15-440
Distributed Systems
Recitation 1

Slides by: Hend Gedawy

& Tamim Jabban

Please open https://pollev.com/hendgedawy084

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Office Hours

Office 1016, Zoom

Sunday, Thursday: 10:00 – 12:00 PM

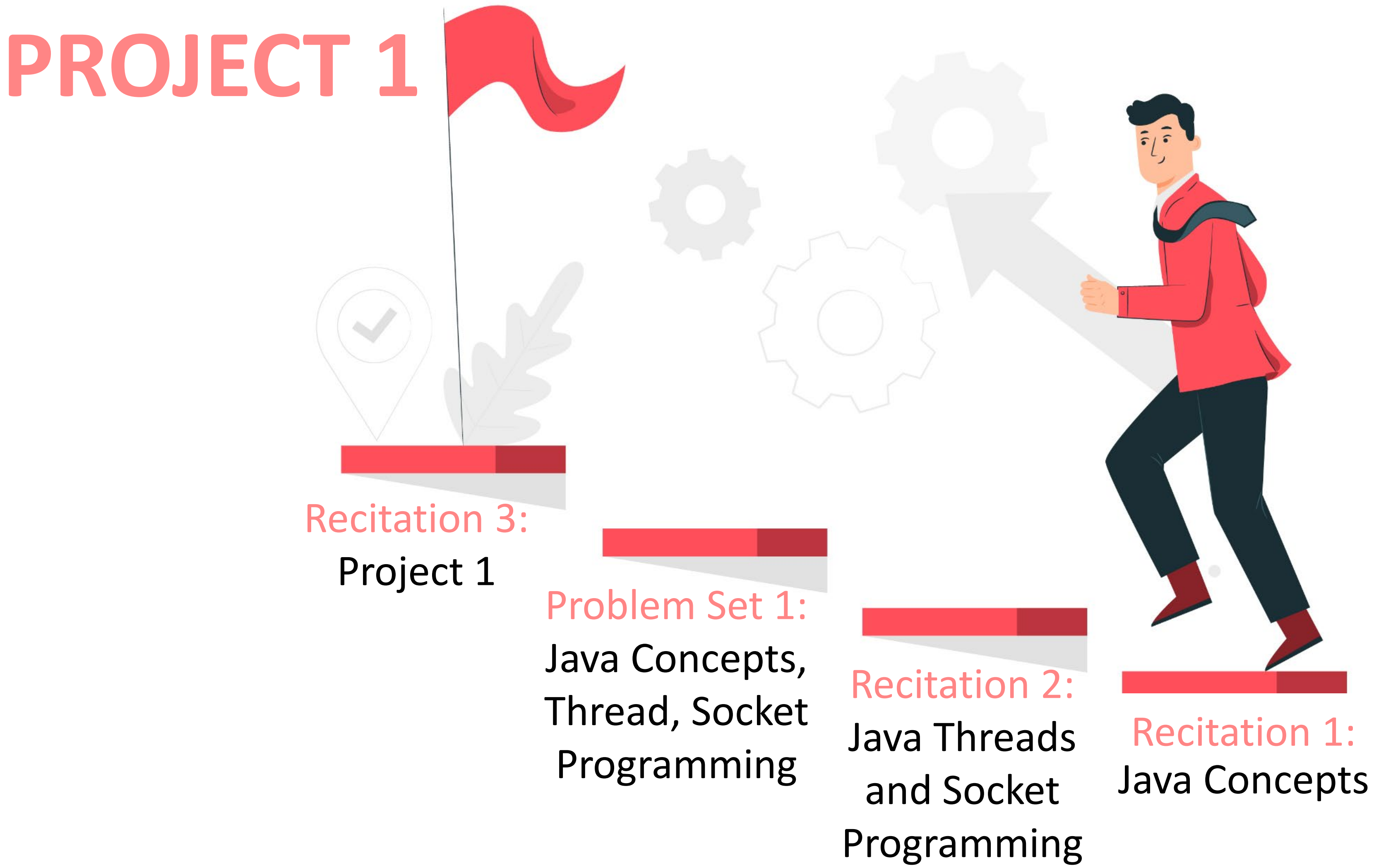
Appointment: send an e-mail

Piazza, Open door policy

Logistics

- PS1 is out on the course website (due on Sep. 3) submit on Gradescope

Big Picture



PROJECT 1

Recitation 3:
Project 1

Problem Set 1:
Java Concepts, Thread, Socket Programming

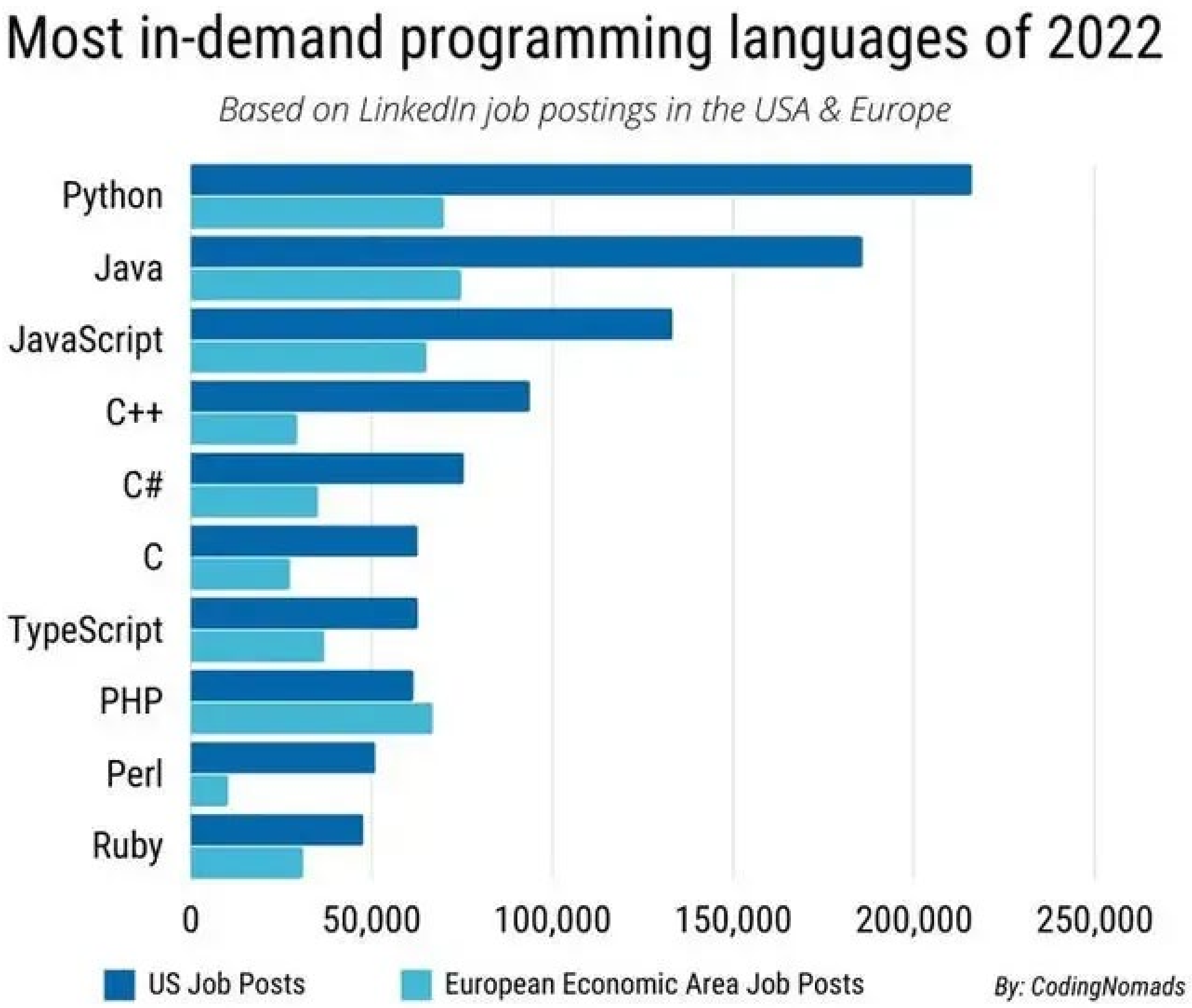
Recitation 2:
Java Threads and Socket Programming

Recitation 1:
Java Concepts

Outline

- **Introduction**
- OOP Structure
- OOP Principles
- More Java Concepts

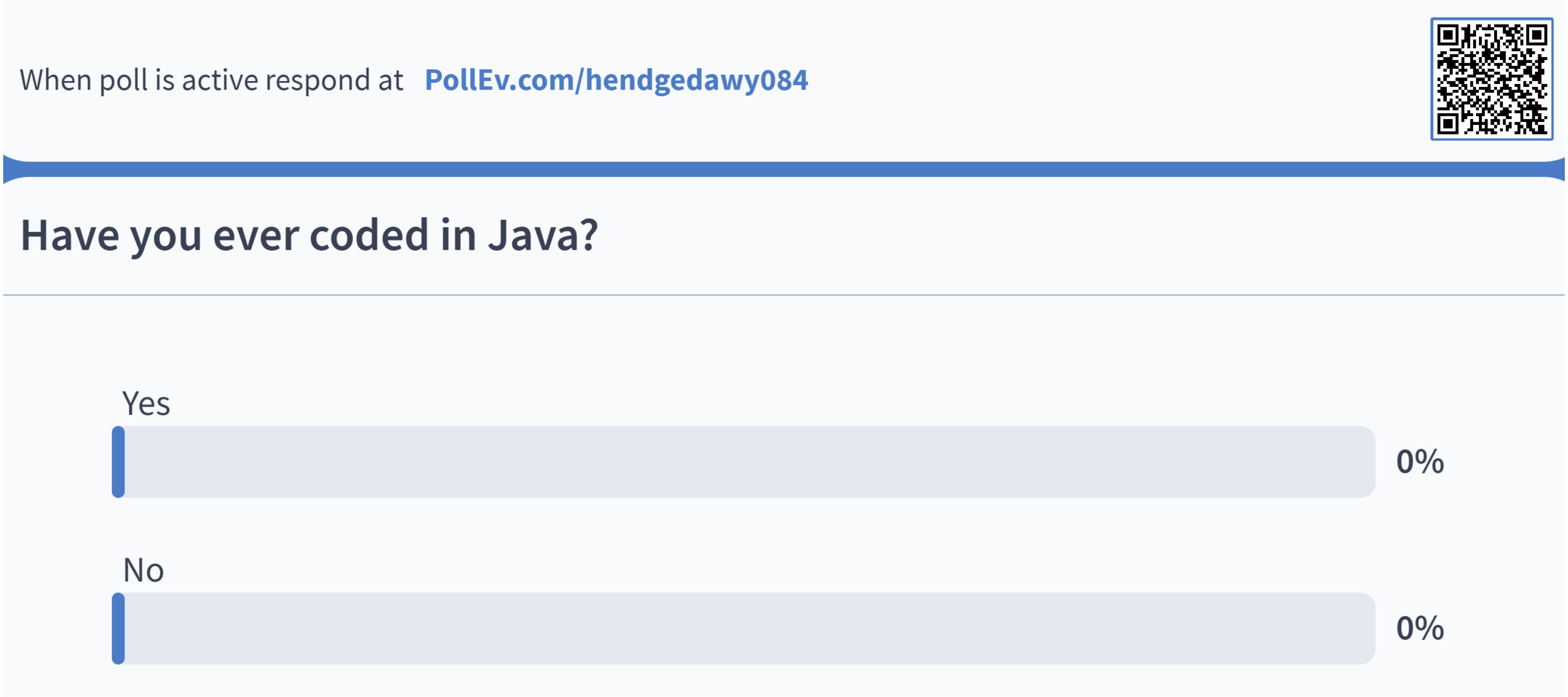
Java Object Oriented Programming (OOP)



Most in-demand programming languages of 2022

Based on LinkedIn job postings in the USA & Europe

By: CodingNomads



When poll is active respond at **PollEv.com/hendgedawy084**

Have you ever coded in Java?

Yes 0%

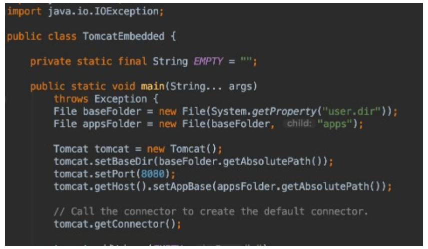
No 0%

Java Introduction

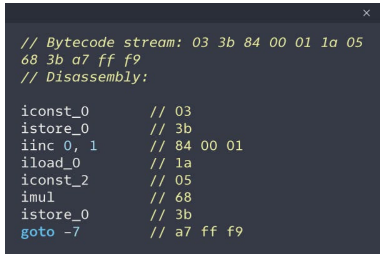
- A class-based, object-oriented programming (OOP) language
- The syntax of Java is similar to C/C++
- Eliminates complex features like pointers and explicit memory allocation and deallocation (garbage collection)

Java Introduction

- Platform-independent *write once run anywhere*
  - Compiler converts source code to bytecode and then the JVM executes the bytecode generated by the compiler
- Javac compiler generate byte code can run on any Java Virtual Machine

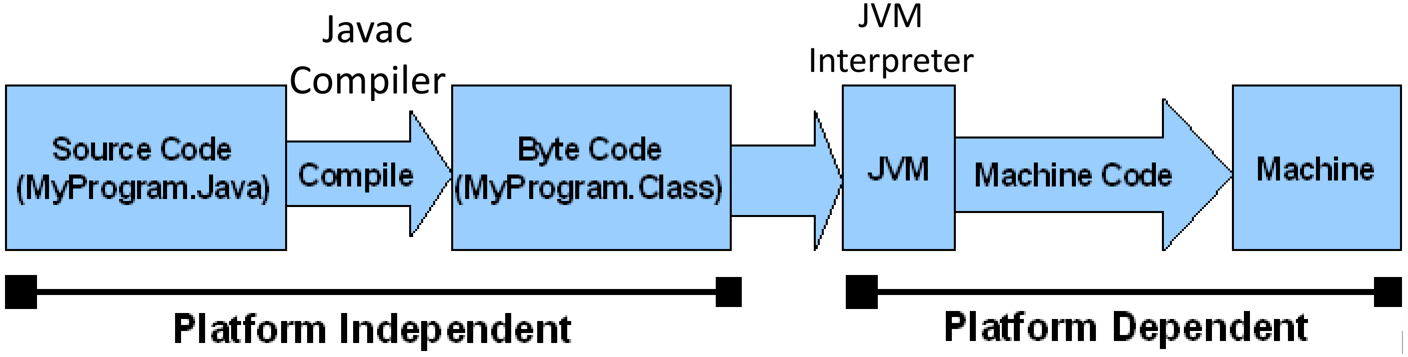


```java
import java.io.IOException;

public class TomcatEmbedded {

    private static final String EMPTY = "";

    public static void main(String... args)
        throws Exception {
        File baseFolder = new File(System.getProperty("user.dir"));
        File appsFolder = new File(baseFolder,  child: "apps");

        Tomcat tomcat = new Tomcat();
        tomcat.setBaseDir(baseFolder.getAbsolutePath());
        tomcat.setPort(8080);
        tomcat.getHost().setAppBase(appsFolder.getAbsolutePath());

        // Call the connector to create the default connector.
        tomcat.getConnector();
```



```
// Bytecode stream: 03 3b 84 00 01 1a 05
68 3b a7 ff f9
// Disassembly:

iconst_0        // 03
istore_0        // 3b
iinc 0, 1       // 84 00 01
iload_0         // 1a
iconst_2        // 05
imul            // 68
istore_0        // 3b
goto -7         // a7 ff f9
```

```
Machine Code
10011101000110100000
01100011010001110110
10000010111101101110
11110110001011011000
10000010011100011011
10010011000111000000
```



Java Introduction: Language Constructs

- Variables
- Datatypes
  - Primitive
    - boolean, char, byte, short, int, long, float, double
  - Non-primitive
    - String, Array, Classes
- Operators
- Flow Control
  - If, switch-case, break, continue
- Loops
  - For, while, for-each loop

- Regular Arrays [] - Immutable (*cannot grow*)
  - Declaring: type var-name[]; OR type[] var-name;
    - E.g.: int ages[]; OR int[] ages;
  - Assigning: var-name = new type [size];
    - E.g.: ages= new int[10];
  - All elements set to their default value (0 or null)
- Dynamic (*resizable*) Arrays: ArrayLists
  - Don’t have to specify the size of the ArrayList at the time of creation
  - Declaring: ArrayList<type> var-name;
    - E.g. : ArrayList<int> ages;
  - Assigning: var-name= new ArrayList<>();
    - E.g.: ages= new new ArrayList<>();
    - Later you can add elements using .add() method
- Strings
- Other classes
- Naming conventions

Outline

- Introduction
- **Java OOP Structure**
  - **Class**
  - **Object**
  - **Attributes**
  - **Methods**
- Java OOP Core Principles
- More Java Concepts

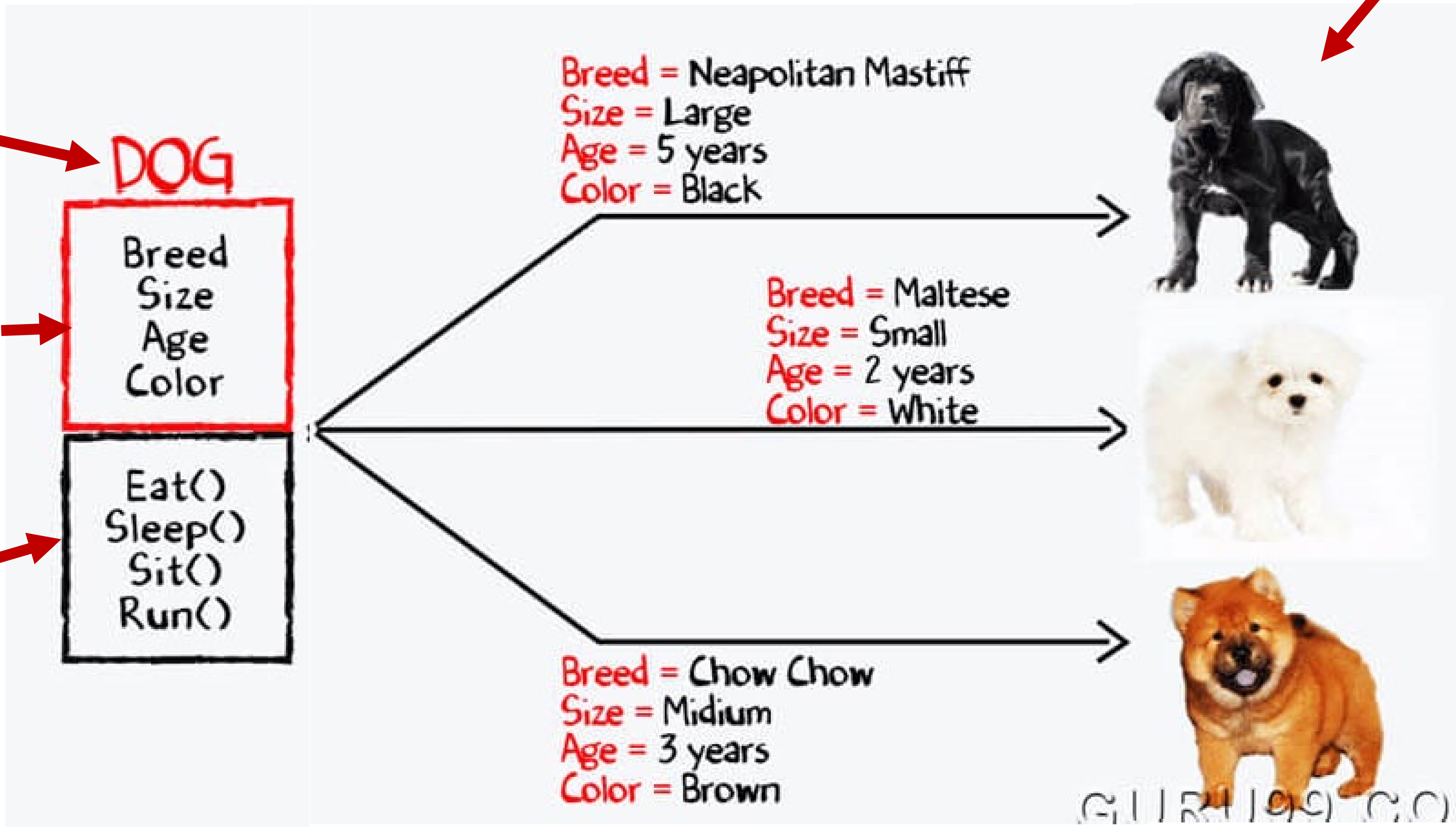
Java OOP: Structure

4) Object

1) Class

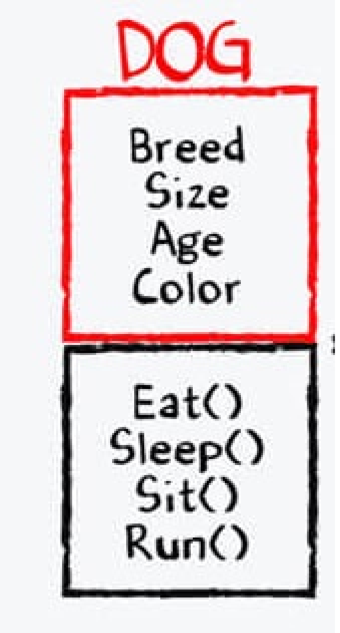
2) Attributes

3) Methods



Java OOP Structure: Class

- A user defined **blueprint or prototype** from which objects are created
- Represents the set of ***properties*** **or** ***methods*** that are common to all objects of one type



Java OOP Structure: Object

- An **Object** consists of
  - *State*: represented by *attributes* of an object
  - *Behavior*: represented by *methods* of an object.
- When an object of a class is created, the class is said to be **instantiated**.
- All the instances (objects of a class) share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object.

Java OOP Structure: Object

How to create an Object of a Class?

Java OOP Structure: Object

- The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory.
- To create an **Dog** **Object**:

```
Dog tuffy = new Dog("tuffy","papillon",5, "white");
```

Constructor

Java OOP Structure: Object Constructors

- A Java constructor is **special method** that is **called when an object is instantiated**
- Constructors take in **zero or more** variables to create an **Object**
- Constructors have the **same name** as the class and have **no return** type
- All classes have **at least one constructor**.
  - If a class does not explicitly declare any, the Java compiler automatically provides a no-argument constructor, also called the default constructor.

```
public class Dog
{
    // Instance Variables
    String name;
    String breed;
    int age;
    String color;

    // Constructor Declaration of Class
    public Dog(String name, String breed,
                  int age, String color)
    {
        this.name = name;
        this.breed = breed;
        this.age = age;
        this.color = color;
    }

    // method 1
```

What is this?

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

         Dog tuffy = new Dog("tuffy",
                        "papillon", 5, "white");
```

```
public class Dog
{
    // Instance Variables
    String name;
    String breed;
    int age;
    String color;
    // Constructor 1
    public Dog(String name, String breed,
             int age, String color)
    {
        this.name = name;
        this.breed = breed;
        this.age = age;
        this.color = color;
    }
    // Constructor 2
    public Dog(String name, String breed)
    {
        this.name = name;
        this.breed = breed;
        this.age = 0;
        this.color = "Black";
    }
```

Constructor overloading is their most useful functionality

More on that Later!

Java OOP Structure:
Object & Class Variables

- Each Dog object has its own `size`, `age`, etc…
    - `size` and `age` are examples of **Object Variables**.
- When an attribute should describe an **entire class** of objects instead of a specific object, we use **Class Variables** (or `static` **Variables**).
- There's only one copy of class variables for the entire class, regardless of how many objects are created from it.
- Called/retrieved using the class name

Java OOP Structure:
Object & Class Variables

```
public class Dog {
    public static final String currentPlanet = “EARTH”;
}

public class Test() {
        public static void main(String[] args) {
            Dog foobar = new Dog();
            String planet = foobar.currentPlanet;
        }
}
```

What’s wrong?

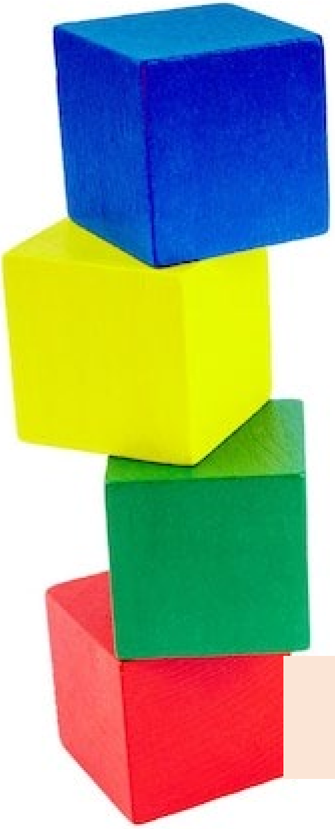
Java OOP Structure:
Object & Class Variables

```
public class Dog {
   public static final String currentPlanet = “EARTH”;
}

   public class Test() {
         public static void main(String[] args) {
               ~~Dog foobar = new Dog();~~
               String planet = Dog.currentPlanet;
         }
}
```

Outline

- Introduction
- Java OOP Structure
  - Class
  - Object
  - Attributes
  - Methods
- **Java OOP Core Principles**
  - **Inheritance**
  - **Encapsulation**
  - **Abstraction**
  - **Polymorphism**
- More Java Concepts

Java OOP: Core Principles



Inheritance

Java OOP Principles: Inheritance

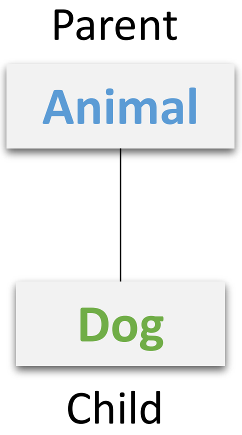
- Enables one class to inherit ***methods*** (*behavior)* and ***attributes*** from another class.
  - It `extends` the functionality of that other class

```java
class Animal{
    void eat(){ System.out.println("eating..."); }
}
```

Superclass

```java
class Dog extends Animal{
    void bark(){ System.out.println("barking..."); }
}
```

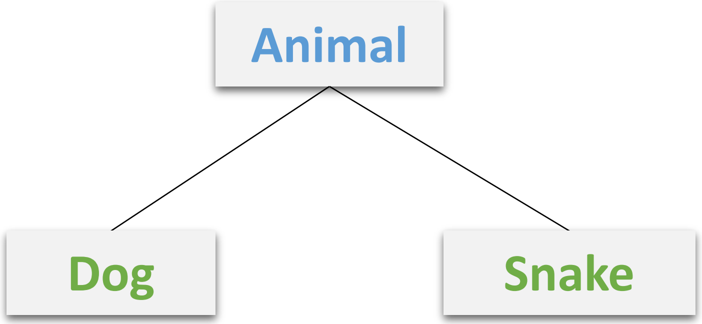
Subclass



```java
class TestInheritance{
    public static void main(String args[]){
    Dog d = new Dog();
    d.bark();
    d.eat();
}
```

Java OOP Principles: Inheritance

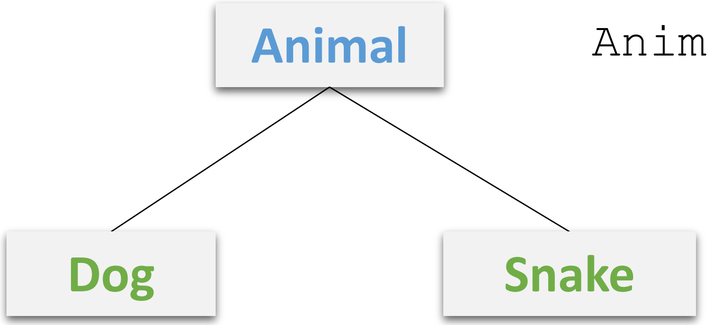
- This introduces **subclasses** and **superclasses**.
- A class that *inherits* from another class is called a subclass:
    - **Dog** *inherits* from **Animal**, and therefore **Dog** is a **subclass**.
- The class that is *inherited* is called a superclass:
    - **Animal** is *inherited*, and is the **superclass**.



Java OOP Principles: Inheritance

- *Organizes* related classes in a *hierarchy*:
  - This allows *reusability and extensibility of common code*
- Subclasses *inherit all the methods* of the superclass (***excluding constructors and privates***)
- Subclasses can **override** methods from the superclass (*more on this later*)
  - i.e. customize implementation of inherited methods

Java OOP Principles: Inheritance (Casting)



```
Animal a = new Animal();
```

```
Dog d = new Dog();
```

```
Snake s = new Snake();
```

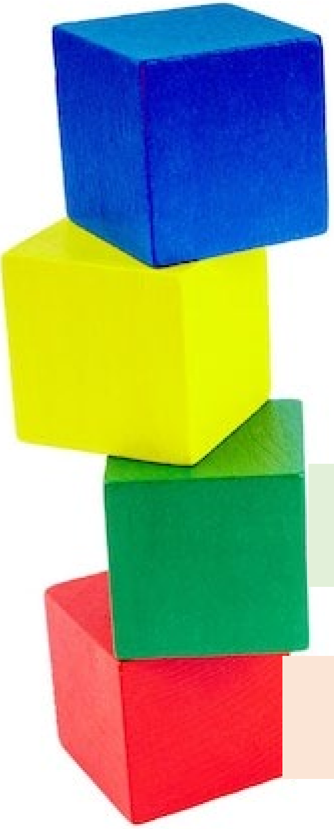
`a = d;` is *Legal* (A Dog is an Animal)

But

`d = a;` is *Illegal* (An Animal isn’t necessarily a Dog)

There are ways to safely UpCast and DownCast:
Beyond the scope of this recitation.
But you can learn more.

Java OOP: Core Principles

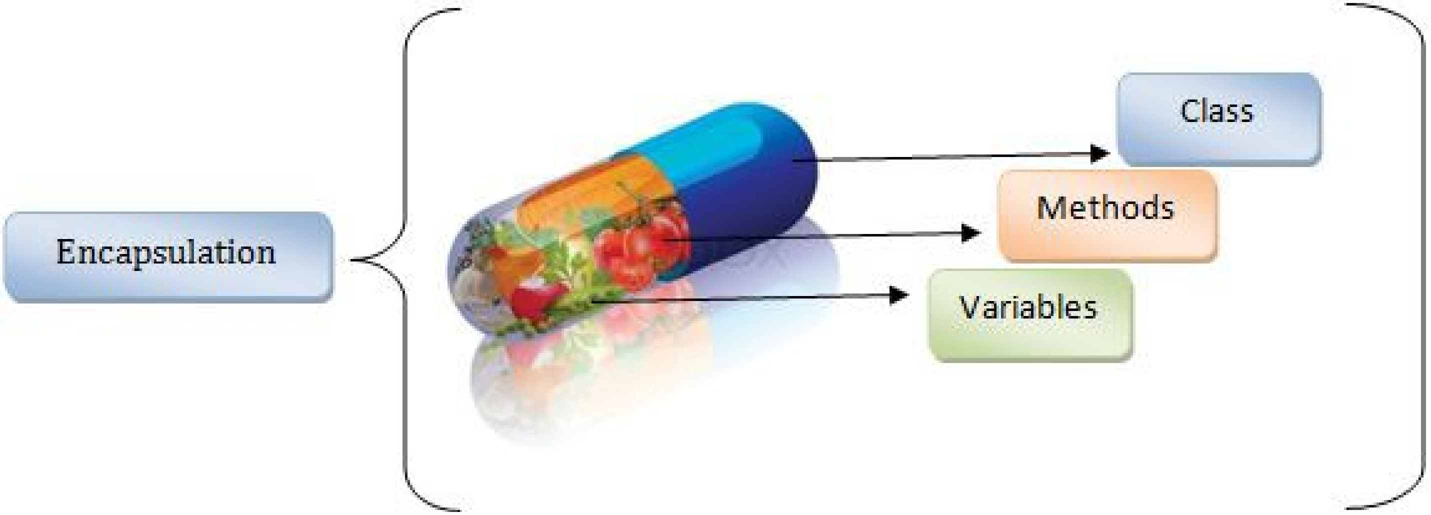


Encapsulation

Inheritance

Java OOP Principles: Encapsulation

Encapsulation is Restricting Access To ….



Java OOP Principles: Encapsulation

Access modifiers describe the accessibility (*scope*) of data like:

- Attributes (Vartiables):

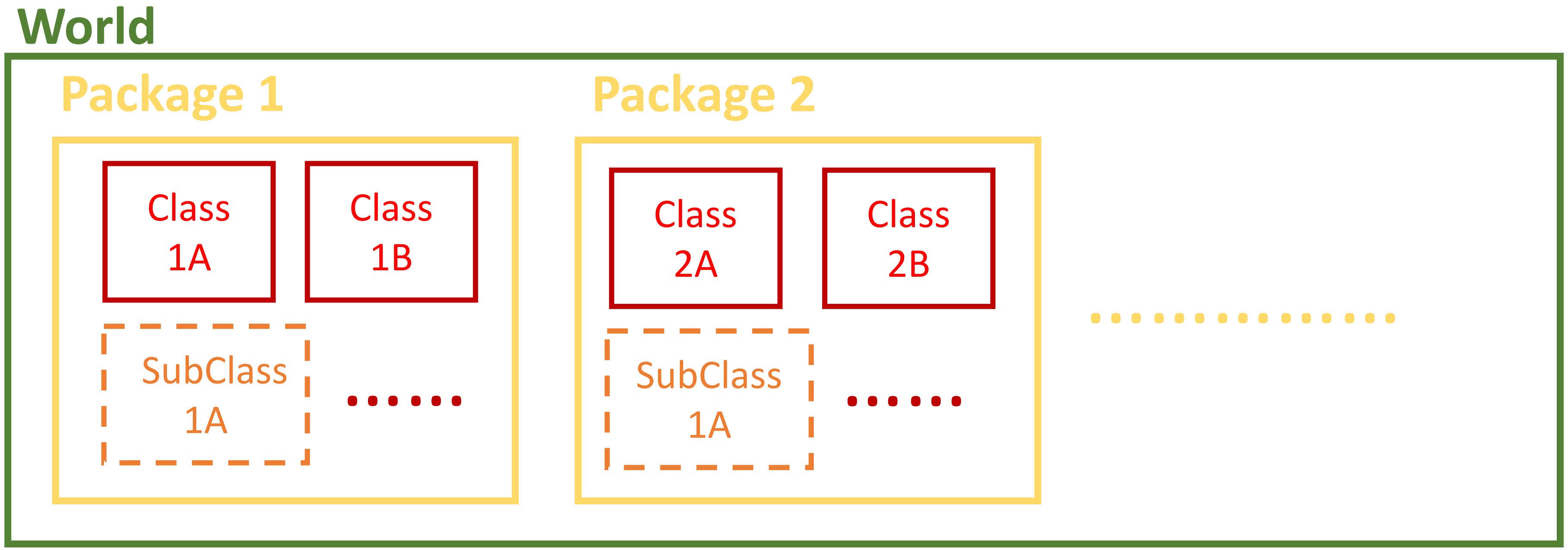
```
public String name;
```

- Methods & Constructors:

```
public String getName() { … }
private Student(String name, int sAge) { … }
```

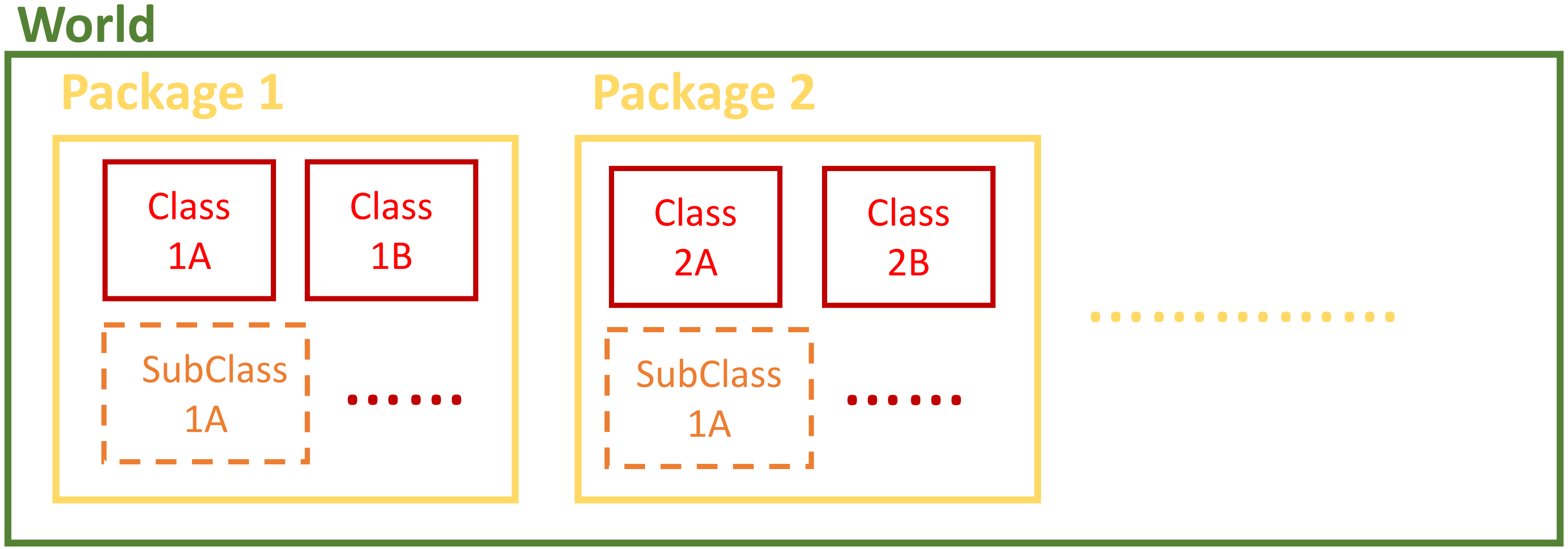
Java OOP Principles: Encapsulation

Accessibility Scope



A **package** is a group of related classes that serve a common purpose.

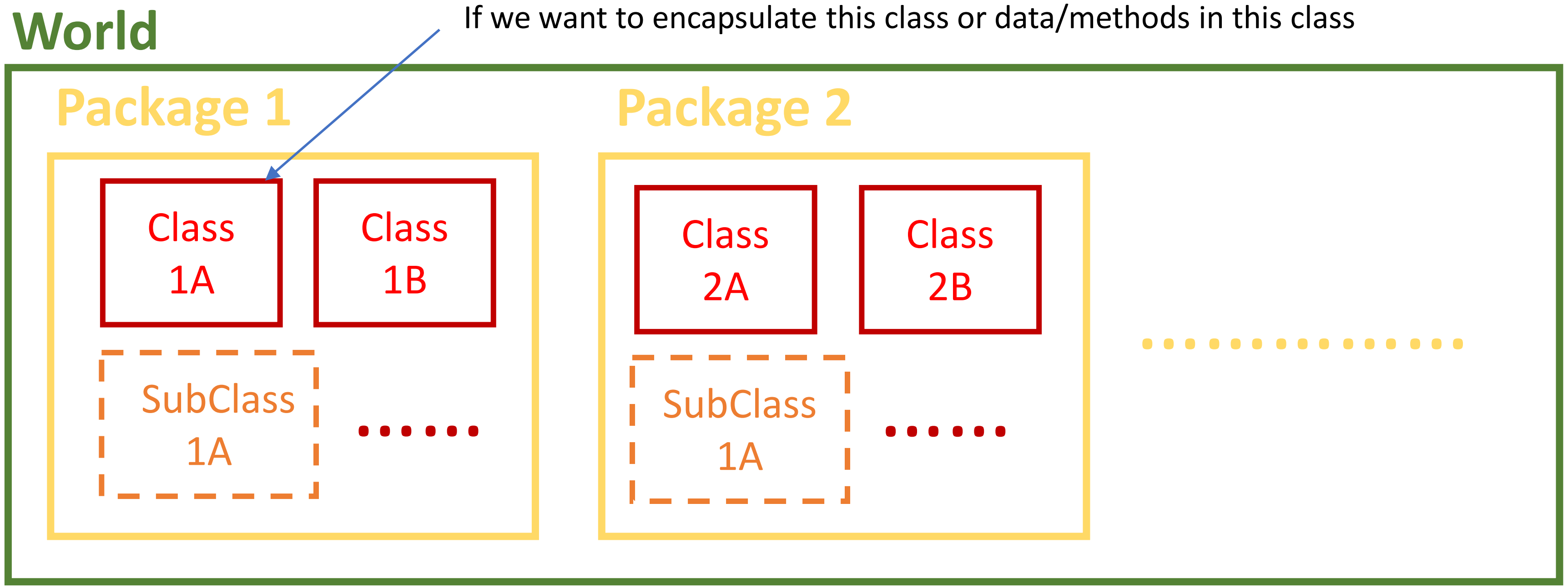
Java OOP Principles: Encapsulation



Access Modifier	Same Class OR Subclass – same package	Same Package	Subclass- different Package	World (Any class, all packages)
Public				
Protected				
Default				
Private				

We Use 4 Different Access Modifiers to Define Accessibility Scope

Java OOP Principles: Encapsulation



Public Access

Access Modifier	Same Class OR Subclass – same package	Same Package	Subclass – different Package	World (Any class, All packages)
Public	Y	Y	Y	Y
Protected				
Default				
Private				

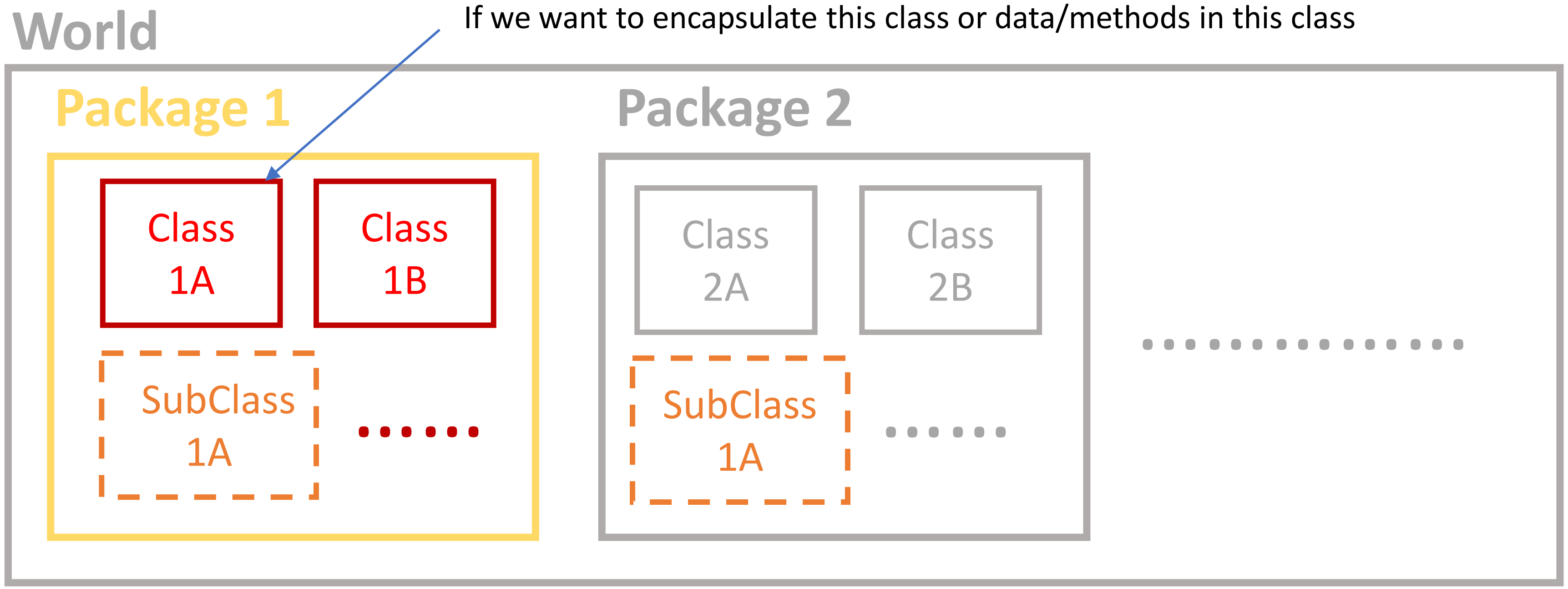
Java OOP Principles: Encapsulation

```
package p1;

class Rec
{
    public void display()
    {
        System.out.println(“Hi!");
    }
}
```

```
package p2;
import p1.*;


class RecNew
{
    public static void main(String args[])
    {
        // Accessing Rec from package p1
        Rec obj = new Rec();

        obj.display();
    }
}
```

Prints “Hi!” → obj.display();

Java OOP Principles: Encapsulation



Protected Access

Access Modifier	Same Class OR Subclass – same package	Same Package	Subclass – Different Package	World (Any class, all packages)
Public	Y	Y	Y	Y
Protected	Y	Y	Y	N
Default				
Private				

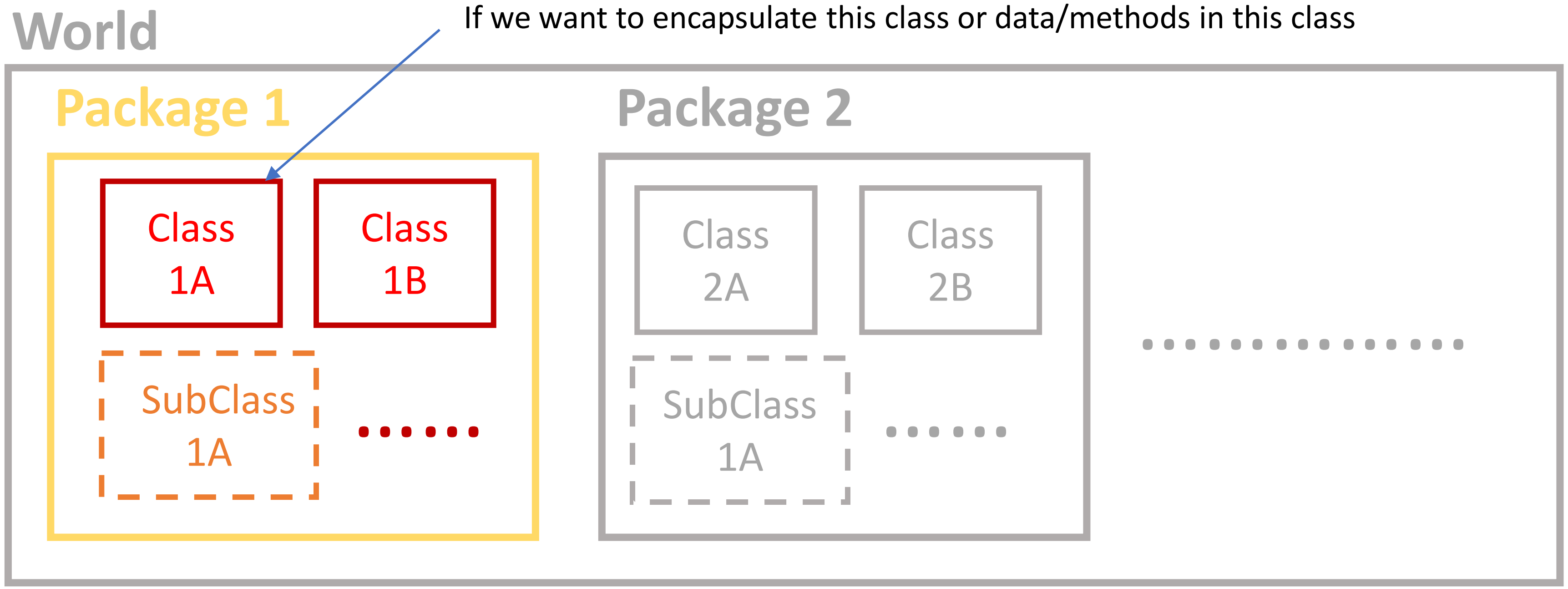
Java OOP Principles: Encapsulation

```
package p1;

class Rec
{
    protected void display()
    {
        System.out.println(“Hi!");
    }
}
```

```
package p2;
import p1.*;

class RecNew extends Rec
{
    public static void main(String args[])
    {
        // Accessing Rec from package p1
        RecNew obj = new RecNew();

        obj.display();
    }
}
```

Prints “Hi!” → obj.display();

Java OOP Principles: Encapsulation



Default Access

Access Modifier	Same Class OR Subclass – same package	Same Package	Subclass – Different Package	World (Any class, all packages)
Public	Y	Y	Y	Y
Protected	Y	Y	Y	N
Default	Y	Y	N	N
Private				

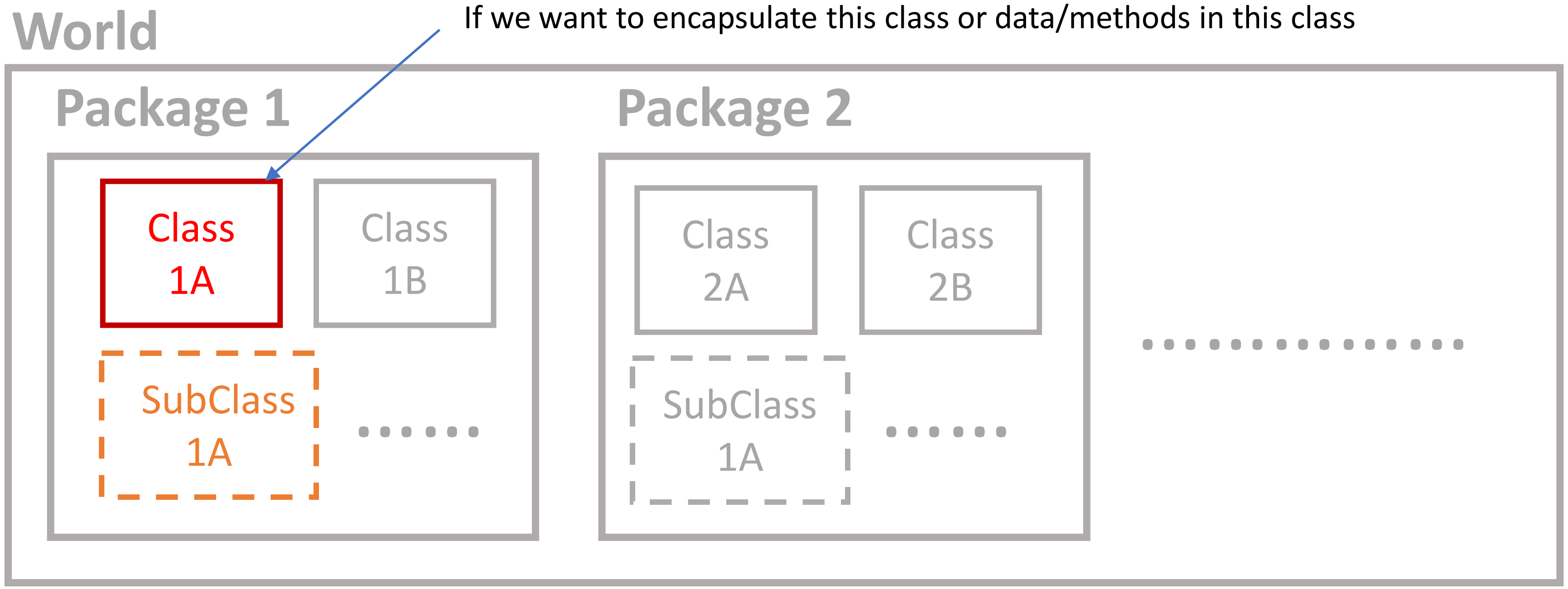
Java OOP Principles: Encapsulation

```java
package p1;

class Rec
{
    void display()
    {
        System.out.println(“Hi!");
    }
}
```

```java
package p2;
import p1.*;

class RecNew
{
    public static void main(String args[])
    {
        // Accessing Rec from package p1
        Rec obj = new Rec();

        obj.display();
    }
}
```

Error → `obj.display();`

Java OOP Principles: Encapsulation



If we want to encapsulate this class or data/methods in this class

Private Access

Access Modifier	Same Class OR Subclass – same package	Same Package	Subclass – different package	World (Any class, all packages)
Public	Y	Y	Y	Y
Protected	Y	Y	Y	N
Default	Y	Y	N	N
Private	Y	N	N	N

Java OOP Principles: Encapsulation

```java
package p1;

class Rec
{
    private void display()
    {
        System.out.println(“Hi!");
    }
}
```

```java
package p2;
import p1.*;

class RecNew extends Rec
{
    public static void main(String args[])
    {
        // Accessing Rec from package p1
        RecNew obj = new RecNew();

        obj.display();
    }
}
```

Error! → `obj.display();`

Java OOP Principles: Encapsulation

If a data is encapsulated,

how can we change it or access **`outside the accessibility scope`**?

Java OOP Principles: Encapsulation

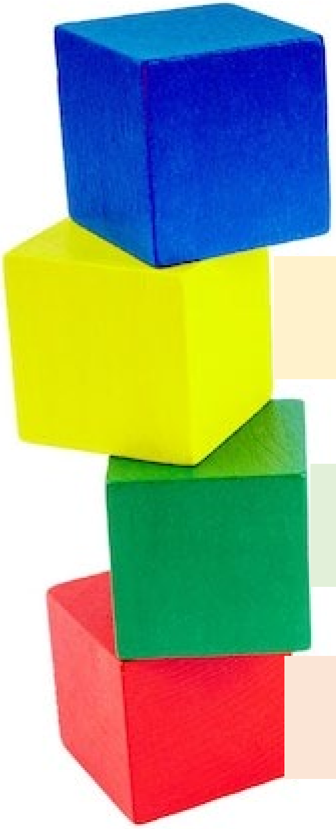
- Using **getters** and **setters**:

```
public class MainClass {

   public static void main(String args[]) {

       Animal foobar = new Animal();
       foobar.setName("Foo Bar");
```

```
public class Animal {
    private String name;
    private int age;

    public void setName(String newName)
    {
        this.name = newName;
    }
    public String getName() {
        return name;
    }
}
```

Why would we do that?

Java OOP: Core Principles



Abstraction

Encapsulation

Inheritance

Java OOP Principles: Abstract Classes

- A class that is **not completely implemented**.
- Contains *one or more* *abstract* methods (methods with no bodies; *only signatures*) that subclasses *must* implement
- Cannot be used to instantiate objects

Java OOP Principles: Abstract Classes

Syntax of defining and using abstract Classes/Methods

```
Abstract Class Header:
     accessModifier abstract class className

Abstract Method signature:
     accessModifier abstract returnType methodName ( args );

Subclass Signature:
     accessModifier class subclassName extends className
```

Project 1 Example

```java
public abstract class Test
{
  protected abstract void perform() throws Throwable;
```

```java
public class SkeletonTest extends Test
{
    /** Performs the test. */
    @Override
    protected void perform() throws TestFailed
    {
        ensureClassRejected();
        ensureNonRemoteInterfaceRejected();
        ensureNullPointerExceptions();
        ensureSkeletonRuns();
    }
```

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Java OOP Principles: Interfaces

- A **special abstract class** in which *all the methods are abstract*
- Contains only abstract methods that **subclasses must implement**
- All **fields** in an interface are automatically **public, static, and final**
- All **methods** that you declare or define (as default methods) are **public**
- An interface ***can extend other interfaces***

Java OOP Principles: Interfaces

Syntax of defining and using interfaces & their abstract methods

Interface header:

```
        accessModifier interface interfaceName
```

Abstract method signature in the interface:

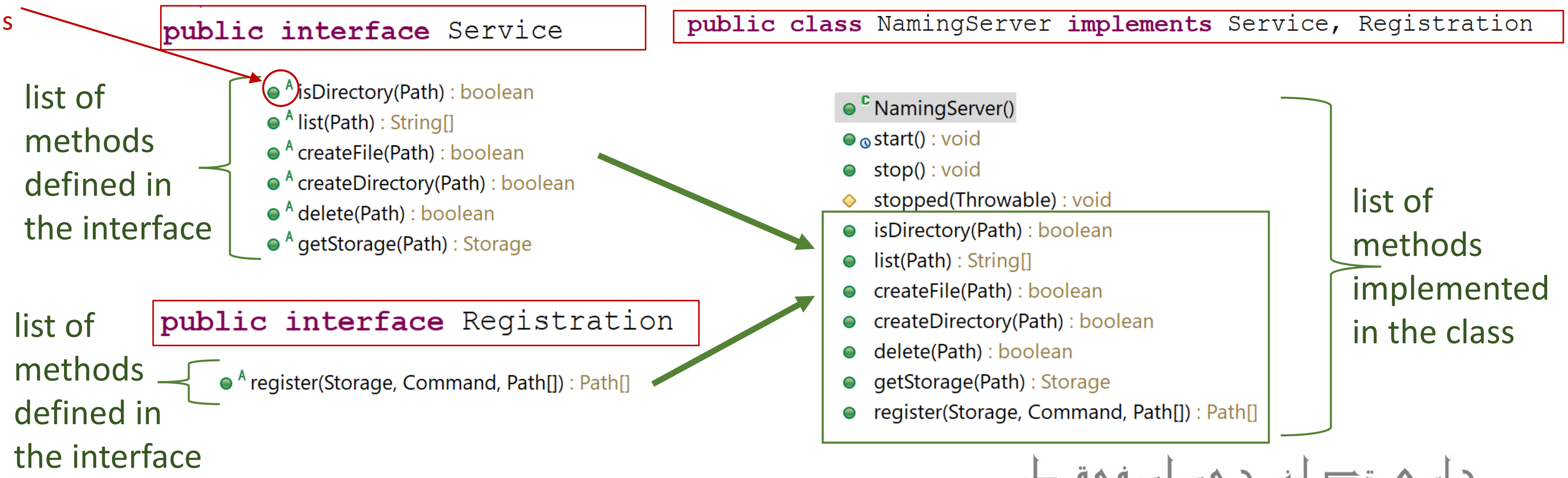
```
        accessModifier abstract returnType methodName ( args );
```

Subclass signature:

```
        accessModifier class subclassName implements someInterface
```

Methods declared as abstract

Project 1 Example



```
public interface Service
```

list of methods defined in the interface

- isDirectory(Path) : boolean
- list(Path) : String[]
- createFile(Path) : boolean
- createDirectory(Path) : boolean
- delete(Path) : boolean
- getStorage(Path) : Storage

```
public interface Registration
```

list of methods defined in the interface

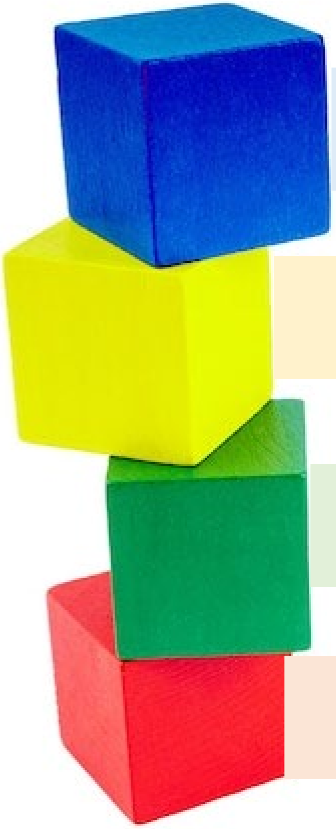
- register(Storage, Command, Path[]) : Path[]

```
public class NamingServer implements Service, Registration
```

- NamingServer()
- start() : void
- stop() : void
- stopped(Throwable) : void
- isDirectory(Path) : boolean
- list(Path) : String[]
- createFile(Path) : boolean
- createDirectory(Path) : boolean
- delete(Path) : boolean
- getStorage(Path) : Storage
- register(Storage, Command, Path[]) : Path[]

list of methods implemented in the class

Java OOP: Core Principles



Polymorphism

Abstraction

Encapsulation

Inheritance

Java OOP Principles: Polymorphism

- Polymorphism means “Many Forms”
- It is applied to methods to decide **what form of method to execute** on different **classes** that are **related** to each other **by Inheritance**.

Java OOP Principles: Polymorphism

Problem Set 1 Exercise:

```java
public abstract class Racer
{
  private String ID;  // racer ID
  private int x;      // x position
  private int y;      // y position

  /** default constructor
  *    Sets ID to blank
  */
  public Racer( )
  {
    ID = "";
  }
 /** abstract method for Racer's move
 */
 public abstract void move( );
```

```java
public class Tortoise extends Racer
{
   private int speed;

   /** Default Constructor: calls Racer default constructor
   */
   public Tortoise( )
   {
     super( );

     // percentage of time (between 90 - 99%) that this tortoise moves each turn
     speed = (int) ( Math.random( ) * 10  + 90 );
   }

   /** Constructor
   *    @param rID  racer Id, passed to Racer constructor
   *    @param rX    x position, passed to Racer constructor
   *    @param rY    y position, passed to Racer constructor
   */
   public Tortoise( String rID, int rX, int rY )
   {
     super( rID, rX, rY );

     // percentage of time (between 90 - 99%) that this tortoise moves each turn
     speed = (int) ( Math.random( ) * 10  + 90 );
   }
   /** move:  calculates the new x position for the racer
   *   Tortoise move characteristics: "slow & steady wins the race"
   *      increment x by 1 most of the time
   */
   public void move( )
   {
     int move =   (int) ( Math.random( ) * 100  + 1 );
     if (move < speed)
         setX( getX( ) + 1 );
   }
```

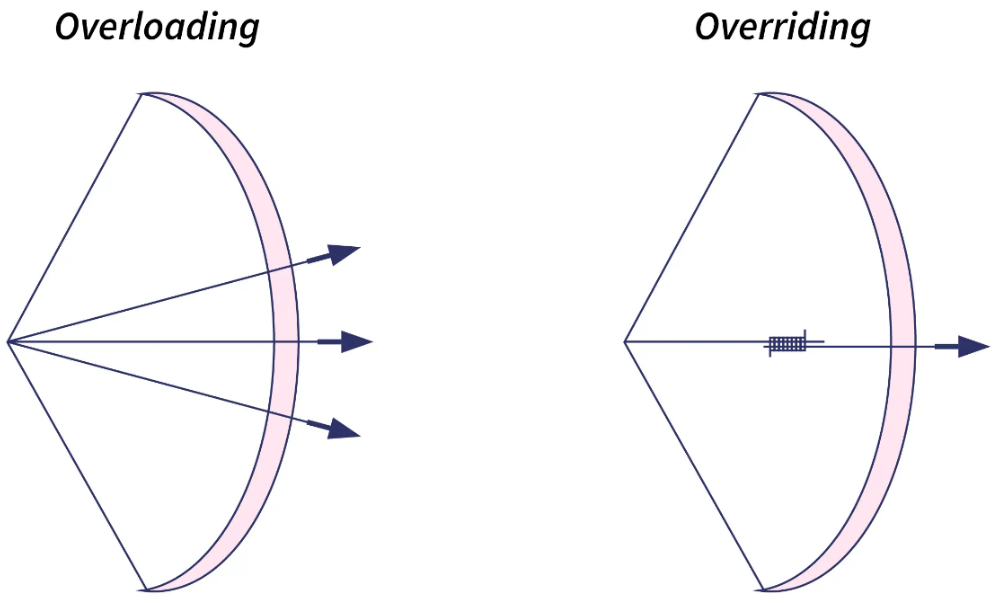
```java
public class Hare extends Racer
{
   /** Default Constructor: calls Racer default constructor
   */
   public Hare( )
   {
     super( );
   }

   /** Constructor
   *    @param rID   racer Id, passed to Racer constructor
   *    @param rX    x position, passed to Racer constructor
   *    @param rY    y position, passed to Racer constructor
   */
   public Hare( String rID, int rX, int rY )
   {
     super( rID, rX, rY );
   }
   /** move:  calculates the new x position for the racer
   *   Hare move characteristics:  30% of the time, Hare jumps 5 pixels
   *                               70% of the time, Hare sleeps (no move)
   *   generates random number between 1 & 10
   *          for 1 - 7,  no change to x position
   *          for 8 - 10, x position is incremented by 5
   */
   public void move( )
   {
     int move =   (int) ( Math.random( ) * 10  + 1 );

    if (getX() < 100)
    {
      if (move > 6)
       setX( getX() + 4);
    }
    else
    {
      if (move > 8)
       setX( getX() + 4);
    }
   }
```

Outline

- Introduction
- Java OOP Structure
  - Class
  - Object
  - Attributes
  - Methods
- Java OOP Core Principles
  - Inheritance
  - Encapsulation
  - Abstraction
  - Polymorphism
- **More Java Concepts**
  - **Overloading Methods**
  - **Overriding Methods**
  - **Generic Classes**
  - **Generic Collections**

More Java Concepts

Overloading & Overriding Methods



Java OOP: Overloading Methods

- Methods overload one another when they have the same method name but:
  - The **number of parameters** is different for the methods
  - The parameter **types** are different (i..e. different signatures)
- **Example:**

```
public void changeDate(int year) {
    // do cool stuff here
}

public void changeDate(int year, int month) {
    // do cool stuff here
}
```

Why would we do that?

Java OOP: Overloading Methods

- Methods overload one another when they have the same method name but:
  - The **number of parameters** is different for the methods
  - The parameter **types** are different (i..e. different signatures)
- **Another Example:**

```
public void addSemesterGPA(float newGPA) {
   // process newGPA
}

public void addSemesterGPA(double newGPA) {
    // process newGPA
}
```

Java OOP: Overloading Methods

- Methods overload one another when they have the same method name but:
  - The **number of parameters** is different for the methods
  - The parameter **types** are different (i..e. different signatures)
- **Another Example:**

```
public void changeDate(int year) {
    // do cool stuff here
}

public void changeDate(int month) {
    // do cool stuff here
}
```

Java OOP: Overloading Methods

- Methods overload one another when they have the same method name but:
  - The **number of parameters** is different for the methods
  - The parameter **types** are different

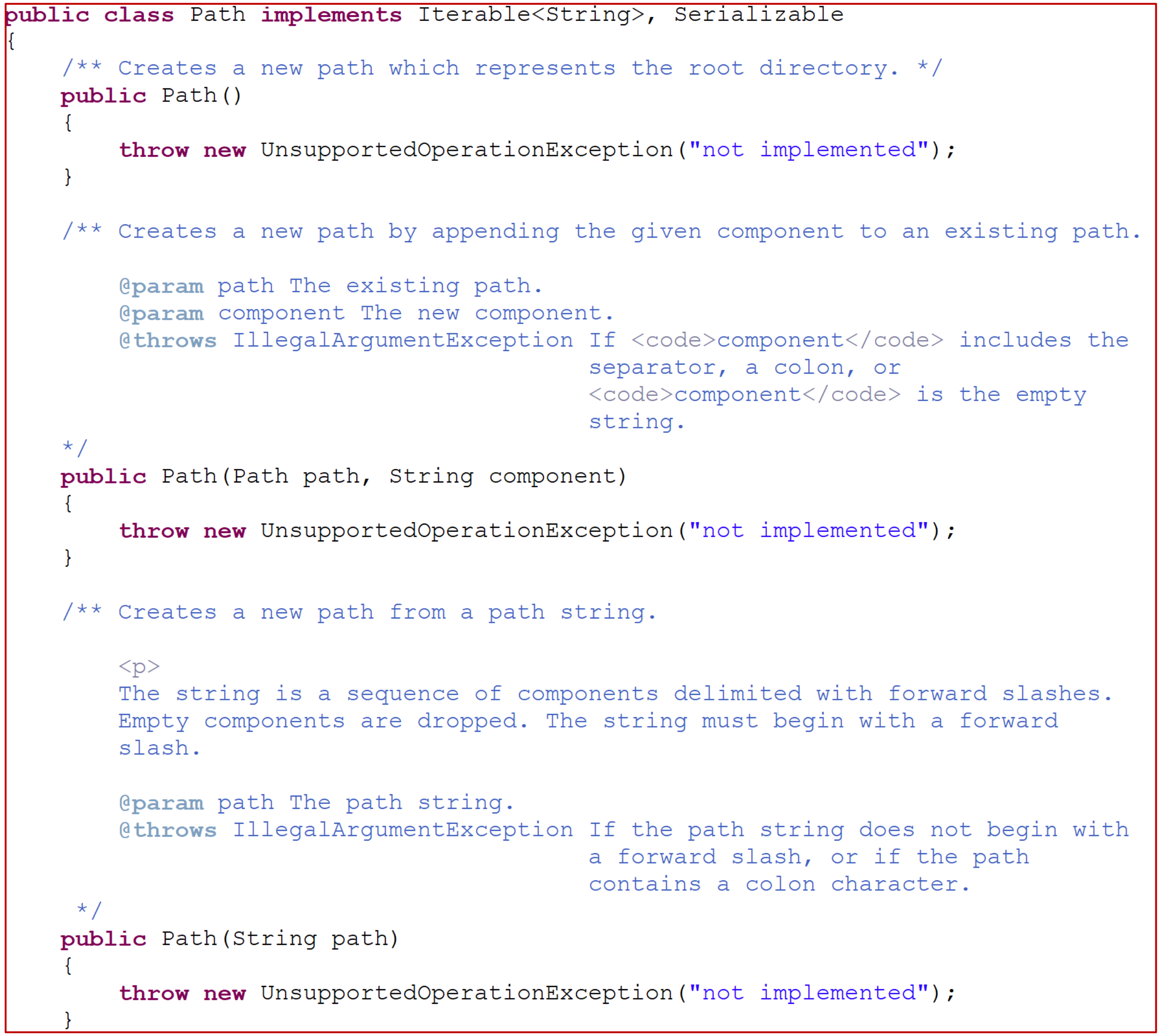
- **Another Example:**

```
public void changeDate(int year) {
    // do cool stuff here
}

public void changeDate(int month) {
    // do cool stuff here
}
```

We can’t overload methods by just changing the parameter name!

Java OOP: Overloading Methods

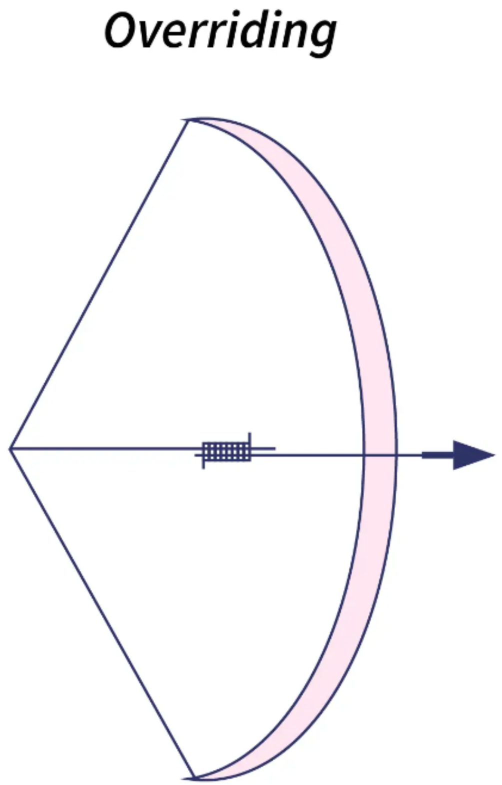
Project 1 Example



```java
public class Path implements Iterable<String>, Serializable
{
    /** Creates a new path which represents the root directory. */
    public Path()
    {
        throw new UnsupportedOperationException("not implemented");
    }

    /** Creates a new path by appending the given component to an existing path.

        @param path The existing path.
        @param component The new component.
        @throws IllegalArgumentException If <code>component</code> includes the
                                         separator, a colon, or
                                         <code>component</code> is the empty
                                         string.
    */
    public Path(Path path, String component)
    {
        throw new UnsupportedOperationException("not implemented");
    }

    /** Creates a new path from a path string.

        <p>
        The string is a sequence of components delimited with forward slashes.
        Empty components are dropped. The string must begin with a forward
        slash.

        @param path The path string.
        @throws IllegalArgumentException If the path string does not begin with
                                         a forward slash, or if the path
                                         contains a colon character.
     */
    public Path(String path)
    {
        throw new UnsupportedOperationException("not implemented");
    }
```

Constructor
Overloading

Java OOP: Overriding Methods



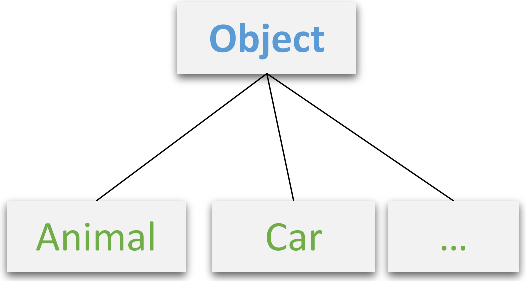
Java OOP: Overriding Methods

- Example:

```
public class ClassA {
   public Integer someMethod() {
               return 3;
   }
}

public class ClassB extends ClassA {

   // this is method overriding:
   public Integer someMethod() {
         return 4;
   }
}
```

Example use case?

Java OOP: Overriding Methods


Object

Animal

Car

…

- Any class extends the **Java** superclass "**Object**".
- The Java "**Object**" class has 3 important methods:
  - public boolean **equals**(Object obj);
  - public int **hashCode**();
  - public String **toString**();
- The hashCode is just a number that is generated by any object:
  - It **shouldn't** be used to compare two objects!
  - Instead, **override** the equals, hashCode, and toString methods.

Java OOP: Overriding Methods

- Example: **Overriding** the `toString` and `equals` methods in our `Dog` class:

```
public class Dog {
    …
    public String toString() {
        return this.name;
    }
}
```

Java OOP: Overriding Methods

- Example: **Overriding** the `toString` and `equals` methods in our `Dog` class:

```
public class Dog {
   …
   public boolean equals(Object obj) {
        if (obj.getClass() != this.getClass())
             return false;
        else {
             Dog s = (Dog) obj;
             return (s.getName().equals(this.name));
        }
   }
}
```

Java OOP: Overriding Methods (Super and Subclasses)

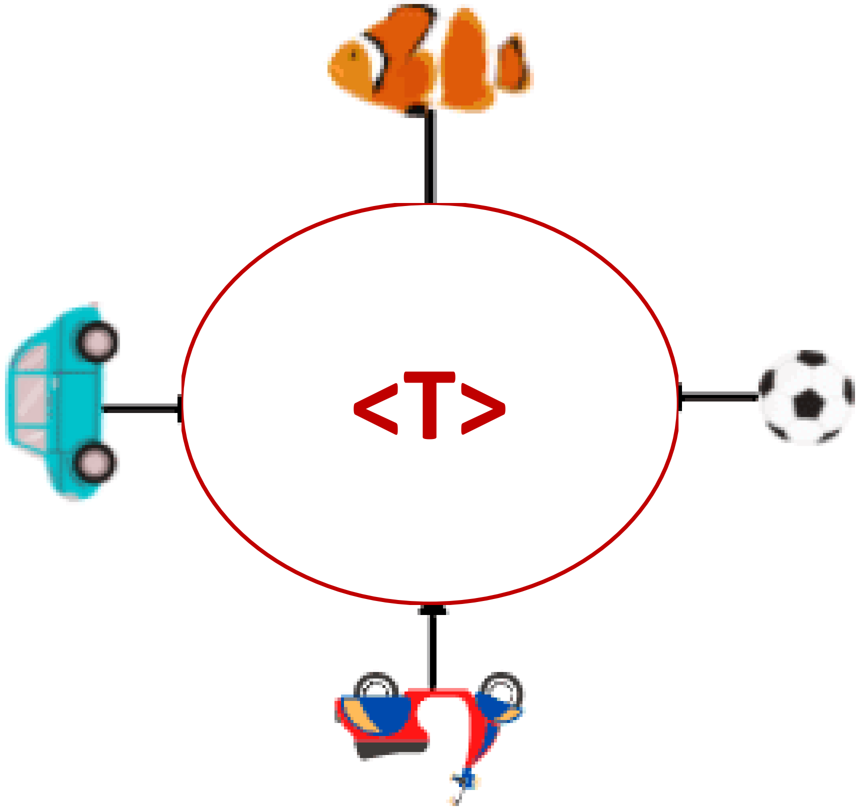
```
class Animal{
    void eat(){ System.out.println(“Animal eating..."); }
}

class Dog extends Animal{
    void eat(){ System.out.println(“Dog eating..."); }

    void bark(){ System.out.println("barking..."); }
}

class TestInheritance{
    public static void main(String args[]){
    Animal a= new Animal();
    Dog d = new Dog();
    a.eat();
    d.eat();
    a=d;
    a.eat();
}
```

What’s the output?

Carnegie Mellon University Qatar

More Java Concepts

Generic Methods, Classes and Collections



Java OOP: Generic Classes&Methods

What if you want to create a class or a method that works for different data types

instead of creating a class or a method for each data type

Java OOP: Generic Classes&Methods

- *“Object”* is the inherent super-type of all types in Java
    - So, would using “Object” work?

```java
public class Box {
      private Object attribute;

      public void set(Object object) {
            this. attribute = object;
      }
      public Object get() {
            return attribute;
      }
}
```

What’s the problem?

Java OOP: Generic Classes&Methods

- Solution:
  - *Generic* or *parameterized* classes/methods receive the data-type of elements as a parameter
  - Generics allow Code Reuse and ensure Type Safety
- A *generic class* is defined with the following format:

```
class my_generic_class <T1, T2, ..., Tn> {
 /* ... */
}
```

Type parameters

Java OOP: Generic Classes&Methods

- Now to make our Box class *generic*:

```
public class Box<T> {
    // T stands for "Type"
    private T t;
    public void set(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
}
```

Generic class

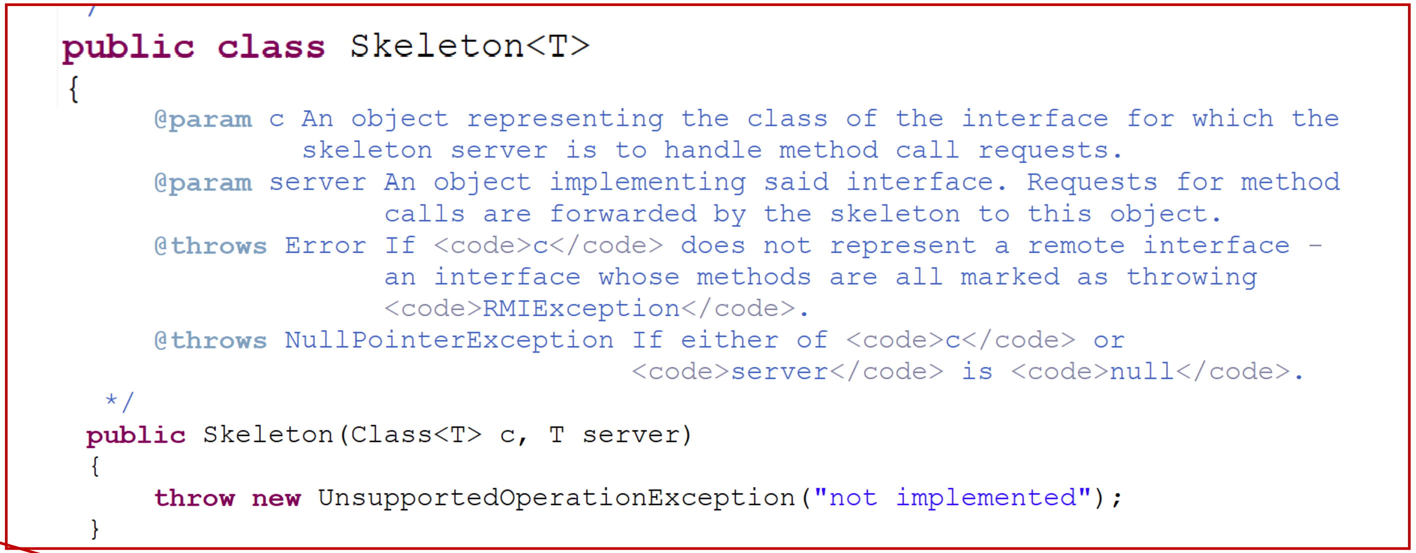
Generic method

To create, for example, an `Integer` “`Box`”:

```
Box<Integer> integerBox;
```

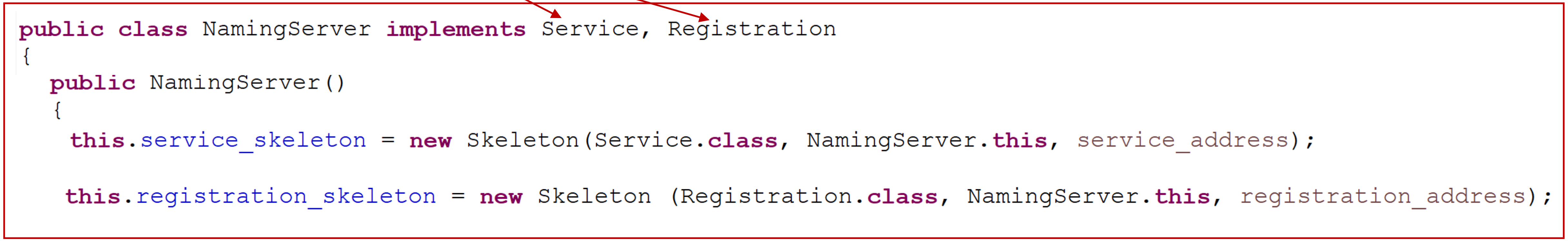
Java OOP: Generic Classes&Methods

Example from Project 1



```
public class Skeleton<T>
{
     @param c An object representing the class of the interface for which the
              skeleton server is to handle method call requests.
     @param server An object implementing said interface. Requests for method
                   calls are forwarded by the skeleton to this object.
     @throws Error If <code>c</code> does not represent a remote interface -
                   an interface whose methods are all marked as throwing
                   <code>RMIException</code>.
     @throws NullPointerException If either of <code>c</code> or
                                  <code>server</code> is <code>null</code>.
  */
  public Skeleton(Class<T> c, T server)
  {
      throw new UnsupportedOperationException("not implemented");
  }
```

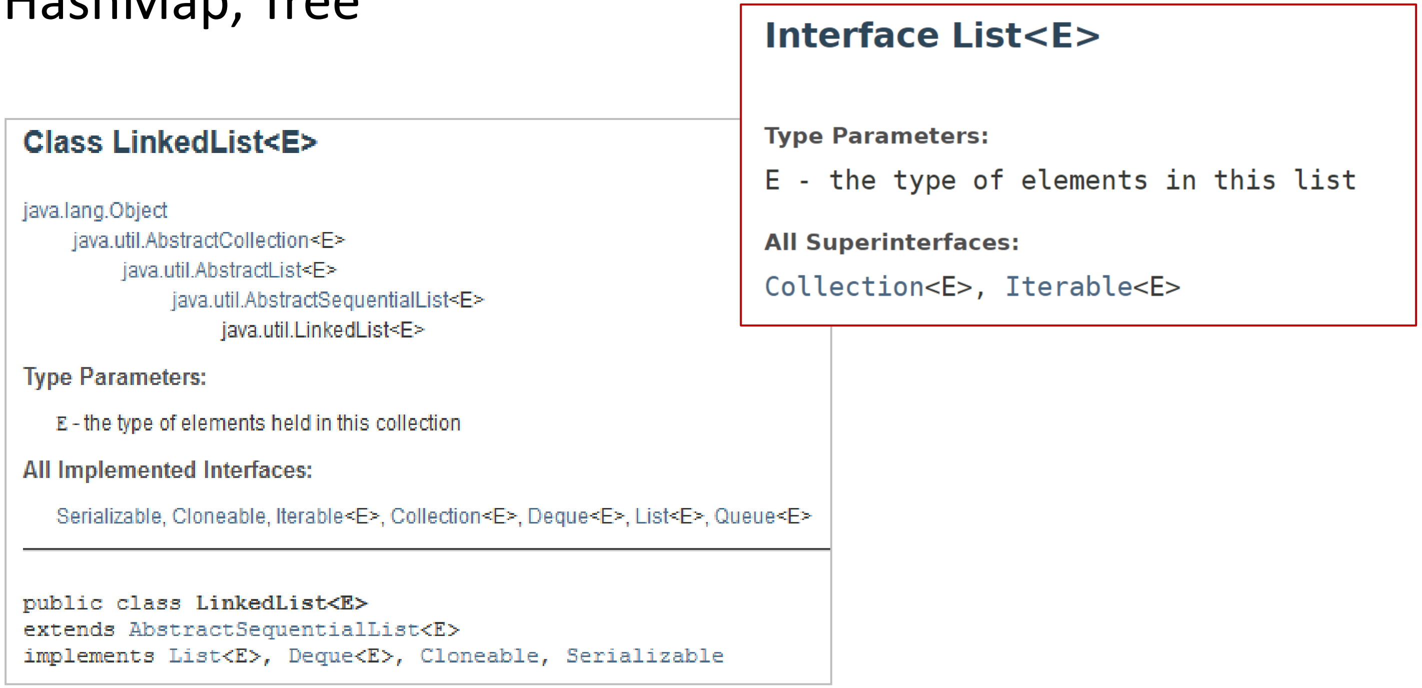
interfaces



```
public class NamingServer implements Service, Registration
{
  public NamingServer()
   {
    this.service_skeleton = new Skeleton(Service.class, NamingServer.this, service_address);

    this.registration_skeleton = new Skeleton (Registration.class, NamingServer.this, registration_address);
```

Java OOP: Generic Collections

- **Classes that represent data-structures**
- *Generic* or *parameterized* since the elements' **data-type is given as a parameter**
- E.g.: LinkedList, Queue, ArrayList, HashMap, Tree
- They provide methods for:
  - Iteration
  - Bulk operations
  - Conversion to/from arrays



Interface List<E>

Type Parameters:

`E - the type of elements in this list`

All Superinterfaces:

`Collection<E>, Iterable<E>`

Class LinkedList<E>

```
java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.AbstractSequentialList<E>
                java.util.LinkedList<E>
```

Type Parameters:

`E` - the type of elements held in this collection

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, List<E>, Queue<E>

```
public class LinkedList<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

Java OOP: Generics Symbols

Bounded Type Parameters

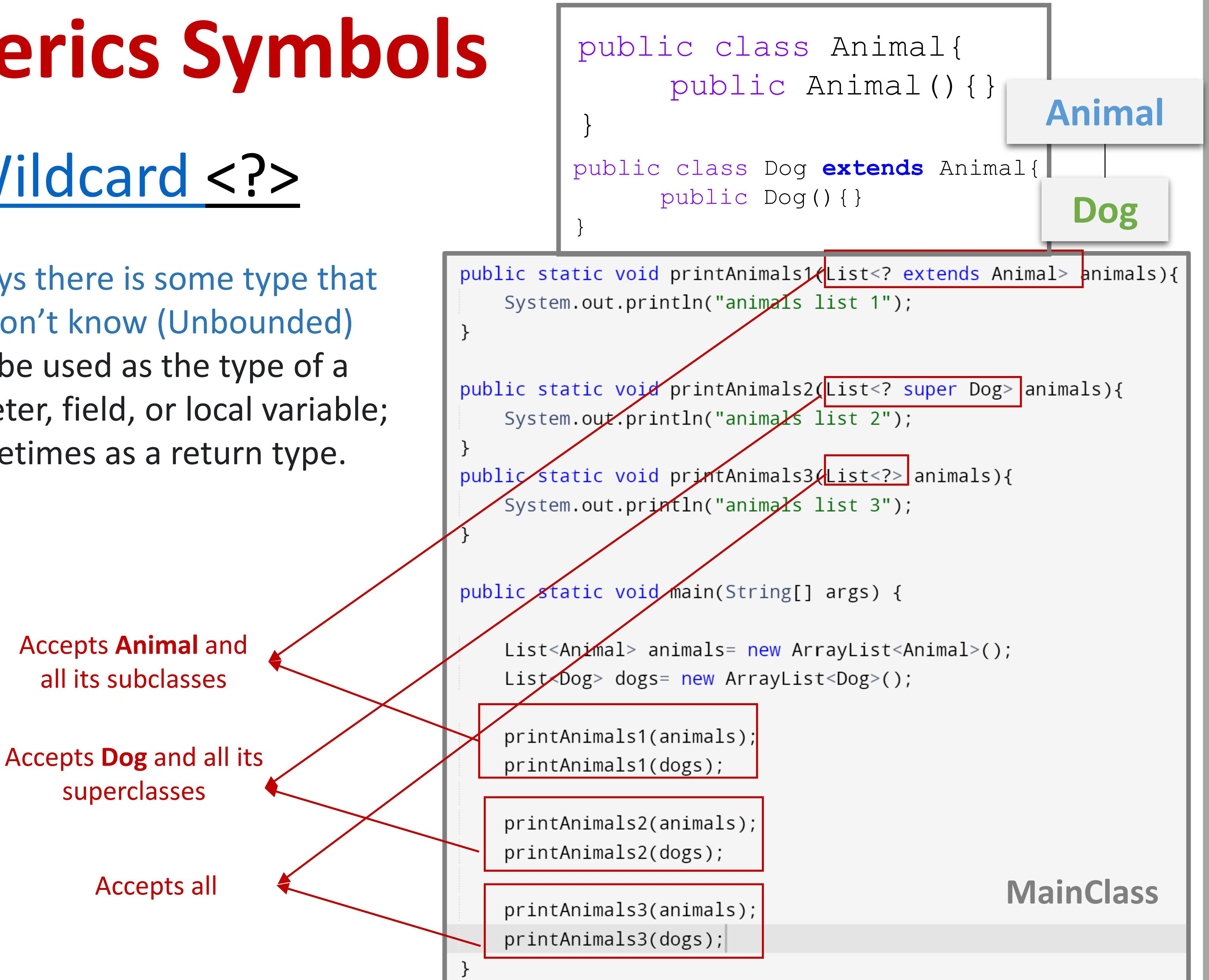
- T – Type
- E – Element
- K – Key
- N – Number
- V – Value

They restrict the type that can be used

More on Generics.

Wildcard <?>

<?> says there is some type that we don't know (Unbounded)
Can be used as the type of a parameter, field, or local variable; sometimes as a return type.



```java
public class Animal{
      public Animal(){}
}
public class Dog extends Animal{
      public Dog(){}
}
```

```java
public static void printAnimals1(List<? extends Animal> animals){
    System.out.println("animals list 1");
}

public static void printAnimals2(List<? super Dog> animals){
    System.out.println("animals list 2");
}
public static void printAnimals3(List<?> animals){
    System.out.println("animals list 3");
}

public static void main(String[] args) {

    List<Animal> animals= new ArrayList<Animal>();
    List<Dog> dogs= new ArrayList<Dog>();

    printAnimals1(animals);
    printAnimals1(dogs);

    printAnimals2(animals);
    printAnimals2(dogs);

    printAnimals3(animals);
    printAnimals3(dogs);
}
```

Accepts **Animal** and all its subclasses

Accepts **Dog** and all its superclasses

Accepts all

More Java Concepts

Try-Catch-Finally

To handle Exceptions that might arise in a piece of Code:

- Write the code within a **try block** followed by *one or more catch blocks*
- Each **catch block** is an exception handler that handles the type of exception indicated by its argument.
- Adding clean up code in a **finally block** is a good practice.
    - It *always* executes
    - Allows programmer to avoid having cleanup code accidently bypassed by a return, continue or break

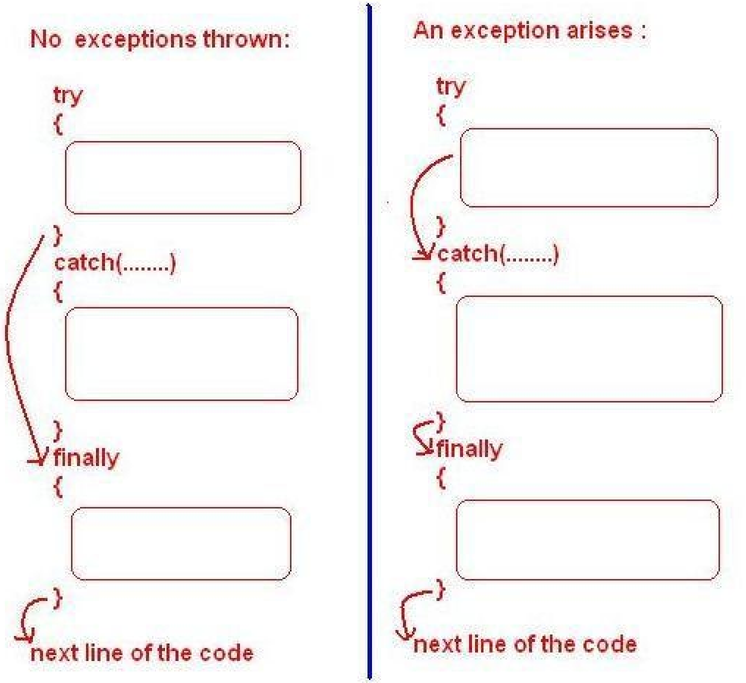
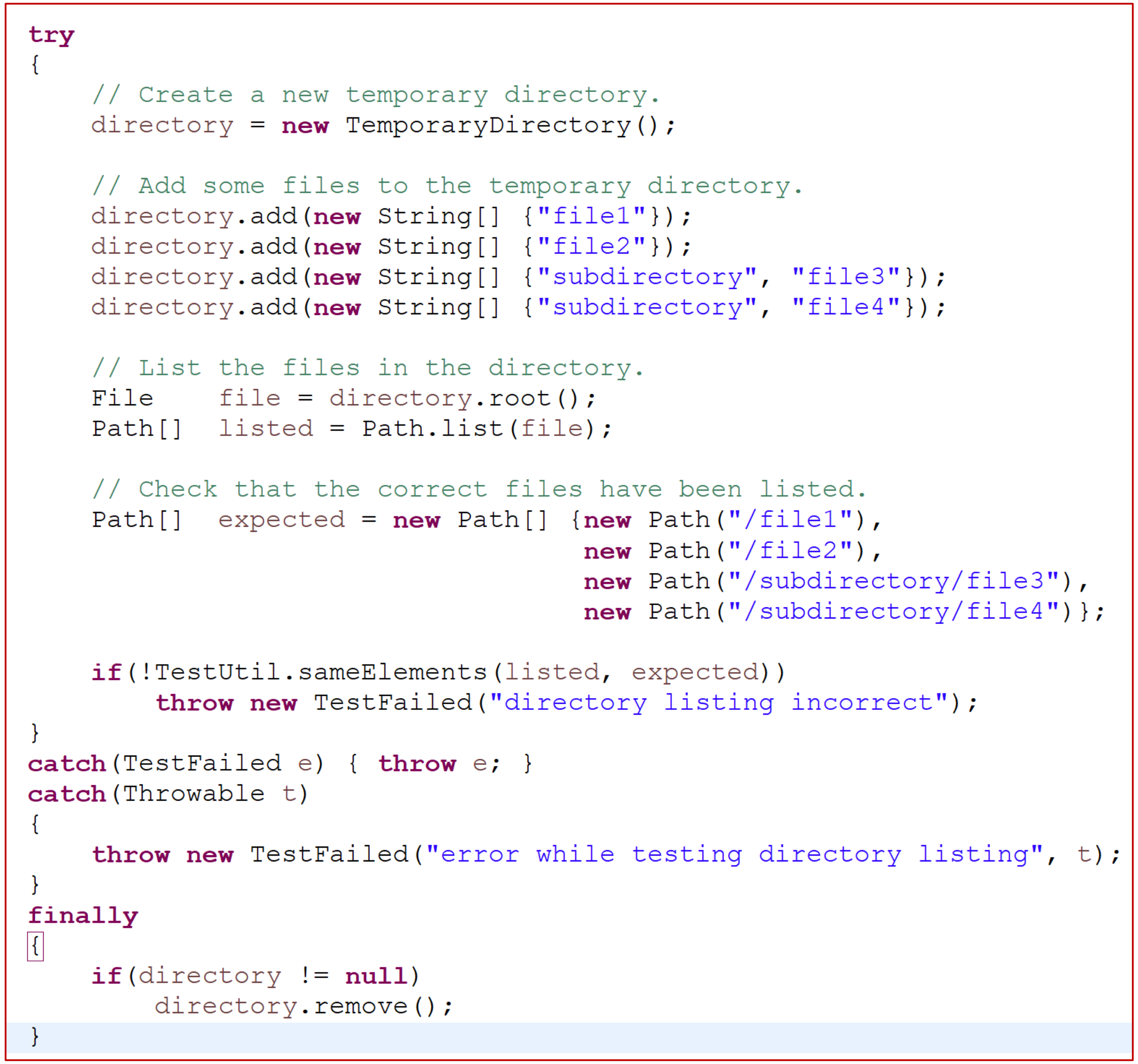


Photo credit: https://howtodoinjava.com/java/exception-handling/try-catch-finally/

Try-Catch-Finally

- **Example from Project 1**



```
try
{
    // Create a new temporary directory.
    directory = new TemporaryDirectory();

    // Add some files to the temporary directory.
    directory.add(new String[] {"file1"});
    directory.add(new String[] {"file2"});
    directory.add(new String[] {"subdirectory", "file3"});
    directory.add(new String[] {"subdirectory", "file4"});

    // List the files in the directory.
    File    file = directory.root();
    Path[]  listed = Path.list(file);

    // Check that the correct files have been listed.
    Path[]  expected = new Path[] {new Path("/file1"),
                                   new Path("/file2"),
                                   new Path("/subdirectory/file3"),
                                   new Path("/subdirectory/file4")};

    if(!TestUtil.sameElements(listed, expected))
        throw new TestFailed("directory listing incorrect");
}
catch(TestFailed e) { throw e; }
catch(Throwable t)
{
    throw new TestFailed("error while testing directory listing", t);
}
finally
{
    if(directory != null)
        directory.remove();
}
```

Recap …

- Introduction
  - What is Java
  - Java Language Constructs
- Java OOP Structure
  - Class
  - Object
  - Attributes
  - Methods
- Java OOP Core Principles
  - Inheritance
  - Encapsulation
  - Abstraction
  - Polymorphism
- More Java Concepts
  - Overloading Methods
  - Overriding Methods
  - Generics
  - Exceptions