Inheritance – Wrap up
Polymorphism in Java

CS 171: Intro to Computer Science II

`final` Keyword

- `final` variables
  - Initialized as a part of declaration but never can get a new value (i.e. a constant)
  - A final field of a class will almost always be static (wasteful to have every instance store an identical value!)

- `final` methods
  - Cannot be overridden by a subclass

- `final` classes
  - Cannot be subclassed

We learned many keywords recently, let’s review some of them.

Take out your phones (or laptops!)

We learned many keywords recently, let’s review some of them.

Take out your phones (or laptops!)

pollev.com/elsayedcs171

pollev.com/elsayedcs171

Question

- A variable declared as `static` inside a Java class:

  A. can never be *updated* and must always have the same value throughout program execution.

  B. is *unique* for each object created of that class.

  ✓ C. gets allocated memory once at the time of class loading, and is used as a common property of all objects of that class.

pollev.com/elsayedcs171

Question

For the shown Dog and Hound classes, what's the result of running:

```
Hound h = new Hound();
h.crazy();
```

✓ A. bowl woof

B. woof bowl

C. bowl bowl

D. woof woof

```java
class Dog{
  public void bark(){ System.out.print("woof "); }
}

class Hound extends Dog{
  public void bark(){ System.out.print("bowl "); }
  public void crazy(){
    this.bark();
    super.bark();
  }
}
```

Object-Oriented Programming

Defining Polymorphism

pol·y·mor·phism

/ˌpälēˈmôrfizəm/

noun

the condition of occurring in several different forms.
"the complexity and polymorphism of human cognition"

In Java:
the ability of a reference variable to take different forms

But first let's talk about "Inheritance Hierarchies" and "is-a" relationship

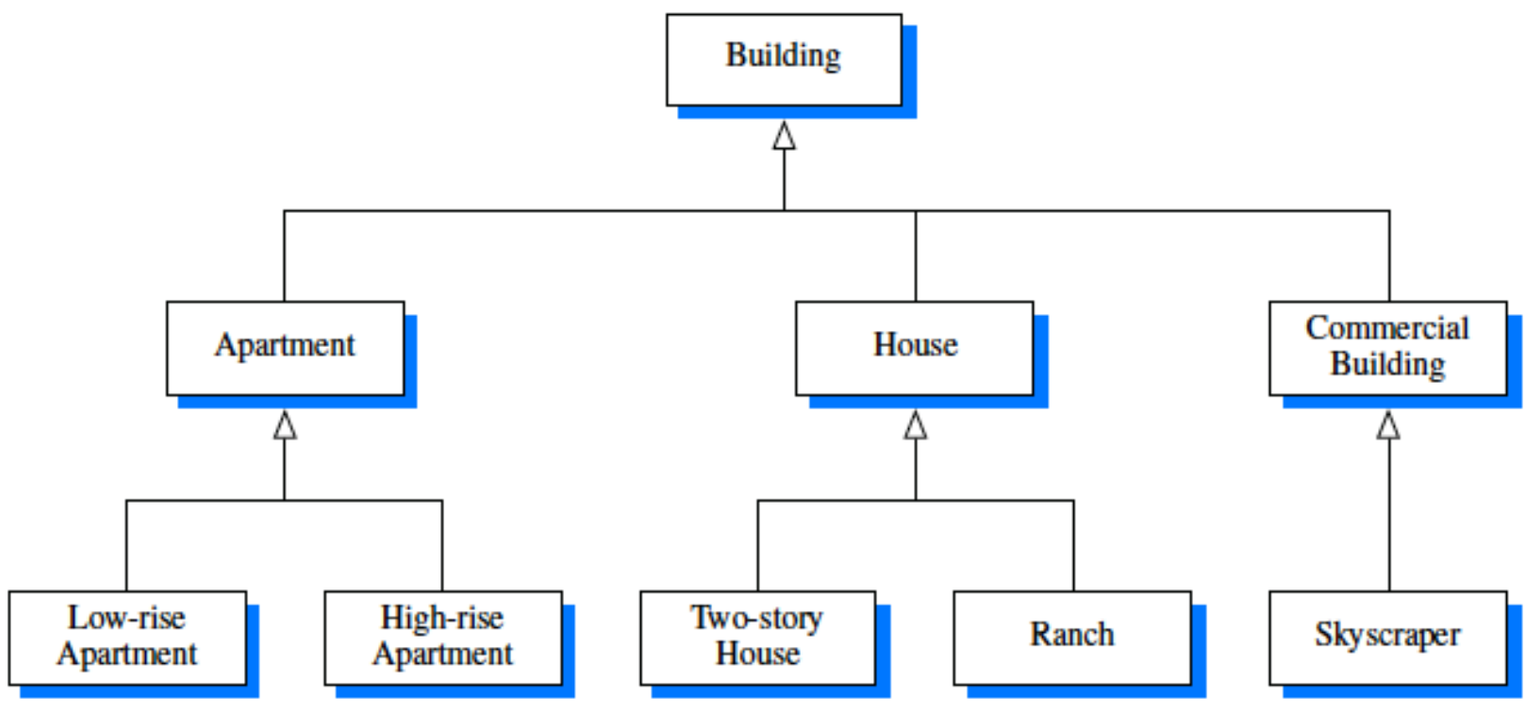
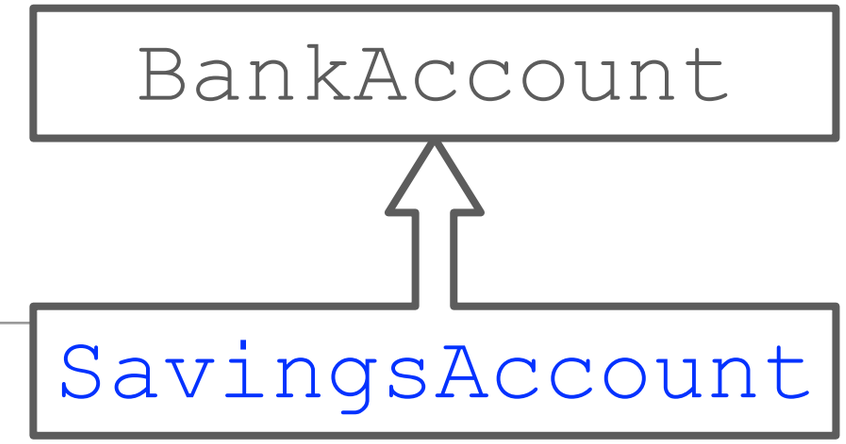


Figure 2.3: An example of an "is a" hierarchy involving architectural buildings.

- An *apartment* is a *building*.
- A *ranch* is a *house*.
- A *ranch* is a *building*.
- A *building* is NOT (always) a *ranch*.
- A *house* is NOT a *skyscraper*.

Example



- A SavingsAccount **is a** BankAccount

- This is allowed Java code:

```
BankAccount accnt = new SavingsAccount();
```

- This is NOT allowed Java code:

```
SavingsAccount accnt = new BankAccount();
```

Casting

- Recall: an int can be cast as a double:

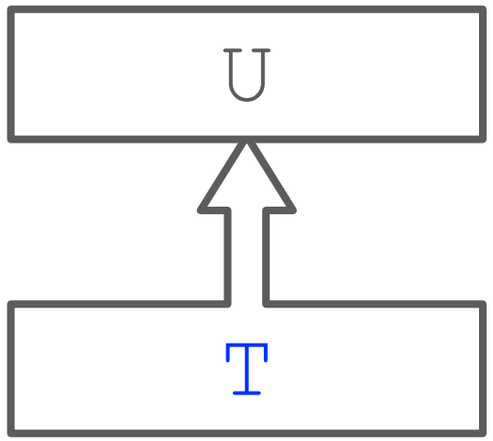
```
int x = 5;
double y = (double)x;
```

- But not the other way around:

```
double x = 5;
int y = (int)x; // loss of precision!
```

Two types of conversions in Java

- **Widening Conversion:**
  - type T converted to wider type U
    where U is a superclass of T
  - Example: BankAccount x = new SavingsAccount();
  - Correctness checked by the compiler



- **Narrowing Conversion (NOT recommended!):**
  - Type U converted to *narrower* type T
    where T is a subclass of U
  - Requires an explicit cast!
  - Example: SavingsAccount save = (SavingsAccount) x;
  - Correctness NOT checked by the compiler;
    can lead to runtime errors!
  - Should avoid doing this when possible…

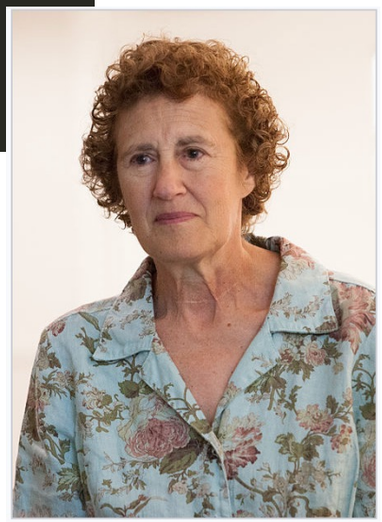
Liskov Substitution Principle

- Says a variable of a declared type can be assigned an instance from any direct or indirect subclass of that type

- If type A "**is a**" type B, an instance of type A can be assigned to a variable of type B

- But not the other way around!

```
BankAccount x = new SavingsAccount(); //valid
SavingsAccount x = new BankAccount(); //NOT valid!
```



Liskov substitution was introduced by Barbara Liskov, photo taken in 2010

Polymorphic Variables

- `BankAccount x = new CheckingAccount();`
- `x` is called a *polymorphic variable*
- `x` references a `CheckingAccount` object
- But because it was declared as a `BankAccount` type, it can only call methods declared in the `BankAccount` definition!

```
BankAccount x = new CheckingAccount();
x.deposit(100); //valid
x.deductFees(); //NOT valid!
```

Question

Given that class `SalariedEmployee` extends class `Employee`, which of the following are allowed in Java?

A. `Employee emp = new Employee("Kiko");`
`SalariedEmployee sEmp = (SalariedEmployee) emp;`

B. `SalariedEmployee sEmp = new SalariedEmployee("Ron", 100.0);`
`Employee emp = sEmp;`

C. `SalariedEmployee sEmp = new SalariedEmployee("Ron", 100.0);`
`Employee emp = (Employee) sEmp;`

D. A and B
✓ E. B and C
F. A and C
G. A, B, and C

Dynamic Dispatch

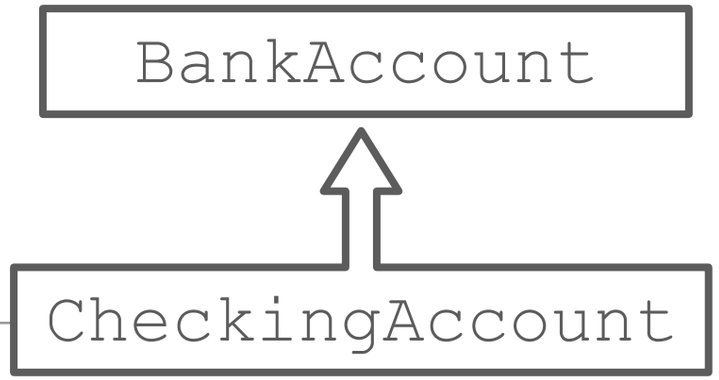
- What about methods of the `CheckingAccount` class that override methods of the `BankAccount` class? Which version is used?

- *Dynamic dispatch*: Java makes a runtime decision to call the method version associated with the actual type of the referenced object (not the declared type)

```
BankAccount x = new CheckingAccount();
```

Q: which version of method `deposit` will `x.deposit(100)` invoke?
Answer: The `CheckingAccount` version!

instanceof



- Operator that tests at runtime if an instance satisfies a particular type

```
BankAccount x = new BankAccount();
boolean b = x instanceof BankAccount; //true
b = x instanceof CheckingAccount; //false

x = new CheckingAccount();
b = x instanceof CheckingAccount; //true
b = x instanceof BankAccount; //true
```

Rule: `x instanceof ClassName` evaluates to `true` if `x` references an object belonging to the `ClassName` class or any further subclass of `ClassName`

pollev.com/elsayedcs171

Question

Given that class `SalariedEmployee` extends class `Employee`:

```
Employee emp = new Employee("Kiko");
SalariedEmployee sEmp = new SalariedEmployee("Ronaldo",100.0);
```

True or False: `sEmp instanceof SalariedEmployee`

✓ A. true
B. false

pollev.com/elsayedcs171

Question

Given that class `SalariedEmployee` extends class `Employee`:

```
Employee emp = new Employee("Kiko");
SalariedEmployee sEmp = new SalariedEmployee("Ronaldo",100.0);
```

True or False: `sEmp instanceof Employee`

✓ A. true
B. false

pollev.com/elsayedcs171

Question

Given that class `SalariedEmployee` extends class `Employee`:

```
Employee emp = new Employee("Kiko");
SalariedEmployee sEmp = new SalariedEmployee("Ronaldo",100.0);
```

True or False: `emp instanceof Employee`

✓ A. true

B. false

pollev.com/elsayedcs171

Question

Given that class `SalariedEmployee` extends class `Employee`:

```
Employee emp = new Employee("Kiko");
SalariedEmployee sEmp = new SalariedEmployee("Ronaldo",100.0);
```

True or False: `emp instanceof SalariedEmployee`

A. true
✓ B. false

Example: PolymorphicEmployee.java

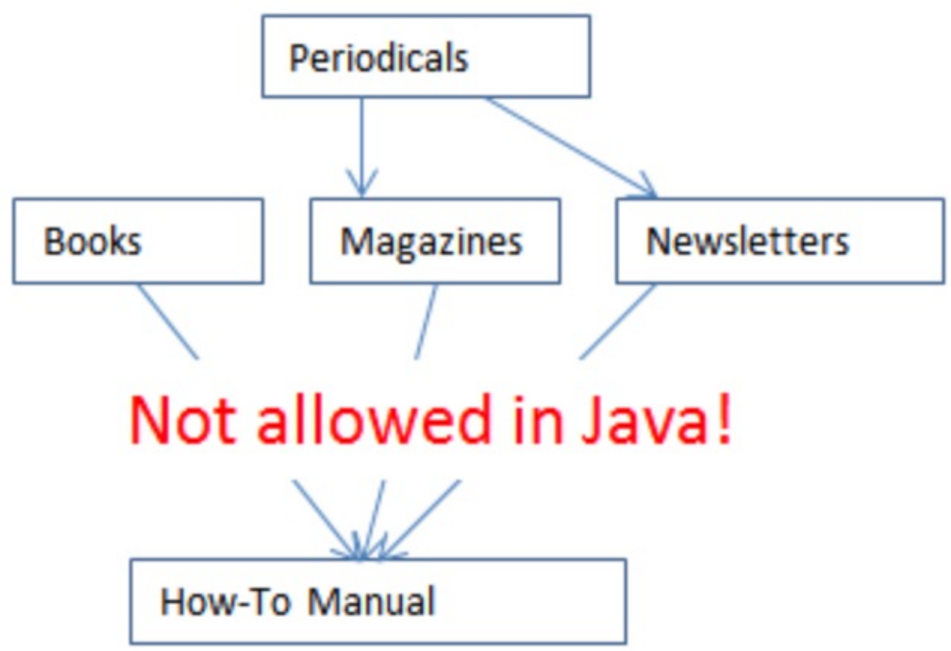
Me: *explains polymorphism*

Friend: So the subclass the same thing as the superclass?

Me:

Limitations of Inheritance

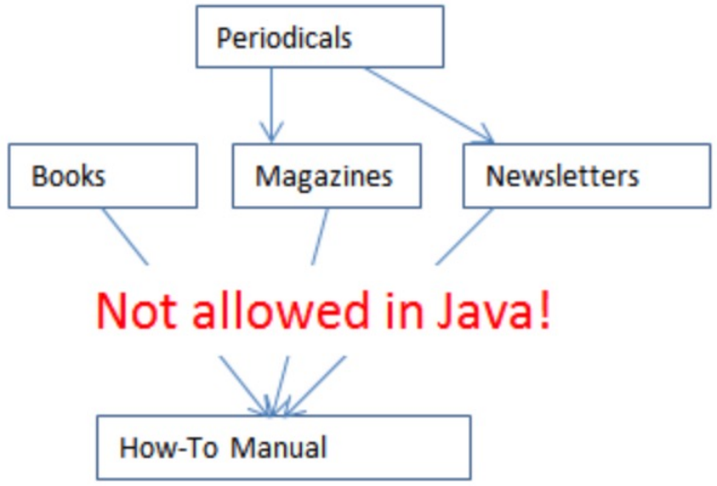
- Can I inherit from more than one parent class?



Inheritance: Limitations [1/2]

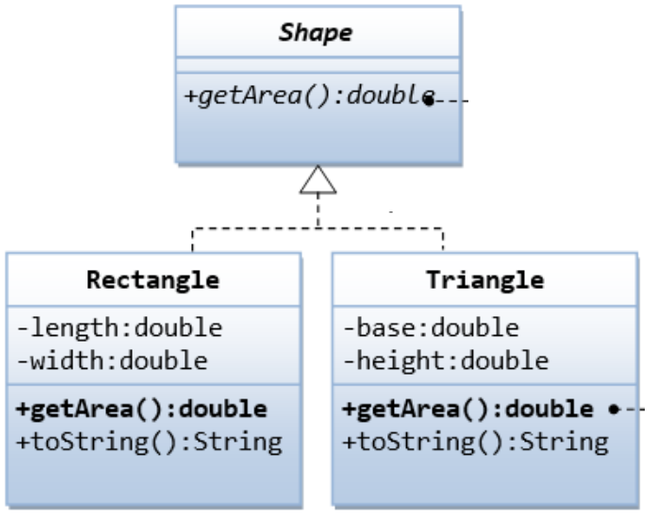
- What if I want to inherit from more than one parent class?
  - In biology: possible

  - In Java: prohibited
  - You can’t inherit from multiple classes



Inheritance: Limitations [2/2]

- What if we don't want to allow creation of superclass objects?
- E.g.: Bank doesn't want customers to open a general BankAccount!
- Or, only *particular* shapes makes sense, not some *generic* shape



Shape sh = new Shape();

Rectangle re = new Rectangle();

Triangle tr = new Triangle();

Solution: "Abstract" Classes!

Review: Object-Oriented Programming

First: Abstract methods

- In Java: You can declare a method *without* defining it

```
public abstract double getArea( );
```

➔ Notice the body of the method { … } is missing!

- A method that has been declared but not *defined* is an abstract method

Abstract class

- Any class containing *one or more* abstract methods is an abstract class

- You must declare the class with the keyword **abstract**:

```
abstract class MyClass {...}
```

- An abstract class is incomplete!

  ➔ Has “missing” method bodies

  ➔ You cannot instantiate (create a new instance of) an abstract class

Why have abstract classes?

- You can **extend** (subclass) an abstract class
  - If subclass defines all inherited abstract methods, it is "complete" and can be instantiated

```
CompleteSubClass sc = new CompleteSubClass ();
```

  - If subclass does not define all inherited abstract methods, it too must be abstract (& can't be instantiated!)

It gets more interesting…

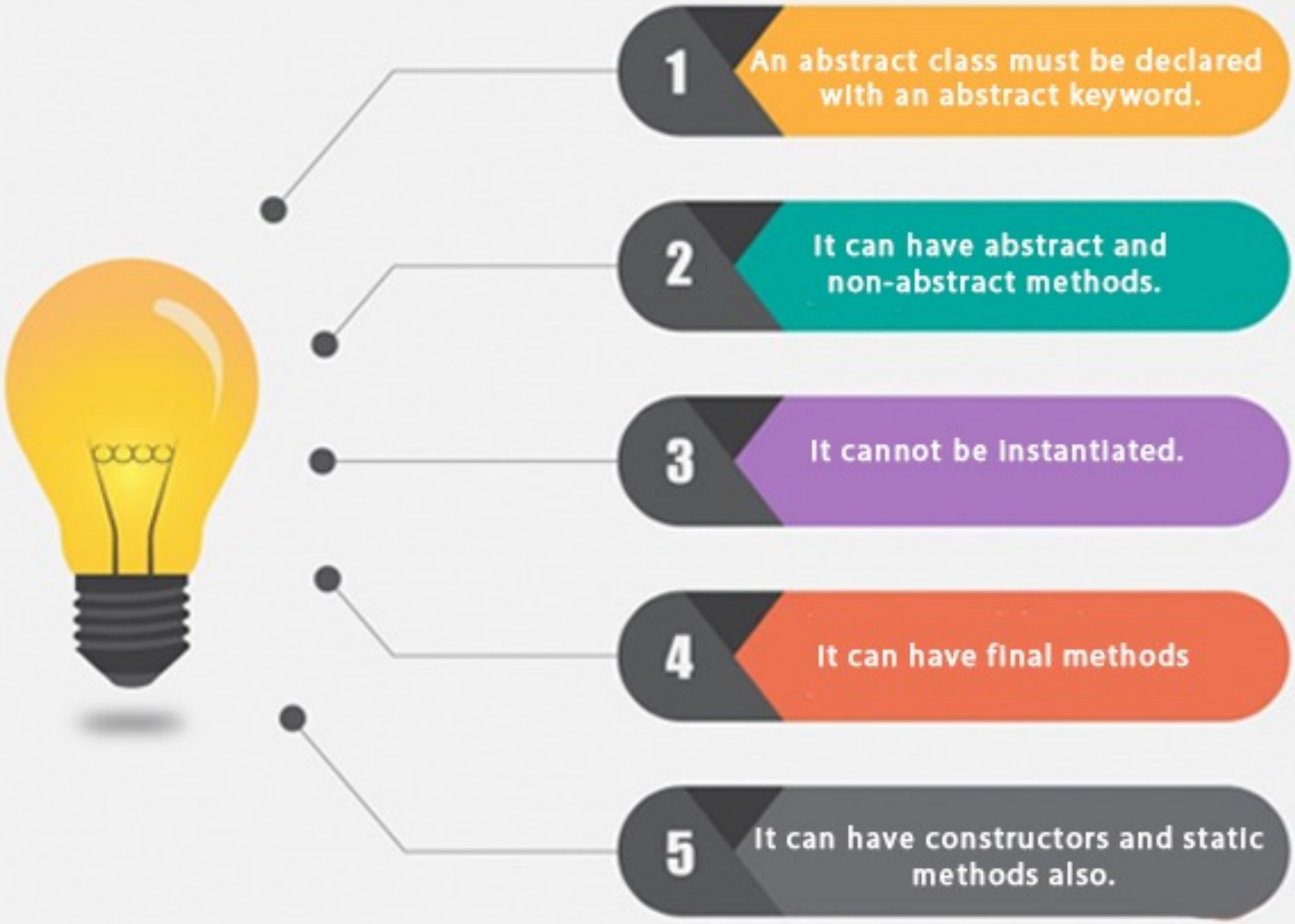
- You can declare a class to be **abstract** even if it doesn't contain any abstract methods!

```
SomeAbstractClass sc = new SomeAbstractClass ();
```

  ➔ Prevents the class from being instantiated!

Think about using abstract classes when something needs to be there but not exactly sure how objects should look

Rules for Java Abstract class



https://www.javatpoint.com/abstract-class-in-java

Example: An Abstract Car

Cannot instantiate an AbstractCar object

```java
public abstract class AbstractCar{
    private String brand;
    private String model;
    private String color;
    private double mileage;

    public AbstractCar(String b, String m, String c){
        brand = b;
        model = m;
        color = c;
        mileage = 0;
    }

    public String toString(){
        return brand + " model " + model + " in " + color;
    }

    public void run(double miles){
        mileage += miles;
    }

    public double getMileage(){
        return mileage;
    }

    public abstract void driveSelf(double miles);
}
```

Honda: Extending AbstractCar

```java
public class Honda extends AbstractCar {
    public Honda(String model, String color) {
        super("Honda", model, color);
    }

    public void driveSelf(double miles) {
        System.out.println("Driving an awesome Honda");
        run(miles);
        System.out.println("Number of miles on the car:" + getMileage());
    }

    public static void main(String[] args) {
        AbstractCar accord = new Honda("Accord", "Green");
        AbstractCar crv = new Honda("CRV", "Blue");

        accord.driveSelf(100);
        crv.driveSelf(20);
    }
}
```

```java
public abstract class AbstractCar {
    private String brand;
    private String model;
    private String color;
    private double mileage;

    public AbstractCar(String b, String m, String c) {
        brand = b;
        model = m;
        color = c;
        mileage = 0;
    }

    public String toString() {
        return brand + " model " + model + " in " + color;
    }

    public void run(double miles) {
        mileage += miles;
    }

    public double getMileage() {
        return mileage;
    }

    public abstract void driveSelf(double miles);
}
```

```java
public class Honda extends AbstractCar {
    public Honda(String model, String color) {
        super("Honda", model, color);
    }

    public void driveSelf(double miles) {
        System.out.println("Driving an awesome Honda");
        run(miles);
        System.out.println("Number of miles on the car:" + getMileage());
    }

    public static void main(String[] args) {
        AbstractCar accord = new Honda("Accord", "Green");
        AbstractCar crv = new Honda("CRV", "Blue");

        accord.driveSelf(100);
        crv.driveSelf(20);
    }
}
```

- Q: Can `AbstractCar` be instantiated?
  - No (it's an abstract class)
- Can `Honda` be instantiated?
  - Yes

What if <u>all</u> methods in a class are abstract?

Declare it as <u>**interface**</u>

Interface

- Implies 100% abstraction (no implemented methods)
- Reference type similar to class (but not a class!)
- Collection of abstract methods or group of related methods with empty bodies
- Can contain variables but must be static and final (can't be changed)
- Cannot be instantiated or contain any constructors

Example: Interface

```
interface FooInterface {
    /* All the methods are public abstract by default
     * Note the methods below have no body
     */
    public void foo1();
    public void foo2();
}
```

Interfaces tell the world: what is the functionality (behavior) that should be offered -- irrespective of underlying implementation