        // Either way, you hit the ground..  
        this.landOnGround();  
    }  
}
```

# When Finally Does NOT Execute

---

- If `System.exit(0)` is called in either **try** or **catch** block
- Catastrophic error occurs
  - `OutOfMemoryError`
  - `StackOverflowError`
- Fatal deployment environment
- Other extreme hardware/software failures



# Review

---

- 1) Intro to Exception Handling
- 2) Try/Catch Blocks
- 3) Exception Messages
- 4) The Stack Trace
- 5) Handling Multiple Exceptions
- 6) Throws Clause
- 7) Throw Statement
- 8) Custom Exceptions
- 9) Finally Block



# Assignment

---

- Create a custom exception class that overrides the `getMessage( )` method
- Create a class that manually throws an exception of your custom type
- Use `System.out.println( )` to note where you are in the control flow. Example: “Starting try block”, “Ending try block”, “Starting catch block”, etc.
- Duck the exception at least once
- Implement a finally block that prints a graceful goodbye message
- Use the `System.exit(0)` command in the try block and rerun the application. Note the console output to see if the finally block executes.

# Advanced Concepts

---



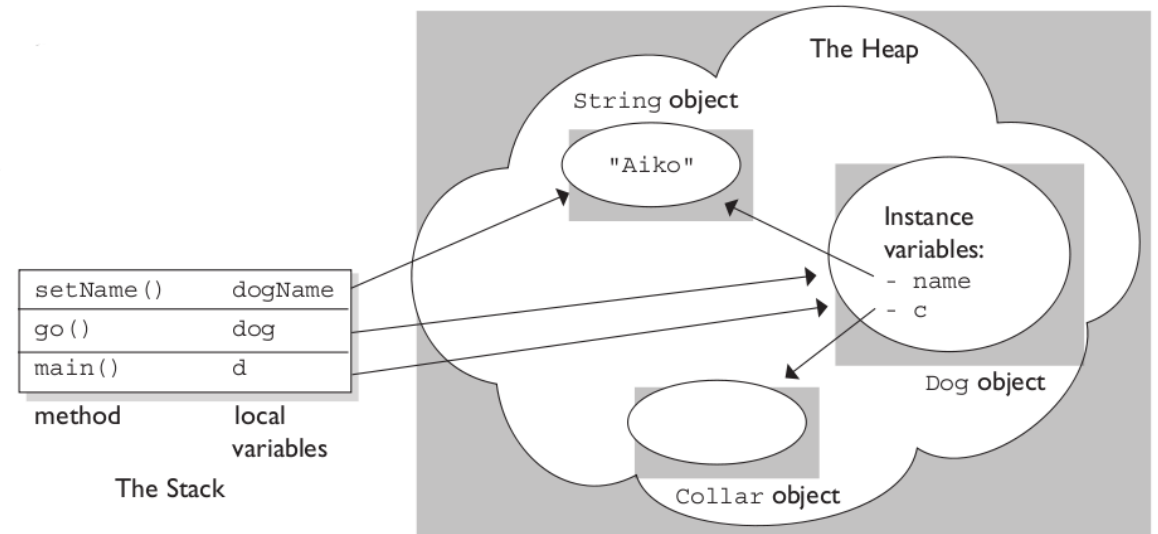
# Topics

---

- 1) Memory management in Java
- 2) How does garbage collection work?
- 3) Requesting garbage collection
- 4) Wrapper classes
- 5) More on String
- 6) StringBuffer / StringBuilder
- 7) StringTokenizer

# Memory Management in Java

- Memory for objects is dynamically allocated
- Remember the Heap?
- When an object is no longer referenced:
  - JRE de-allocates the memory for you
- Process is known as:
- Garbage Collection



# How does garbage collection work?

---

➤ Variables and objects are marked for de-allocation:

➤ When the reference goes out of scope

➤ The reference is assigned to another object

➤ The reference is explicitly set to **null**

➤ Garbage collector calls `Object.finalize( )`

```
public void go(){  
    Employee ref = new Employee();  
}
```

```
public void go(){  
    Employee ref = new Employee();  
    ref = new Employee();  
}
```

```
public void go(){  
    Employee ref = new Employee();  
    ref = null;  
}
```

# Requesting garbage collection

---

- You cannot force garbage collection
- JRE collects garbage on its own time
- You can only request garbage collection
- JRE will collect all objects marked for de-allocation

# Requesting Garbage Collection

```
System.gc();
```

OR

```
Runtime runtime = Runtime.getRuntime();  
runtime.gc();
```

Could you please  
free up some  
memory for me at  
your earliest  
convenience? I  
mean, if you can  
get around to it.



# Wrapper Classes

---

- Let you treat primitive data types as objects
- Each data type has a Wrapper class

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



# Using Wrapper Classes

---

➤ Simply pass a primitive value into the Wrapper constructor

➤ Primitive to Wrapper is called “wrapping”

➤ Wrapper to primitive is called “unwrapping”

```
int number = 5;
```

```
Integer integer = new Integer(number);
```

```
number = integer.intValue();
```

➤ Since Java 5, the JRE can perform “auto-boxing”

```
int number = 5;
```

➤ Wrappers can be directly assigned to primitive variables

```
Integer integer = number;
```

➤ And vice versa

```
number = integer;
```

# More on String

---

- Strings are immutable objects
- Once created they cannot be changed
- Compare Strings using the equals( ) method

```
String str1 = "String";
String str2 = "String";
if(str1.equals(str2)){
    System.out.println("Equal");
}else{
    System.out.println("Not equal");
}
```

- Using == will compare reference variable memory locations
- First output is “Equal”
- Second output is “Not equal”

```
String str1 = "String";
String str2 = "String";
if(str1 == str2){
    System.out.println("Equal");
}else{
    System.out.println("Not equal");
}
```

# More String Methods

---

**toUpperCase()** -Converts all the characters of a string to upper case.

**toLowerCase()**-Converts all the characters of a string to lower case

**charAt(int index)** -This returns the indexed character of a string, where the index of the initial character is 0

**concat(String s)** -This returns a new string consisting which has the old string + s

**equals(String s)** -Checks if two strings are equal

**equalsIgnoreCase(String s)** -This is like equals(), but it ignores the case(Ex: 'Hello' and 'hello' are equal)

**length( )** -Returns the number of characters in the current string.

**replace(char old, char new)** -This returns a new string, generated by replacing every occurrence of old with new.

**trim()** -Returns the string that results from removing white space characters from the beginning and ending of the current string.

# StringBuffer / StringBuilder

---

➤ Allows dynamic modification of Strings

➤ Easy to create

➤ Has several String manipulation methods

➤ append

➤ insert

➤ deleteCharAt

➤ setCharAt

➤ Substring

➤ StringBuffer is a **synchronized** version of StringBuilder

```
StringBuffer str1 = new StringBuffer("String1");  
StringBuilder str2 = new StringBuilder("String2");
```

```
str1.append(" more text");  
str2.deleteCharAt(6);
```

# StringTokenizer

---

- Parses a String
- Splits into tokens based on delimiter
- Second parameter is the delimiter
- Delimiter can be any character, or:
  - Space
  - \t (tab)
  - \n (new line)
  - \r (carriage-return)
  - \f (form-feed)

```
String str = "Mike:Joe:Smith";  
  
StringTokenizer tokenizer = new StringTokenizer(str, ":");  
  
while(tokenizer.hasMoreTokens()){  
    System.out.println(tokenizer.nextToken());  
}
```



"Mike"

"Joe"

"Smith"

# Review

---

- 1) Memory management in Java
- 2) How does garbage collection work?
- 3) Requesting garbage collection
- 4) Wrapper classes
- 5) More on String
- 6) StringBuffer / StringBuilder
- 7) StringTokenizer

# Assignment

---

- Create a `StringBuilder` object. Use at least three methods to manipulate the `String`.
- Create a new `String` with delimited tokens, such as “pickles:ketchup:mustard:onion” and use `StringTokenizer` to parse out and print each token.
- Create two `String` objects with number values (i.e. “20”). Write a method that adds the two.
- Request garbage collection in your method.
- Create a `Runtime` object and note at least three methods. Imagine how you would use them.

# Data Structures

---





# Topics

---

- 1) Collections
- 2) Collections Hierarchy
- 3) Collection Interface
- 4) List
- 5) Set
- 6) Queue
- 7) Collection classes
- 8) Generics
- 9) Non-Collection Data Structures
- 10) Hashtable
- 11) HashMap

# Collections

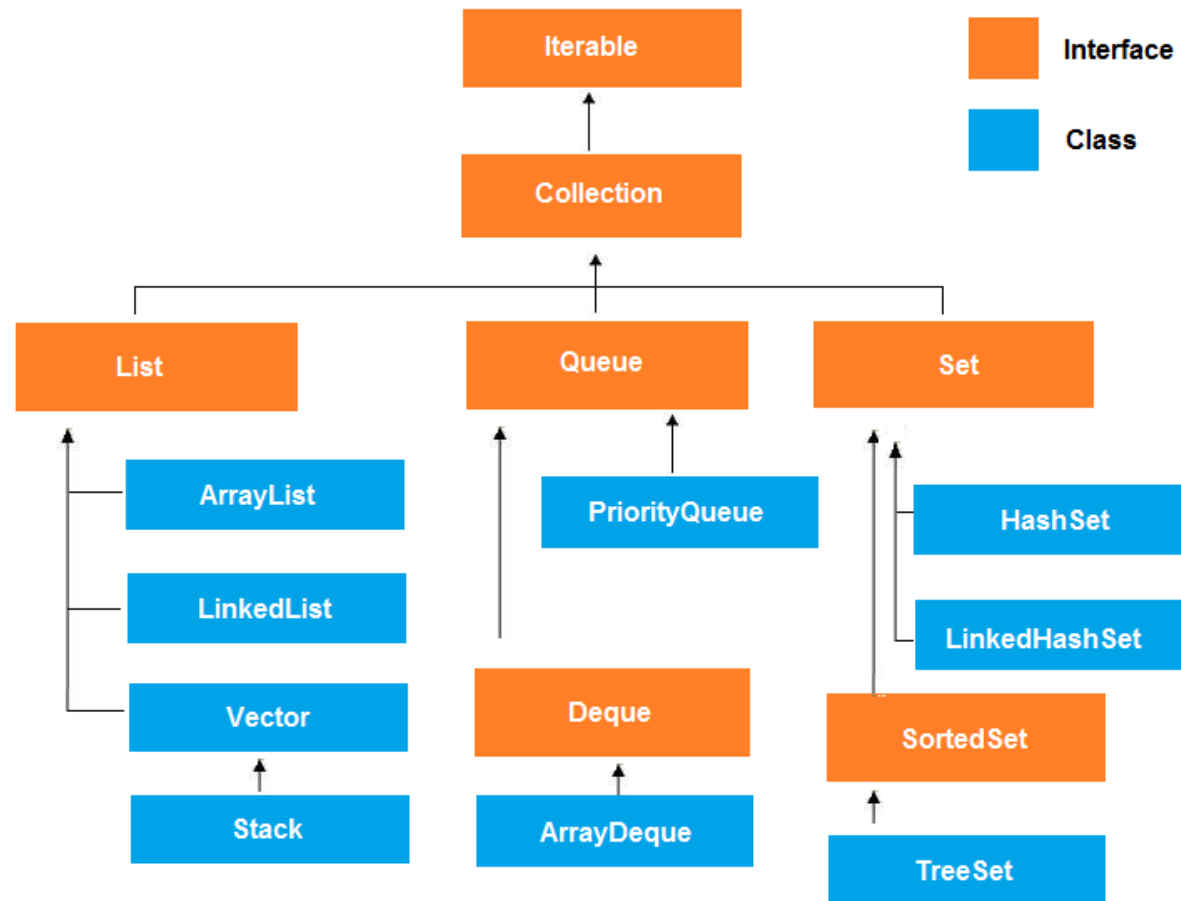
---

- A collection is an object that groups multiple elements into a single unit
- Used to store, retrieve, transform, and manipulate data
- Reduces programming effort by providing useful data structures and algorithms
- Increases program speed and quality
- Collections “collect” things



# Collections Hierarchy

---



# Collection Interface

---

- `add(Object obj)`
- `addAll(Collection c)`
- `clear()`
- `contains(Object obj)`
- `equals (Object obj)`
- `isEmpty()`
- `iterator()`
- `remove(Object obj)`
- `removeAll(Object obj)`
- `size()`

# List Interface

---

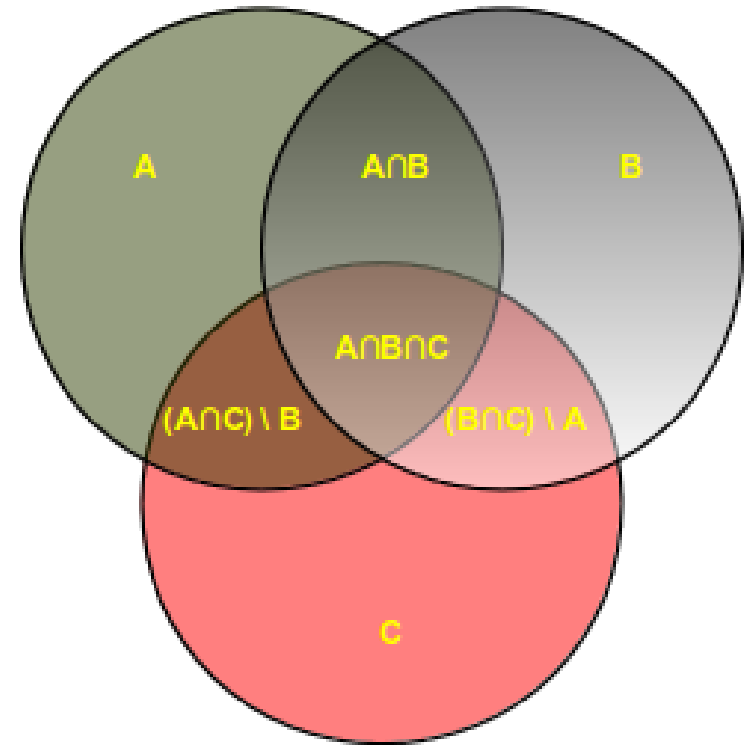
- Elements can be inserted or accessed by their position in the list
- Like array, List uses a zero-based index
- May contain duplicate elements
- Methods include:
  - add (int index, Object obj)
  - get (int index)
  - remove (int index)



# Set Interface

---

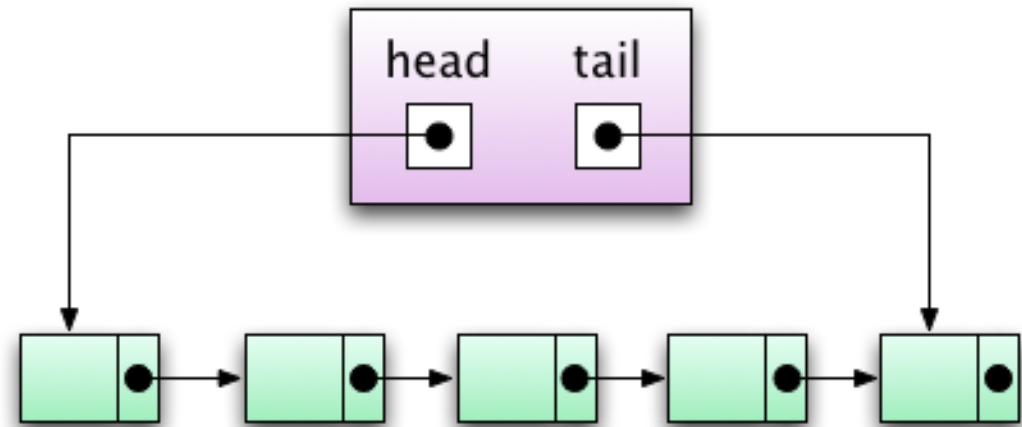
- The Set interface defines a collection of distinct elements
- Set does NOT allow duplicate elements
- Elements are accessed by iterating over the whole set
- Methods include:
  - `add(Object obj)`
  - `clear()`
  - `remove(Object obj)`
  - `size()`
  - `toArray()`



# Queue Interface

---

- Places objects on a “waiting list”
- Useful for storing objects prior to processing
- Elements are added to the tail of the queue
- Elements can be popped off the head of the queue
- Methods include:
  - `add(Object obj)`
  - `element()`
  - `peek()`
  - `poll()`
  - `remove()`



# Common Concrete Collection Classes

---

- ArrayList – Resizable array; allows fast retrieval by index
- LinkedList – Doubly-linked list; each object is a node with reference to nearby nodes
- HashSet – Maintains a set of objects; each assigned a hash code
- TreeSet – An ordered set of elements
- PriorityQueue – Queue ordered by priority

```
List<Employee> al = new ArrayList<Employee>();  
List<Employee> ll = new LinkedList<Employee>();  
Set<Employee> hs = new HashSet<Employee>();  
Set<Employee> ts = new TreeSet<Employee>();  
Queue<Employee> pq = new PriorityQueue<Employee>();
```



# Generics

- Generics enforce the type of object allowed in a Collection
- Uses the Diamond operator < >
- Insert the Class in the Diamond: <Employee>
- Generics provide compile-time safety

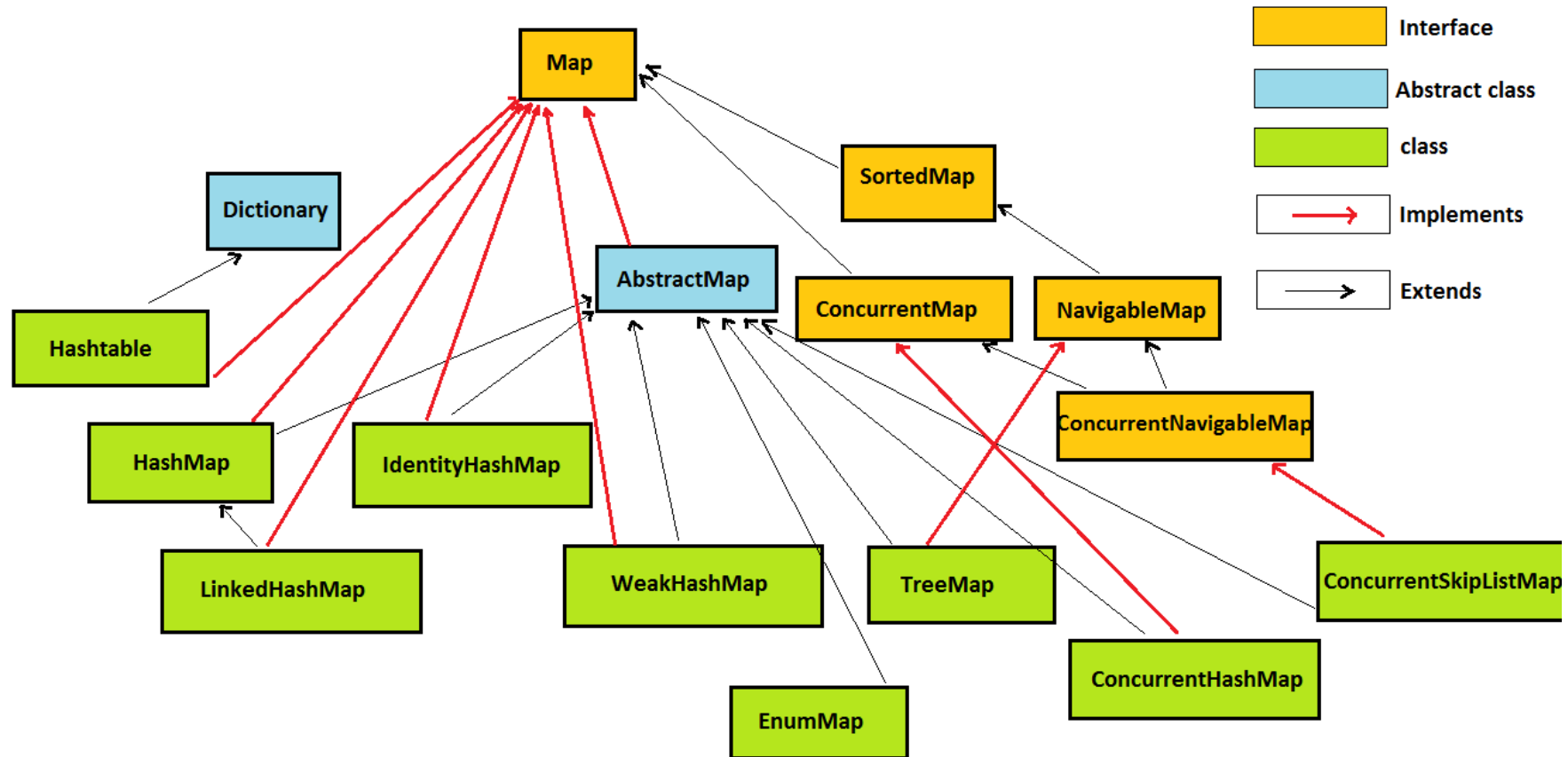
```
List<Monkey> monkeyBarrel = new ArrayList<Monkey>();
```

```
List<Monkey> monkeyBarrel2 = new ArrayList<Lion>();
```

Type mismatch: cannot convert from ArrayList<Lion> to List<Monkey>  
1 quick fix available:  
➤ [Change type of 'monkeyBarrel2' to 'List<Lion>'](#)  
Press 'F2' for focus



# Non-Collection Data Structures



# Hashtable

---

- Hashtable stores key/value pairs
- When using a Hashtable, you must specify:
  - An object that is used as a key
  - The value that you want linked to that key
- Allows random access by key
- Iterate over key set

```
Hashtable<String,Double> balance = new Hashtable<String,Double>();

balance.put("John", new Double(1000.50));
balance.put("Jane", new Double(2560.99));
balance.put("Tom", new Double(5678.00));
balance.put("Todd", new Double(4567.50));

// Random access
System.out.println(balance.get("John"));

Enumeration names;
String str;

// Using key set
names = balance.keys();
while(names.hasMoreElements())
{
    str = (String) names.nextElement();
    System.out.println(str + ":" + balance.get(str));
}
```

# HashMap

---

- HashMap stores key/value pairs
- When using a HashMap, you must specify:
  - An object that is used as a key
  - The value that you want linked to that key
- Allows random access by key
- Iterate over key set and get values

```
HashMap<String,Double> balance = new HashMap<String,Double>();

balance.put("John", new Double(1000.50));
balance.put("Jane", new Double(2560.99));
balance.put("Tom", new Double(5678.00));
balance.put("Todd", new Double(4567.50));

// Random access
System.out.println(balance.get("John"));

// Using key set
Set<String> keys = balance.keySet();
for(String key: keys){
    System.out.println("Value of "+key+" is: "+ balance.get(key));
}
```

# Hashtable VS HashMap

---

## HASHTABLE

- Thread-safe, **synchronized**
- Does not allow **null** keys and **null** values
- Uses Enumeration to iterate key set
- Legacy class

## HASHMAP

- Not thread-safe
- Allows one **null** key and any number of **null** values
- Uses iterator or for:each loop to iterate over key set
- Better performance

# Difference between HashSet and TreeSet in Java

Property	HashSet	TreeSet
Ordering or Sorting	HashSet doesn't provide any ordering guarantee.	TreeSet provides ordering /sorting guarantee.
Comparison and Duplicate detection	HashSet uses equals() method for comparison.	TreeSet uses compareTo() method for comparison
Underlying data structure	HashSet is backed by hash table	TreeSet is backed by Red-Black Tree.
Null element	HashSet allows one null element	TreeSet doesn't allows null objects.
Implementation	Internally implemented using HashMap	Internally implemented using TreeMap.
Performance	HashSet is faster	TreeSet is slower for most of the general purpose operation e.g. add, remove and search

# Review

---

- 1) Collections
- 2) Collections Hierarchy
- 3) Collection Interface
- 4) List
- 5) Set
- 6) Queue
- 7) Collection classes
- 8) Generics
- 9) Non-Collection Data Structures
- 10) Hashtable
- 11) HashMap

# Assignment

---

- Create an ArrayList and a HashSet. Insert 3 objects into each.
- Iterate over each collection and print each object.
- Review Vector and other collections online.
- Review Comparator and Comparable in your book or online.
- Review java.util.Collections methods (sort, reverseOrder, shuffle, etc.)



# File I/O

---



# Topics

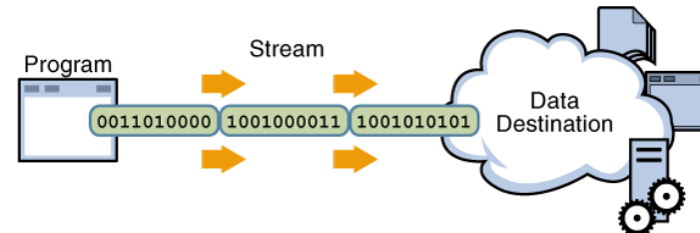
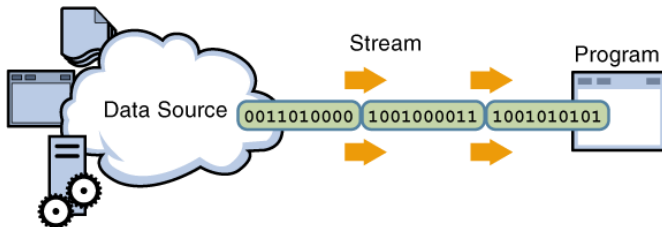
---

- 1) File Input and Output
- 2) Character Streams
- 3) BufferedReader and BufferedWriter
- 4) File Management
- 5) Random Access Files
- 6) Object Serialization
- 7) Java Properties file

# File I/O

- The Java API has many utilities for File I/O
- Classes associated with input inherit from **InputStream**
- Classes associated with output inherit from **OutputStream**
- `FileInputStream` reads raw bytes from a file
- `FileOutputStream` writes raw bytes to a file
- **Always close resources after using them**

```
public void byteStreamMethod() throws Exception{  
    FileInputStream in = null;  
    FileOutputStream out = null;  
    try {  
        in = new FileInputStream("xanadu.txt");  
        out = new FileOutputStream("outagain.txt");  
  
        int c;  
        while ((c = in.read()) != -1) {  
            out.write(c);  
        }  
    } finally {  
        if (in != null) {  
            in.close();  
        }  
        if (out != null) {  
            out.close();  
        }  
    }  
}
```



# Character Streams

---

- FileReader lets you read characters from a file
- FileWriter lets you write characters to a file
- This time actual characters are read/written
- Always close resources after using them

```
public void charStreamMethod() throws Exception{
    FileReader inputStream = null;
    FileWriter outputStream = null;

    try {
        inputStream = new FileReader("xanadu.txt");
        outputStream = new FileWriter("characteroutput.txt");

        int c;
        while ((c = inputStream.read()) != -1) {
            outputStream.write(c);
        }
    } finally {
        if (inputStream != null) {
            inputStream.close();
        }
        if (outputStream != null) {
            outputStream.close();
        }
    }
}
```

# Line-Oriented I/O

---

- Character files can be read line-by-line
- Line is read until it reaches:
  - A new line
  - A carriage feed
- Wrap FileReader into BufferedReader
- Wrap FileWriter in PrintWriter
- Always close resources after using them

```
public void lineStreamMethod() throws Exception{
    BufferedReader inputStream = null;
    PrintWriter outputStream = null;

    try {
        inputStream = new BufferedReader(new FileReader("xanadu.txt"));
        outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));

        String l;
        while ((l = inputStream.readLine()) != null) {
            outputStream.println(l);
        }
    } finally {
        if (inputStream != null) {
            inputStream.close();
        }
        if (outputStream != null) {
            outputStream.close();
        }
    }
}
```

# Buffered Reader and Writer

---

- For faster performance:
- Use Buffered Reader and Writer
- Wrap FileReader into BufferedReader
- Wrap FileWriter in BufferedWriter
- Always close resources after using them

```
public void lineStreamMethod() throws Exception{
    BufferedReader inputStream = null;
    BufferedWriter outputStream = null;

    try {
        inputStream = new BufferedReader(new FileReader("xanadu.txt"));
        outputStream = new BufferedWriter(new FileWriter("characteroutput.txt"));

        String l;
        while ((l = inputStream.readLine()) != null) {
            outputStream.write(l);
        }
    } finally {
        if (inputStream != null) {
            inputStream.close();
        }
        if (outputStream != null) {
            outputStream.close();
        }
    }
}
```

# File Management

---

- Where are the files? What about folders?
- The File Object can help work with files
- Methods for making folders
- Listing directory contents
- Finding absolute path
- Deleting files

```
public void manageFiles() throws Exception{

    // Make a File object that represents an existing file
    File file = new File("myFile.txt");

    // Make a directory
    File dir = new File("Folder");
    dir.mkdir();

    // List contents of a directory
    if(dir.isDirectory()){
        String[] dirContents = dir.list();
        for (int i = 0; i < dirContents.length; i++) {
            System.out.println(dirContents[i]);
        }
    }

    // Get absolute path of a file or directory
    System.out.println(dir.getAbsolutePath());

    // Delete a file - return true if successful
    boolean isDeleted = file.delete();
}
```

# Random Access Files

---

➤ Random access files permit access to a file's contents:

➤ Non-sequential order

➤ Random order

➤ To access a file randomly:

➤ Open the file

➤ Seek a particular location

➤ Read from or write to that file

➤ A file can be Read-Only “r”

➤ Or Read-Write “rw”

➤ Always close resources after using them

```
public void accessAtRandom() throws Exception{

    RandomAccessFile rf = new RandomAccessFile("rtest.dat", "rw");
    for(int i= 0; i < 10; i++)
        rf.writeDouble(i*1.414);
    rf.close();

    //List the file
    rf = new RandomAccessFile("rtest.dat", "r");
    for(int i = 0; i < 10; i++)
        System.out.println("Value " + i + ": " +rf.readDouble());
    rf.close();

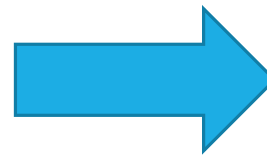
    rf = new RandomAccessFile("rtest.dat", "rw");
    rf.seek(5*8); //Change the 5th position. double is 8 bytes
    rf.writeDouble(47.0001);
    rf.close();
    //List the file and check the 5th position is changed
    rf = new RandomAccessFile("rtest.dat", "r");
    for(int i = 0; i < 10; i++)
        System.out.println("Value " + i + ": " + rf.readDouble());
    rf.close();
}
```



# Serialization

---

- Serialization is the process of writing the state of an object to a byte stream
- Serializable
  - Any object which is
    - Passed across the network
    - Saved to a DAT file
  - Object must **implement** Serializable interface.
  - The Serializable interface defines no members
  - It is simply used to “mark” that class may be serialized.
    - Serializable is a “marker” interface
  - If a class is serializable, all of its subclasses are also serializable.



# Serializable Object

---

- Serialization transforms variables into bytes
- Any **transient** variables will not be serialized

```
public class Employee implements Serializable{

    private static final long serialVersionUID = -2301264712036302390L;
    private long employeeId;
    private String name;
    private String address;
    private transient String jobTitle;

    public String getName() {
        return name;
    }

    public long getEmployeeId() {
        return employeeId;
    }

    public void setEmployeeId(long employeeId) {
        this.employeeId = employeeId;
    }
}
```

# Serializing/Deserializing an Object

---

## ➤Serialization

- Wrap a FileOutputStream in an ObjectOutputStream
- Write the Object
- Always close resources after using them

## ➤Deserialization

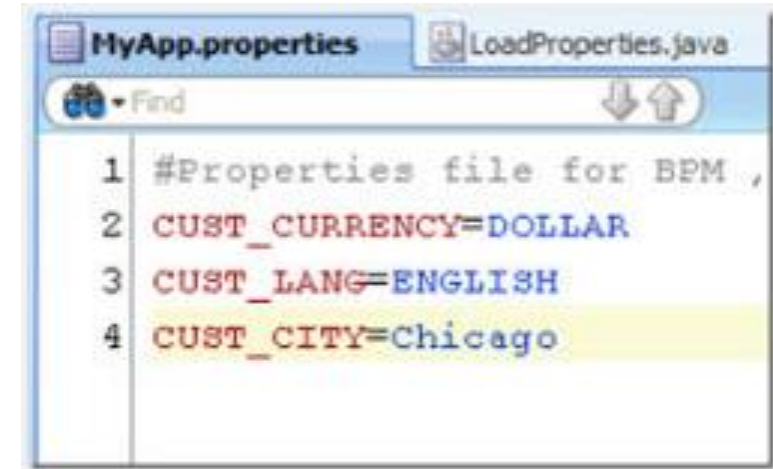
- Wrap a FileInputStream in an ObjectInputStream
- Read the Object
- Cast the Object back into its original type
- Always close resources after using them

```
public void serializeEmployee(Employee emp){  
    try {  
        FileOutputStream fos = new FileOutputStream("employees.ser");  
        ObjectOutputStream oos = new ObjectOutputStream(fos);  
        oos.writeObject(emp);  
        oos.close();  
    }catch(Exception e){  
        e.printStackTrace();  
    }  
}
```

```
public void deserializeEmployee(Employee emp){  
    try {  
        FileInputStream fis = new FileInputStream("employees.ser");  
        ObjectInputStream ois = new ObjectInputStream(fis);  
        Employee emp1 = (Employee) ois.readObject();  
        Employee emp2 = (Employee) ois.readObject();  
        Employee emp3 = (Employee) ois.readObject();  
        ois.close();  
    }catch(Exception e){  
        e.printStackTrace();  
    }  
}
```

# Properties Files

- Create key value pairs for your applications
- Set method adds a property (key, value)
- Store saves a file with .properties extension
- Load will load a properties file



```
public void createProperties(){
    Properties properties = new Properties();
    try {
        properties.setProperty("server", "Saturn");
        properties.setProperty("environment", "Development");
        properties.setProperty("database", "Sybase");
        properties.setProperty("username", "JTutor");
        properties.store(new FileOutputStream("my.properties"), "New properties added on 01/01/2005");
        properties.load(new FileInputStream("my.properties"));
        System.out.println("Username from properties file is: " + properties.getProperty("username"));
        properties.list(System.out);
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}
```

# Review

---

- 1) File Input and Output
- 2) Character Streams
- 3) BufferedReader and BufferedWriter
- 4) File Management
- 5) Random Access Files
- 6) Object Serialization
- 7) Java Properties file

# Assignment

---

- Consider a text file has the following colon-separated lines:
  - Employee id:First Name>Last Name:Role
- Read the file, and parse(tokenize) the fields using StringTokenizer with delimiter(:)
- While parsing the file, place the information in an object, which should have four instance variables: Employee Id, First Name, Last Name, Role.
- Place these objects in a HashMap with the key as employee id and value as the object
- i.e HashMap should have
  - Key: 01 Value : Object representing first line
  - Key: 02 Value: Object representing second line

# Multithreading

---



# Topics

---

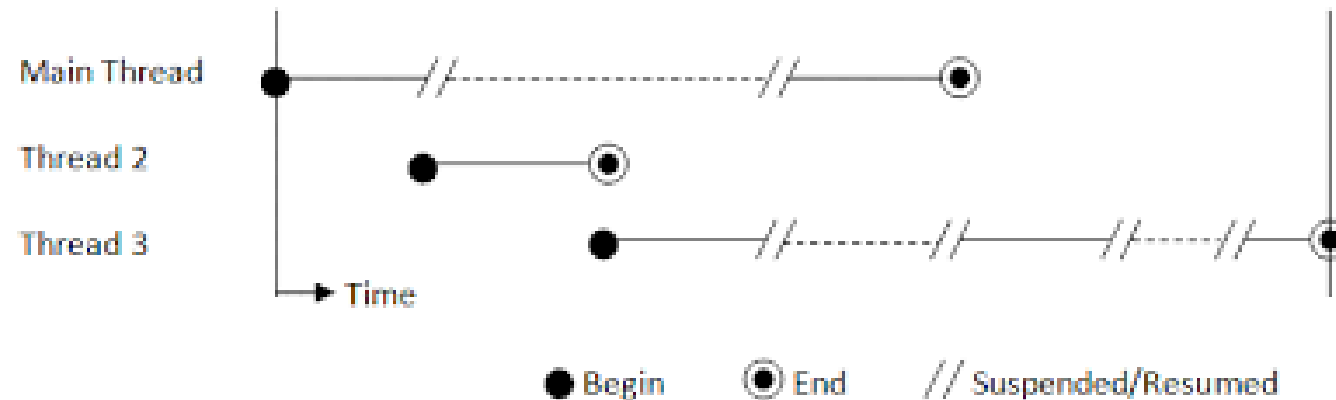
- 1) Multithreading
- 2) Lifecycle of a Thread
- 3) Priorities
- 4) Creating a Thread
- 5) Runnable interface VS Thread Class
- 6) Thread methods
- 7) Synchronization
- 8) Deadlock




# Multithreading

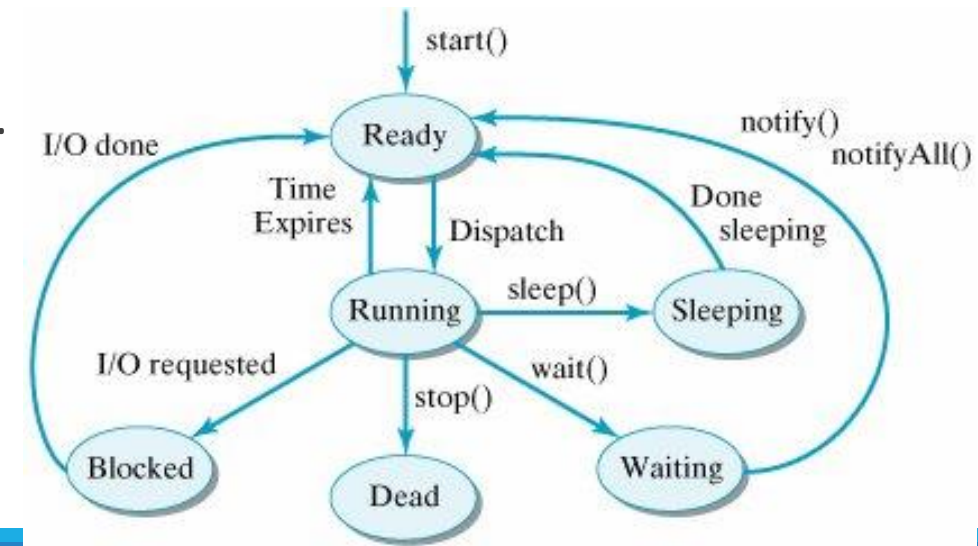
---

- A multithreaded program can have two or more threads running concurrently
- Each thread can have its own task
- Program becomes optimized and runs faster
- OS divides processing time not just with applications but between threads as well



# Lifecycle of a Thread

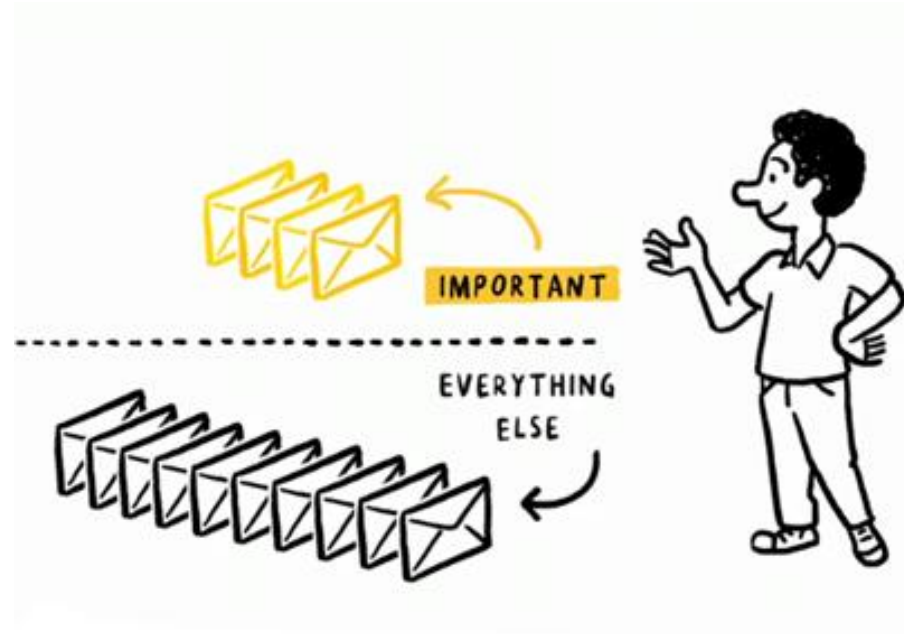
- **New:** A thread's lifecycle begins. Stays in this state until the program starts the thread.
  - **Runnable:** After the thread starts, its in the runnable state.
  - **Waiting:** Waits for other threads to complete the task. Will go back to runnable state only when the other thread signals it to continue executing.
  - **Timed Waiting:** A thread can be in a waiting state for a specified interval time. It goes back to the runnable state after the time has expired.
  - **Terminated:** A thread that completes the task is terminated.
- 
- The diagram shows a state transition for a thread. A blue oval labeled 'Ready' is the central focus. A blue arrow labeled 'start()' points down into the 'Ready' state from above. A blue arrow labeled 'UO done' points from the left into the 'Ready' state. A blue arrow labeled 'notifi' points from the right into the 'Ready' state.



# Priorities

---

- Threads can be configured with a priority number which signifies which order threads are to be run in (Range is 1 – 10).
- MIN\_PRIORITY (typically a 1)
- NORM\_PRIORITY (defaults to 5)
- MAX\_PRIORITY (typically a 10)



# Creating a Thread

---

- Create a class that **implements** the Runnable interface
  - Implement the run() method
  - Pass an object of it into the Thread constructor
  - Call the start() method
- Create a class that **extends** the Thread class
  - Override the run() method
  - Create an object of the class
  - Call the start() method

Thread	Runnable
When you want to override other Thread utility methods	When you want to extend another class

# Implementing Runnable

---

```
class RunnableDemo implements Runnable {
    private Thread t;
    private String threadName;

    RunnableDemo( String name){
        threadName = name;
        System.out.println("Creating " + threadName );
    }
    public void run() {
        System.out.println("Running " + threadName );
        try {
            for(int i = 4; i > 0; i--) {
                System.out.println("Thread: " + threadName + ", " + i);
                // Let the thread sleep for a while.
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread " + threadName + " interrupted.");
        }
        System.out.println("Thread " + threadName + " exiting.");
    }

    public void start ()
    {
        System.out.println("Starting " + threadName );
        if (t == null)
        {
            t = new Thread (this, threadName);
            t.start ();
        }
    }
}

public class TestThread {
    public static void main(String args[]) {

        RunnableDemo R1 = new RunnableDemo( "Thread-1");
        R1.start();

        RunnableDemo R2 = new RunnableDemo( "Thread-2");
        R2.start();
    }
}
```

# Extending Thread

---

```
class ThreadDemo extends Thread {
    private Thread t;
    private String threadName;

    ThreadDemo( String name){
        threadName = name;
        System.out.println("Creating " + threadName );
    }
    public void run() {
        System.out.println("Running " + threadName );
        try {
            for(int i = 4; i > 0; i--) {
                System.out.println("Thread: " + threadName + ", " + i);
                // Let the thread sleep for a while.
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread " + threadName + " interrupted.");
        }
        System.out.println("Thread " + threadName + " exiting.");
    }

    public void start ()
    {
        System.out.println("Starting " + threadName );
        if (t == null)
        {
            t = new Thread (this, threadName);
            t.start ();
        }
    }
}

public class TestThread {
    public static void main(String args[]) {

        ThreadDemo T1 = new ThreadDemo( "Thread-1");
        T1.start();

        ThreadDemo T2 = new ThreadDemo( "Thread-2");
        T2.start();
    }
}
```

# Thread Methods

---

## ➤ Non Static methods

- `public void start()`
- `public void run()`
- `public final void setName(String name)`
- `public final void setPriority(int priority)`
- `public final void setDaemon(boolean on)`
- `public final void join(long millisecond)`
- `public void interrupt()`
- `public final boolean isAlive()`

# Thread Methods

---

- Static method
  - `public static void yield()`
  - `public static void sleep(long milliseconds)`
  - `public static boolean holdsLock(Object x)`
  - `public static Thread currentThread()`
  - `public static void dumpStack()`



# Thread Methods

---

- Object methods that threads can use
  - `notify()`
  - `notifyAll()`
  - `wait()`

# Synchronization

---

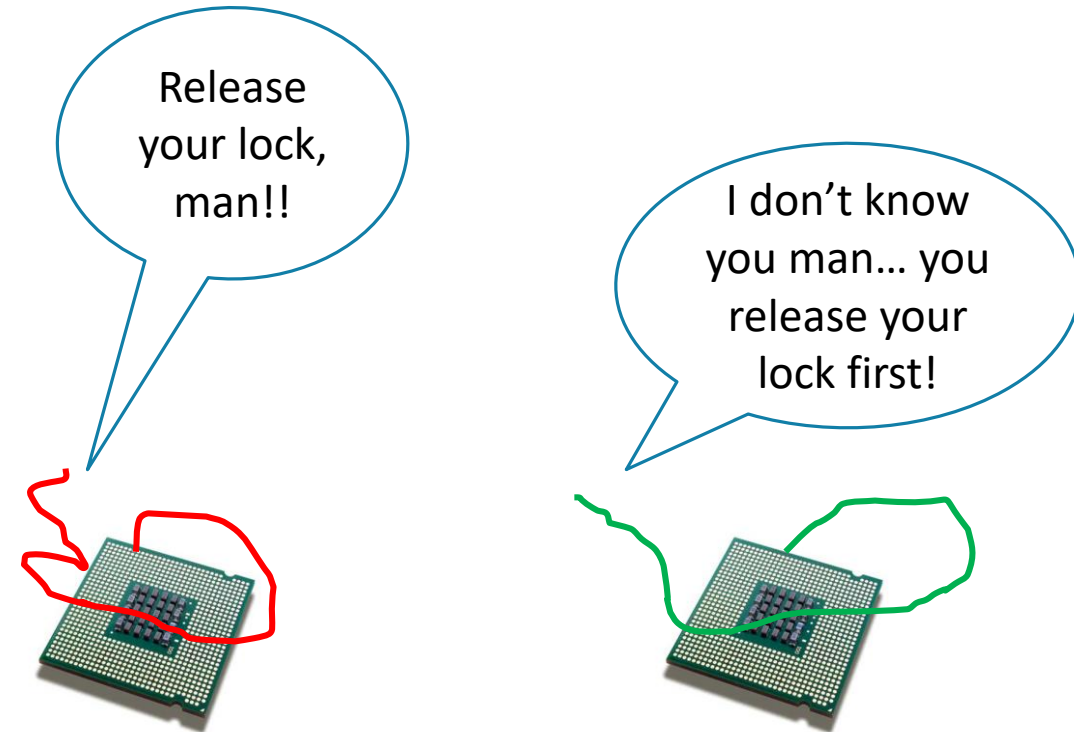
- Problem: When two or more threads are trying to access a method or process, a race condition may occur. If one thread is writing data to a file, the other thread may overwrite that data which may cause issues in the program.
- Solution: synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called *monitors*. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.
- The Java programming language provides a very handy way of creating threads and synchronizing their task by using **synchronized** blocks. You keep shared resources within this block.



# Deadlock

---

- Occurs due to resource locking
- Thread 'A' locks resource '1'
- Thread 'B' locks resource '2'
- Thread 'A' requests resource '2'
- Thread 'B' requests resource '1'
- Neither thread can continue!
- Each holds locks on resources the other needs



# Review

---

- 1) Multithreading
- 2) Lifecycle of a Thread
- 3) Priorities
- 4) Creating a Thread
- 5) Runnable interface VS Thread Class
- 6) Thread methods
- 7) Synchronization
- 8) Deadlock

# Assignment

---

- Write a program that creates a thread using either a Runnable interface or Thread class. Have it call a method which prints out numbers from 1 to 10. Hint: Loop the thread.
- Write a program which demonstrates synchronization between two threads. Print out the duration a thread has to wait till the prior one has completed. Loop the threads 5 times each.
- Review deadlock information in your book or online.

# JDBC

---



# Topics

---

- 1) Intro to JDBC
- 2) Establishing a Connection
- 3) Loading Drivers
- 4) Connection Object
- 5) Statement
- 6) PreparedStatement
- 7) ResultSet
- 8) Transactions
- 9) Commit/Rollback

# Intro to JDBC

---

- Java Database Connectivity
- Standard framework for handling tabular/relational data
- Located in java.sql package





# Establishing a Connection

---

- The first thing to do is establish a connection with the DBMS
- This involves two steps:
  - Loading the driver
  - Making the connection



# Loading Drivers

---

➤ Loading a driver is very simple

➤ One line of code

```
Class.forName ("oracle.jdbc.OracleDriver"); ORACLE
```

```
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

```
Class.forName("com.mysql.jdbc.Driver").newInstance();
```



Microsoft  
SQL Server

➤ You also need the respective JAR file in your build

➤ Drivers can be easily downloaded online

# Connection Object

---

➤ To make the driver connect to the database:

➤ One line of code

```
Connection conn = DriverManager.getConnection(url, user, password);
```

➤ What's my URL?

➤ Every vendor is different

```
String url = "jdbc:oracle:thin:@localhost:1521:xe";
```

```
String url = "jdbc:sqlserver://localhost:1433;databaseName=MyDatabase";
```

```
String url = "jdbc:mysql://localhost:3306/MyDatabase";
```

# Statement

---

- Once connected, you can execute SQL statements
- A Statement object wraps and executes SQL (including DDL, DML, etc.)

```
Statement stmt = conn.createStatement();
```

- For a SELECT statement, the method to use is executeQuery

```
stmt.executeQuery("SELECT * FROM TRAINEES");
```

- To INSERT or UPDATE, use executeUpdate

```
stmt.executeUpdate("UPDATE TRAINEES SET RANK = 'EXPERT' WHERE USER_ID = '5'");
```

- Review other methods of the Statement class

# PreparedStatement

---

- Precompiled SQL statement
- Reduce execution time for repetitious SQL statements
- Parameterized inputs using '?' placeholders
  - Easier to read
- Database reserved characters automatically escaped
  - SQL Injection prevention

```
PreparedStatement updateEmp= conn.prepareStatement("UPDATE student SET fees = ? WHERE name LIKE ?");  
updateEmp.setInt(1, 750);  
updateEmp.setString(2, "John Doe");  
updateEmp.executeUpdate();
```

- pstmt.method(Placeholder position, value to insert)

# ResultSet

---

➤ Object that wraps the rows returned by the query

➤ Return type of a query

```
ResultSet rs = stmt.executeQuery("SELECT * FROM TRAINEES");
```

➤ Loop over the ResultSet to inspect each row

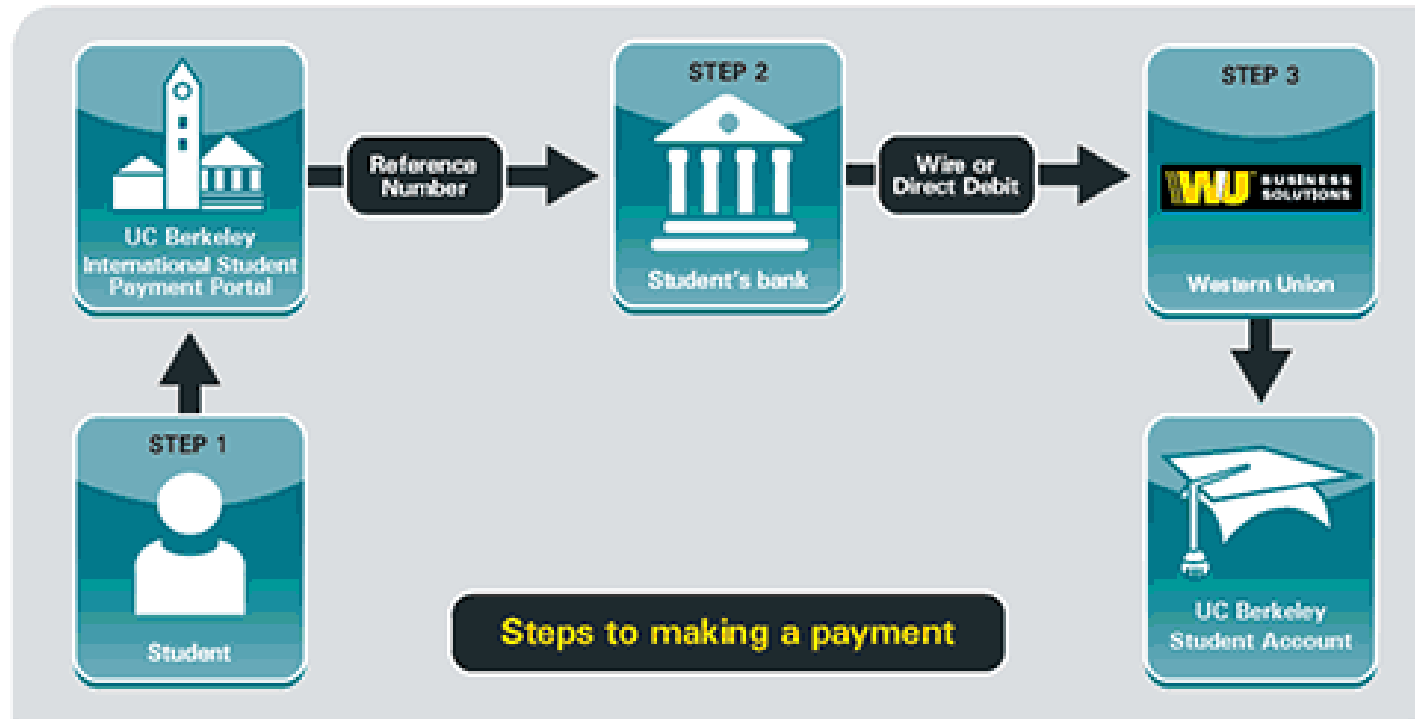
➤ Select each column by “column\_name” or column\_index

➤ At end of loop, ResultSet points to the next row

```
while(rs.next()){  
    System.out.println(  
        rs.getString("name")  
        + "\t" + rs.getInt("age")  
        + "\t" + rs.getFloat("fees")  
        + "\t" + rs.getDate("date_of_hire"));  
}
```

# Transactions

- ACID: Atomic, Consistent, Isolation, Durable



# Transactions

---

- JDBC is in auto-commit mode by default
- Each Statement is immediately committed
- Disable auto-commit mode to execute Statements as a unit
- Re-enable auto-commit mode at the end of transaction

```
conn.setAutoCommit(false);

String deductFromStudent = "UPDATE STUDENT SET BALANCE = BALANCE - ? WHERE STUDENT_ID = ?";
String postPayment = "UPDATE ACCOUNT SET BALANCE = BALANCE + ? WHERE STUDENT_ID = ?";

PreparedStatement pstmt = conn.prepareStatement(deductFromStudent);
pstmt.executeUpdate();

pstmt = conn.prepareStatement(postPayment);
pstmt.executeUpdate();

conn.commit();
conn.setAutoCommit(true);
```



# Commit/Rollback

---

- To make changes permanent, use commit( ) method
- To return to the previous state, use rollback( ) method

```
try{
    conn.setAutoCommit(false);

    String deductFromStudent = "UPDATE STUDENT SET BALANCE = BALANCE - ? WHERE STUDENT_ID = ?";
    String postPayment = "UPDATE ACCOUNT SET BALANCE = BALANCE + ? WHERE STUDENT_ID = ?";

    PreparedStatement pstmt = conn.prepareStatement(deductFromStudent);
    pstmt.executeUpdate();

    pstmt = conn.prepareStatement(postPayment);
    pstmt.executeUpdate();

    conn.commit();
}catch(Exception e){
    conn.rollback();
}finally{
    conn.setAutoCommit(true);
}
```

# Just like Files...

---

- Always close resources after using them
- All JDBC resources have a `close()` method

```
statement.close();  
preparedStatement.close();  
connection.close();
```

# Review

---

- 1) Intro to JDBC
- 2) Establishing a Connection
- 3) Loading Drivers
- 4) Connection Object
- 5) Statement
- 6) PreparedStatement
- 7) ResultSet
- 8) Transactions
- 9) Commit/Rollback

# Assignment

---

- Create a Connection where the database URL, username, and password is retrieved from a Properties file.
- Create a table with at least 3 columns within the Java application.
- Insert test data into the table.
- Query the test data using Statement and PreparedStatement.
- Study CallableStatement for calling stored procedures in your database (PL/SQL, T-SQL, etc.)