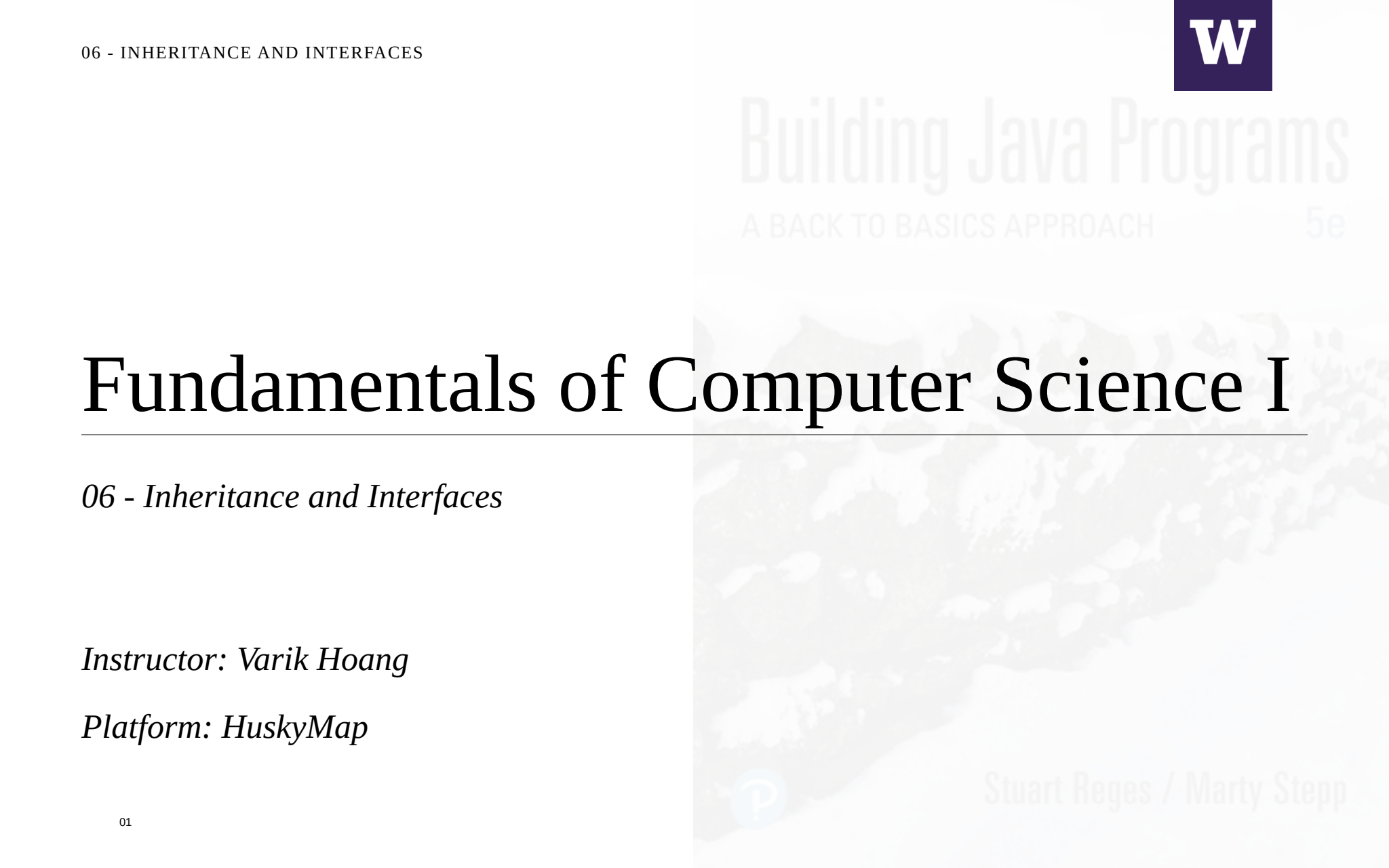


Fundamentals of Computer Science I

06 - Inheritance and Interfaces

Instructor: Varik Hoang

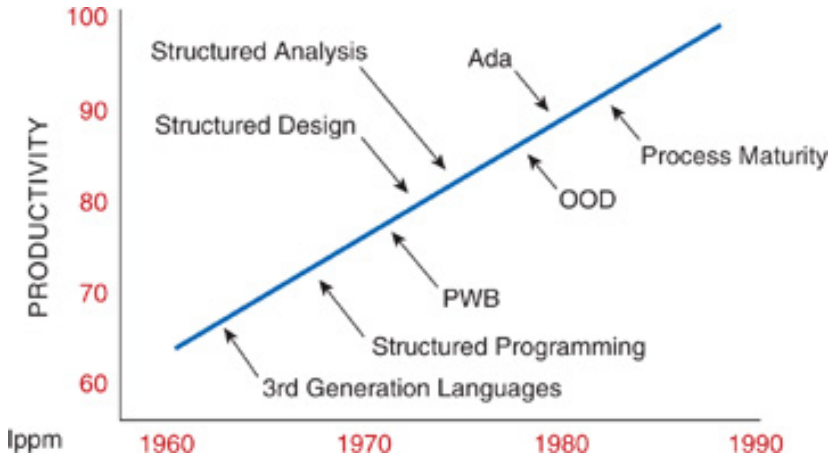
Platform: HuskyMap

Objectives

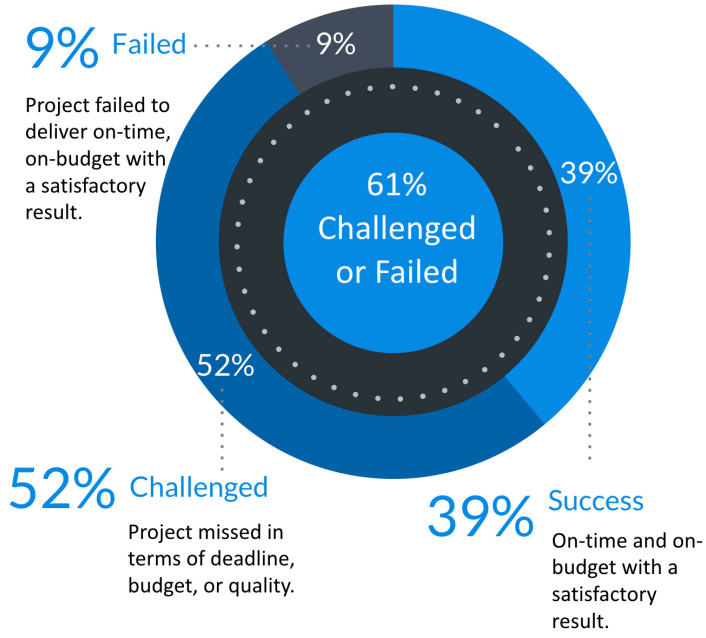
- Inheritance (Pillar)
  - Subclass Constructors
  - Overriding Methods
  - Access Modifiers
  - The Object Class
  - The **equals** Method
  - The **instanceof** Keyword
- Polymorphism (Pillar)
  - Interfaces
  - Abstract Class
  - Access Modifiers
- Overloading & Overriding

https://www.mygreatlearning.com/blog/oops-concepts-in-java/

The Software Crisis



https://www.informit.com/articles/article.aspx?p=2246403/



https://lucidlift.com/agile-coaching/agile-crisis/

Inheritance

- **Inheritance** is a way to form new classes based on existing classes, taking on their attributes/behavior.
  - A way to *group related classes*
  - A way to *share code* between two or more classes
- One class can extend another, absorbing its data/behavior
  - **superclass**: The parent class that is being extended.
  - **subclass**: The child class that extends the superclass and inherits its behavior and gets a copy of every field and method from *superclass*

Inheritance - Syntax

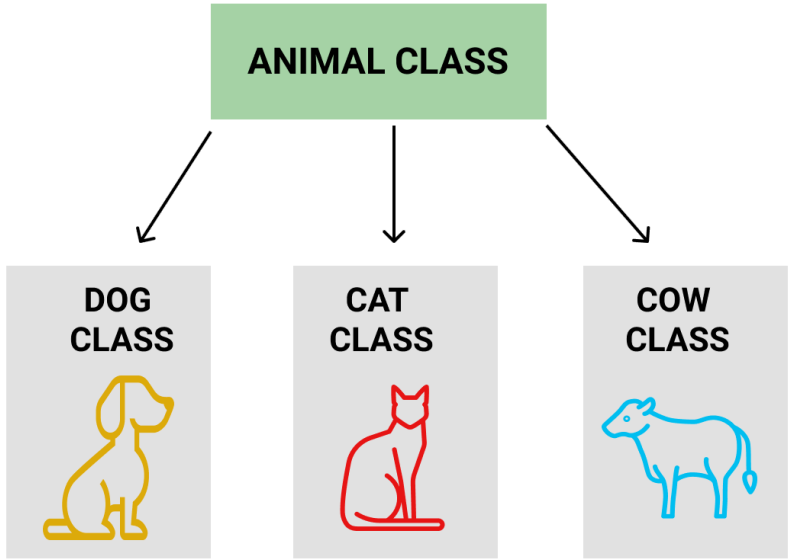
— The **extends** keyword: to make a new class that derives from existing class

— The meaning of "**extends**" is to increase the functionality.

```
public class <SUB_CLASS> extends <SUPER_CLASS>
{
    ...
}
```

Inheritance - Purposes

- **Extending a class** and **overriding methods**
- Allows greater flexibility in **reusing code**
- Creates hierarchies of related object types



https://eng.libretexts.org/

Inheritance - **Is-a** Relationship & Hierarchy

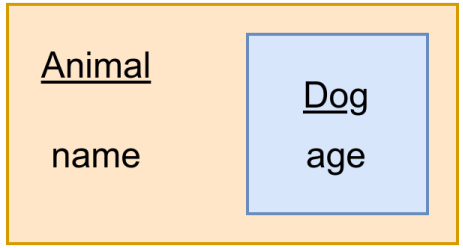
- **Is-a** Relationship: a hierarchical connection where one category can be treated as a specialized version of another.
  - **Subclasses** have an **is-a** relationship with **superclass** that they are derived from
    - A **Dog** is a type of **Animal**
    - **Animal** includes **Dog**, **Cat**, **Bird**, etc.
  - The **subclasses** consists of all *attributes* and *methods* that their **superclass** has
  - The **subclasses** inherit all the state and behaviors of the **superclass**
  - The **subclass** and **superclass** are known as **child** and **parent** classes *respectively*
- Inheritance Hierarchy: a set of classes connected by **is-a** relationships that can share common code.

Inheritance - Subclass Constructors

- The constructors of subclass should **always** be overridden using **super** keyword
- The local constructor needs to **initiate** the local attributes itself

```
public class Animal
{
    private String name;

    public Animal(String name)
    {
        this.name = name;
    }
}
```

```
public class Dog extends Animal
{
    private int age;

    public Dog(String name, int age)
    {
        super(name);
        this.age = age;
    }
}
```



Inheritance - Overriding Methods

- Implement a **new** version of the inherited method of the superclass in the subclass
- The new method **must have the same signature** as the method inherited from the superclass

```
public class Animal
{
    private String name;

    public Animal(String name)
    {
        this.name = name;
    }


    public void speak()
    {
        System.out.println("...");
    }
}
```

```
public class Dog extends Animal
{
    private int age;

    public Dog(String name, int age)
    {
        super(name);
        this.age = age;
    }

    public void speak()
    {
        System.out.println("Woof");
    }
}
```

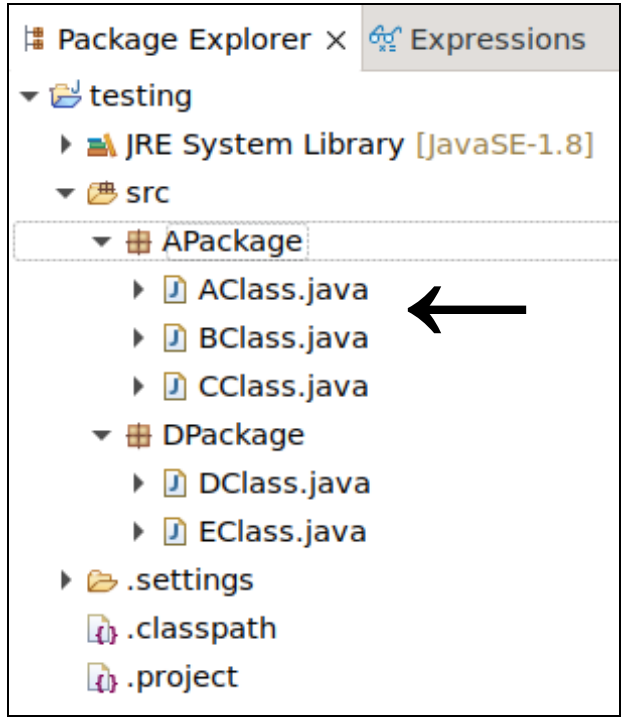
```
public class Cat extends Animal
{
    private int age;

    public Cat(String name, int age)
    {
        super(name);
        this.age = age;
    }

    public void speak()
    {
        System.out.println("Meow");
    }
}
```

Inheritance - Access Modifiers

Access Modifier	In Class	In Package (Subclass)	Out of Package (Subclass)	Out of Package (Not subclass)
private	**Yes**	No	No	No
[Default]	**Yes**	**Yes**	No	No
protected	**Yes**	**Yes**	**Yes**	No
public	**Yes**	**Yes**	**Yes**	**Yes**

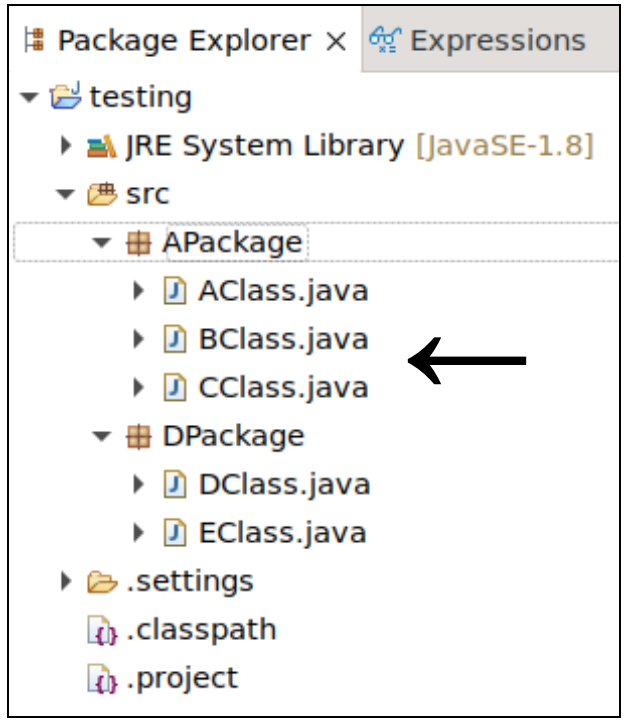
Inheritance - Access Modifiers (cont.)

```
package APackage;

public class AClass
{
    // same package - same class
    public void testMeAll()
    {
        tryMePrivate();   // Worked
        tryMeDefault();   // Worked
        tryMeProtected(); // Worked
        tryMePublic();    // Worked
    }

    private   void tryMePrivate()   {};
              void tryMeDefault()   {};
    protected void tryMeProtected() {};
    public    void tryMePublic()    {};
}
```



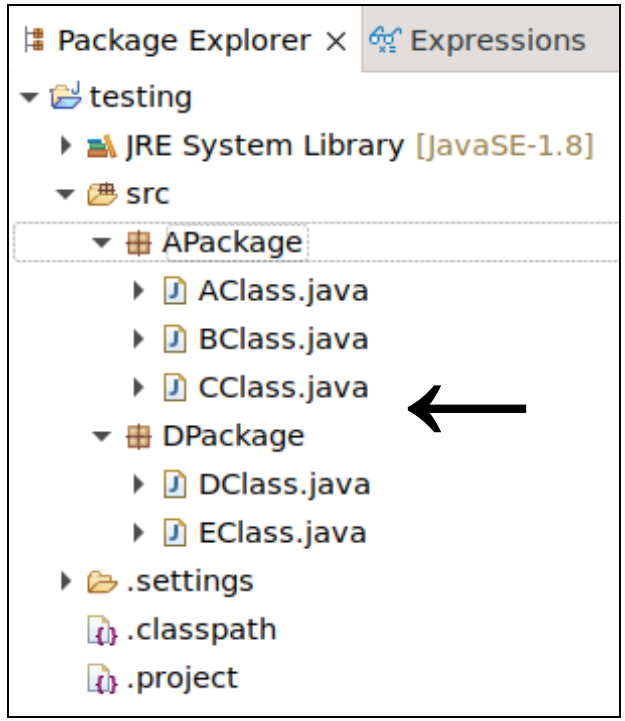
Inheritance - Access Modifiers (cont.)

```
package APackage;

public class BClass
extends AClass
{
    // same package - subclass
    public void testMeAll()
    {
        tryMePrivate();   // Compiled Error
        tryMeDefault();   // Worked
        tryMeProtected(); // Worked
        tryMePublic();    // Worked
    }
}
```



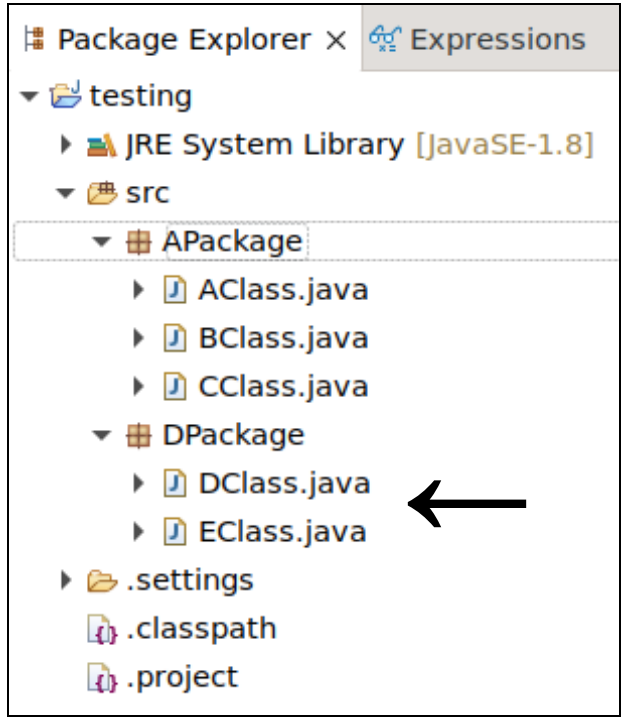
Inheritance - Access Modifiers (cont.)

```
package APackage;

public class CClass
{
    // same package - not subclass
    public void testMeAll()
    {
        AClass me = new AClass();
        me.tryMePrivate();   // Compiled Error
        me.tryMeDefault();   // Worked
        me.tryMeProtected(); // Worked
        me.tryMePublic();    // Worked
    }
}
```



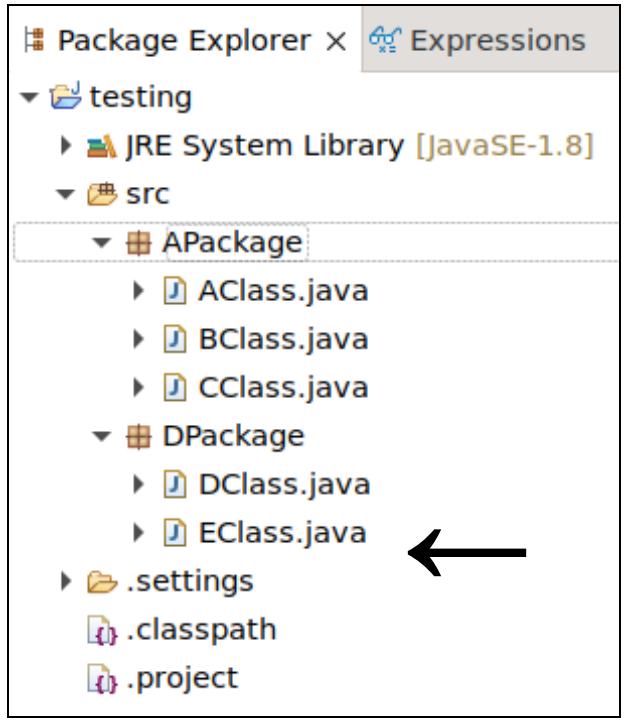
Inheritance - Access Modifiers (cont.)

```
package DPackage;

import APackage.AClass;

public class DClass
extends AClass
{
    // different package - subclass
    public void testMeAll()
    {
        tryMePrivate();   // Compiled Error
        tryMeDefault();   // Compiled Error
        tryMeProtected(); // Worked
        tryMePublic();    // Worked
    }
}
```



Inheritance - Access Modifiers (cont.)

```
package DPackage;

import APackage.AClass;

public class EClass
{
    // different package - not subclass
    public void testMeAll()
    {
        AClass me = new AClass();
        me.tryMePrivate();   // Compiled Error
        me.tryMeDefault();   // Compiled Error
        me.tryMeProtected(); // Compiled Error
        me.tryMePublic();    // Worked
    }
}
```



Inheritance - The **Object** Class

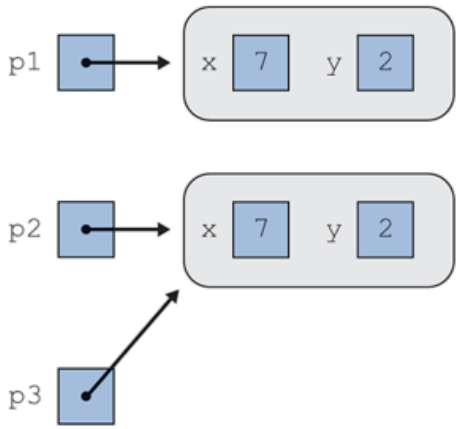
- In Java, **all** classes are derived either directly or indirectly from **Object** class
- There are certain methods that are inherited by **all** objects
  - Two important examples are **toString**() and **equals**(**Object**)
- These methods **don't always** behave the way you want them to
- Typically, we want to write new local versions (**overriding methods**) for the inherited classes you create to override the undesirable default behavior of the method

```
public class MyClass
extends Object // no error but unnecessary
{
    ... the content of the class
}
```

Inheritance - The **equals**(obj) Method

- The equality operator compares the **memory addresses** of two objects
- The equality method compares based on the **overridden implementation**

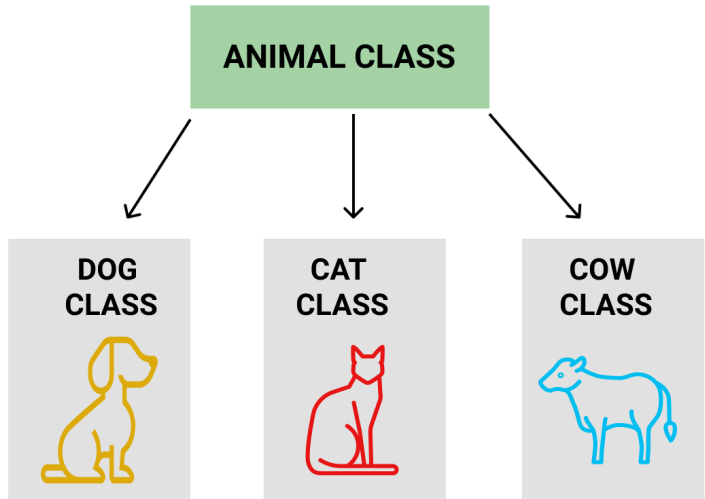
```
Point p1 = new Point(7, 2);
Point p2 = new Point(7, 2);
Point p3 = p2;

System.out.println(p1.equals(p2)); // ?????
System.out.println(p1 == p2);      // false
System.out.println(p2 == p3);      // true
```




```
String str1 = new String("MyString");
String str2 = new String("MyString");
String str3 = "MyString";
String str4 = "MyString";

System.out.println(str1.equals(str2)); // true
System.out.println(str1 == str2);      // false
System.out.println(str1.equals(str3)); // true
System.out.println(str1 == str3);      // false
System.out.println(str3.equals(str4)); // true
System.out.println(str3 == str4);      // true
```

Inheritance - The **equals**(obj) Method (cont.)

- Dictate how the comparison that determines equality is made
- The **equals** method should **always** take a parameter of type **Object**
  - This creates a potential issue because direct comparisons of the derived object with an object of type Object will **not** work
  - The Java compiler will **not** know ahead of time what type of "**Object**" that need to be compared.



Inheritance - The **equals**(obj) Method (cont.)

```java
public class Dog extends Animal
{
    private int age;

    public Dog(String name, int age)
    {
        super(name);
        this.age = age;
    }

    public static void main(String[] args)
    {
        Dog d1 = new Dog("Henry", 3);
        Dog d2 = new Dog("Henry", 8);
        System.out.println(d1.equals(d2));  // false
        System.out.println(d1 == d2);       // false
    }
}
```

```java
public class Dog extends Animal
{
    private int age;

    public Dog(String name, int age)
    {
        super(name);
        this.age = age;
    }

    public static void main(String[] args)
    {
        Dog d1 = new Dog("Henry", 3);
        Dog d2 = new Dog("Henry", 8);
        System.out.println(d1.equals(d2));  // true
        System.out.println(d1 == d2);       // false
    }

    public boolean equals(Object object)
    {
        Dog other = (Dog) object;
        if (this.getName().equals(other.getName()))
            return true;
        else return false;
    }
}
```

The **instanceof** Keyword

- To **identify** the type of subclass (two child classes can have different attributes)
- To avoid comparisons between two **incomparable** objects
- Always **cast** the object type after being identified
- The **instanceof** keyword accomplishes **this**

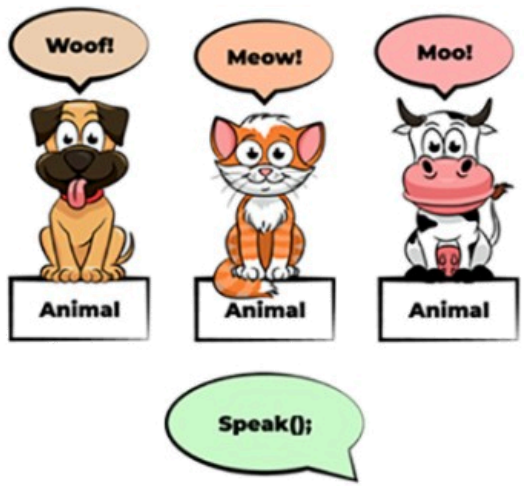
```
21    public boolean equals(Object object)
22    {
23        if (object instanceof Dog)
24        {
25            Dog other = (Dog) object;
26            if (this.getName().equals(other.getName()))
27                return true;
28            else return false;
29        }
30
31        return false;
32    }
```

The **final** Keyword

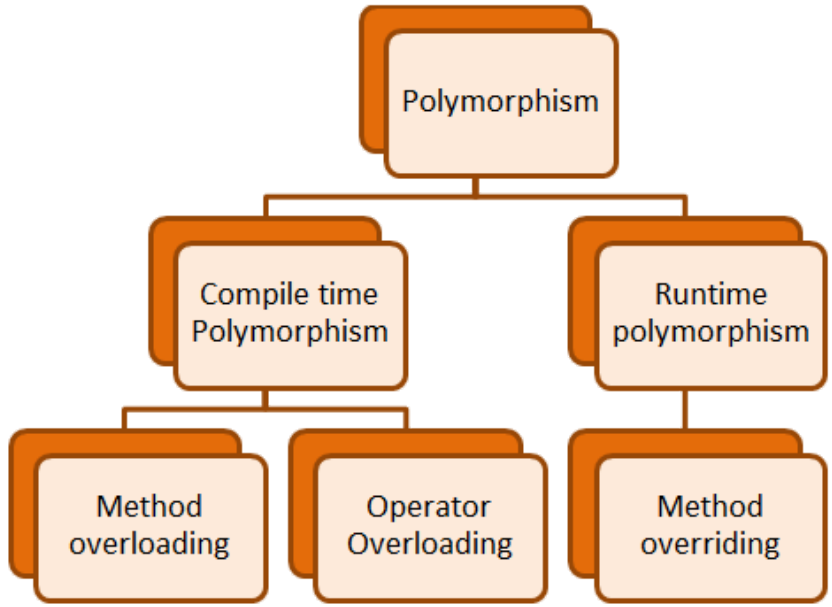
- The **final** keyword generally mean **"no change after being set"**
  - When we use **final** keyword on a variable, it's called **constant**
    - We *cannot assign* a different value once it's initiated
  - When we use **final** keyword on a method, no formal terminology to call it
    - We *cannot overwrite* this method from the parent class
  - When we use **final** keyword on a class, no formal terminology to call it
    - We *cannot inherit* from this class (it has no child class)

Polymorphism

- Polymorphism: ability for the same code to be used with different types of objects and behave differently with each.
  - Compile-time Polymorphism (static or early binding)
  - Runtime Polymorphism (dynamic or late binding)



https://codegym.cc/groups/posts/polymorphism-in-java



https://www.softwaretestinghelp.com/polymorphism-in-java/

Polymorphism - Parameters

We can pass any subtype of a parameter's type.

```
public class AnimalMain
{
    public static void main(String[] args)
    {
        Animal dog = new Dog("Teddy", 5);
        Animal cat = new Cat("Lucy", 4);
        printInfo(dog);
        printInfo(cat);
    }

    public static void printInfo(Animal animal)
    {
        animal.speak();
    }
}
```

```
varikmp@gazelle:~/$ javac *.java
varikmp@gazelle:~/$ java AnimalMain
Woof
Meow
varikmp@gazelle:~/$ _
```

Polymorphism - Array

We can have an array of objects that inherit from one parent.

```
public class AnimalMain
{
    public static void main(String[] args)
    {
        Animal[] animals = {
            new Animal("Rosie"),
            new Dog("Lucky"),
            new Cat("Gary")
        };

        int index, len = animals.length;
        for (index = 0; index < len; index++)
            animals[index].speak();
    }
}
```

```
varikmp@gazelle:~/$ javac *.java
varikmp@gazelle:~/$ java AnimalMain
...
Woof
Meow
varikmp@gazelle:~/$ _
```

Polymorphism - Mechanics

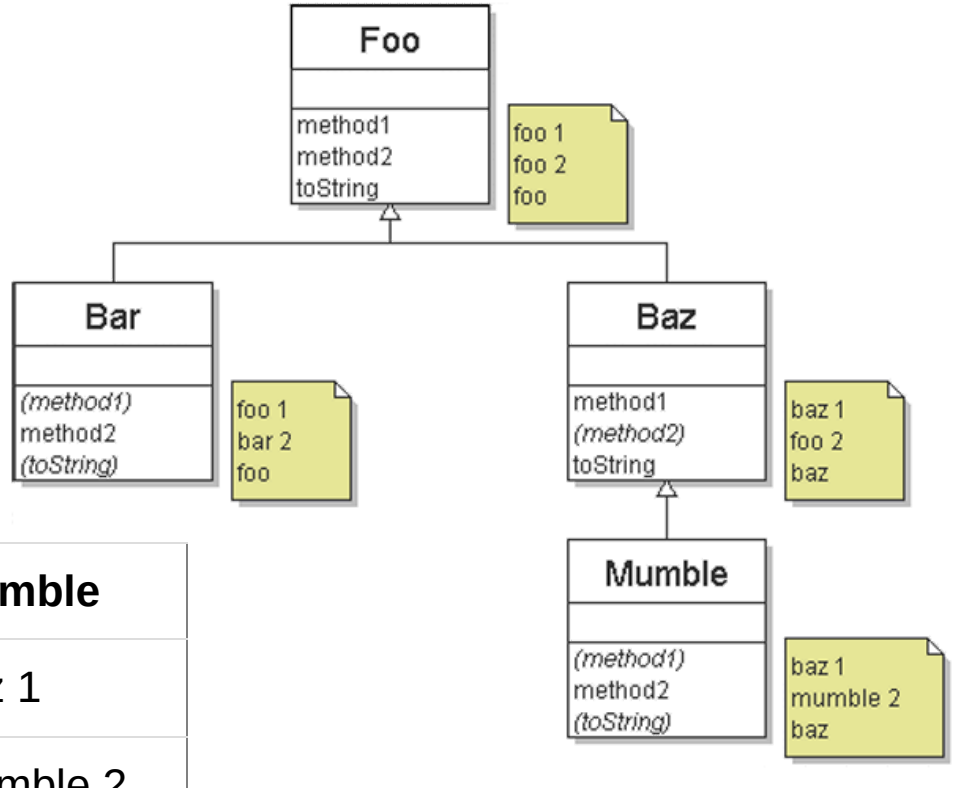
```
public class Foo {
    public void method1() {
        System.out.println("foo 1");
    }

    public void method2() {
        System.out.println("foo 2");
    }

    public String toString() {
        return "foo";
    }
}

public class Bar extends Foo {
    public void method2() {
        System.out.println("bar 2");
    }
}
```

```
public class Baz extends Foo {
    public void method1() {
        System.out.println("baz 1");
    }

    public String toString() {
        return "baz";
    }
}

public class Mumble extends Baz {
    public void method2() {
        System.out.println("mumble 2");
    }
}
```

```
Foo[] elements = { new Foo(), new Bar(),
                   new Baz(), new Mumble() };
for (int i = 0; i < elements.length; i++) {
    System.out.println(elements[i]);
    elements[i].method1();
    elements[i].method2();
}
```

What would be the output of the following?

Polymorphism - Mechanics (cont.)



Method	Foo	Bar	Baz	Mumble
method1	foo 1	foo 1	baz 1	baz 1
method2	foo 2	bar 2	foo 2	mumble 2
toString	foo	foo	baz	baz

Polymorphism - Mechanics (cont.)

```
public class Foo {
    public void method1() {
        System.out.println("foo 1");
    }

    public void method2() {
        System.out.println("foo 2");
    }

    public String toString() {
        return "foo";
    }
}
```

```
public class Bar extends Foo {
    public void method2() {
        System.out.println("bar 2");
    }

    // this method doesn't have in the parent class
    public void method3() {
        System.out.println("bar 3");
    }
}
```

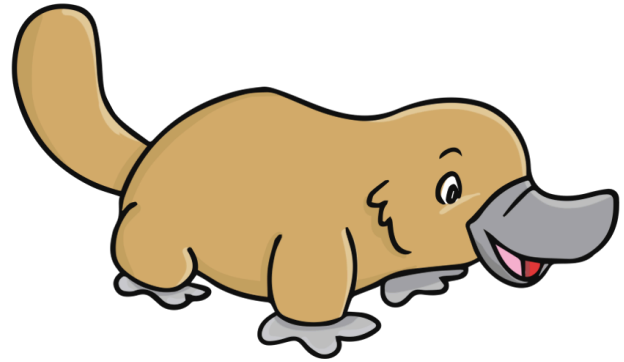
What would be the output of the following?

```
Foo element = new Bar();
element.method1(); // "foo 1" -> Bar doesn't have method1(). It calls method1() from Foo where it inherited
element.method2(); // "bar 2" -> Bar does    have method2(). It calls method2()          where it defined
element.method3(); // this line causes compiled error because Foo doesn't have method3() defined
```

Interfaces

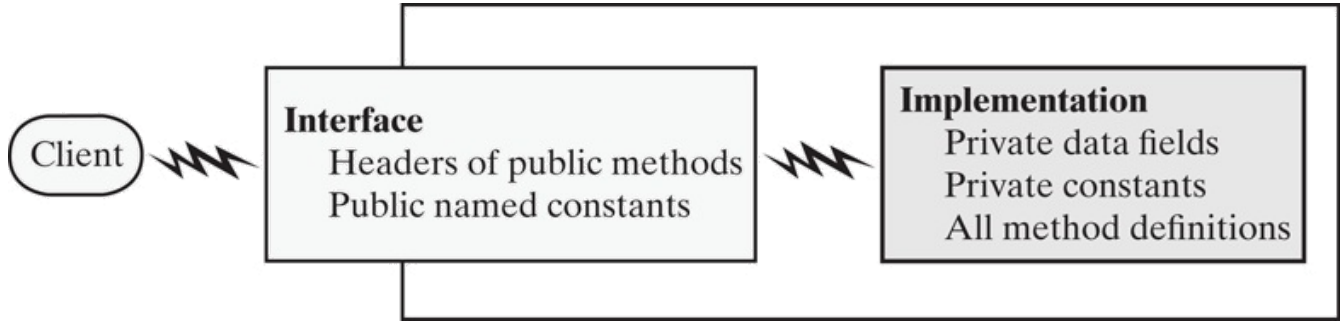
- **Snake** can "implements" **Reptile**
- **Eagle** can "implements" **Bird**
- **Dog** can "implements" **Mammal**
- **Reptile**, **Bird**, and **Mammal** are **Animal**

Reptile	Bird	Mammal
isDeadlyVenom	isLayingEgg	isLactating



Interfaces (cont.)

- Inheritance is useful as a way to share code but limited to single inheritance
  - A class can **extend only one superclass**
- What if we want to set up **multiple "is-a" relationships**?
- A feature called an interface can represent a common supertype between **multiple classes without code sharing**
- Interface fundamentally provides **only method declarations**
- The class implements an interface **must overwrite all methods** in the interface



Interfaces (Example)

```java
public interface Shape
{
    public double getArea();
    public double getPerimeter();
}
```

```java
public class Rectangle
implements Shape
{

}
```

```java
public class Rectangle
implements Shape
{
    private int height;
    private int width;

    public Rectangle(int height, int width)
    {
        this.height = height;
        this.width = width;
    }

    @Override
    public double getArea()
    {
        return this.height * this.width;
    }

    @Override
    public double getPerimeter()
    {
        return (this.height + this.width) * 2;
    }
}
```

Inheritance & Interfaces

- **Inheritance** shares the same **characteristics** in code implementation.
- **Inheritance** is to derive new classes from existing classes.
- **Interface** shares the same **prototypes** in the objects (no code sharing).
- **Interface** is to implement abstract classes and multiple inheritance.

Abstract Class

- Abstract class is a **hybrid** of **concrete class** and **abstract interfaces**
- Abstract class **cannot** be initiated itself
- Abstract class can **have attributes**
- Abstract method is **only allowed** to be in abstract class
- Abstract method can be **optionally** implemented
- Abstract class can:
  - **Extend** from *a* superclass
  - **Implement** *multiple* interfaces

Abstract Class - Properties

Functionality or Object Types	**Interfaces**	**Abstract Classes**	**Concrete Classes**
Object can have attributes	No	**Yes**	**Yes**
Object can have methods implemented	No	**Yes**	**Yes**
Object can be initiated	No	No	**Yes**
Object must implement all methods that inherited from the interfaces	No	No	**Yes**

Abstract Class - Example

```
public abstract class Animal
{
    String name;

    public Animal(String name) { this.name = name; }
    public void move()
    {
        System.out.println("I am moving");
        speak(); // not implemented yet
    }

    public abstract void speak();
}
```

```
public class Main
{
    public static void main(String[] args)
    {
        Animal myDog = new Dog("lucky");
        myDog.speak();
        myDog.move();

        // this line below causes compiled error
        // Animal myDog = new Animal("lucky");
    }
}
```

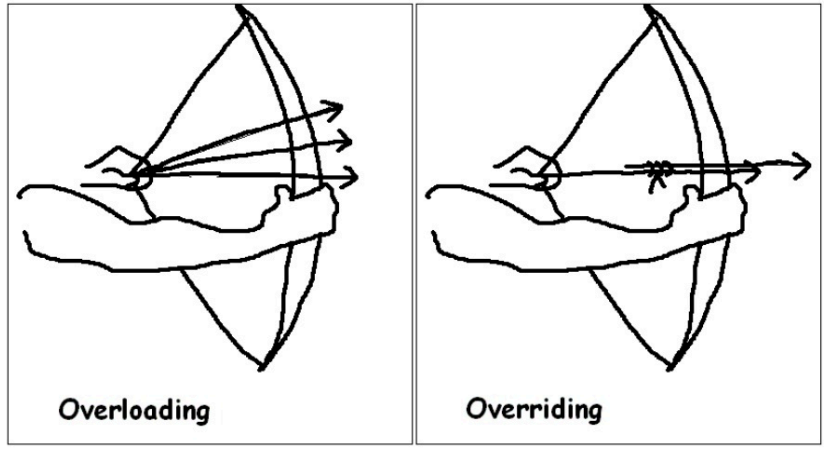
```
public class Dog extends Animal
{
     // always require to create constructor(s)
    public Dog(String name) { super(name); }

    public void speak() // required to implement
    {
        System.out.println("I am speaking");
    }
}
```

```
I am speaking
I am moving
I am speaking
```

Overloading & Overriding

- **Overloading**: a feature of Java in which a class has more than one method of the same name and their parameters are different
- **Overriding**: a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes

```
class OverloadingExample{
 static int add(int a, int b)
 {
  return a+b;
 }
 static int add(int a, int b, int c)
 {
  return a+b+c;
  }
}
```




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

Questions & Answer


Thank You!