1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Day 3: Basics of object oriented programming (AI, CS)

Wednesday, 1 November / Thursday, 2 November

JC2002 Java Programming

Day 3, Session 1: Objects and classes

Object oriented programming (OOP)

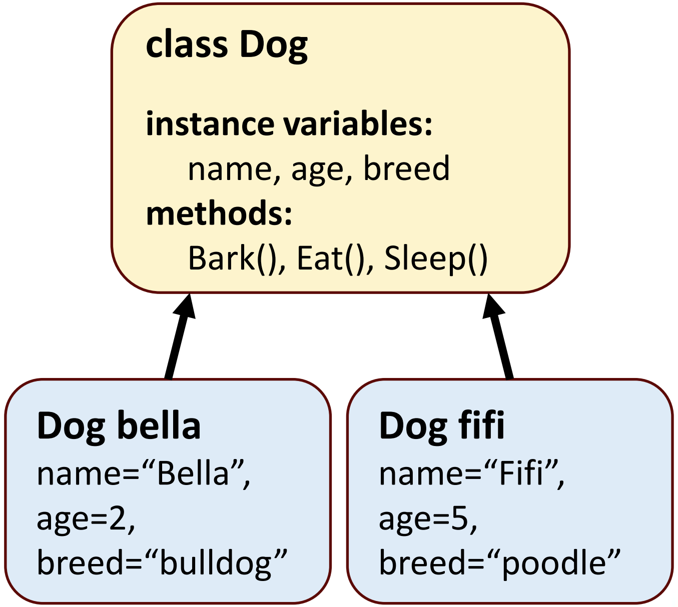
- Today, we will cover the fundamentals of object oriented programming (OOP) in Java
    - Basic concepts of classes and objects
    - Instance variables, set and get methods
    - Scope and access modifiers
    - Enum types
    - Inheritance, composition, and polymorphism
- Much of the material is based on slides from ***Java: How to Program***, chapter 7, available via MyAberdeen

Learning objectives

- After the theory sessions today, you should be able to:
  - Explain the basic concepts of classes and objects
  - Declare classes and instantiate objects in your Java programs
  - Select appropriate access modifiers for your classes
  - Use inheritance and composition in your Java programs

Concepts of classes and objects

- ***Class*** is a data structure that represents a category of objects with some shared characteristics
  - Class can include *instance variables* defining its state, as well as *methods* implementing its behavior
- ***Object*** is an instance of a class
  - For example, class "Person" represents human beings, and object "John" is an instance of class "Person", representing a specific person



class Dog

instance variables:
name, age, breed

methods:
Bark(), Eat(), Sleep()

Dog bella
name="Bella",
age=2,
breed="bulldog"

Dog fifi
name="Fifi",
age=5,
breed="poodle"

Classes and objects in Java

- In Java, you can declare new classes as needed; this is one reason Java is known as an *extensible* language
- Each class you create becomes a new type that can be used to declare variables and create objects
  - By convention, class names, method names and variable names are all identifiers and all use the camel-case naming scheme
  - Also, by convention, class names begin with an initial uppercase letter, and method names and variable names begin with an initial lowercase letter
  - Note that these conventions are *not* forced by Java syntax; however, it is highly recommended to follow them

Instance variables

- An object has attributes that are implemented as instance variables and carried with it throughout its lifetime
- Each object (instance) of the class has its own copy of each of the class’s instance variables
- Instance variables are declared inside a class declaration but outside the bodies of the class’s method declarations
- A class normally contains one or more methods that manipulate the instance variables that belong to particular objects of the class

Getter and setter methods

- By convention, we use *set* and *get* methods to store / obtain instance variable values (i.e., attributes) in an object
    - If variable is defined as private, it is not possible to access directly
    - If variable is defined as public, it can be accessed directly, but even then, it is best to use set and get methods to modify the variable
- *Set* methods are commonly called *mutator methods*
- *Get* methods are commonly called *accessor methods* or *query methods*

Get and Set example

Account.java

```java
public class Account {
  private String name; // instance variable
  // method to set the name
  public void setName(String name) {
    this.name = name;
  }
  // method to retrieve the name
  public String getName() {
     return name; // return name value
  }
}
```

AccountTest.java

```java
import java.util.Scanner;
public class AccountTest {
  public static void main(String[] args) {
    // create a Scanner object for input
    Scanner input = new Scanner(System.in);
    // create an Account object myAccount
    Account myAccount = new Account();
    // display initial value of name (null)
    System.out.printf("Initial name is: %s%n%n",
      myAccount.getName());
    // prompt for and read name
    System.out.println("Please enter the name:");
    String theName = input.nextLine();
    myAccount.setName(theName);
    System.out.println(); // outputs a blank line
    // display the name stored in object myAccount
    System.out.printf("Name in myAccount is:%n%s%n",
      myAccount.getName());
  }
}
```

```
Initial name is: null

Please enter the name:
Jane Green

Name in object myAccount is:
Jane Green
```

Get and Set example

Account.java

```java
public class Account {
  private String name; // instance varia
  // method to set the name
  public void setName(String name) {
    this.name = name;
  }
  // method to retrieve the name
  public String getName() {
     return name; // return name value
  }
}
```

Setter takes one parameter, return is void

Getter takes no parameter, return is String

AccountTest.java

```java
import java.util.Scanner;
public class AccountTest {
  public static void main(String[] args) {
    // create a Scanner object for input
    Scanner input = new Scanner(System.in);
    // create an Account object myAccount
    Account myAccount = new Account();
```

```java
of name (null)
tial name is: %s%n%n",
;
d name
ease enter the name:");
      String theName = input.nextLine();
ame);
// outputs a blank line
tored in object myAccount
e in myAccount is:%n%s%n",
;
    }
  }
```

```
Initial name is: null

Please enter the name:
Jane Green

Name in object myAccount is:
Jane Green
```

UNIVERSITY OF ABERDEEN

Access modifiers (public and private)

- Most instance-variable declarations are preceded with the keyword `private`, which is an access modifier
- Variables or methods declared with access modifier `private` are accessible only to methods of the class in which they're declared
- Declaring instance variables with access modifier private is known as *information hiding*
    - When a program creates (instantiates) an object of class *Account*, variable *name* is encapsulated (hidden) in the object and can be accessed only by methods of the object's class

Method’s local variables

- Parameters of a method are *local variables* of the method
  - Local variables declared in the body of a particular method can be used *only* in that method
  - When a method terminates, the values of its local variables are lost
  - Local variables are not automatically initialized
- If a method contains a local variable and instance variable with the same name, the method’s body will refer to the local variable rather than the instance variable
  - Local variable *shadows* the instance variable in the method’s body.
  - Keyword **`this`** can be used to refer to the shadowed instance variable explicitly

Using keyword *this*

```
1   public class Account {
2         private String name; // instance variable
3
4         // method to set the name in the obj
5         public void setName(String name) {
6               this.name = name; // store the
7         }
8
9         // method to retrieve the name from
10        public String getName() {
11              return name; // return value of name to caller
12        }
13  }
```

We could have avoided the need for keyword **this** by choosing different parameter name on line 5, but using **this** keyword is a widely accepted practice.

UNIVERSITY OF ABERDEEN

More about keyword *this*

- Every object can access a reference to itself with keyword **`this`** (sometimes called the **`this`** reference)
- When an instance method is called for a particular object, the method’s body implicitly uses keyword **`this`** to refer to the object’s instance variables and other methods
    - Therefore, the class’s code knows which object should be manipulated
- There is only one copy of each method per class; every object of the same class shares the method’s code
- On the other hand, each object has its own copy of the class’s instance variables, and the non-static methods implicitly use **`this`** to determine the specific object to manipulate

Instantiating an object

- A class instance (object) is created using keyword **`new`**
- A *constructor* is similar to a method, but it is called implicitly by the **`new`** operator to initialize an object's instance variables when the object is created
  - If a class does not define a constructor, the compiler provides a default constructor with no parameters, and the class's instance variables are initialized to their default values
  - Every instance variable has a default initial value (a value provided by Java) if you do not specify the initial value
  - The default value for an instance variable of type `String` is `null`

Constructor example

- In this example, instance variable name is set using the constructor, so we do not need to call `setName` after creating the object

Account.java

```java
public class Account {
  private String name; // instance variable
  // constructor initializes name
  public Account(String name) {
    this.name = name;
  }
  // method to set the name
  public void setName(String name) {
    this.name = name;
  }
  // method to retrieve the name
  public String getName() {
     return name; // return name value
  }
}
```

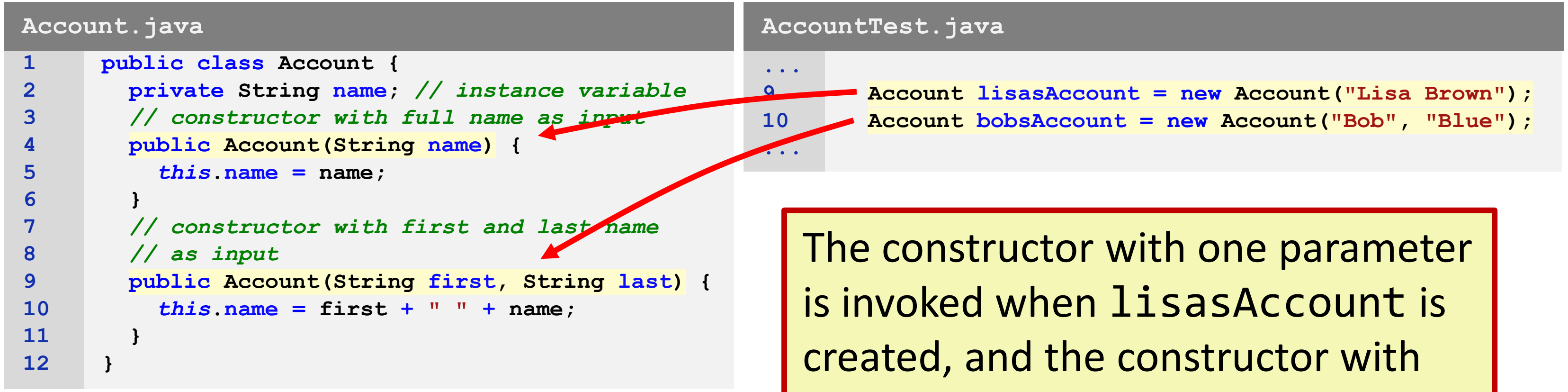
AccountTest.java

```java
...
    System.out.println("Please enter the name:");
    String theName = input.nextLine();
    Account myAccount = new Account(theName);
...
```

The constructor is a method with the same name as the class. It is invoked when an object is instantiated using the keyword new.

Constructor overloading example

- *Overloaded constructors* allow different ways to initialise objects
  - Only the parameters for the constructors are different



Account.java

```java
public class Account {
  private String name; // instance variable
  // constructor with full name as input
  public Account(String name) {
    this.name = name;
  }
  // constructor with first and last name
  // as input
  public Account(String first, String last) {
    this.name = first + " " + name;
  }
}
```

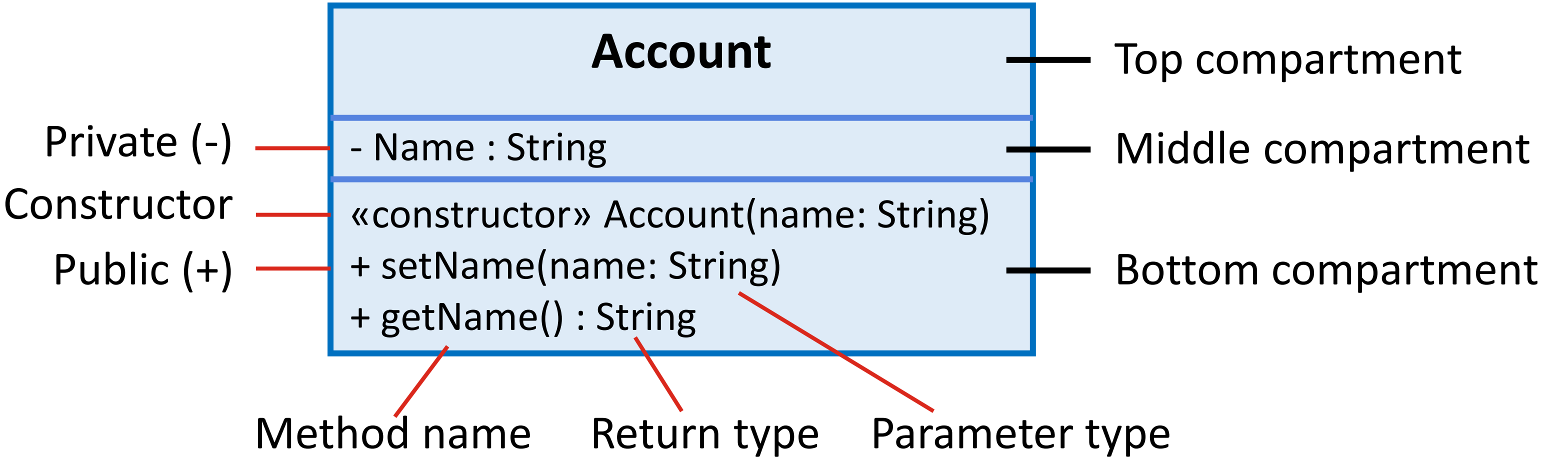
AccountTest.java

```java
...
Account lisasAccount = new Account("Lisa Brown");
Account bobsAccount = new Account("Bob", "Blue");
...
```

The constructor with one parameter is invoked when `lisasAccount` is created, and the constructor with two parameters is invoked when `bobsAccount` is created.

UML class diagram

- UML class diagrams are often used to illustrate classes



Account

- Name : String

«constructor» Account(name: String)
+ setName(name: String)
+ getName() : String

Top compartment

Middle compartment

Bottom compartment

Private (-)

Constructor

Public (+)

Method name

Return type

Parameter type

Questions, comments?

JC2002 Java Programming

Day 3, Session 2: Enum types, static, and final

Enum types and keywords static and final

- What are enum types?
  - Enum declaration
- Keyword static
  - Static class members
  - Static import
- Keyword final
  - Principle of least privilege
  - Final instance variables
- Much of the material is based on slides from ***Java: How to Program***, chapter 8, which is available via MyAberdeen

What are enum types?

- Like classes, all `enum` types are reference types
- The basic `enum` type defines a set of constants represented as unique identifiers
- For every `enum`, the compiler generates the `static` method **`values()`** that returns an array of the `enum`’s constants
- The `enum` constants can be used anywhere constants can be used, such as in the `case` labels of `switch` statements and to control enhanced `for` statements

Enum declaration

- An `enum` type is declared with an *enum declaration*, which is a *comma-separated* list of enum *constants*
- The declaration may optionally include other components of traditional classes, such as constructors, fields and methods
  - An `enum` constructor can specify any number of parameters and it can be overloaded
- Each `enum` declaration declares an `enum` class with the following restrictions:
  - Enum constants are implicitly `final` and `static`
  - Any attempt to create an object of an `enum` type with operator new results in a compilation error

Enum declaration example

Book.java

```java
public enum Book {
  // declare constants of enum type
  JHTP("Java How to Program", "2018"),
  CHTP("C How to Program", "2016"),
  IW3HTP("Internet & World Wide Web How to Program", "2012"),
  CPPHTP("C++ How to Program", "2017"),
  VBHTP("Visual Basic How to Program", "2014"),
  CSHARPHTP("Visual C# How to Program", "2017");

  // instance fields
  private final String title;
  private final String copyrightYear;

 // enum constructor
  Book(String title, String copyrightYear) {
    this.title = title;
    this.copyrightYear = copyrightYear;
  }
```

```java
  // accessor for field title
  public String getTitle() {
    return title;
  }
  // accessor for field copyrightYear
  public String getCopyrightYear() {
    return copyrightYear;
  }
}
```

Enum methods

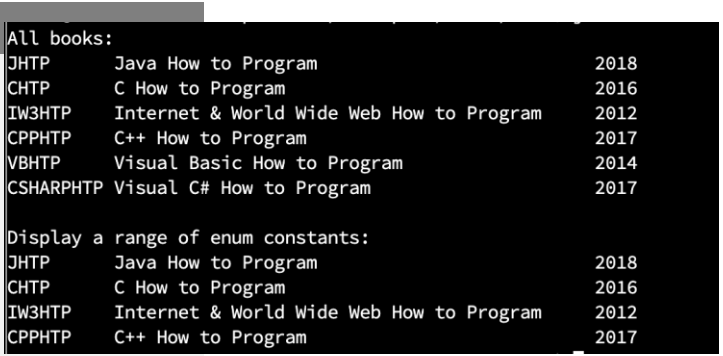
- The enhanced `for` statement can be used with an `EnumSet` just as it can with an array
- Method **`range()`** of class **`EnumSet`** (declared in package `java.util`) can be used to access a range of an `enum`’s constants
  - Method `range` takes two parameters: the first and the last enum constant in the range
  - Returns an `EnumSet` that contains all the constants between these two constants, both inclusive
- Class `EnumSet` provides several other `static` methods

Enum usage example

EnumTest.java

```java
import java.util.EnumSet;

public class EnumTest {
  public static void main(String[] args) {
    System.out.println("All books:");
    // print all books in enum Book
    for (Book book : Book.values()) {
      System.out.printf("%-10s%-45s%s%n", book,
        book.getTitle(), book.getCopyrightYear());
    }
    System.out.printf("%nDisplay a range of enum constants:%n");
    // print first four books
    for (Book book : EnumSet.range(Book.JHTP, Book.CPPHTP)) {
      System.out.printf("%-10s%-45s%s%n", book,
        book.getTitle(), book.getCopyrightYear());
    }
  }
}
```

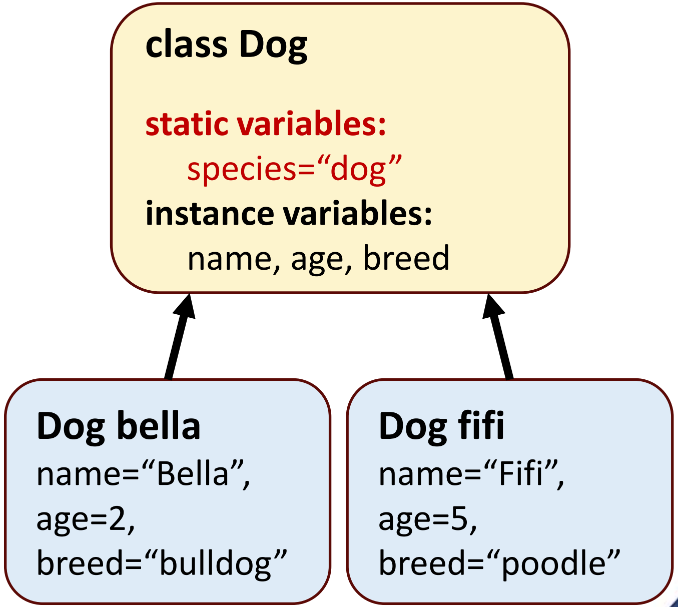


```
All books:
JHTP      Java How to Program                          2018
CHTP      C How to Program                             2016
IW3HTP    Internet & World Wide Web How to Program     2012
CPPHTP    C++ How to Program                           2017
VBHTP     Visual Basic How to Program                  2014
CSHARPHTP Visual C# How to Program                     2017

Display a range of enum constants:
JHTP      Java How to Program                          2018
CHTP      C How to Program                             2016
IW3HTP    Internet & World Wide Web How to Program     2012
CPPHTP    C++ How to Program                           2017
```

Static class members

- A `static` field (called a *class variable*) is used in the case of only one copy of a particular variable should be *shared* by all objects of a class
- A `static` variable have *class scope*, which represents *class-wide* information: all objects of the class share the *same* piece of data, and it can also be used in all of the class's methods
- The declaration of a `static` variable begins with the keyword `static`



Features of static class members

- Static class members are available as soon as the class is loaded into memory at execution time
  - Class members declared as `private static` can be accessed by client code only through methods of the class
  - A class's `public static` members can be accessed through a reference to any object of the class, or by qualifying the member name with the class name and a dot (`.`), as in `Math.random()`
- When no objects of the class exist:
  - To access a `public static` member, prefix the class name and a dot (`.`) to the `static` member, as in `Math.PI`
  - To access a `private static` member, provide a `public static` method and call it by qualifying its name with the class name and a dot

Features of static methods

- Since a `static` method can be called even when no objects of the class have been instantiated, a `static` method *cannot* access a class's instance variables and instance methods
  - The `this` reference *cannot* be used in a `static` method: the `this` reference must refer to a specific object of the class, but when a `static` method is called, there might not be any objects of its class in memory
- If a `static` variable is not initialized, the compiler assigns it a default value (e.g., the default value for type `int` is `0`)

Static class member example (1)

Employee.java

```java
public class Employee {
  private static int count = 0;
  private String firstName;
  private String lastName;
  // Constructor
  public Employee(String firstName,
      String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    ++count; // increment static count
    System.out.printf("Name %s %s; count = %d%n",
        firstName, lastName, count);
  }
  public String getFirstName() {
    return firstName;
  }
  public String getLastName() {
    return lastName;
  }
  public static int getCount() {
    return count;
  }
}
```

EmployeeTest.java

```java
public class EmployeeTest {
  public static void main(String[] args) {
  System.out.printf("Employees before: %d\n",
      Employee.getCount());
  // create two Employees; count should be 2
  Employee e1 = new Employee("Susan", "Baker");
  Employee e2 = new Employee("Bob", "Blue");

  // show that count is now 2
  System.out.printf("\nEmployees after:\n");
  System.out.printf("via e1.getCount(): %d\n",
      e1.getCount());
  System.out.printf("via e2.getCount(): %d\n",
      e2.getCount());
  System.out.printf("via Employee.getCount(): %d\n",
      Employee.getCount());
  // get names of Employees
  System.out.printf("\nEmployee 1: %s %s%n",
      e1.getFirstName(), e1.getLastName());
  System.out.printf("\nEmployee 2: %s %s%n",
      e2.getFirstName(), e2.getLastName());
  }
}
```

1495 UNIVERSITY OF ABERDEEN

Static class member example (2)

Employee.java

```java
public class Employee {
  private static int count = 0;
  private String firstName;
  private String lastName;
  // Constructor
  public Employee(String firstName
```

```java
                                             %d%n",
```

```java
  }
  public String getFirstName() {
    return firstName;
  }
  public String getLastName() {
    return lastName;
  }
  public static int getCount() {
    return count;
  }
}
```

Counter variable count is a static variable shared by all the instances of class Employee.

EmployeeTest.java

```java
public class EmployeeTest {
  public static void main(String[] args) {
  System.out.printf("Employees before: %d\n",
      Employee.getCount());
  // create two Employees; count should be 2
  Employee e1 = new Employee("Susan", "Baker");
  Employee e2 = new Employee("Bob", "Blue");

  // show that count is now 2
  System.out.printf("\nEmployees after:\n");
  System.out.printf("via e1.getCount(): %d\n",
      e1.getCount());
  System.out.printf("via e2.getCount(): %d\n",
      e2.getCount());
  System.out.printf("via Employee.getCount(): %d\n",
      Employee.getCount());
  // get names of Employees
  System.out.printf("\nEmployee 1: %s %s%n",
      e1.getFirstName(), e1.getLastName());
  System.out.printf("\nEmployee 2: %s %s%n",
      e2.getFirstName(), e2.getLastName());
  }
}
```

Static class member example (3)

Employee.java

```java
public class Employee {
 private static int count = 0;
 private String firstName;
 private String lastName;
 // Constructor
 public Employee(String firstName,
    String lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
  ++count; // increment static count
  System.out.printf("Name %s %s; count = %d%n",
    firstName, lastName, count);
 }
```

```
Employees before: 0
```

```java
 public String getLastName() {
  return lastName;
 }
 public static int getCount() {
  return count;
 }
}
```

EmployeeTest.java

```java
public class EmployeeTest {
 public static void main(String[] args) {
 System.out.printf("Employees before: %d\n",
   Employee.getCount());
 // create two Employees; count should be 2
 Employee e1 = new Employee("Susan", "Baker");
 Employee e2 = new Employee("Bob", "Blue");

 // show that count is now 2
 System.out.printf("\nEmployees after:\n");
 System.out.printf("via e1.getCount(): %d\n",
   e1.getCount());
 System.out.printf("via e2.getCount(): %d\n",
   e2.getCount());
 System.out.printf("via Employee.getCount(): %d\n",
   Employee.getCount());
 // get names of Employees
 System.out.printf("\nEmployee 1: %s %s%n",
   e1.getFirstName(), e1.getLastName());
 System.out.printf("\nEmployee 2: %s %s%n",
   e2.getFirstName(), e2.getLastName());
 }
}
```

Static class member example (4)

Employee.java

```java
public class Employee {
  private static int count = 0;
  private String firstName;
  private String lastName;
  // Constructor
  public Employee(String firstName,
      String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    ++count; // increment static count
    System.out.printf("Name %s %s; count = %d%n",
        firstName, lastName, count);
  }
```

```
Employees before: 0
Name: Susan Baker; count = 1
Name: Bob Blue; count = 2
```

```java
  }
  public static int getCount() {
    return count;
  }
}
```

EmployeeTest.java

```java
public class EmployeeTest {
  public static void main(String[] args) {
  System.out.printf("Employees before: %d\n",
      Employee.getCount());
  // create two Employees; count should be 2
  Employee e1 = new Employee("Susan", "Baker");
  Employee e2 = new Employee("Bob", "Blue");

  // show that count is now 2
  System.out.printf("\nEmployees after:\n");
  System.out.printf("via e1.getCount(): %d\n",
      e1.getCount());
  System.out.printf("via e2.getCount(): %d\n",
      e2.getCount());
  System.out.printf("via Employee.getCount(): %d\n",
      Employee.getCount());
  // get names of Employees
  System.out.printf("\nEmployee 1: %s %s%n",
      e1.getFirstName(), e1.getLastName());
  System.out.printf("\nEmployee 2: %s %s%n",
      e2.getFirstName(), e2.getLastName());
  }
}
```

Static class member example (5)

Employee.java

```java
public class Employee {
  private static int count = 0;
  private String firstName;
  private String lastName;
  // Constructor
  public Employee(String firstName,
      String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    ++count; // increment static count
    System.out.printf("Name %s %s; count = %d%n",
        firstName, lastName, count);
  }
```

```
Employees before: 0
Name: Susan Baker; count = 1
Name: Bob Blue; count = 2

Employees after:
via e1.getCount(): 2
via e2.getCount(): 2
via Employee.getCount(): 2
```

EmployeeTest.java

```java
public class EmployeeTest {
  public static void main(String[] args) {
  System.out.printf("Employees before: %d\n",
      Employee.getCount());
  // create two Employees; count should be 2
  Employee e1 = new Employee("Susan", "Baker");
  Employee e2 = new Employee("Bob", "Blue");

  // show that count is now 2
  System.out.printf("\nEmployees after:\n");
  System.out.printf("via e1.getCount(): %d\n",
      e1.getCount());
  System.out.printf("via e2.getCount(): %d\n",
      e2.getCount());
  System.out.printf("via Employee.getCount(): %d\n",
      Employee.getCount());
  // get names of Employees
  System.out.printf("\nEmployee 1: %s %s%n",
      e1.getFirstName(), e1.getLastName());
  System.out.printf("\nEmployee 2: %s %s%n",
      e2.getFirstName(), e2.getLastName());
  }
}
```

Static class member example (6)

Employee.java

```java
public class Employee {
  private static int count = 0;
  private String firstName;
  private String lastName;
  // Constructor
  public Employee(String firstName,
      String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    ++count; // increment static count
```

```
Employees before: 0
Name: Susan Baker; count = 1
Name: Bob Blue; count = 2

Employees after:
via e1.getCount(): 2
via e2.getCount(): 2
via Employee.getCount(): 2

Employee 1: Susan Baker
Employee 2: Bob Blue
```

EmployeeTest.java

```java
public class EmployeeTest {
  public static void main(String[] args) {
    System.out.printf("Employees before: %d\n",
        Employee.getCount());
    // create two Employees; count should be 2
    Employee e1 = new Employee("Susan", "Baker");
    Employee e2 = new Employee("Bob", "Blue");

    // show that count is now 2
    System.out.printf("\nEmployees after:\n");
    System.out.printf("via e1.getCount(): %d\n",
        e1.getCount());
    System.out.printf("via e2.getCount(): %d\n",
        e2.getCount());
    System.out.printf("via Employee.getCount(): %d\n",
        Employee.getCount());
    // get names of Employees
    System.out.printf("\nEmployee 1: %s %s%n",
        e1.getFirstName(), e1.getLastName());
    System.out.printf("\nEmployee 2: %s %s%n",
        e2.getFirstName(), e2.getLastName());
  }
}
```

Static import

- A *static import* declaration enables you to import the `static` members of a class or interface so you can access them via their *unqualified names* in your class. i.e., the class name and a dot (`.`) are *not* required when using an imported `static` member
- Two forms of static import:
  - One that imports a particular `static` member (which is known as *single static import*)
  - One that imports all `static` members of a class (which is known as *static import on demand*)

Static import syntax

- The following syntax imports a particular `static` member:

  `import static` ***`packageName.ClassName.staticMemberName;`***

- The following syntax imports *all* `static` members of a class:

  `import static` ***`packageName.ClassName.*;`***

  - where *`packageName`* is the package of the class, *`ClassName`* is the name of the class and *`staticMemberName`* is the name of the `static` field or method
  - Wildcard * indicates that *all* `static` members of the specified class should be imported
- Note that static import declarations import only `static` class members: Regular `import` statements should be used to specify the classes used in a program

Static import example

```java
// Static import of Math class methods.
import static java.lang.Math.*;

public class StaticImportTest {
    public static void main(String[] args) {
        System.out.printf("sqrt(900.0) = %.1f\n", sqrt(900.0));
        System.out.printf("ceil(-9.8) = %.1f\n", ceil(-9.8));
        System.out.printf("E = %f\n", E);
        System.out.printf("PI = %f\n", PI);
    }
}
```

```
sqrt(900.0) = 30.0
ceil(-9.8) = -9.0
E = 2.718282
PI = 3.141593
```

Final instance variables

- Keyword `final` specifies that a variable is not modifiable (i.e., it is a constant) and any attempt to modify it gives an error
  - A `final` variable cannot be modified by assignment after it has been initialized
  - A `final` variable can be initialised when is declared, e.g., to declare a `final` (constant) instance variable `INCREMENT` of type `int`, use:

    ```
    private final int INCREMENT;
    ```

  - Different objects of the class can have different value for the `final` variable, if it is initialised with a different value in different constructors of the class

Why to use final variables?

- The *principle of least privilege* is fundamental to good software engineering
  - Code should be granted only the amount of privilege and access that it needs to accomplish its designated task, but no more
  - This principle makes your programs more robust by preventing code from accidentally (or maliciously) modifying variable values and calling methods that should not be accessible

Questions, comments?

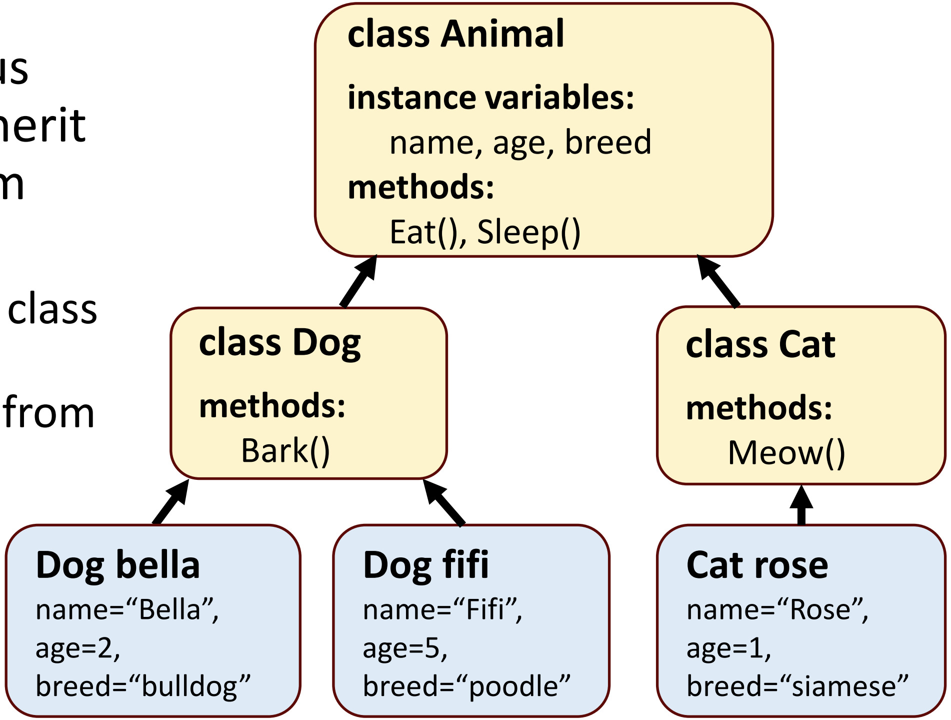
1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Day 3, Session 3: Class inheritance and access modifiers

Class inheritance

- ***Class inheritance*** lets us declare classes that inherit common structure from higher level classes
  - Objects of an inherited class can use the member variables and methods from the class it inherits



class Animal

instance variables:
name, age, breed

methods:
Eat(), Sleep()

class Dog

methods:
Bark()

class Cat

methods:
Meow()

Dog bella
name=“Bella”,
age=2,
breed=“bulldog”

Dog fifi
name=“Fifi”,
age=5,
breed=“poodle”

Cat rose
name=“Rose”,
age=1,
breed=“siamese”

Benefits of class inheritance

- DRY: don’t repeat yourself
  - Inheritance lets us pass on common structure and messages to similar objects
- Class inheritance allows “reuse” parts of objects
  - We can pull out common attributes and move them up to higher level object, and then differentiate them at the lower level
  - Reduces repetition and eases code maintenance and reusability

Superclasses and subclasses

- The class that inherits from another class is ***subclass*** (child)
  - Java does not support multiple inheritance directly: you can only inherit from one class
- The class being inherited from is ***superclass*** (parent)
  - Objects of all classes that extend a common superclass can be treated as objects/members of that superclass
- To inherit from a class, use **`extends`** keyword, for example:

```
class Dog extends Animal { … }
```

Inheritance example

Vehicle.java

```
class Vehicle {
  protected String brand = "Ford";
  public void honk() {
    System.out.println("Tuut tuut!");
  }
}
```

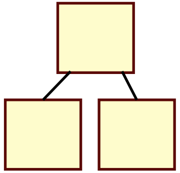
Car.java

```
class Car extends Vehicle {
  private String modelName = "Mustang";
  public static void main(String[] args) {
    Car myCar = new Car();
    myCar.honk();
    System.out.println(myCar.brand + " " + myCar.model);
  }
}
```

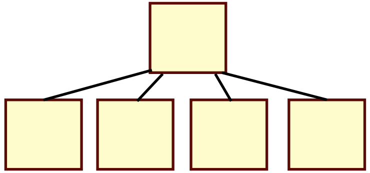
```
$ javac Car.java
$ java Car
Tuut tuut!
Ford Mustang
$
```

Inheritance hierarchies

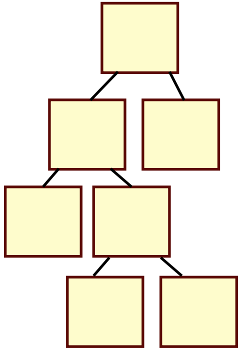
- Different class hierarchies can be constructed via inheritance
  - Deep hierarchies are complicated and tend to get wider over time, making them harder to maintain and use
  - For simplicity, shallow hierarchies are more recommended



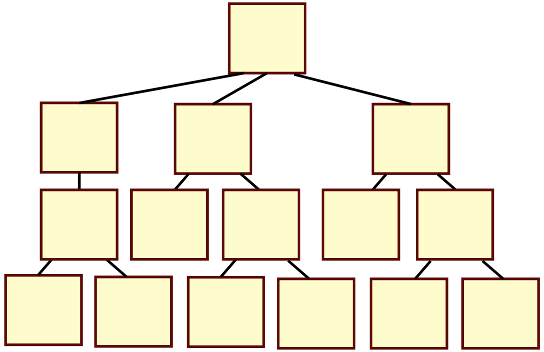
Shallow, Narrow



Shallow, Wide



Deep, Narrow



Deep, Wide

Using constructors with subclasses

- The first task of a subclass constructor is to call its direct superclass’s constructor *explicitly* or *implicitly*
    - Ensures that the instance variables inherited from the superclass are initialized properly.
- If the code does not include an explicit call to the superclass’s constructor, Java implicitly calls the superclass’s default or no-argument constructor

Constructor example

TestCar.java

```java
class Vehicle {
  public Vehicle() {
    System.out.println("this is Vehicle constructor");
  }
}
class Car extends Vehicle {
  public Car() {
    System.out.println("this is Car constructor");
  }
}
public class TestCar {
  public static void main(String[] arg) {
    Car ford = new Car();
  }
}
```

```
$ javac TestCar.java
$ java TestCar
this is Vehicle constructor
this is Car constructor
$
```

Redefine (override) methods

- Even when a superclass method is appropriate for a subclass, that subclass often needs a customized version of the method
- The subclass can *override* (i.e., redefine) the superclass method with an appropriate implementation
  - In Java, you can use optional **`@Override`** annotation to tell the compiler that the method is supposed to override another method; this can help to find errors during compilation time
- If keyword **`final`** is used for a method, it cannot be overridden; an attempt to override a `final` method gives a compilation error

Overriding example

```java
class Vehicle {
  void engine() {
    System.out.println("this is vehicle engine");
  }
}
class Car extends Vehicle {
  void engine() {
    System.out.println("this is car engine");
  }
}
class MotorBike extends Vehicle {
  void engine() {
    System.out.println("this is motorbike engine");
  }
}
public class TestEngines {
  public static void main(String[] arg) {
    MotorBike honda = new MotorBike ();
    honda.engine();
    Car ford = new Car ();
    ford.engine ();
  }
}
```

```
$ javac TestEngines.java
$ java TestEngines
this is motorbike engine
this is car engine
$
```

Overriding example with @Override

@Override annotation reveals a typing error in the method name

```java
    System.out.println("this is vehicle engine");
  }
}
class Car extends Vehicle {
  @Override
  void engne() {
    System.out.println("this is car engine");
  }
}
class MotorBike extends Vehicle {
  @Override
  void engine() {
    System.out.println("this is motorbike engine");
  }
}
```

```java
public class TestEngines {
  public static void main(String[] arg) {
    MotorBike honda = new MotorBike ();
    honda.engine();
    Car ford = new Car ();
    ford.engine ();
  }
}
```

```
$ javac TestEngines.java
error: method does not override or
implement a method from a supertype
  @Override
  ^
1 error
$
```

Overriding example with final

```java
class Vehicle {
  final void engine() {
    System.out.println("this is vehicle engine");
  }
}
class Car extends Veh
  @Override
  void engine() {
    System.out.println("this is car engine");
  }
}
class MotorBike extends Vehicle {
  @Override
  void engine() {
    System.out.println("this is motorbike engine");
  }
}
```

```java
public class TestEngines {
  public static void main(String[] arg) {
    MotorBike honda = new MotorBike ();
    honda.engine();
    Car ford = new Car ();
    ford.engine ();
```

Method defined as final cannot be overriden

```
$ javac TestEngines.java
error: engine() in Car cannot override
engine() in Vehicle
  void engine() {
       ^
  overridden method is final
1 error
$
```

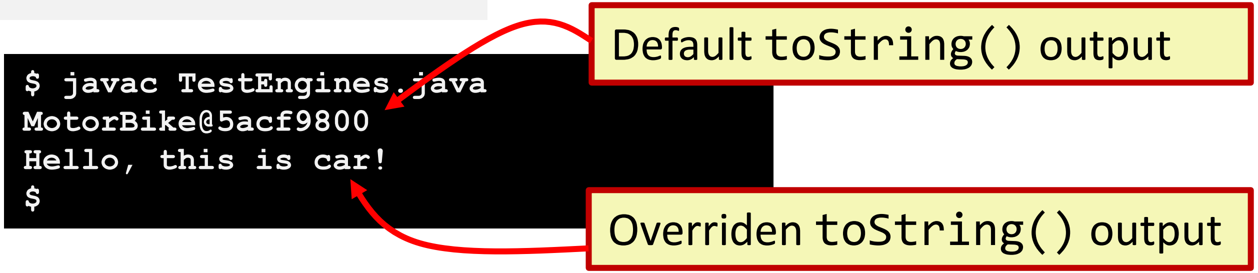
Method inheritance

- In Java, *every class* is a subclass of **`class Object`**, even if not explicitly defined to extend `Object`
- Some methods, such as `toString`, are inherited from `Object` and therefore defined for every class
  - Called implicitly whenever an object must be converted to a string representation
  - The default `toString` method returns a `String` with the name of the object's class
  - More appropriate `String` representation can be specified by overriding `toString`

Overriding example of toString() method

```java
class Vehicle {
}
class Car extends Vehicle {
  @Override
  public String toString() {
    return "Hello, this is car!";
  }
}
class MotorBike extends Vehicle {
}
```

```java
public class TestEngines {
  public static void main(String[] arg) {
    MotorBike honda = new MotorBike ();
    Car ford = new Car();
    System.out.println(honda.toString());
    System.out.println(ford.toString());
  }
}
```



```
$ javac TestEngines.java
MotorBike@5acf9800
Hello, this is car!
$
```

Default toString() output

Overriden toString() output

Access modifiers

- A class's *public* members are accessible wherever the program has a reference to an object of that class *or one of its subclasses*
- A class's *private* members are accessible only within the class itself
- To enable a subclass to directly access superclass instance variable, we can declare those members as *protected* in the superclass
  - Protected access is an intermediate level of access between public and private
  - All public and protected superclass members retain their original access modifier when they become members of the subclass

Access modifier protected

- A superclass's protected members can be accessed by members of *that superclass*, its *subclasses*, and *other classes in the same package* (protected members also have package access)
  - Subclass methods can refer to public and protected members inherited from the superclass simply by using the member names
- Superclass's private members are hidden from its subclasses
  - They can be accessed only through the public or protected methods inherited from the superclass
  - In many cases, it is better to use private instance variables to encourage proper software engineering

Disadvantages of protected variables

- With protected instance variables, we may need to modify all the subclasses of a superclass if the superclass implementation changes
  - Such a class is said to be fragile or brittle, because a small change in the superclass can “break” subclass implementation
  - You should be able to change the superclass implementation while still providing the same services to the subclasses
- A class’s protected members are visible to all classes in the same package as the class containing the protected members – this is not always desirable (the principle of minimum privilege)

Summary of access modifiers

Access to	default	private	protected	public
Same class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

- Access modifiers allow *encapsulation* (data hiding from other classes), one of the fundamental concepts of OOP

Calling superclass constructor

- Each subclass constructor must implicitly or explicitly call one of its superclass's constructors to initialize the instance variables inherited from the superclass
  - The syntax for calling superclass constructor: **`super(arguments)`**
  - Must be the first statement in the constructor's body
  - This lets you specify how to instantiate the object
- If the subclass constructor did not invoke the superclass's constructor explicitly, the compiler would attempt to insert a call to the superclass's default or no-argument constructor
  - You can also explicitly use `super()` to call the superclass's no-argument or default constructor, but this is not usually done

Superclass constructor example

```java
class Vehicle {
  private String type;
  public Vehicle() {
    this.type = "undefined";
  }
  public Vehicle(String type) {
    this.type = type;
  }
}
class Car extends Vehicle {
  private Engine engine;
  public Car() {
    super("car");
  }
}
```

```java
public class TestEngines {
  public static void main(String[] arg) {
    Car ford = new Car();
    System.out.print("Type: ");
    ford.printType();
  }
}
```

```
$ javac TestEngines.java
Type: car
$
```

Invokes superclass's constructor with a parameter. Note that variable **type** is private, so it cannot be accessed directly outside the superclass **Vehicle**.

Reference super methods

- When a subclass method overrides an inherited superclass method, the superclass version of the method can be accessed from the subclass by preceding the superclass method name with keyword **`super`** and dot(**`.`**) separator

```java
class Vehicle {
  public void engine() {
    System.out.println("this is vehicle engine");
  }
}
class Car extends Vehicle {
  public void engine() {
    super.engine();
    System.out.println("this is car engine");
  }
}
```

```java
public class TestEngines {
  public static void main(String[] arg) {
    Car ford = new Car ();
    ford.engine();
  }
}
```

```
$ java TestEngines
this is vehicle engine
this is car engine
$
```

Questions, comments?

1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Day 3, Session 4: Composition and polymorphism

Class relationships

- Inheritance relationship is basically ***is-a*** relationship
    - Car (subclass) *is a* vehicle (superclass)
    - Dog (subclass) *is a* mammal (superclass)
- However, some class relationships are ***has-a*** relationships
    - Car *has* an engine
    - Dog *has* a tail
    - Person *has* a name
    - Has-a relationships should be created by *composition* of existing classes, rather than inheritance

Composition

- A class can have references to objects of other classes as members
  - This is called composition and is sometimes referred to as has-a relationship
- Composition is used to ease complexity, which lets us create objects with fewer dependencies
  - Example: An `AlarmClock` object needs to know the current time and the time when it is supposed to sound its alarm, so it is reasonable to include two references to `Time` objects in an `AlarmClock` object

Composition example

Car.java

```java
public class Car {
  private Engine engine;
  public Car() {
    this.engine = new Engine();
  }
  public void startCar() {
    engine.makeNoise();
  }
}
```

Engine.java

```java
public class Engine {
  public void makeNoise() {
    System.out.println("Wrroom!");
  }
}
```

Car *has an* Engine

Composition or inheritance?

- There has been much discussion in the software engineering community about the relative merits of composition and inheritance
    - Each has its own place, but inheritance is often overused and composition is more appropriate in many cases
- A mix of composition and inheritance often is the best approach
    - It is best to think whether *is-a* or *has-a* relationship represents your case more naturally

Composition vs. inheritance

Composition

- Composition and aggregation form has-a relationships where sum is greater than its parts
- Objects stand alone, so development cost is higher: fewer built-in dependencies that can be reused

Inheritance

- Inheritance for when message delegation is free within hierarchy
- Easier to develop, but more dependencies: it is easy to break things by changing something in a superclass that affects all the subclasses

Nested classes

- Java allows declaring classes inside classes (nested classes)
  - To instantiate a nested (inner) class, you need to first instantiate the enclosing (outer) class
  - Non-static inner classes have access to other members of the outer class, even if declared **`private`**
- Nested classes can be considered as a kind of "composition", since the outer class "owns" the inner class
  - However, some benefits of composition are lost, such as polymorphic behavior and reusability: only use a nested class, if you are absolutely sure that you do not need it anywhere else!

Nested class example

Car.java

```java
public class Car {
  private class Engine {
    public void makeNoise() {
      System.out.println("Wrroom!");
    }
  }
  private Engine engine;
  public Car() {
    this.engine = new Engine();
  }
  public void startCar() {
    engine.makeNoise();
  }
  public static void main(String[] args) {
    Car car = new Car();
    car.startCar();
  }
}
```

Nested class defined here

```
$ java Car
Wrroom!
$
```

Anonymous classes

- In Java, you can declare anonymous classes
  - Anonymous classes are like local classes, except that they do not have a name
  - Use them if you only need to use a local class in one place
- Anonymous classes are defined in their initialisation statements when they are instantiated
  - Declare anonymous classes using the following syntax:

```java
SuperClass myClass = new SuperClass() {
    // override methods here as needed
};
```

Anonymous class example

Car.java

```java
class Engine {
  public void makeNoise() {
    System.out.println("Put put put!");
  }
}
public class Car {
  private Engine engine;
  public Car() {
    this.engine = new Engine() {
      public void makeNoise() {
        System.out.println("Wrrooom!");
      }
    };
  }
  public void startCar() {
    engine.makeNoise();
  }
  public static void main(String[] args) {
    Car car = new Car();
    car.startCar();
  }
}
```

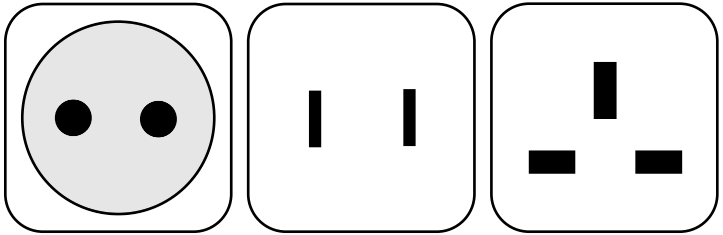
```
$ java Car
Wrrooom!
$
```

Anonymous subclass of Engine defined here

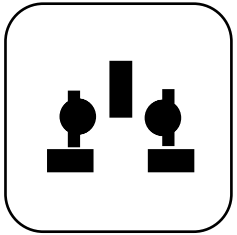
UNIVERSITY OF ABERDEEN

Polymorphism

- *Polymorphism* allows you to define one interface and have multiple implementations
  - The word “poly” means many and “morphs” means forms: polymorphism means “many forms”



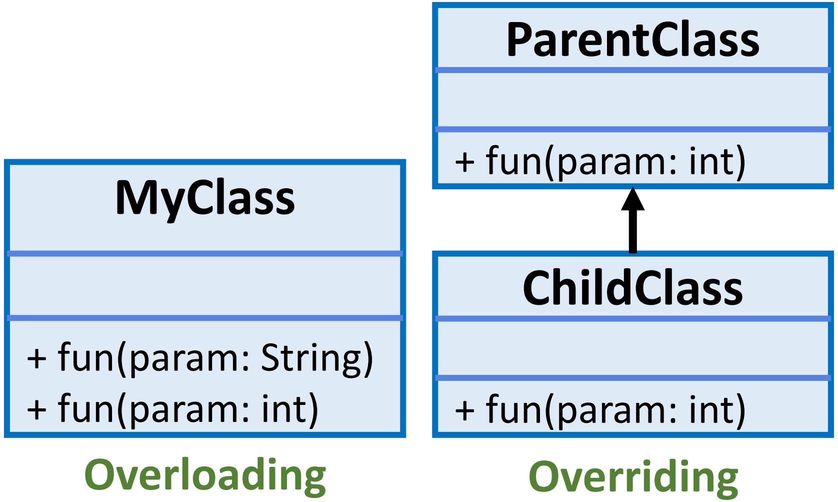
Without polymorphism



With polymorphism

Method overloading and overriding

- In Java, polymorphism is mainly divided into two types:
  - Compile-time polymorphism (static polymorphism achieved by ***method overloading***)
  - Runtime polymorphism (dynamic method dispatch achieved by ***method overriding***)



Overloading example

- We discusses overloading of constructors already, but other methods can be overloaded as well

```
class Helper {
  static int Multiply(int a, int b) {return a * b;}
  static double Multiply(double a, double b) {return a * b;}
  public static void main(String[] args)
  {
    System.out.println(Helper.Multiply(2, 4));
    System.out.println(Helper.Multiply(4.2, 3.8));
  }
}
```

```
$ java Helper
8
15.540000000000001
```

Overloading example (2)

- Different versions of the method can differ in parameter types or the number of parameters

```
class Helper {
  static int Multiply(int a, int b) {return a * b;}
  static int Multiply(int a, int b, int c) {return a * b * c;}
  public static void main(String[] args)
  {
    System.out.println(Helper.Multiply(2, 4));
    System.out.println(Helper.Multiply(2, 4, 8));
  }
}
```

```
$ java Helper
8
64
```

Runtime overriding example

```java
class Vehicle {
  public void printType() {
    System.out.println("undefined");
  }
}
class Car extends Vehicle {
  public void printType() {
    System.out.println("car");
  }
}
class MotorBike extends Vehicle {
  public void printType() {
    System.out.println("motorbike");
  }
}
```

```java
public class TestEngines {
  public static void main(String[] arg) {
    Vehicle vehicle = new MotorBike();
    System.out.print("Vehicle type 1: ");
    vehicle.printType();
    vehicle = new Car();
    System.out.print("Vehicle type 2: ");
    vehicle.printType();
  }
}
```

```
$ java TestEngines
Vehicle type 1: motorbike
Vehicle type 2: car
$
```

Overriding data members

- Note that overriding works for methods but not data members!
  - Runtime polymorphism cannot be achieved by inherited variables

```java
class Vehicle {
  int maxSpeed = 50;
}
class Car extends Vehicle {
  int maxSpeed = 150;
}
public class TestEngines {
  public static void main(String[] arg) {
    Car ford = new Car();
    System.out.printf("Max speed: %d%n", ford.maxSpeed);
  }
}
```

```
$ javac TestEngines.java
Max speed: 50
$
```

Summary

- Java is an *object oriented language*; therefore, to understand Java, it is essential to understand the OOP concepts of Java
  - ***Abstraction:*** *classes*, *objects, methods* and *variables* provide simple representations of complex underlying data and behavior
  - ***Encapsulation:*** access to private members of a class can be controlled via *access modifiers*
  - ***Inheritance:*** inherited *subclasses* can be declared to share the attributes of the higher level *superclasses*
  - ***Polymorphism:*** allows methods with the same name to work in different contexts

Questions, comments?