Java
Objects and inheritance

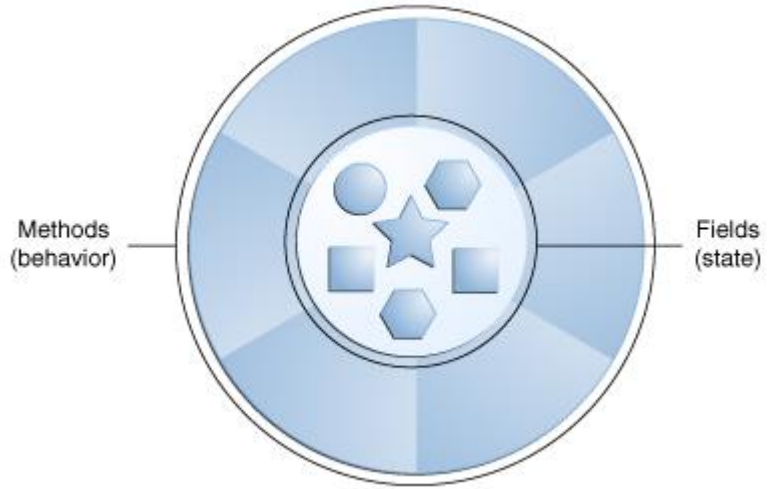
Objects

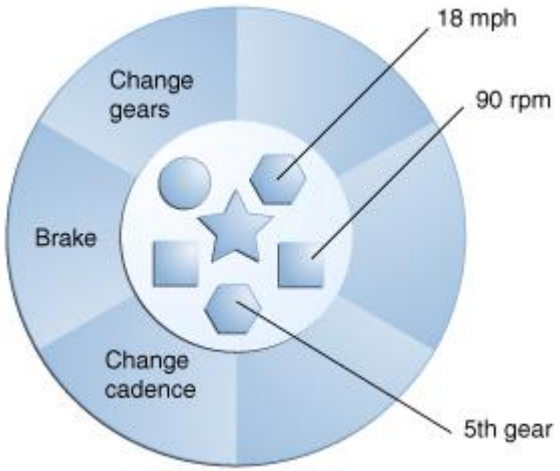
Characteristics of **objects** (real-world or software)

- State
- Behaviour

Why program with objects?

- Modularity
- Information-hiding
- Code re-use
- Pluggability and debugging ease





Java Tutorial "What is an Object?"

Classes vs. objects

Classes are **types**

Objects are **instances of types**

An object is an **instance** of a general **class** of objects

In other words, a class is a **blueprint** from which individual objects are created

Example 1: `boolean` primitive type has instance values **true** and **false**

Example 2: imagine a `Dog` class

Instances: toto, lassie, brianGriffin, scoobyDoo, …

Class declaration in Java

Use the `class` keyword

Give the class a name (use *CamelCase*)

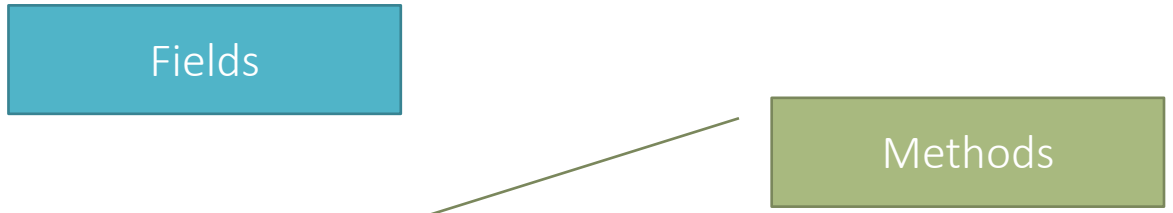
Specify class body inside curly brackets

- Fields (properties)
- Methods (behaviours)

(Optional other things: access modifier(s), superclass, interface(s) – we will address these later in the course)

Example class: bank account

```
class BankAccount {

  int balance;
  String name;

  void deposit(int value) { this.balance += value; }

  void withdraw(int value) { this.balance -= value; }

}
```



Class members: fields and methods

Data fields:

Store state that represent some attributes of the object

For `Dog` class: name, breed, size, age, …

Methods:

Represent behaviour that processes and transforms the object state

For `Dog` class: eat(), sleep(), goForWalk(), …

Special method: `public static void main (String[] args)`

If a class has a `main` method, then you can run it directly (Eclipse: “Run as Java application”)

Fields in Java

Three types of variables:

- Local variables (declared in a method)
- Method parameters (in a method header)
- Member variables in a class – also known as **fields**

Field declarations look the same as local variable declarations, but occur **outside any methods**

An instance variable is accessible in **all methods of a class**

Bank account class revisited

```
class BankAccount {

   int balance;
  String name;

   void deposit(int value) { this.balance += value; }

   void withdraw(int value) { this.balance -= value; }

   BankAccount(String name, int initialAmount) {

      this.name = name;
     this.balance = initialAmount;

   }
}
```

Fields

Methods

Constructor

Constructors

```
BankAccount(String name, int
initialAmount)
{
    this.name = name;
    this.balance = initialAmount;
}
```

Looks like a method with the **same name as the class**

No return type specified (not even `void`)

Sets up initial values for the data fields to initialise object state

Using `this` keyword to refer to current object being created

Use `new` keyword to create a new object:

```
BankAccount b = new BankAccount ("Mary", 0);
```

If no constructor is specified, a default *no-args* constructor is automatically created

No arguments

Sets all fields to their default values (usually 0 or `null`)

Question

For the `BankAccount` class …

How do we stop other code from directly modifying the fields of individual `BankAccount` objects?

```
class BankAccount {
  int balance;
  String name;
  void deposit(int value) { this.balance += value; }
  void withdraw(int value) { this.balance -= value; }
  BankAccount(String name, int initialAmount) {
    this.name = name;
    this.balance = initialAmount;
  }
}
```

```
BankAccount b = new BankAccount("Mary", 1000);
b.value += 1000;
b.name = "Eve";
```

Visibility modifiers

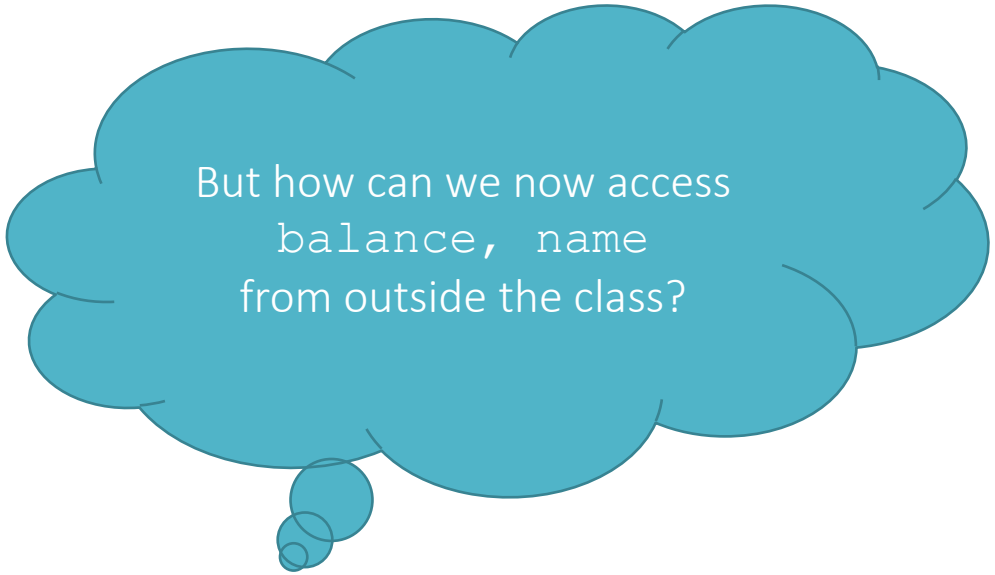
Modifier	Same class	Same package	Any subclass	Any class
`public`	●	●	●	●
`protected`	●	●	●	
(default)	●	●		
`private`	●			

Used to limit the visibility of class members (fields and methods)

Specify as part of member declaration – `private int balance;`

Bank account revisited (again)

```
public class BankAccount {

  private int balance;
  private String name;

  public void deposit(int value) { this.balance += value; }

  public void withdraw(int value) { this.balance -= value; }

  public BankAccount(String name, int initialAmount) {

    this.name = name;
    this.balance = initialAmount;
  }
}
```



Getters and setters

Give controlled access to private properties

```
public String getName() {
      return name;
}

public void setName (String name) {
      this.name = name;
}

// … and so on
```

Bank account class again – added ID field

```
public class BankAccount {

   private int balance;
   private String name;
   private int id;

   private static int NEXT_ID = 0;

   public BankAccount(String name, int initialAmount) {

      this.name = name;
      this.balance = initialAmount;
      this.id = BankAccount.NEXT_ID++;

   }
}
```

Static members

Associated directly with the class itself, not with any object of that class

Static field: only one variable, no matter how many objects of the class have been created (even zero)

Examples you have seen: `Double.MAX_VALUE`, `System.out`

Static method: performs a general task for the class; can only access other static members

Examples you may have seen: `Math.random()`, `Integer.parseInt()`

In updated `BankAccount` class, `NEXT_ID` field is static (why?)

Inheritance

Objects (world or software) have some features in common

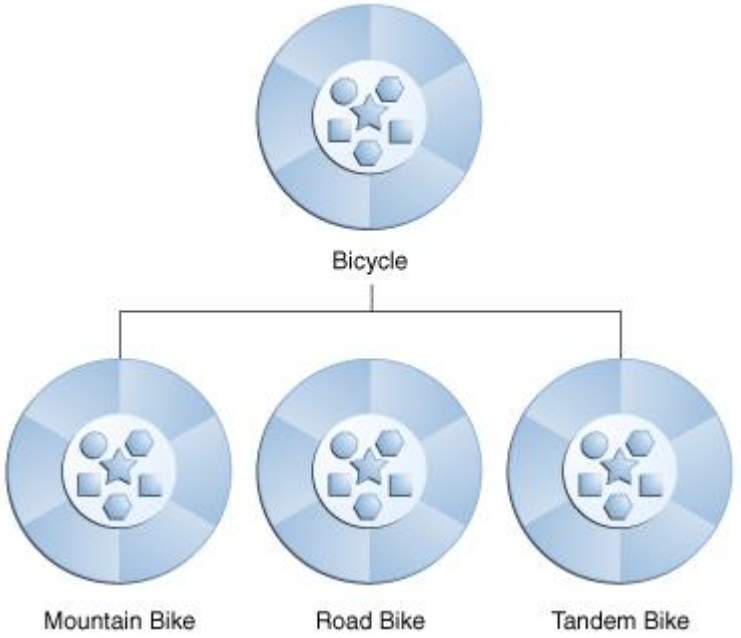
In OO programming, classes can **inherit** state and behaviour (fields and methods) from other classes

- Subclass is a **specialised version** of the superclass

In Java, a class can have **exactly one** superclass

- If superclass isn't specified, then it inherits from `Object`

Subclasses can **override** superclass methods to provide specialised behaviour



Java Tutorial "What is Inheritance?"

Inheritance in Java

```
public class Animal {
    protected String name;

    public void move() {
        System.out.println(name
              +  “can move”);
    }
    public String getName() {
        return this.name;
    }
}
```

```
public class Dog extends Animal {
    private String breed;

    public void move() {
        System.out.println(name
            + “can walk and run”);
    }
}
```

What you can do in a subclass (Fields)

Use the inherited fields just like other fields *(except if they are private)*

Declare a field in the subclass with the same name as in the superclass

This is known as **hiding** the parent field (not recommended!)

Declare new fields that are not in the superclass

What you can do in a subclass (Methods)

Use the inherited methods directly *(unless they are private)*

Override an instance method by writing a new method with the same signature

Hide a static method by writing a new method with the same signature

Declare new methods that are not in the superclass

Method overriding

If a subclass has an instance method with an **identical** signature (name, parameters, return type*) as a parent class …

… then the subclass method **overrides** the parent method

This is a method for specifying alternative (specialised) behaviour for the subclass

```
public class Animal {
    public void move() {
        System.out.println("Animals can move");
    }
}

public class Dog extends Animal {
    public void move() {
        System.out.println("Dogs can walk and run");
    }
}
```

https://www.tutorialspoint.com/java/java_overriding.htm

* Actually, the subclass method's return type could also be a subclass of the parent class method's return type – this is known as a **covariant return type**.

Polymorphism

Literal meaning: “many forms”

In OO design: whenever an instance of class A is expected, you can also use an instance of any subclass of A

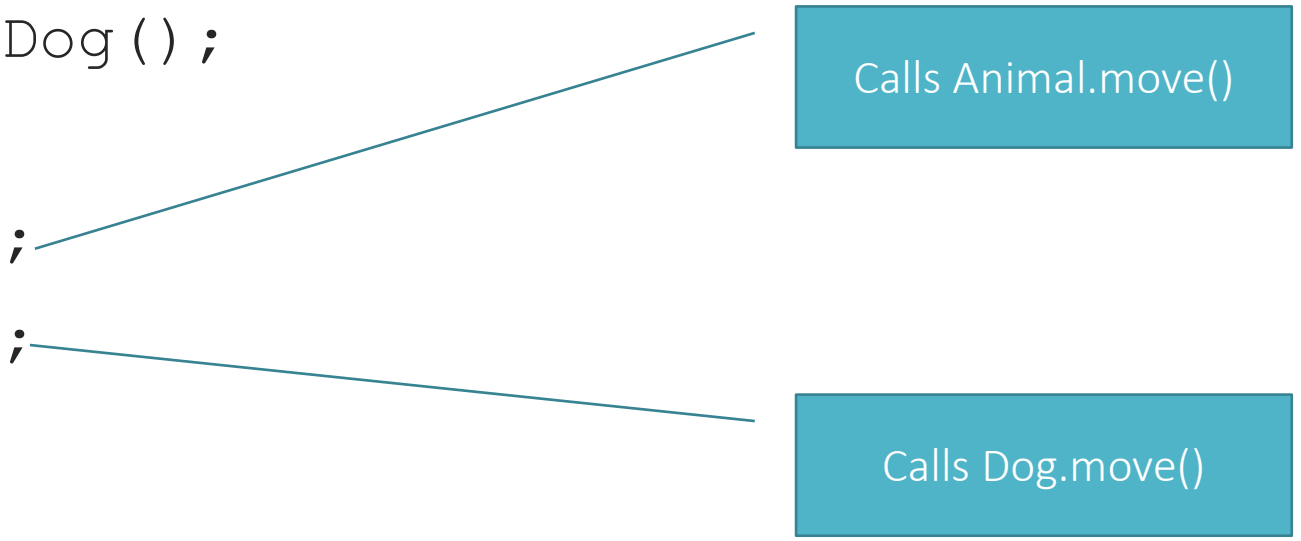
If a method is overridden in a subclass, Java will always use the most-specific overridden version

Even when the variable type is the superclass

Supported by **virtual method invocation**: method calls are dynamically dispatched based on the **runtime** type of the receiver object

Polymorphism example

```
Animal a1, a2;
a1 = new Animal();
a2 = new Dog();

a1.move();
a2.move();
```



Built-in methods

All objects inherit some methods from the root class `java.lang.Object`

- `toString()`: generates a `String` representation of the object
- `equals()`: compares two objects for equality and returns a `boolean` result
- `hashCode()`: returns an integer associated with the class for use in more complex data structures

You can override these methods to provide class-specific behaviour

toString() for BankAccount

Default behaviour: prints the class and the memory address of the BankAccount (rarely useful)

Overridden behaviour: prints the information in the relevant bank account fields

```
public String toString() {
    return "Account #" + id
           + " Name=" + name
           + ", balance="
           + balance;
}
```

Default (“no-args”) constructor

If you do not specify a constructor for your class, a default constructor is created

Properties of the default constructor

- Takes no parameters (“no-args”)
- Sets all fields to default values (0 for numeric types, false for Boolean, null for non-primitive types)

This constructor is **only created if you do not specify your own constructor**

```
public Bicycle() {
    this.gear = 0;
    this.cadence = 0;
    this.speed = 0;
}
```

Constructors and inheritance

Recall:

- Constructors look like a method with same name as class; no return type
- If no constructor is specified, a default **no-args** constructor is created

What about inheritance?

- "Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass." *(Java Tutorial)*

Also:

- "If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the super class does not have a no-argument constructor, you will get a compile-time error. `Object` does have such a constructor, so if `Object` is the only superclass, there is no problem."

Constructors and inheritance

A subclass **does not inherit** the superclass's constructors

… But can call them from its own constructor using `super`

```
public class MountainBike extends Bicycle {
    public int seatHeight;

    public MountainBike(int seatHeight, int cadence, int speed, int gear) {
        super(cadence, speed, gear);
        this.seatHeight = seatHeight;
    }
}
```

If this line wasn't here, Java would automatically insert

`super();`

This would not compile!!!