

CSE201: Monsoon 2024

Advanced Programming

Lecture 02: Classes and Objects

Dr. Arun Balaji Buduru

Head, Center of Technology in Policing

Founding Head, Usable Security Group (USG)

Associate Professor, Dept. of CSE | HCD

IIIT-Delhi, India

Last Lecture

- Introduction to OOP
 - What, Why, and Advantages of OOP
 - Encapsulation
 - Procedural programming v/s OO programming

Today's Lecture

- Identifying classes and objects
- Working with objects

Ideas to Program

Analysis (common sense)



Design (object oriented)



Implementation (actual programming)



Testing

- Analysis
 - **What to do and not how to do it**
 - Decide corner cases and exact functionalities
- Design
 - Define classes, their attributes and methods, objects, and class relationships
- Implementation
 - Novice programmers often think that writing code is the heart of software development, but actually it should be the least creative step
- Testing
 - A program should be free of errors

Analysis: Identifying Classes & Responsibilities

- Identifying classes
 - Good first step: look for **nouns** in use cases. Then...
 - **Actors**- objects that perform tasks
 - **Events** - store information about events
- Identifying responsibilities
 - Good first step: look for **verbs, actions** in use cases
 - These actions may directly describe responsibilities, or
 - may depend on other responsibilities

Analysis: Identifying Classes

- A partial requirement document

The user must be allowed to specify each product by its primary characteristics, including its name and product number. If the bar code does not match the product, then an error should be generated to the message window and entered into the error log. The summary report of all transactions must be structured as specified in section 7.A.

Of course, not all nouns will correspond to a class or object in the final solution

Analysis: Guidelines for Discovering Classes

- Limit responsibility of each analysis class
 - Clear purpose for existence
 - Avoid giving too many responsibilities to one class
- Use clear and consistent names
 - Class names should be nouns
 - Not finding good name implies class is too fuzzy
- Keep analysis classes simple
 - In first step don't worry about class relationships

Exercise: Logging into Email Account

- A partial requirement document

For accessing an online email account, the customer will first click the login button on the home page of the email account. This will display the login page of email account. Once the customer gets directed to the login page, he will enter his user id and password, and then click the OK button. The email account will first validate the customer credentials and then grant access to his email account.

Exercise: Logging into Email Account

- Step -1: Identifying classes (nouns) and objects

For accessing an online email account, the customer will first click the login button on the home page of the email account. This will display the login page of email account. Once the customer gets directed to the login page, he will enter his user id and password, and then click the OK button. The email account will first validate the customer credentials and then grant access to his email account.

Exercise: Logging into Email Account

- Step -2: Identifying methods (verbs)

For accessing an online email account, the customer will first click the login button on the home page of the email account. This will display the login page of email account. Once the customer gets directed to the login page, he will enter his user id and password, and then click the OK button. The email account will first validate the customer credentials and then grant access to his email account.

Design: Classes and Objects

- Recall, class represents a group of objects with similar behaviors
 - Instantiate as many objects as you like!
- If a class becomes too complex, decompose into multiple smaller classes
- Assign responsibilities to each class
 - Every activity in a program represents methods in a class
 - In early stages, begin with primary responsibilities and evolve the design

Design: Interaction Between Objects

- Sequence diagrams
 - Interaction diagrams that details how operations are carried out in a program
 - **Messages:** Interaction between two objects is performed as a message sent from one object to another
 - Help tracing object methods and interactions
 - UML is significantly improved version of sequence diagram
 - *We will cover this in depth in later lectures*

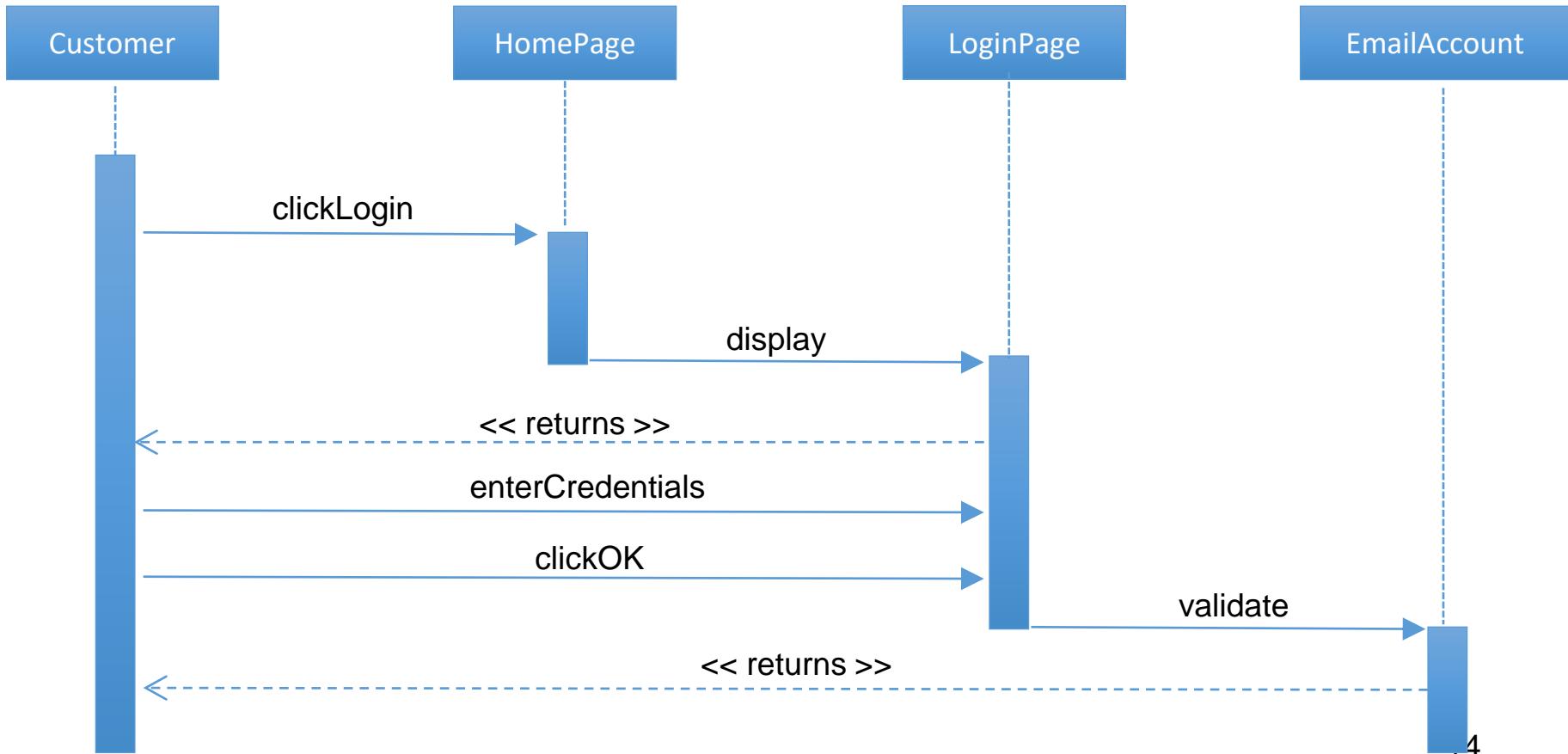
Exercise: Logging into Email Account

- Step-3: Draw the sequence diagram

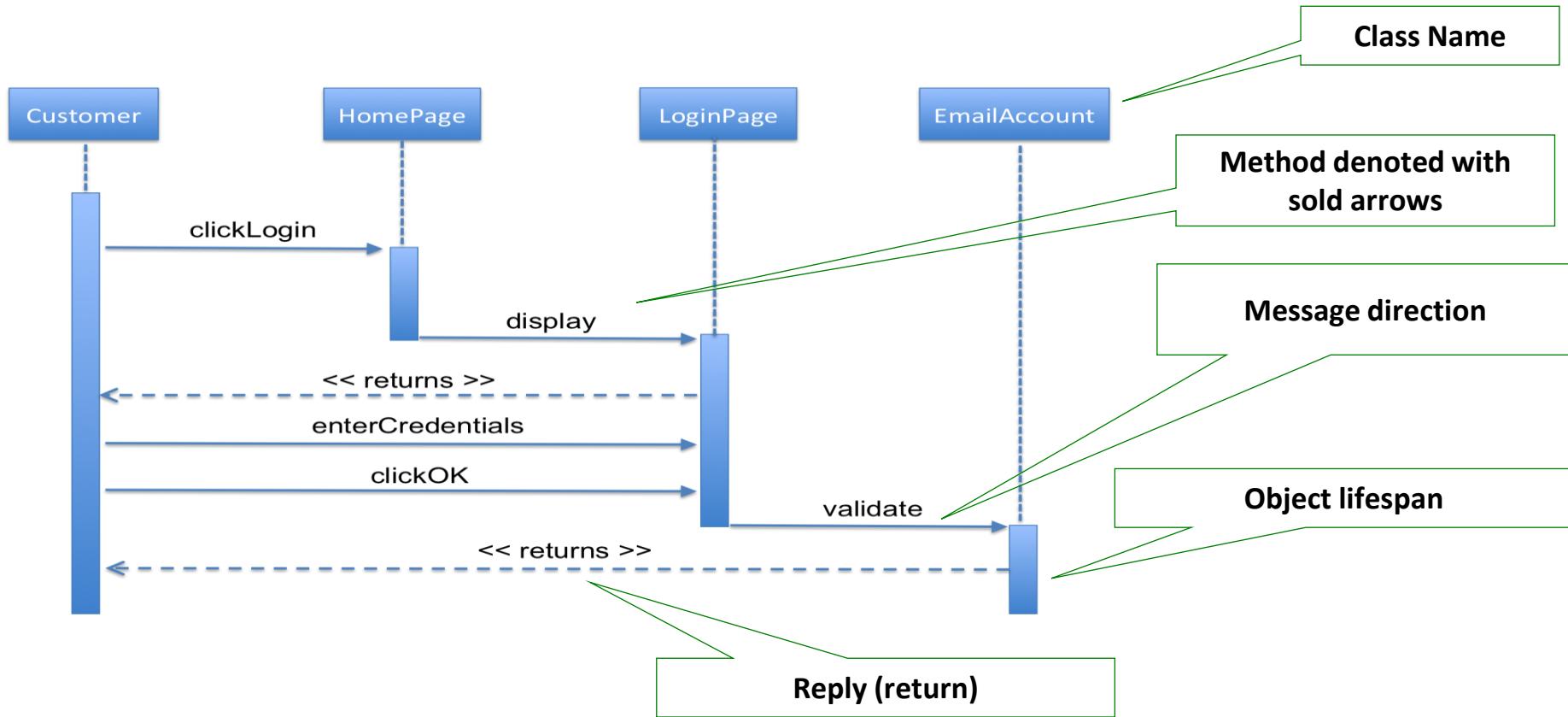
For accessing an online email account, the customer will first click the login button on the home page of the email account. This will display the login page of email account. Once the customer gets directed to the login page, he will enter his user id and password, and then click OK button. The email account will first validate the customer credentials and then grant access to his email account.

Classes	Methods
Customer	clickLogin
HomePage	display
LoginPage	enterCredentials
EmailAccount	clickOK
	validate

Sequence Diagram



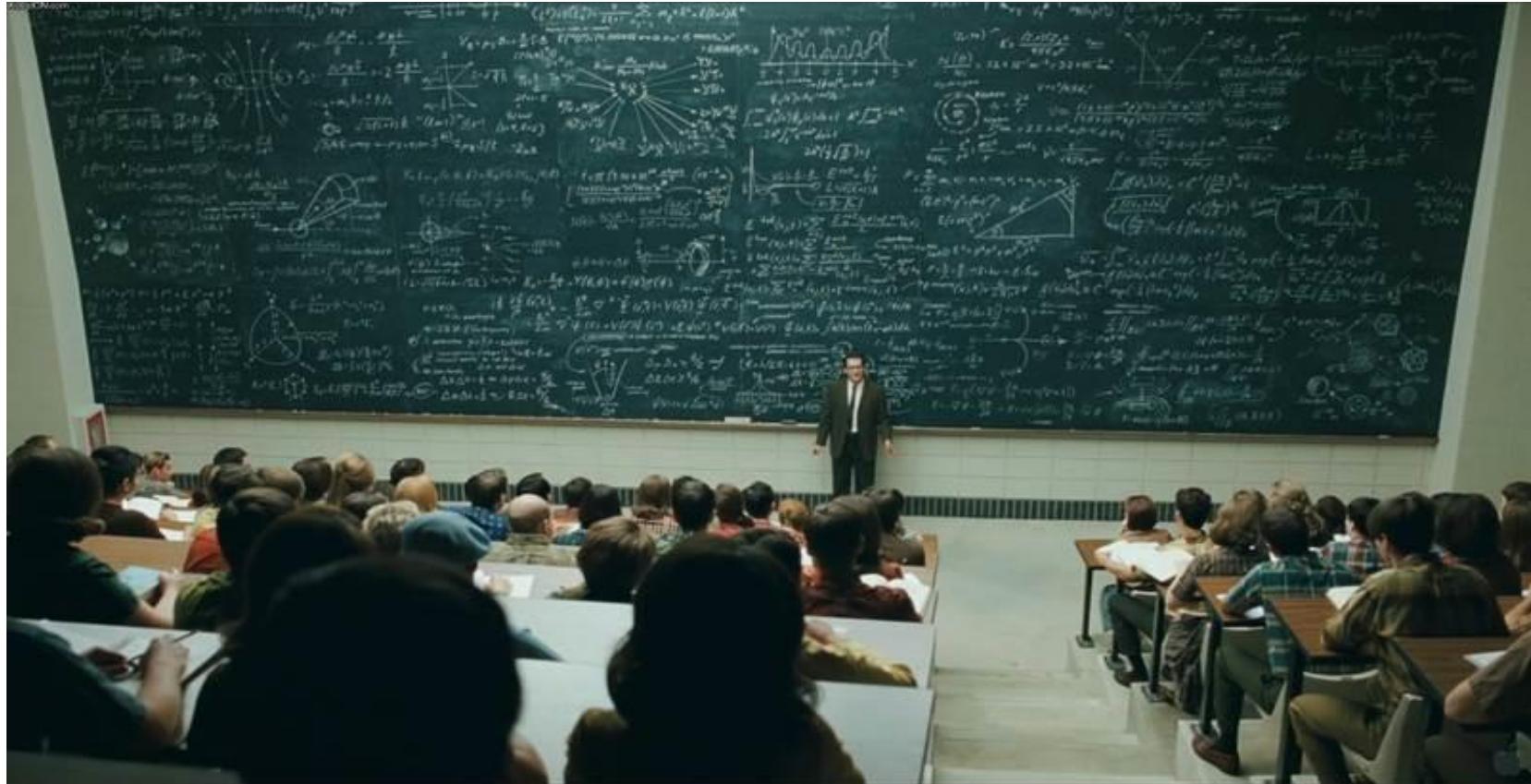
Sequence Diagram



Cohesion Between Methods

- Methods of an object should be in harmony. If a method seems out of place, then your object might be better off by giving that responsibility to somewhere else
- E.g., for *LoginPage class*, `enterCredentials()`, `clickOK()` are in harmony but not if we make `validate()` as method of *LoginPage*

Identify Classes Below



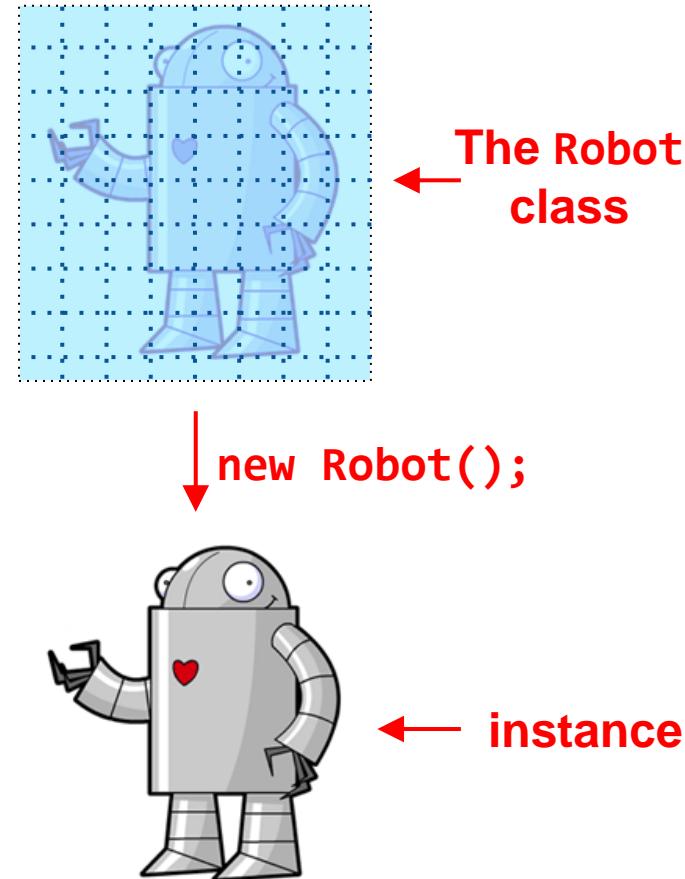


Let's change gears...

How to Work with Objects ?

Review: Instantiation

- **Instantiation** means building an object from its class “blueprint”
- Ex: `new Robot();` creates an instance of Robot
- This calls the `Robot` class’s **constructor**: a special kind of method



Review: Constructors

- A **constructor** is a method that is called to create a new object
- Let's define one for the **Dog** class
- All **Dogs** know how to bark, eat, and wag their tails

```
public class Dog {  
  
    public Dog() {  
        // this is the constructor!  
    }  
  
    public void bark(int numTimes) {  
        // code for barking goes here  
    }  
  
    public void eat() {  
        // code for eating goes here  
    }  
  
    public void wagTail() {  
        // code for wagging tail goes here  
    }  
}
```

Review: Constructors

- Constructors do not specify a return type
- Name of constructor must exactly match name of class
- Now we can instantiate a Dog in some method:
`new Dog();`

```
public class Dog {  
  
    public Dog() {  
        // this is the constructor!  
    }  
  
    public void bark(int numTimes) {  
        // code for barking goes here  
    }  
  
    public void eat() {  
        // code for eating goes here  
    }  
  
    public void wagTail() {  
        // code for wagging tail goes here  
    }  
}
```

Review: Constructors

Question:

- Find the order of execution for following statement

Dog djangho = new Dog("Djangho");

```
public class Dog {  
    private String name;  
    private int breed_id;  
    private int rego_id;  
    private static int rego_counter;  
  
    { // initialization block  
        rego_id = ++rego_counter; // line-z  
    }  
    public Dog(int _breed) {  
        this.breed_id = _breed; // line-y  
    }  
    public Dog(String _name) {  
        this(20);  
        this.name = _name; // line-x  
    }  
    .....  
}
```

Variable Declaration & Assignment

```
Dog django = new Dog();  
<type> <name> = <value>;
```

- The “=” operator **assigns** the instance of **Dog** that we created to the variable **django**. We say “**django gets a new Dog**”
- Note that we can reassign as many times as we like (example soon)

django



Variables Store Information: Values vs. References

- A variable stores information as either:
 - a *value* of a *primitive* (aka *base*) *type* (like `int` or `float`)
 - or a *reference* to an *instance* (like an instance of `Dog`) of an arbitrary type stored elsewhere in memory – we symbolize a reference with an arrow

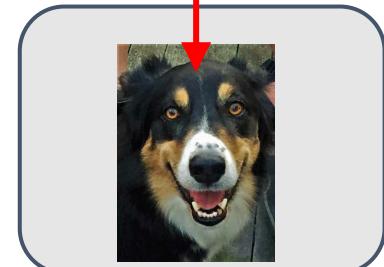
```
int favoriteNumber = 9;
```

favNumber
9

- Think of the variable like a box; storing a value or reference is like putting something into the box
- Primitives have a predictable memory size, while arbitrary objects vary in size, hence Java simplifies its memory management by having a fixed size reference to an instance elsewhere in memory
 - “one level of indirectness”

```
Dog django = new Dog();
```

django



Example: Instantiation

```
public class PetShop {  
  
    /*constructor of trivial PetShop! */  
    public PetShop() {  
        this.testDjango();  
    }  
  
    public void testDjango() {  
        Dog django = new Dog();  
        django.bark(5);  
        django.eat();  
        django.wagTail();  
    }  
    ...  
}
```

**This doesn't seem
to be the job of
PetShop owner!
Maybe
DogGroomer
should be hired..**

- Let's call the `testDjango()` method within the constructor of the `PetShop` class
- Whenever someone instantiates a `PetShop`, it in turn calls `testDjango()`, which in turn instantiates a `Dog`
- Then it tells the `Dog` to bark, eat, and wag its tail

Objects as Parameters (1/2)

- Methods can take in objects as parameters
- The **DogGroomer** class has a method **groom**
- **groom** method needs to know which **Dog** to groom

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
    public void groom(Dog shaggyDog) {  
        // code that grooms shaggyDog  
    }  
}
```

type **name**
 ↑ ↑

Objects as Parameters (2/2)

- How to call the *groom* method?
- Do this in the *PetShop* helper method *testGroomer()*
- *PetShop*'s call to *testGroomer()* instantiates a *Dog* and a *DogGroomer*, then tells the *DogGroomer* to groom the *Dog*

```
public class PetShop {  
  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
    }  
}
```

What is Memory?

- Memory (system memory, not disk or other peripheral devices) is the hardware in which computers store information, both temporary and permanent
- Think of memory as a list of slots; each slot holds information (e.g., a local `int` variable, or a reference to an instance of a class)
- Here, two references are stored in memory: one to a `Dog` instance, and one to a `DogGroomer` instance

```
//Elsewhere in the program
Petshop petSmart = new Petshop();

public class PetShop {

    public PetShop() {
        this.testGroomer();
    }

    public void testGroomer() {
        Dog django = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(django);
    }
}
```



Objects as Parameters: Under the Hood (1/6)

```
public class PetShop {  
  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        // code that grooms shaggyDog goes here!  
    }  
}
```

Somewhere in memory...



Objects as Parameters: Under the Hood (2/6)

```
public class PetShop {  
  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        // code that grooms shaggyDog goes here!  
    }  
}
```

Somewhere in memory...



When we instantiate a Dog, he's stored somewhere in memory. Our PetShop will use the name django to refer to this particular Dog, at this particular location in memory.

Objects as Parameters: Under the Hood (3/6)

```
public class PetShop {  
  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        // code that grooms shaggyDog goes here!  
    }  
}
```

Somewhere in memory...



The same goes for the DogGroomer—we store a particular DogGroomer somewhere in memory. Our PetShop knows this DogGroomer by the name `groomer`.

Objects as Parameters: Under the Hood (4/6)

```
public class PetShop {  
  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        // code that grooms shaggyDog goes here!  
    }  
}
```

Somewhere in memory...



We call the `groom` method on our `DogGroomer`, `groomer`. We need to tell her which `Dog` to groom (since the `groom` method takes in a parameter of type `Dog`). We tell her to groom `django`.

Objects as Parameters: Under the Hood (5/6)

```
public class PetShop {  
  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        // code that grooms shaggyDog goes here!  
    }  
}
```

Somewhere in memory...



When we pass in django as an argument to the groom method, we're telling the groom method about him. When groom executes, it sees that it has been passed that particular Dog. 33

Objects as Parameters: Under the Hood (6/6)

```
public class PetShop {  
  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        // code that grooms shaggyDog goes here!  
    }  
}
```

Somewhere in memory...



The `groom` method doesn't really care which `Dog` it's told to `groom`—no matter what another object's name for the `Dog` is, `groom` is going to know it by the name `shaggyDog`.

Variable Reassignment (1/2)

- After giving a variable an initial value, we can **reassign** it (make it refer to a different object)
- What if we wanted our **DogGroomer** to **groom** two different **Dogs** when the **PetShop** opened?
- Could re-use the variable **django** to first point to one **Dog**, then another!

```
public class PetShop {  
  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
    }  
}
```

Variable Reassignment (2/2)

- First, instantiate another `Dog`, and reassign variable `django` to point to it
- Now `django` no longer refers to the first `Dog` instance we created, which has already been groomed
- We then tell `groomer` to `groom` the newer `Dog`

```
public class PetShop {  
  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
        django = new Dog(); // reassign django  
        groomer.groom(django);  
    }  
}
```

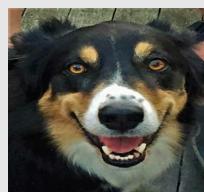
Variable Reassignment: Under the Hood (1/5)

```
public class PetShop {  
  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
        django = new Dog();  
        groomer.groom(django);  
    }  
}
```



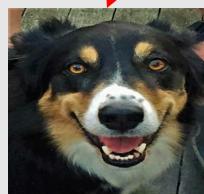
Variable Reassignment: Under the Hood (2/5)

```
public class PetShop {  
  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
        django = new Dog();  
        groomer.groom(django);  
    }  
}
```



Variable Reassignment: Under the Hood (3/5)

```
public class PetShop {  
  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
        django = new Dog();  
        groomer.groom(django);  
    }  
}
```



Variable Reassignment: Under the Hood (4/5)

```
public class PetShop {  
  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
        django = new Dog(); //old ref garbage collected  
        groomer.groom(django);  
    }  
}
```



Variable Reassignment: Under the Hood (5/5)

```
public class PetShop {  
  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
        django = new Dog(); //old ref garbage collected  
        groomer.groom(django);  
    }  
}
```

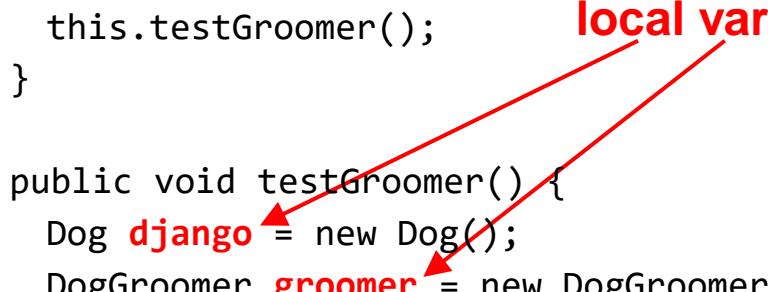


Local Variables (1/2)

- All variables we've seen so far have been **local variables**: variables declared *within a method*
- Problem: the **scope** of a local variable (where it is known and can be accessed) is limited to its own method—it cannot be accessed from anywhere else
 - the same is true of method parameters

```
public class PetShop {  
  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
        django = new Dog();  
        groomer.groom(django);  
    }  
}
```

local variables



Local Variables (2/2)

- We created `groomer` and `django` in our `PetShop`'s helper method, but as far as the rest of the class is concerned, they don't exist
- What happens to `django` after the method is executed?
 - “Garbage Collection”

```
public class PetShop {  
  
    /* This is the constructor! */  
    public PetShop() {  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        DogGroomer groomer = new DogGroomer();  
        groomer.groom(django);  
        django = new Dog();  
        groomer.groom(django);  
    }  
}
```

local variables

Accessing Local Variables

- If you try to access a local variable outside of its method, you'll receive a “cannot find symbol” compilation error.

In Terminal:

```
Petshop.java:13: error: cannot find symbol
    django.playCatch();
           ^
  symbol: variable django
  location: class PetShop
```

```
public class PetShop {

    private DogGroomer _groomer;

    /* This is the constructor! */
    public PetShop() {
        _groomer = new DogGroomer();
        Dog django = new Dog();
    }
}
```

```
public void exerciseDjango() {
    django.playCatch();
}
```

Instance Variables for the Rescue

- Local variables aren't always what we want. We'd like every `PetShop` to come with a `DogGroomer` who exists for as long as the `PetShop` exists
- That way, as long as the `PetShop` is in business, we'll have our `DogGroomer` on hand
- We can accomplish this by storing the `DogGroomer` in an **instance variable**

What's an Instance Variable?

- An **instance variable** models a property that all instances of a class have
 - its *value* can differ from instance to instance (e.g, the dog's breed, name, color, ...)
- Instance variables are declared within a class, not within a single method, and are accessible from anywhere within the class – its **scope** is the entire class
- Instance variables and local variables are identical in terms of what they can store—either can store a base type (like an **int**) or a reference to an object (instance of some other class)



Instance Variables

- We've modified PetShop example to make our DogGroomer an **instance variable**
- Split up declaration and assignment of instance variable:
 - **declare** instance variable
 - **initialize** the instance variable by **assigning** a value to it in the constructor
 - purpose of constructor is to initialize all instance variables so the instance has a valid initial "state" at its "birth"
 - **state** is the set of all values for all properties—local variables don't hold properties - they are "temporaries"

```
public class PetShop {    declaration  
    private DogGroomer _groomer;  
  
    /* This is the constructor! */  
    public PetShop() {        assignment  
        _groomer = new DogGroomer();  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog(); //local var  
        _groomer.groom(django);  
    }  
}
```

Always Remember to Initialize!

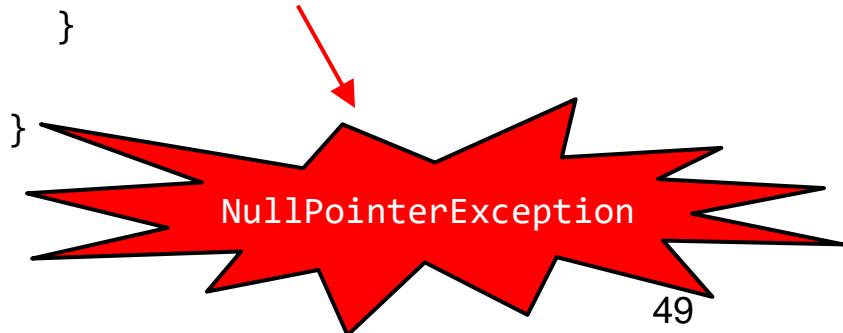
- What if you declare an instance variable, but forget to initialize it?
- The instance variable will assume a “default value”
 - if it’s an `int`, it will be 0
 - if it’s an object, it will be `null`—a special value that means your variable is not referencing any instance at the moment

```
public class PetShop {  
  
    private DogGroomer _groomer;  
  
    /* This is the constructor! */  
    public PetShop() {  
        //oops!  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog(); //local var  
        _groomer.groom(django);  
    }  
}
```

NullPointerExceptions

- If a variable's value is null and you try to give it a command, you'll be rewarded with a *runtime error*—you can't call a method on “nothing”!
- This particular error yields a **NullPointerException**
- When you run into one of these (we promise, you will)—edit your program to make sure you have explicitly initialized all variables

```
public class PetShop {  
  
    private DogGroomer _groomer;  
  
    public PetShop() {  
        //oops!  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog(); //local var  
        _groomer.groom(django);  
    }  
}
```



Next Lecture

- Class relationships

CSE201: Monsoon 2024

Advanced Programming

Lecture 03: Class Relationships

Dr. Arun Balaji Buduru

Head, Center of Technology in Policing

Founding Head, Usable Security Group (USG)

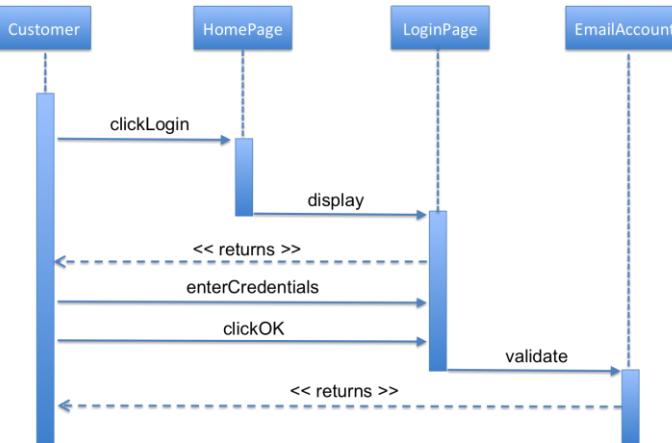
Associate Professor, Dept. of CSE | HCD

IIIT-Delhi, India

Last Lecture

- Program development
- Identifying classes and objects
- Sequence diagrams
- Working with objects
 - Objects as parameters
 - Variable reassignment
 - Instance variables

For accessing an online email account, the customer will first click the login button on the home page of the email account. This will display the login page of email account. Once the customer gets directed to the login page, he will enter his user id and password, and then click OK button. The email account will first validate the customer credentials and then grant access to his email account.

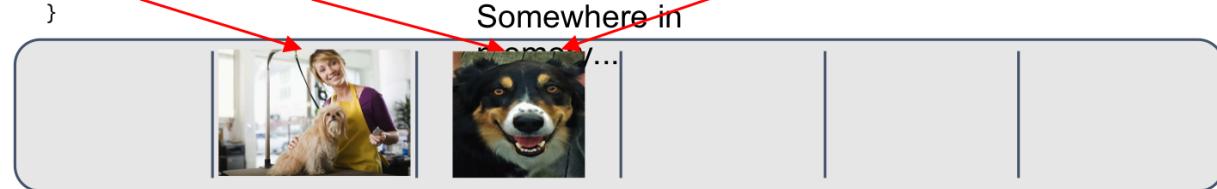


```

public class PetShop {
    public PetShop() {
        this.testGroomer();
    }

    public void testGroomer() {
        Dog django = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(django);
    }
}

```

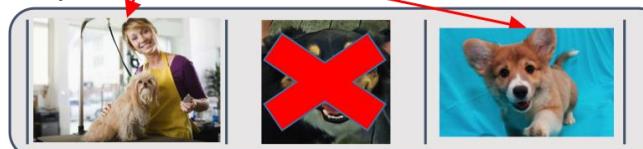


```

public class PetShop {
    /* This is the constructor! */
    public PetShop() {
        this.testGroomer();
    }

    public void testGroomer() {
        Dog django = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(django);
        django = new Dog(); //old ref garbage collected
        groomer.groom(django);
    }
}

```



```

public class DogGroomer {
    public DogGroomer() {
        // this is the constructor!
    }

    public void groom(Dog shaggyDog) {
        // code that grooms shaggyDog goes here!
    }
}

```

Somewhere in

declaration

```

public class PetShop {
    private DogGroomer _groomer;
}

/* This is the constructor! */
public PetShop() {
    _groomer = new DogGroomer();
    this.testGroomer();
}

public void testGroomer() {
    Dog django = new Dog(); //local var
    _groomer.groom(django);
}

```

assignment

This Lecture

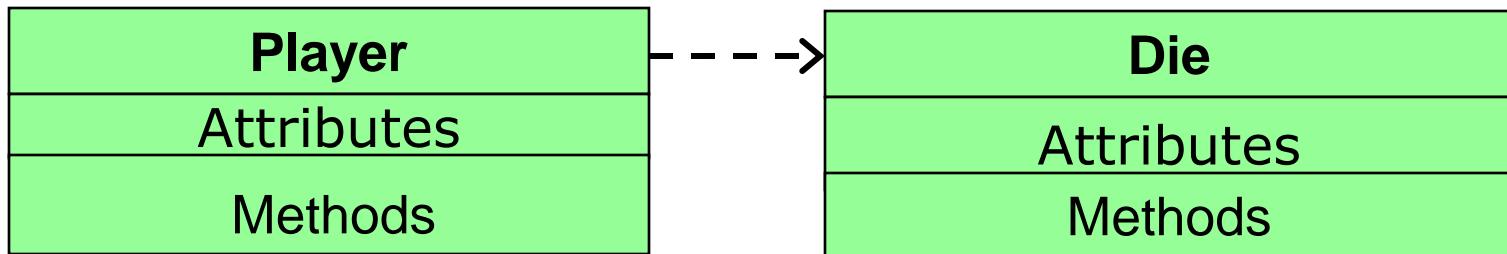
- Class relationships

Slide acknowledgements: Internet resources + CS15, Brown University

UML: Quick Introduction

- UML stands for the *Unified Modeling Language*
 - We will cover this in depth in later lectures
- Much detailed than sequence diagrams
- *UML diagrams* show relationships among classes and objects
 - Lines connecting the classes
- A UML *class diagram* consists of one or more classes, each with sections for the class name, attributes (data), and operations (methods)

A Sample UML Class Diagram



Class Relationships

- The whole point of OOP is that your code replicates real world objects, thus making your code readable and maintainable
- When we say real world, the real world has relationships
- When writing a program, need to keep in mind “big picture”—how are different classes related to each other?

Most Common Class Relationships

- **Composition**
 - A “contains” B – “part-of” relationship
- **Association/Aggregation**
 - A “knows-about” B – “uses-a” or “has-a” relationship
- **Dependency**
 - A “depends on” B – Sort of “depends-on” relationship
- Inheritance
 - HarleyDavidson “is-a” Bike

Composition Relationship

- Class A **contains** object of class B
 - A **instantiates** B
- Thus A knows about B and can call methods on it
- But this is **not symmetrical!** B can't automatically call methods on A
- Lifetime?
 - **The death relationship**
 - Garbage collection of A means B also gets garbage collected

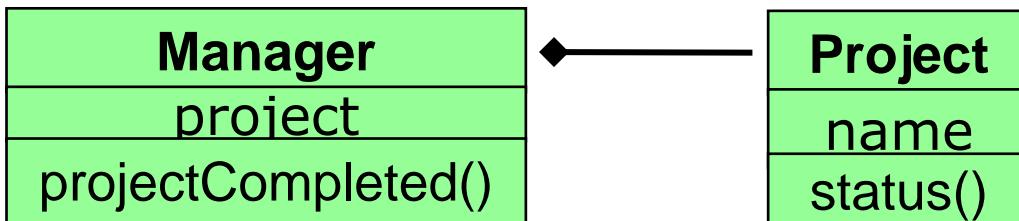
Composition in UML

- Represented by a solid arrow with diamond head
- In below UML diagram, A is composed of B



Composition Example (1/2)

- Manager is fixed for a project and is responsible for the timely completion of the project. If manager leaves, project is ruined



```
class Project {  
    private String name;  
    public boolean status() { ... }  
    ....  
}  
// A manager is fixed for a project  
class Manager {  
    private Project project;  
    public Manager() {  
        this.project = new Project("ABC");  
    }  
    public boolean projectCompleted() {  
        return project.status();  
    }  
}
```

Composition Example (2/2)

- PetShop **contains** a DogGroomer
- Composition relationship because PetShop itself instantiates a DogGroomer with “new DogGroomer();”
- Since PetShop created a DogGroomer and stored it in an instance variable, all PetShop’s methods “know” about the _groomer and can access it

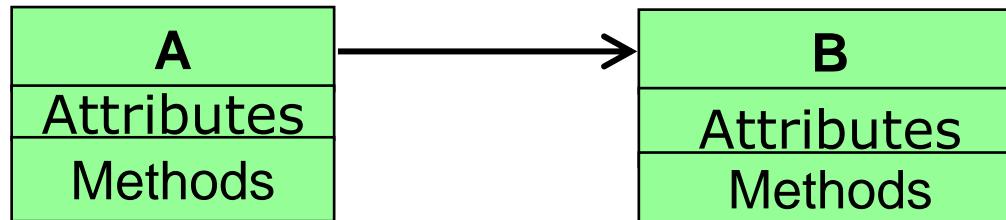
```
public class PetShop {  
  
    private DogGroomer _groomer;  
  
    public PetShop() {  
        _groomer = new DogGroomer();  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog(); //local var  
        _groomer.groom(django);  
    }  
}
```

Association Relationship

- Association is a relationship between two objects
- Class A and class B are **associated** if A “knows about” B, but B is not a component of A
- But this is **not symmetrical!**
- **Class A holds a class level reference to class B**
- Lifetime?
 - Objects of class A and B have their own lifetime, i.e., they can exist without each other

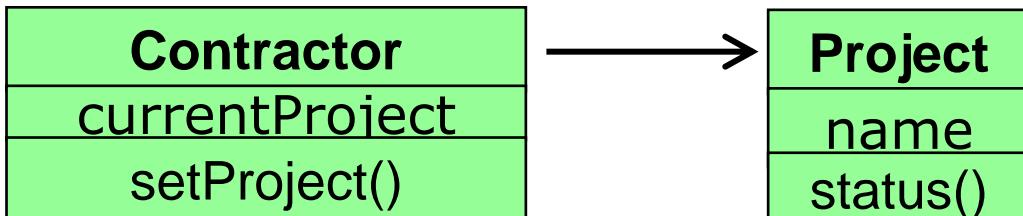
Association in UML

- Represented by a solid arrow
- In below UML diagram, A holds a reference of B



Association Example (1/4)

- A contractor's project keeps changing as per company's policy and contractor's performance



```
class Project {  
    private String name;  
    public boolean status() { ... }  
    .....  
}  
// Contractor's project keep changing  
class Contractor {  
    private Project currentProject;  
    public Contractor(Project proj) {  
        this.currentProject = proj;  
    }  
    public void setProject(Project proj){  
        this.currentProject = proj;  
    }  
}
```

Associations Example (2/4)

- **Association** means that one object knows about another object that is not one of its components

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        shaggyDog.setHairLength(1);  
    }  
}
```

Associations Example (2/4)

- As noted, **PetShop** contains a **DogGroomer**, so it can send messages to the **DogGroomer**
- But what if the **DogGroomer** needs to send messages to the **PetShop** she works in?
 - the **DogGroomer** probably needs to know several things about her **PetShop**: for example, operating hours, grooming supplies in stock, customers currently in the shop...

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        shaggyDog.setHairLength(1);  
    }  
}
```

Associations Example (2/4)

- The **PetShop** keeps track of such information in its properties
- Can set up an **association** so that **DogGroomer** can send her **PetShop** messages to retrieve information she needs

```
public class DogGroomer {  
  
    public DogGroomer() {  
        // this is the constructor!  
    }  
  
    public void groom(Dog shaggyDog) {  
        shaggyDog.setHairLength(1);  
    }  
}
```

Associations Example (2/4)

- This is what the full association looks like
- Let's break it down line by line
- **But note we're not yet making use of the association in this fragment**

```
public class DogGroomer {  
  
    private PetShop _petShop;  
  
    public DogGroomer(PetShop myPetShop) {  
        _petShop = myPetShop; // store the assoc.  
    }  
  
    public void groom(Dog shaggyDog) {  
        shaggyDog.setHairLength(1);  
    }  
}
```

Associations Example (2/4)

- We declare an instance variable named _petShop
- We want this variable to record the instance of PetShop that the DogGroomer belongs to

```
public class DogGroomer {  
  
    private PetShop _petShop;  
  
    public DogGroomer(PetShop myPetShop) {  
        _petShop = myPetShop; // store the assoc.  
    }  
  
    public void groom(Dog shaggyDog) {  
        shaggyDog.setHairLength(1);  
    }  
}
```

Associations Example (2/4)

- Modified `DogGroomer`'s constructor to take in a parameter of type `PetShop`
- Constructor will refer to it by the name `myPetShop`
- Whenever we instantiate a `DogGroomer`, we'll need to pass it an instance of `PetShop` as an argument. Which? The `PetShop` instance that created the `DogGroomer`, hence use `this`

```
public class DogGroomer {  
    private PetShop _petShop;  
  
    public DogGroomer(PetShop myPetShop) {  
        _petShop = myPetShop; // store the assoc.  
    }  
    //groom method elided  
}  
  
public class PetShop {  
    private DogGroomer _groomer;  
  
    public PetShop() {  
        _groomer = new DogGroomer(this);  
        this.testGroomer();  
    }  
  
    //testGroomer() elided  
}
```

Associations Example (2/4)

- Now store `myPetShop` in instance variable `_petShop`
- `_petShop` now points to same `PetShop` instance passed to its constructor
- After constructor has been executed and can no longer reference `myPetShop`, any `DogGroomer` method can still access same `PetShop` instance by the name `_petShop`

```
public class DogGroomer {  
  
    private PetShop _petShop;  
  
    public DogGroomer(PetShop myPetShop) {  
        _petShop = myPetShop; // store the assoc.  
    }  
  
    public void groom(Dog shaggyDog) {  
        shaggyDog.setHairLength(1);  
    }  
}
```

Associations Example (2/4)

- Let's say we've written an accessor method and a mutator method in the `PetShop` class:
`getClosingTime()` and
`setNumCustomers(int customers)`
- If the `DogGroomer` ever needs to know the closing time, or needs to update the number of customers, she can do so by calling
 - `getClosingTime()`
 - `setNumCustomers(int customers)`

```
public class DogGroomer {  
  
    private PetShop _petShop;  
    private Time _closingTime;  
  
    public DogGroomer(PetShop myPetShop) {  
        _petShop = myPetShop; // store assoc.  
        _closingTime = myPetShop.getClosingTime();  
        _petShop.setNumCustomers(10);  
    }  
}
```

Association: Under the Hood (1/5)

```
public class PetShop {  
    private DogGroomer _groomer;  
  
    public PetShop() {  
        _groomer = new DogGroomer(this);  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        _groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
    private PetShop _petShop;  
  
    public DogGroomer(PetShop myPetShop) {  
        _petShop = myPetShop;  
    }  
  
    /* groom and other methods elided for this  
       example */  
}
```

Somewhere in memory...



Association: Under the Hood (2/5)

```
public class PetShop {  
    private DogGroomer _groomer;  
  
    public PetShop() {  
        _groomer = new DogGroomer(this);  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        _groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
    private PetShop _petShop;  
  
    public DogGroomer(PetShop myPetShop) {  
        _petShop = myPetShop;  
    }  
  
    /* groom and other methods elided for this  
    example */  
}
```

Somewhere in memory...



Somewhere else in our code, someone calls new PetShop(). An instance of PetShop is created somewhere in memory and PetShop's constructor initializes all its instance variables (just a DogGroomer here)

Association: Under the Hood (3/5)

```
public class PetShop {  
    private DogGroomer _groomer;  
  
    public PetShop() {  
        _groomer = new DogGroomer(this);  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        _groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
    private PetShop _petShop;  
  
    public DogGroomer(PetShop myPetShop) {  
        _petShop = myPetShop;  
    }  
  
    /* groom and other methods elided for this  
    example */  
}
```

Somewhere in memory...



The PetShop instantiates a new DogGroomer, passing itself in as an argument to the DogGroomer's constructor (remember the this keyword?)

Association: Under the Hood (4/5)

```
public class PetShop {  
    private DogGroomer _groomer;  
  
    public PetShop() {  
        _groomer = new DogGroomer(this);  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        _groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
    private PetShop _petShop;  
  
    public DogGroomer(PetShop myPetShop) {  
        _petShop = myPetShop;  
    }  
  
    /* groom and other methods elided for this  
       example */  
}
```

Somewhere in memory...



When the DogGroomer's constructor is called, its parameter, myPetShop, points to the same PetShop that was passed in as an argument.

Association: Under the Hood (5/5)

```
public class PetShop {  
    private DogGroomer _groomer;  
  
    public PetShop() {  
        _groomer = new DogGroomer(this);  
        this.testGroomer();  
    }  
  
    public void testGroomer() {  
        Dog django = new Dog();  
        _groomer.groom(django);  
    }  
}
```

```
public class DogGroomer {  
    private PetShop _petShop;
```

```
    public DogGroomer(PetShop myPetShop) {  
        _petShop = myPetShop;  
    }  
  
    /* groom and other methods elided for this  
       example */  
}
```

Somewhere in memory...



The DogGroomer sets its `_petShop` instance variable to point to the same `PetShop` it received as an argument. Now it “knows about” the `PetShop` that instantiated it! And therefore so do all its methods...

Associations Example (3/4)

- Here we have the class **Professor**
- We want **Professor** to know about his TAs—he didn't create them or vice versa, hence no containment – they are peer objects
- Let's set up associations!

```
public class Professor {  
    // declare instance variables here  
    // and here...  
    // and here...  
    // and here!  
  
    public Professor(/* parameters */) {  
  
        // initialize instance variables!  
        // ...  
        // ...  
        // ...  
    }  
  
    /* additional methods elided */  
}
```

Associations Example (3/4)

- The Professor needs to know about 4 TAs, all of whom will be instances of the class TA
- Once he knows about them, he can call methods of the class TA on them: remindTA, runRefresherModule, etc.
- Take a minute and try to fill in this class

```
public class Professor {  
    // declare instance variables here  
    // and here...  
    // and here...  
    // and here!  
  
    public Professor(/* parameters */) {  
  
        // initialize instance variables!  
        // ...  
        // ...  
        // ...  
    }  
  
    /* additional methods elided */  
}
```

Associations Example (3/4)

- Here's our solution!
- Remember, you can choose your own names for the instance variables and parameters
- The Professor can now send a message to one of his TAs like this:

```
_ta1._runRefresherModule();
```

```
public class Professor {  
  
    private TA _ta1;  
    private TA _ta2;  
    private TA _ta3;  
    private TA _ta4;  
  
    public Professor(TA firstTA,  
                    TA secondTA, TA thirdTA  
                    TA fourthTA) {  
  
        _ta1 = firstTA;  
        _ta2 = secondTA;  
        _ta3 = thirdTA;  
        _ta4 = fourthTA;  
    }  
  
    /* additional methods elided */  
}
```

Associations Example (3/4)

- We've got the Professor class down
- Now let's create a professor and TAs from a class that contains all of them: Course
- Try and fill in this class!
 - You can assume that the TA class takes no parameters in its constructor.

```
public class Course {  
    // declare Professor instance var.  
    // declare four TA instance vars.  
    // ...  
    // ...  
    // ...  
  
    public Course() {  
        // instantiate the four TAs  
        // ...  
        // ...  
        // instantiate the professor!  
    } }
```

Associations Example (3/4)

- We declare `_vivek`, `_akanksha`, `_akash`, `_alind` and `_abhiprayah` as instance variables
- In the constructor, we instantiate them
- Since the constructor of `Professor` takes in 4 TAs, we pass in `_akanksha`, `_akash`, `_alind` and `_abhiprayah`

```
public class Course {  
  
    private Professor _Arun;  
    private TA _Madiha;  
    private TA _Daksh;  
    private TA _Ramu;  
    private TA _Binu;  
  
    public Course() {  
        _akanksha = new TA();  
        _akash = new TA();  
        _alind = new TA();  
        _abhiprayah = new TA();  
        _vivek = new Professor(_akanksha,  
                             _akash , _alind , _abhiprayah);  
    }  
}
```

Associations Example (3/4)

```
public class Professor {  
  
    private TA _ta1;  
    private TA _ta2;  
    private TA _ta3;  
    private TA _ta4;  
  
    public Professor(TA firstTA,  
                    TA secondTA, TA thirdTA  
                    TA fourthTA){  
  
        _ta1 = firstTA;  
        _ta2 = secondTA;  
        _ta3 = thirdTA;  
        _ta4 = fourthTA;  
        _ta1.runRefresherModule();  
    }  
    /* additional methods elided */  
}
```

```
public class Course {  
  
    private Professor _Arun;  
    private TA _Madiha;  
    private TA _Daksh;  
    private TA _Ramu;  
    private TA _Binu;  
  
    public Course() {  
        _Madiha = new TA();  
        _Daksh = new TA();  
        _Ramu = new TA();  
        _Binu = new TA();  
        _Arun = new Professor(_Madiha,  
                            _Daksh , _Ramu , _Binu);  
    }  
}
```

Associations Example (4/4)

- What if we want the TAs to know about **Professor** too?
- Need to set up another association
- Can we just do the same thing?

```
public class Course {  
  
    private Professor _Arun;  
    private TA _Madiha;  
    private TA _Daksh;  
    private TA _Ramu;  
    private TA _Binu;  
  
    public Course() {  
        _Madiha = new TA();  
        _Daksh = new TA();  
        _Ramu = new TA();  
        _Binu = new TA();  
        _Arun = new Professor(_Madiha,  
                             _Daksh , _Ramu , _Binu);  
    }  
}
```

Associations Example (4/4)

- This doesn't work: when we instantiate `_Madiha`, `_Daksh`, `_Ramu` and `_binu`, we would like to pass them an argument, `_vivek`
- But `_Arun` hasn't been instantiated yet! And can't initialize `_Arun` first because the TAs haven't been created yet...
- What can we try instead?

```
public class Course {  
  
    private Professor _Arun;  
    private TA _Madiha;  
    private TA _Daksh;  
    private TA _Ramu;  
    private TA _Binu;  
  
    public Course() {  
        _Madiha = new TA();  
        _Daksh = new TA();  
        _Ramu = new TA();  
        _Binu = new TA();  
        _Arun = new Professor(_Madiha,  
        _Daksh , _Ramu , _Binu);  
    }  
}
```

Associations Example (4/4)

- Need a way to pass `_Arun` to `_Madiha`, `_Daksh`, `_Ramu` and `_binu` after we instantiate `_Arun`
- Use a new method, `setProf`, and pass each TA `_Arun`

```
public class Course {  
  
    private Professor _Arun;  
    private TA _Madiha;  
    private TA _Daksh;  
    private TA _Ramu;  
    private TA _Binu;  
  
    public Course() {  
        _Madiha = new TA();  
        _Daksh = new TA();  
        _Ramu = new TA();  
        _Binu = new TA();  
        _Arun = new Professor(_Madiha,  
        _Daksh , _Ramu , _Binu);  
    }  
}
```

Associations Example (4/4)

```
public class TA {  
  
    private Professor _professor;  
  
    public TA() {  
  
        //Other code elided  
  
    }  
  
    public void setProf(Professor prof) {  
        _professor = prof;  
    }  
}
```

- Now each TA will know about _Arun!

```
public class Course {  
  
    private Professor _Arun;  
    private TA _Madiha;  
    private TA _Daksh;  
    private TA _Ramu;  
    private TA _Binu;  
  
    public Course() {  
        _Madiha = new TA();  
        _Daksh = new TA();  
        _Ramu = new TA();  
        _Binu = new TA();  
        _Arun = new Professor(_Madiha,  
        _Daksh , _Ramu , _Binu);  
  
        _Madiha.setProf(_Arun);  
        _Daksh.setProf(_Arun);  
        _Ramu.setProf(_Arun);  
        _Binu.setProf(_Arun);  
    }  
}
```

Question

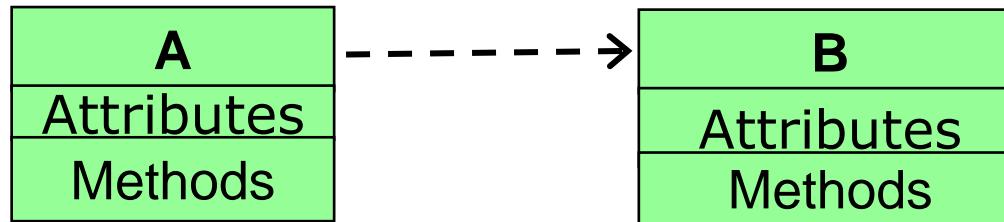
- What happens if `setProf` is never called?
- Will the TAs be able to call methods on the `Professor`?

Dependency

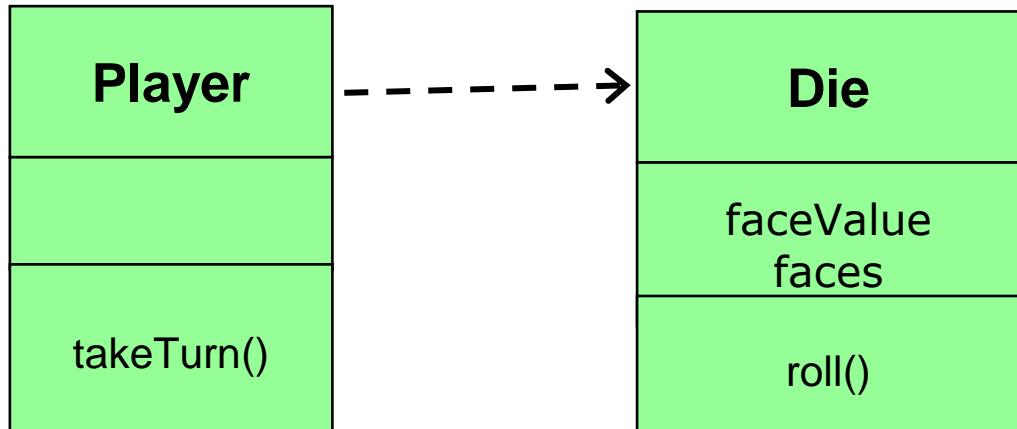
- Class A **depends** on class B if A cannot carry out its work without B, but B is neither a component of A nor it has association with A
- **A is requesting service from an object of class B**
 - A or B “**doesn’t know**” about each other (no association)
 - A or B “**doesn’t contain**” each other (no composition)
- But this is **not symmetrical!** B doesn’t depends on A

Dependency in UML

- Represented by a dashed arrow starting from the dependent class to its dependency
 - A is dependent on B
 - A is requesting service from B

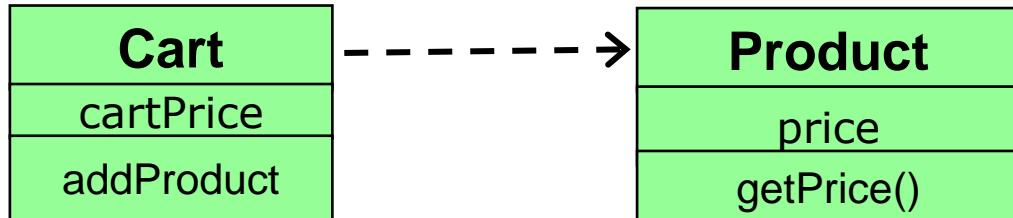


Dependency Example (1/3)



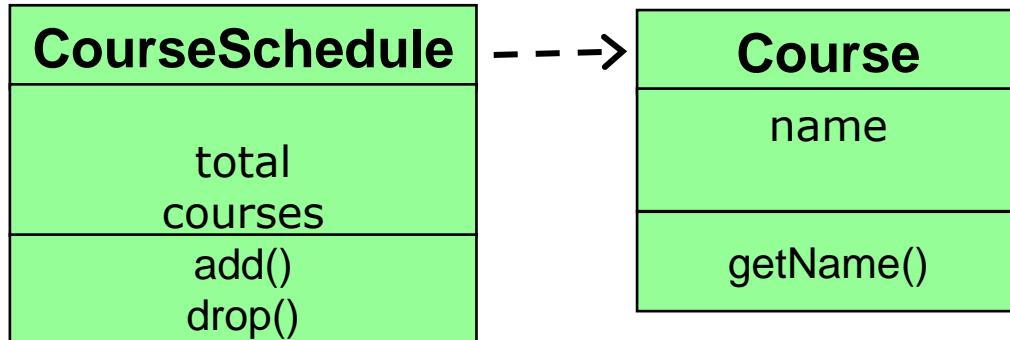
```
class Die {  
    private int faceValue, faces;  
    ....  
    public void roll() { .... }  
}  
  
class Player {  
    public void takeTurn(Die die) {  
        die.roll();  
    }  
}
```

Dependency Example (2/3)



```
class Product {  
    private double price;  
    ....  
    public double getPrice() { .... }  
}  
  
class Cart {  
    private double cartPrice;  
    public void addProduct(Product p) {  
        cartPrice += p.getPrice();  
    }  
}
```

Dependency Example (3/3)



```
class Course {  
    private String name;  
    ....  
    public String getName() { ..... }  
}  
  
class CourseSchedule {  
    private int total;  
    private String courses[];  
    public void addCourse(Course c) {  
        courses[total++] = c.getName();  
    }  
    ....  
}
```

CSE201: Monsoon 2024 Advanced Programming

Lecture 04: Interfaces

Dr. Arun Balaji Buduru

Head, Center of Technology in Policing

Founding Head, Usable Security Group (USG)

Associate Professor, Dept. of CSE | HCD

IIT-Delhi, India

Last Lecture

- Class relationships

- When writing a program, need to keep in mind “big picture”— how are different classes related to each other?

- Association
 - Class A and class B are **associated** if A “knows about” B, but B is not a component of A
 - Class A holds a class level reference to class B

- Composition
 - Class A **contains** object of class B
 - A **instantiates** B
 - The death relationship
 - B is garbage collected when A gets garbage collected

- Dependency
 - Neither class A or class B “knows about” each other, nor one of them is a “component” of the other. However, if A requests a service from B then A is said to be dependent on B

```
class Cart {  
    private double price;  
    public void addProduct(Product P) {  
        price+=P.getPrice();  
    }  
}
```

```
class Project {  
    private String name;  
    public boolean status() { ... }  
    ....  
}  
// Contractor's project keep changing  
class Contractor {  
    private Project currentProject;  
    public Contractor(Project proj) {  
        this.currentProject = proj;  
    }  
    public void setProject(Project proj){  
        this.currentProject = proj;  
    }  
}
```

```
class Project {  
    private String name;  
    public boolean status() { ... }  
    ....  
}  
// A manager is fixed for a project  
class Manager {  
    private Project project;  
    public Manager() {  
        this.project = new Project("ABC");  
    }  
    public boolean projectCompleted() {  
        return project.status();  
    }  
}
```

This Lecture

- Interfaces in Java
 - Declaring
 - Defining

Slide acknowledgements: CS15, Brown University

Recall: Declaring vs. Defining Methods

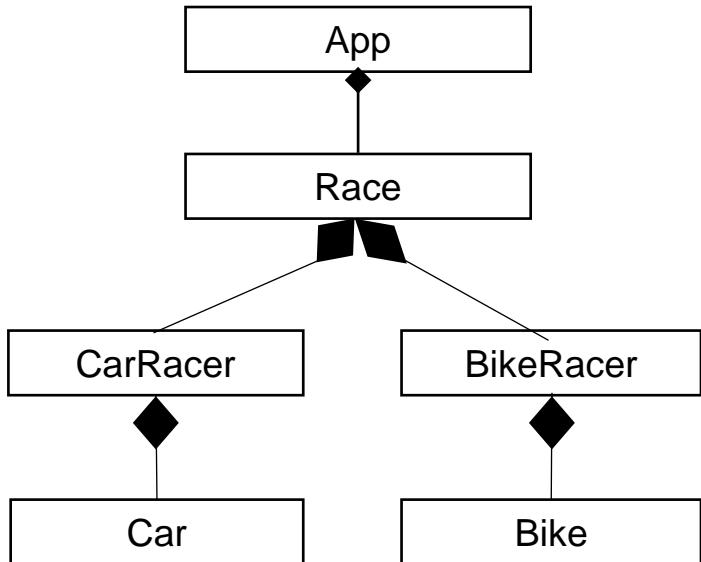
- What's the difference between **declaring** and **defining** a method?
 - method **declaration** is the scope (**public**), return type (**void**), name and parameters (**makeSounds()**)
 - method **definition** is the body of the method – the actual implementation (the code that actually makes the sounds)

```
public class Dog {  
    //constructor elided  
  
    public void makeSounds() {  
        this.bark();  
        this.whine();  
        this.bark();  
    }  
    public void bark() {  
        //code elided  
    }  
    public void whine() {  
        //code elided  
    }  
}
```

Using What You Know

- Imagine this program:
 - Sophia and Dan are racing from their home to city center
 - whoever gets there first, wins!
 - catch: they don't get to choose their method of transportation
- Design a program that
 - assigns mode of transportation to each racer
 - starts the race
- For now, assume transportation options are **Car** and **Bike**

What does our design look like?



- Imagine this program:
 - Sophia and Dan are racing from their home to city center
 - whoever gets there first, wins!
 - catch: they don't get to choose their method of transportation
- Design a program that
 - assigns mode of transportation to each racer
 - starts the race
- For now, assume transportation options are **Car** and **Bike**

Goal 1: Assign transportation to each racer

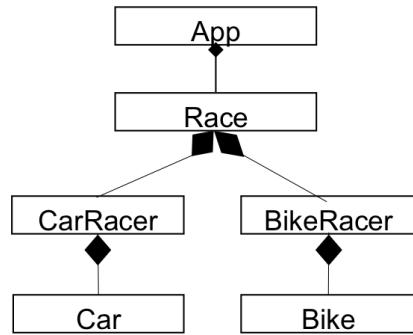
- Need transportation classes (something to give to racers)
- Let's use **Car** and **Bike** classes
- Both classes will need to describe how the transportation moves
 - **Car** needs **drive** method
 - **Bike** needs **pedal** method

Coding the project (1/4)

- Let's build transportation classes

```
public class Car {  
  
    public Car() {//constructor  
        //code elided  
    }  
    public void drive(){  
        //code elided  
    }  
    //more methods elided  
}
```

```
public class Bike {  
  
    public Bike() {//constructor  
        //code elided  
    }  
    public void pedal(){  
        //code elided  
    }  
    //more methods elided  
}
```



Goal 1: Assign transportation to each racer

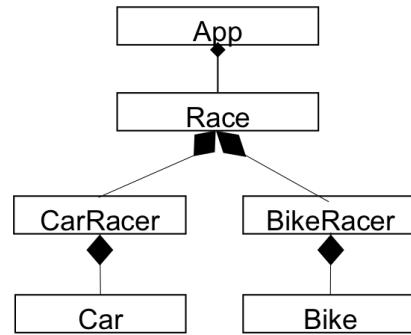
- Need racer classes that will use their type of transportation
 - `CarRacer`
 - `BikeRacer`
- What methods will we need? What capabilities should each `-Racer` class have?
- `CarRacer` needs to know when to use the car
 - write `useCar()` method
- `BikeRacer` needs to know when to use the bike
 - write `useBike()` method

Coding the project (2/4)

- Let's build the racer classes

```
public class CarRacer {  
    private Car _car;  
  
    public CarRacer() {  
        _car = new Car();  
    }  
  
    public void useCar(){  
        _car.drive();  
    }  
    //more methods elided  
}
```

```
public class BikeRacer {  
    private Bike _bike;  
  
    public BikeRacer() {  
        _bike = new Bike();  
    }  
  
    public void useBike(){  
        _bike.pedal();  
    }  
    //more methods elided  
}
```



Goal 2: Tell the racers to start the race

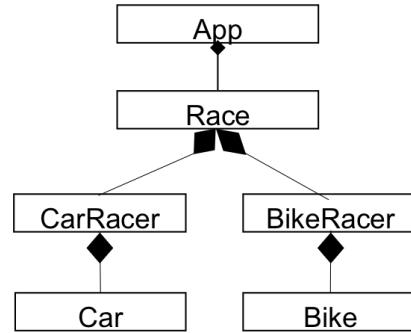
- Race class contains Racers
 - App contains Race
- Race class will have `startRace()` method
 - `startRace()` tells each racer to use their transportation
- `startRace()` gets called in App

`startRace:`

Tell `_dan` to `useCar`
Tell `_sophia` to `useBike`

Coding the project (3/4)

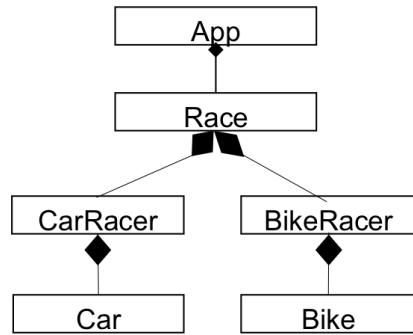
- Let's build the Race class



```
public class Race {  
    private CarRacer _dan;  
    private BikeRacer _sophia;  
  
    public Race() {  
        _dan = new CarRacer();  
        _sophia = new BikeRacer();  
    }  
  
    public void startRace() {  
        _dan.useCar();  
        _sophia.useBike();  
    }  
}
```

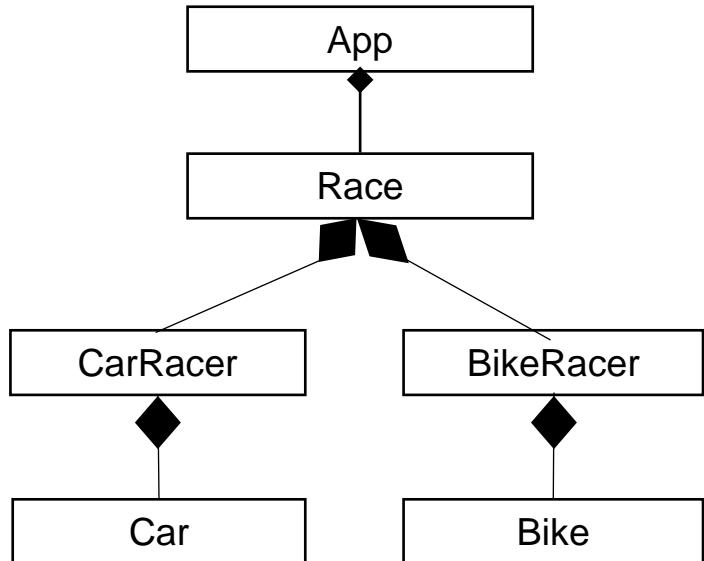
Coding the project (4/4)

```
public class App {  
    Race race;  
    public App() {  
        race = new Race();  
        race.startRace();  
    }  
  
    public static void main (String[] args) {  
        new App();  
    }  
}
```



- Now build the `App` class
- Now the race to the city center!

Recap: What does our design look like?



How would this program run?

- An instance of `App` gets initialized
- `App`'s constructor initializes an instance of `Race`
- `Race`'s constructor initializes `_dan` (`CarRacer`) and `_sophia` (`BikeRacer`)
 - `CarRacer`'s constructor initializes a `_car` (`Car`)
 - `BikeRacer`'s constructor initializes a `_bike`
- `App` calls `race.startRace()`
- `race` calls `_dan.useCar()` and `_sophia.useBike()`
- `_dan` calls `_car.drive()`
- `_sophia` calls `_bike.pedal()`

Can we do better?

Things to think about

- Do we need two different Racer classes?
 - Want multiple instances of Racers that use different modes of transportation
 - But how?

Solution 1: Create one Racer class with methods!

- Create one **Racer** class
 - define different methods for each type of transportation
- _dan is instance of **Racer** and elsewhere we have:

```
Car dansCar = new Car();
_dan.useCar(dansCar);
```

- **Car's drive()** method will be invoked
- But any given instance of **Racer** will need a new method to accommodate every kind of transportation!

```
public class Racer {
    public Racer(){
        //constructor
    }

    public void useCar(Car myCar){
        myCar.drive();
    }

    public void useBike(Bike myBike){
        myBike.pedal();
    }
}
```

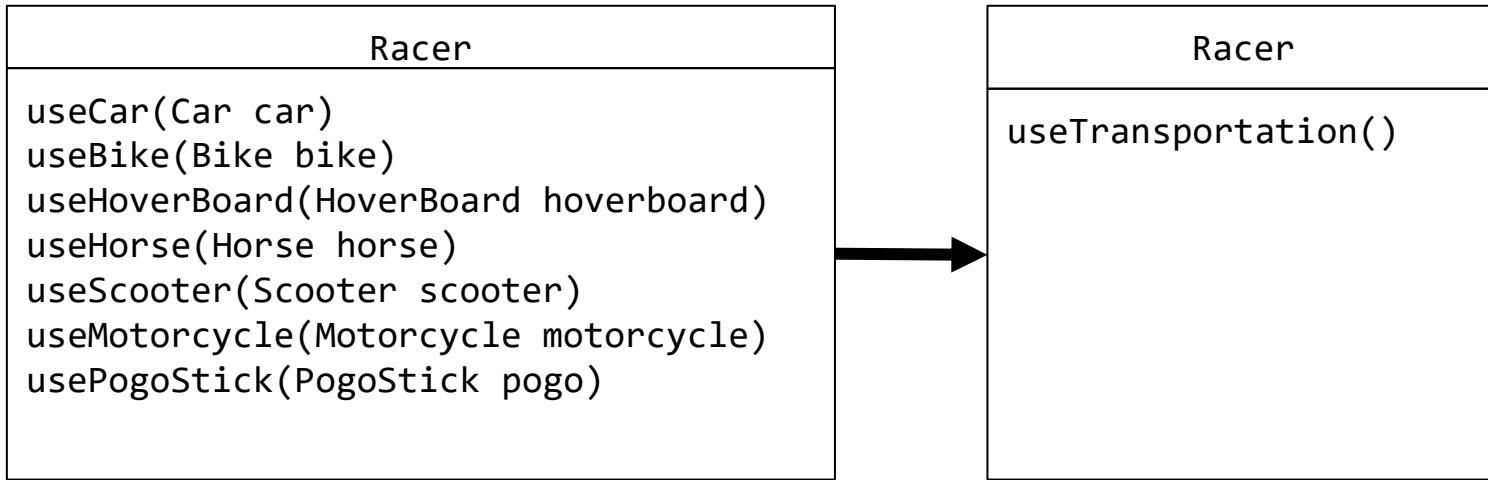
Question: What is the relationship between Racer+Car and Racer+Bike?

Solution 1 Drawbacks

- Now imagine 10 people join the race and so there are 10 different modes of transportation
- Writing these similar `useType()` methods are a lot of work for you, the developer, and inefficient coding style

```
public class Racer {  
  
    public Racer() {  
        //constructor  
    }  
  
    public void useCar(Car myCar){//code elided}  
    public void useBike(Bike myBike){//code elided}  
    public void useHoverboard(Hoverboard myHb){//code elided}  
    public void useHorse(Horse myHorse){//code elided}  
    public void useScooter(Scooter myScooter){//code elided}  
    public void useMotorcycle(Motorcycle myMc) { //code elided}  
    public void usePogoStick(PogoStick myPogo){//code elided}  
    // And more...  
}
```

Is there another solution?



- Can we go from left to right?

Interfaces: Spot the Similarities

- What do cars and bikes have in common?
- What do cars and bikes not have in common?



Cars vs. Bikes

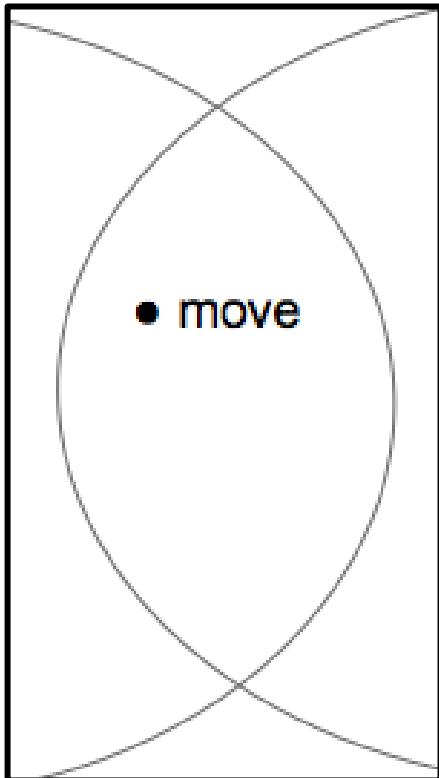
Cars

- Play radio
- Turn off/on headlights
- Turn off/on turn signal
- Lock/unlock doors

Bikes

- Move
- Drop kickstand
- Change gears

Digging deeper into the similarities



- How similar are they when they move?
 - do they move in same way?
- Not very similar
 - cars drive
 - bikes pedal
- Both can move, but in different ways

Can we model this in code?

- Many real-world objects have several broad similarities
 - cars and bikes can move
 - cars and laptops can play radio
- Take **Car** and **Bike** class
 - how can their similar functionalities get enumerated in one place?
 - how can their broad relationship get portrayed through code?

Car

- playRadio()
- lockDoors()
- unlockDoors()
- drive()

Bike

- dropKickstand()
- changeGears()
- pedal()

Introducing Interfaces

- **Interfaces** group similar capabilities/function of different classes together
- Model “acts-as” relationship
- **Cars** and **Bikes** could implement a **Transporter** interface
 - they can transport people from one place to another
 - “act as” transporters
 - objects that can move
 - have shared functionality, such as moving, braking, turning etc.
 - for this lecture, interfaces are **green** and classes that implement them **pink**

Introducing Interfaces

- Interfaces are contracts that classes agree to
- If classes choose to **implement** given interface, it must define all methods declared in interface
 - if classes don't implement one of interface's methods, the compiler raises error
 - later we'll discuss strong motivations for this contract enforcement
- Interfaces don't define their methods - implementing classes do
 - Interfaces **only** care about the fact that the methods get defined - not **how** – *implementation-agnostic*
- **Models similarities while ensuring consistency**
 - **What does this mean?**

Let's break that down

1) Models Similarities

2) Ensures Consistency

Models Similarities While Ensuring Consistency

- How does this help our program?
- We know **Cars** and **Bikes** both need to move
 - i.e., should all have some `move()` method
 - let compiler know that too!
- Let's make the **Transporter** interface!
 - what methods should the **Transporter** interface declare?
 - `move()`
 - only using a `move()` for simplicity, but `brake()`, etc. would also be useful
 - compiler doesn't care how method is defined, just that it's been defined
 - general tip: methods that interface declares **should model functionality all implementing classes share**

Declaring an Interface (1/4)

What does this look like?

```
public interface Transporter {  
    public void move();  
}
```

- That's it!
- Interfaces, just like classes, have their own .java file.
This file would be **Transporter.java**

Declaring an Interface (2/4)

What does this look like?

```
public interface Transporter {  
    public void move();  
}
```

- Declare it as **interface** rather than class

Declaring an Interface (3/4)

What does this look like?

```
public interface Transporter {  
    public void move();  
}
```

- Declare methods - the contract
- In this case, only one method required: `move()`
- All classes that sign contract (implement this interface) **must define actual implementation** of any declared methods

Declaring an Interface (4/4)

What does this look like?

```
public interface Transporter {  
    public void move();  
}
```

- Interfaces are only contracts, not classes that can be instantiated
- Interfaces can only declare methods - not define them
- Notice: method declaration end with **semicolons**, not curly braces!

Implementing an Interface (1/6)

Let's modify `Car`

```
public class Car implements Transporter {  
  
    public Car() {  
        // constructor  
    }  
  
    public void drive() {  
        // code for driving the car  
    }  
  
}
```

- Let's modify `Car` to implement `Transporter`
 - declare that `Car` “acts-as” `Transporter`
- Add `implements Transporter` to class declaration
- Promises compiler that `Car` will define all methods in `Transporter` interface
 - i.e., `move()`
- Will this code compile?

Implementing an Interface (2/6)

```
public class Car implements Transporter {  
  
    public Car() {  
        // constructor  
    }  
  
    public void drive() {  
        // code for driving the car  
    }  
}
```

“Error: **Car** does not override method **move()** in **Transporter**” *

- Will this code compile?
 - nope :(
- Never implemented **move()** and **drive()** - doesn't suffice.
Compiler will complain accordingly

*Note: the full error message is “**Car** is not abstract and does not override abstract method **move()** in **Transporter**.” We’ll get more into the meaning of abstract in a later lecture.

Implementing an Interface (3/6)

```
public class Car implements Transporter {  
  
    public Car() {  
        // constructor  
    }  
  
    public void drive() {  
        //code for driving car  
    }  
  
    @Override  
    public void move() {  
        this.drive();  
    }  
}
```

- Next: honor contract by defining a `move()` method
- Method ***signature*** (name and number/type of arguments) **must match how its declared in interface**

Implementing an Interface (4/6)

What does `@Override` mean?

```
public class Car implements Transporter {  
  
    public Car() {  
        // constructor  
    }  
  
    public void drive() {  
        //code for driving car  
    }  
  
    @Override  
    public void move() {  
        this.drive();  
    }  
}
```

- Include `@Override` right above the method signature
- `@Override` is an annotation – a signal to the compiler (and to anyone reading your code)
 - allows compiler to enforce that interface actually has method declared
 - more explanation of `@Override` in next lecture
- Annotations, like comments, have **no effect on how code behaves** at runtime

Implementing an Interface (5/6)

- Defining interface method is like defining any other method
- Definition can be as complex or as simple as it needs to be
- Ex.: Let's modify `Car`'s move method to include braking
- What will instance of `Car` do if `move()` gets called on it?

```
public class Car implements Transporter {  
    public Car() {  
        //code elided  
    }  
    public void drive(){  
        //code elided  
    }  
    @Override  
    public void move(){  
        this.drive();  
        this.brake();  
        this.drive();  
    }  
    //more methods elided  
}  
  
public class Racer {  
    //previous code elided  
    public void useTransportation(  
        Transporter transport) {  
        transport.move(); //Polymorphism  
    }  
}
```

Implementing an Interface (6/6)

- As with signing multiple contracts, classes can implement multiple interfaces
 - “I signed my rent agreement, so I'm a renter, but I also signed my employment contract, so I'm an employee. I'm the same person.”
 - what if I wanted `Car` to change color as well?
 - create a `Colorable` interface
 - add that interface to `Car`'s class declaration
- Implementing class must define **every single method** in each of its interfaces

```
public interface Colorable {  
  
    public void setColor(Color c);  
    public Color getColor();  
  
}  
  
public class Car implements Transporter, Colorable{  
  
    public Car(){ //body elided }  
    public void drive(){ //body elided }  
    public void move(){ //body elided }  
    public void setColor(Color c){ //body elided }  
    public Color getColor(){ //body elided }  
}
```

Summary

- Interfaces are **formal contracts** and **ensure consistency**
 - compiler will check to ensure all methods declared in interface are defined
- Can trust that any object from class that implements **Transporter** can **move()**
- Will know how 2 classes are related if both implement **Transporter**

Question

Given the following interface:

```
public interface Clickable {  
    public void click();  
}
```

Which of the following would work as an implementation of the Clickable interface? (don't worry about what changeXPosition does)

A.

```
public void click() {  
    this.changeXPosition(100.0);  
}
```

C.

```
public void clickIt() {  
    this.changeXPosition(100.0);  
}
```

B.

```
public void click(double xPositon) {  
    this.changeXPosition(xPosition);  
}
```

D.

```
public double click() {  
    return this.changeXPosition(100.0);  
}
```

CSE201: Monsoon 2024 Advanced Programming

Lecture 05: Interfaces and Polymorphism

Dr. Arun Balaji Buduru

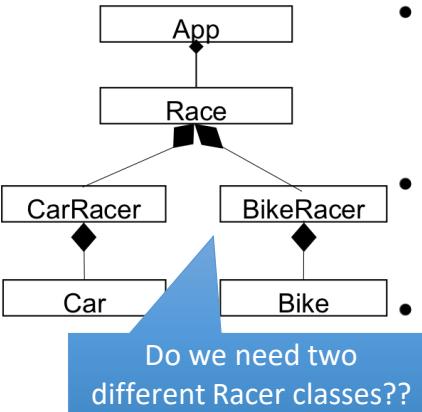
Head, Center of Technology in Policing

Founding Head, Usable Security Group (USG)

Associate Professor, Dept. of CSE | HCD

IIT-Delhi, India

Last Lecture

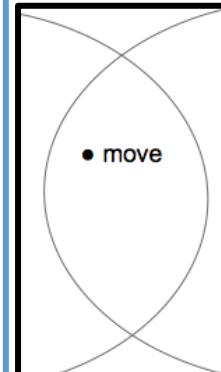


- Imagine this program:
 - Sophia and Dan are racing from their home to city center
 - whoever gets there first, wins!
 - catch: they don't get to choose their method of transportation
- Design a program that
 - assigns mode of transportation to each racer
 - starts the race
- For now, assume transportation options are **Car** and **Bike**

How about one Racer class with different methods?

```
public class Racer {  
    public Racer() {  
        //constructor  
    }  
    public void useCar(Car myCar){//code elided}  
    public void useBike(Bike myBike){//code elided}  
    public void useHoverboard(Hoverboard myHb){//code elided}  
    public void useHorse(Horse myHorse){//code elided}  
    public void useScooter(Scooter myScooter){//code elided}  
    public void useMotorcycle(Motorcycle myMc) { //code elided}  
    public void usePogoStick(PogoStick myPogo){//code elided}  
    // And more...  
}
```

Any similarities?



Interfaces in Java

- Group similar capabilities/function of different classes together
- Interfaces can only declare methods - not define them
- Interfaces are contracts that classes agree to
- If classes choose to **implement** given interface, it must define all methods declared in interface
 - if classes don't implement one of interface's methods, the compiler raises error

Declaring an Interface

```
public interface Transporter {  
    public void move();  
}
```

@Override is an annotation – a signal to the compiler (and to anyone reading your code)

Implementing an Interface

```
public class Car implements Transporter {  
    public Car() {  
        //code elided  
    }  
    public void drive(){  
        //code elided  
    }  
    @Override  
    public void move(){  
        this.drive();  
        this.brake();  
        this.drive();  
    }  
    //more methods elided  
}
```

This Lecture

- Interfaces and Polymorphism

Slide acknowledgements: CS15, Brown University

Back to the Race

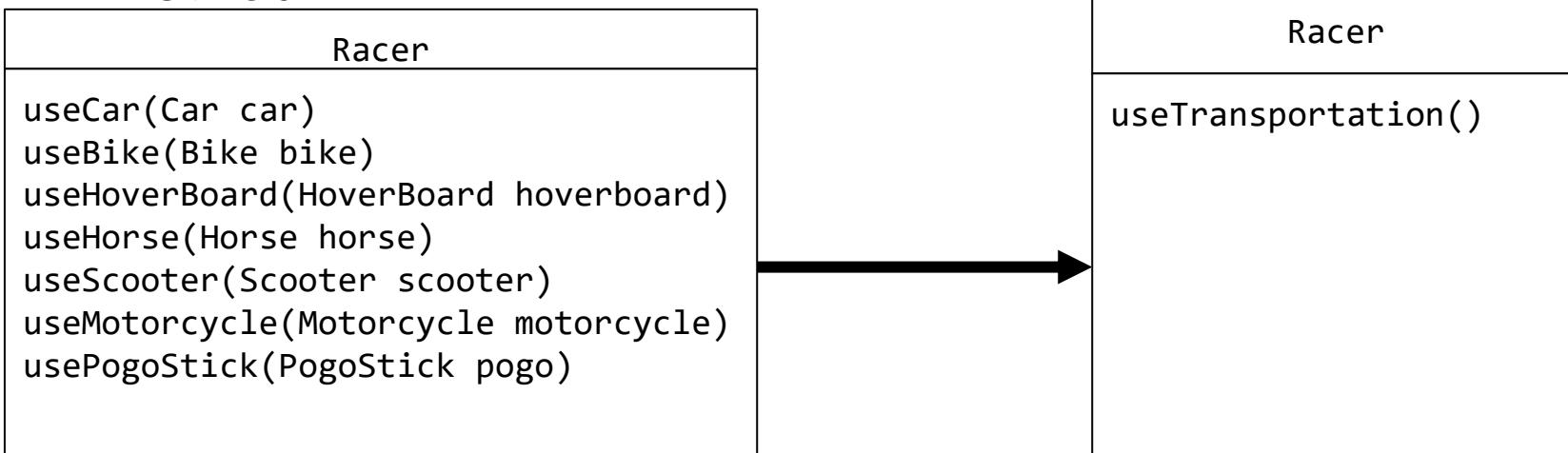
- Let's make transportation classes use an interface

```
public class Car implements Transporter{  
  
    public Car() {  
        //code elided  
    }  
    public void drive(){  
        //code elided  
    }  
    @Override  
    public void move() {  
        this.drive();  
    }  
    //more methods elided  
}
```

```
public class Bike implements Transporter{  
  
    public Bike() {  
        //code elided  
    }  
    public void pedal(){  
        //code elided  
    }  
    @Override  
    public void move() {  
        this.pedal();  
    }  
    //more methods elided  
}
```

Leveraging Interfaces

- Given that there's **guarantee** anything that implements **Transporter** knows how to **move**, how can it be leveraged to create single **useTransportation()** method?



Introducing Polymorphism

- Poly = many, morph = forms
- A way of coding **generically**
 - way of referencing many related objects as one generic type
 - cars and bikes can both `move()` → refer to them as **Transporter** objects
 - phones and camera can both `getCharged()` → refer to them as **Chargeable** objects, i.e., objects that implement **Chargeable** interface
 - cars and mobile phones can both `playRadio()` → refer to them as **RadioPlayer** objects
- How do we write one generic `useTransportation()` method?

What would this look like in code?

```
public class Racer {  
  
    //previous code elided  
    public void useTransportation(Transporter transporter transportation) {  
        transportation.move();  
    }  
  
}
```



This is polymorphism!
transportation object passed
in could be instance of **Car**,
Bike, etc., i.e., any class that
implements the interface

Let's break this down.

```
public class Racer {  
  
    //previous code elided  
    public void useTransportation(Transporter transportation) {  
        transportation.move();  
    }  
  
}
```

1. Actual vs. Declared Type
2. Method resolution

Actual vs. Declared Type (1/2)

- Consider following piece of code:

```
Transporter dansCar = new Car();
```

- ...is that legal?
 - doesn't Java do strict type checking? (type on LHS = type on RHS)
 - how can instances of `Car` get stored in `Transporter` variable?

Actual vs. Declared Type (2/2)

- Can treat **Car/Bike** object as **Transporter** objects
- **Car** is the **actual type**
 - Java will look in this class for the definition of the method
- **Transporter** is **declared type**
 - Java will limit caller so it can only call methods on instances that are declared as **Transporter** objects
- If **Car** defines **playRadio()** method. Is **transportation.playRadio()** correct?

```
Transporter transportation = new Car();  
transportation.playRadio();
```



Nope. The **playRadio()** method is not declared in **Transporter** interface, therefore Java does not recognize it as viable method call

Determining the Declared Type

- What methods do **Car** and **Bike** have in common?
 - `move()`
- How do we know that?
 - they implement **Transporter**
 - guarantees that they have `move()` method
- Think of **Transporter** like the “lowest common denominator”
 - it’s what all transportation classes will have in common

```
Bike implements Transporter
void move();
void dropKickstand();//etc.
```

```
Car implements Transporter
void move();
void playRadio();//etc.
```

Is this legal?

Transporter sophiasBike = new Bike(); 

Transporter sophiasCar = new Car(); 

Transporter sophiasRadio = new Radio(); 

Radio wouldn't implement Transporter. Since Radio cannot “act as” a Transporter, you cannot treat it as Transporter.

Motivations for Polymorphism

- Many different kinds of transportation but only care about their shared capability
 - i.e. how they move
- Polymorphism let programmers sacrifice specificity for generality
 - treat any number of classes as their lowest common denominator
 - limited to methods declared in that denominator
 - can only use methods declared in `Transporter`
- For this program, that sacrifice is ok!
 - `Racer` doesn't care if instance of `Car` can `playRadio()` or if instance of `Bike` can `dropKickstand()`
 - only method `Racer` wants to call is `move()`

Polymorphism in Parameters

- What are implications of this method declaration?

```
public void useTransportation(Transporter transportation) {  
    //code elided  
}
```

- `useTransportation` will accept any object that implements `Transporter`
- `useTransportation` can only call methods declared in `Transporter`

Is this legal?

```
Transporter sophiasBike = new Bike();  
_sophia.useTransportation(sophiasBike);
```



Even though sophiasCar is declared as a Car, the compiler can still verify that it implements Transporter.

```
Car sophiasCar = new Car();  
_sophia.useTransportation(sophiasCar);
```



```
Radio sophiasRadio = new Radio();  
_sophia.useTransportation(sophiasRadio);
```



A Radio wouldn't implement Transporter. Therefore, useTransportation() cannot treat it like a Transporter object.

Why move()? (1/2)

- Why call `move()`?
- What `move()` method gets executed?

```
public class Racer {  
  
    //previous code elided  
    public void useTransportation(Transporter transportation) {  
        transportation.move();  
    }  
  
}
```

Why move()? (2/2)

- Only have access to `Transporter` object
 - cannot call `transportation.drive()` or `transportation.pedal()`
 - that's okay, because all that's needed is `move()`
 - limited to the methods declared in `Transporter`

Method Resolution: Which move() is executed?

- Consider this line of code in `Race` class:

```
_sophia.useTransportation(new Bike());
```

- Remember what `useTransportation` method looked like

```
public void useTransportation(Transporter transportation) {  
    transportation.move();  
}
```

What is “actual type” of `transportation` in this method invocation?

Method Resolution (1/4)

```
public class Race {  
  
    private Racer_sophia;  
    //previous code elided  
  
    public void startRace() {  
        _sophia.useTransportation(new Bike());  
    }  
}
```

```
public class Racer {  
    //previous code elided  
  
    public void useTransportation(Transporter  
        transportation) {  
        transportation.move();  
    }  
}
```

- Bike is actual type
 - Racer was handed instance of Bike
 - new Bike() is argument
- Transporter is declared type
 - Racer treats Bike object as Transporter object
- So... what happens in transportation.move()?
 - What move() method gets used?

Method Resolution (2/4)

```
public class Race {  
    //previous code elided  
    public void startRace() {  
        _sophia.useTransportation(new Bike());  
    }  
}
```

```
public class Racer {  
    //previous code elided  
    public void useTransportation(Transporter  
        transportation) {  
        transportation.move();  
    }  
}
```

```
public class Bike implements Transporter {  
    //previous code elided  
    public void move() {  
        this.pedal();  
    }  
}
```

- Sophia is a Racer
- Bike's move() method gets used
- Why?
 - Bike is actual type
 - Java will execute methods defined in Bike class
 - Transporter is declared type
 - Java limits methods that can be called to those declared in Transporter interface

Method Resolution (3/4)

- What if `_sophia` received instance of `Car`?
 - What `move()` method would get called then?
 - `Car`'s!

```
public class Race {  
  
    //previous code elided  
  
    public void startRace() {  
        _sophia.useTransportation(new Car());  
    }  
}
```

Method Resolution (4/4)

- This method resolution is example of **dynamic binding**, which is when actual method implementation used is not determined until runtime
 - contrast with **static binding**, in which method gets resolved at compile time
- **move()**method is bound dynamically – Java does not know which **move()** method to use until program runs
 - same “**transport.move()**” line of code could be executed indefinite number of times with different method resolution each time

Clicker Question

Given the following class:

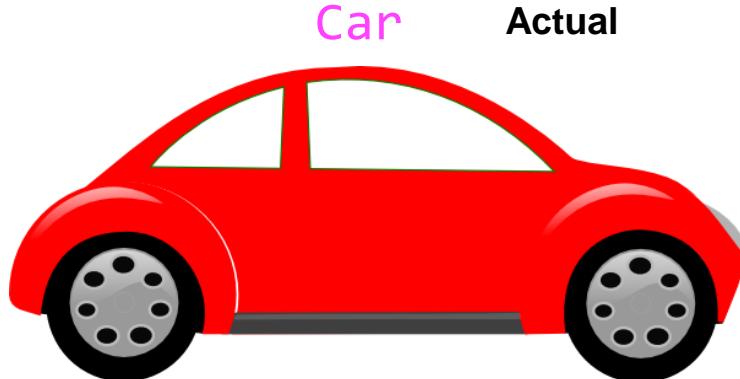
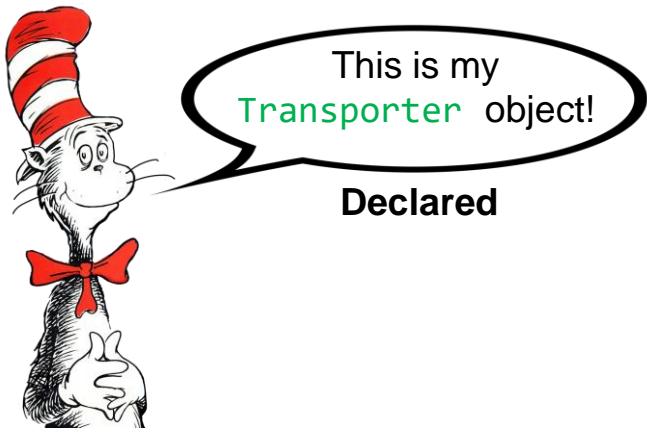
```
public class Laptop implements Typeable, Clickable {  
    public void type() {  
        // code elided  
    }  
    public void click() {  
        //code elided  
    }  
}
```

Given that typeable has declared the type method and clickable has declared the click method, which of the following calls is/are **valid**?

- A. `Typeable macBook= new Typeable();
macBook.type();`
- C. `Typable macBook= new Laptop();
macBook.click();`
- B. `Clickable macBook = new Clickable();
macBook.type();`
- D. `Clickable macBook = new Laptop();
macBook.click();`

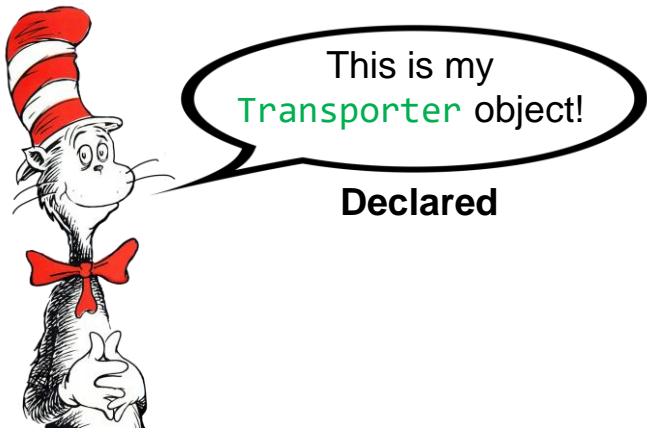
Why does that work? (1/2)

- Declared type and actual type work together
 - declared type keeps things generic
 - can reference a lot of objects using one generic type
 - actual type ensures specificity
 - when defining implementing class, the methods can get implemented in any way



Why does that work? (2/2)

- Declared type and actual type work together
 - declared type keeps things generic
 - can reference a lot of objects using one generic type
 - actual type ensures specificity
 - when defining implementing class, the methods can get implemented in any way



When to use polymorphism?

- Using only functionality declared in interface or specialized functionality from implementing class?
 - if only using functionality from the interface → polymorphism!
 - if need specialized methods from implementing class, don't use polymorphism

Why use interfaces?

- Contractual enforcement
 - will guarantee that class has certain capabilities
 - `Car` implements `Transporter`, therefore it must know how to `move()`
- Polymorphism
 - Can have implementation-agnostic classes and methods
 - know that these capability exists, don't care how they're implemented
 - allows for more generic programming
 - `useTransportation` can take in any `Transporter` object
 - can easily extend this program to use any form of transportation, with minimal changes to existing code
 - an extremely powerful tool for extensible programming

Why is this important?

- With 2 modes of transportation!
- Old Design:
 - need more classes → more specialized methods
(`useRollerblades()`, `useBike()`, etc)
- New Design:
 - as long as the new classes implement `Transporter`, `Racer` doesn't care what transportation it has been given
 - **don't need to change Racer!**
 - less work for you!
 - just add more transportation classes that implement `Transporter`

The Program

```
public class App {  
    public App() {  
        Race r = new Race();  
        r.startRace();  
    }  
  
    public class Race {  
        private Racer _dan, _sophia;  
  
        public Race(){  
            _dan = new Racer();  
            _sophia = new Racer();  
        }  
        public void startRace() {  
            _dan.useTransportation(new Car());  
            _sophia.useTransportation(new Bike());  
        }  
    }  
  
    public interface Transporter {  
        public void move();  
    }
```

```
public class Racer {  
    public Racer() {}  
  
    public void useTransportation(Transporter transport){  
        transport.move();  
    }  
}  
  
public class Car implements Transporter {  
    public Car() {}  
    public void drive() {  
        //code elided  
    }  
    public void move() {  
        this.drive();  
    }  
}  
  
public class Bike implements Transporter {  
    public Bike() {}  
    public void pedal() {  
        //code elided  
    }  
    public void move() {  
        this.pedal();  
    }  
}
```

In Summary

- Interfaces are contracts
 - force classes to define certain methods
- Polymorphism allows for extremely generic code
 - treats multiple classes as their “generic type” while still allowing specific method implementations to be executed
- Polymorphism + Interfaces
 - generic coding
- Why is it helpful?
 - want you to be the laziest (but cleanest) programmer you can be

Next Lecture

- Inheritance and polymorphism

CSE201: Monsoon 2024 Advanced Programming

Lecture 06: Inheritance - Abstract Class and Immutable Class

Dr. Arun Balaji Buduru

Head, Center of Technology in Policing
Founding Head, Usable Security Group (USG)
Associate Professor, Dept. of CSE | HCD
IIIT-Delhi, India

Convertibles vs. Sedans

Convertible

- Top Down Roof
(Retractable Roof)

Sedan

- Fixed Roof

- Drive
- Brake
- Play radio
- Lock/unlock doors
- Turn off/on turn engine

Can we model this in code?

- In some cases, objects can be very closely related to each other
 - Convertibles and sedans drive the same way
 - Flip phones and smartphones call the same way
- Imagine we have an Convertible and a Sedan class
 - Can we enumerate their similarities in one place?
 - How do we portray their relationship through code?

Convertible

- putTopDown()
- turnOnEngine()
- turnOffEngine()
- drive()

Sedan

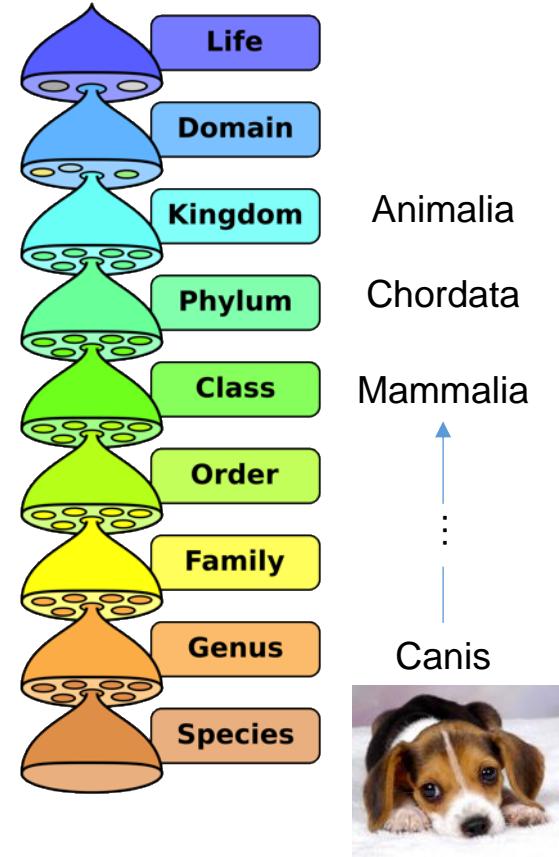
- parkInCompactSpace()
- turnOnEngine()
- turnOffEngine()
- drive()

Can we use Interfaces?

- We could build an interface to model their similarities
 - Build a Car interface with the following methods:
 - turnOnEngine()
 - turnOffEngine()
 - drive()
 - etc.
- Remember: interfaces only declare methods
 - Each class will need to implement the method in its own way
 - Thinking ahead: a lot of these method implementations would be the same across classes
 - Convertible and Sedan would have the same definition for drive()
 - startEngine, shiftToDrive, etc
- Is there a better way where we can reuse the code?

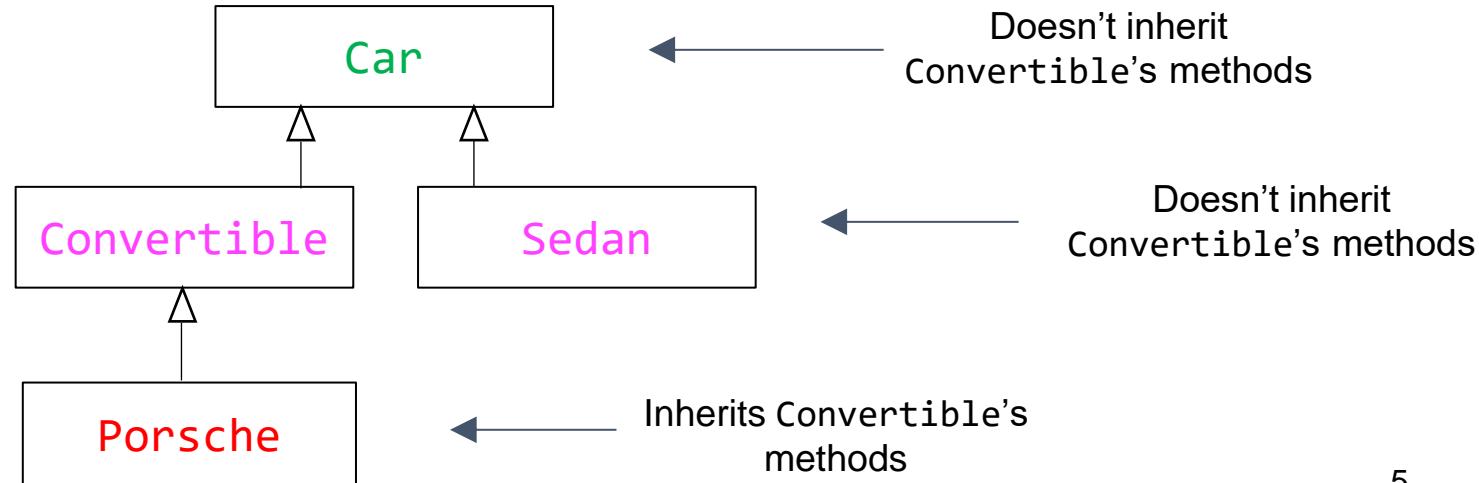
Inheritance

- In OOP, inheritance is a way of modeling very similar classes
- **Inheritance** models an “**is-a**” relationship
 - A **sedan** “is a” **car**
 - A **dog** “is a” **mammal**
- Remember: **Interfaces** model an “**acts-as**” relationship
- You’ve probably seen inheritance before!
 - Taxonomy from biology class



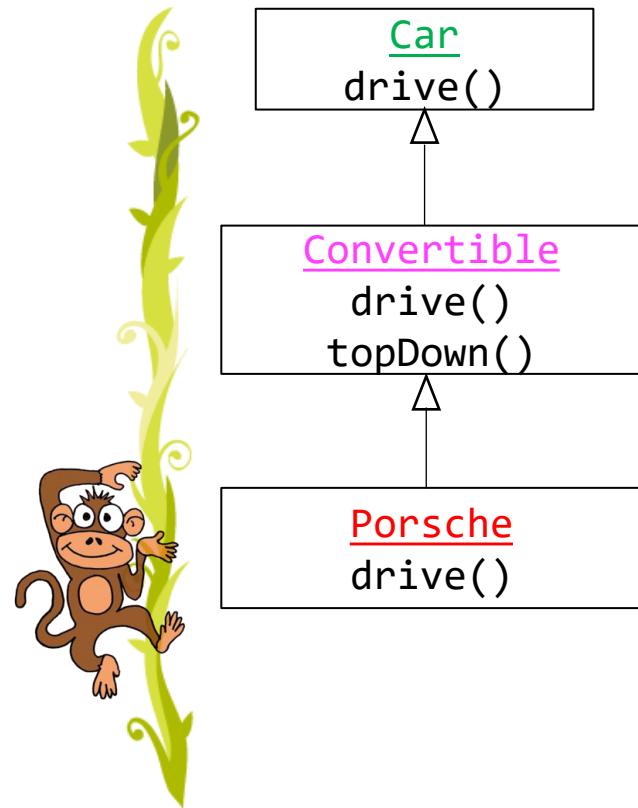
Adding new methods

- You can add specialized functionality to a subclass by defining methods
- These methods can only be inherited if a class extends this subclass



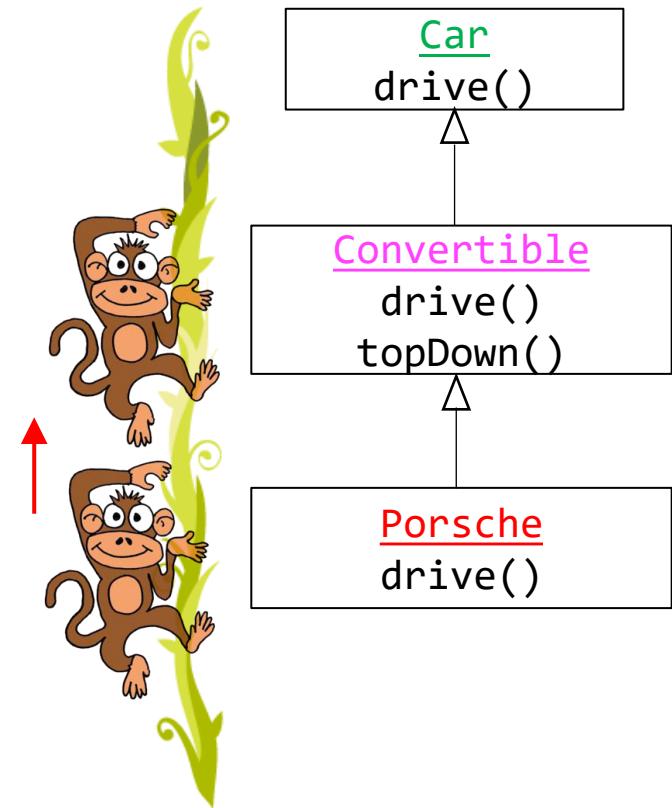
Method Resolution (1/2)

- When we call `drive()` on some instance of **Porsche**, how does Java know which version of the method to call?
- Essentially, Java “walks up the class inheritance tree” from subclass to superclass until it either:
 - finds the method, and calls it
 - doesn’t find the method, and generates a compile-time error. You can’t send a message for which there is no method!



Method Resolution (2/2)

- When we call `drive()` on a **Porsche**, Java executes the `drive()` method defined in **Porsche**
- When we call `topDown()` on a **Porsche**, Java executes the `topDown()` method defined in **Convertible**



Indirectly Accessing private Instance Variables in Superclass by defining Accessors and Mutators

```
public class Car {  
  
    private Radio _myRadio;  
  
    public Car() {  
        _myRadio = new Radio();  
    }  
  
    protected Radio getRadio(){  
        return _myRadio;  
    }  
    protected void setRadio(Radio radio){  
        _myRadio = radio;  
    }  
}
```

- Remember from earlier that private variables are not directly inherited by subclasses
- If `Car` does want its subclasses to be able to access and change the value of `_myRadio`, it can **define protected accessor and mutator methods**
 - Will non-subclasses be able to access `getRadio()` and `setRadio()` ?
- Very carefully consider these design decisions in your own programs – which properties will need to be accessible to other classes?

Calling Accessors/Mutators From Subclass

- `Convertible` can get a reference to `_radio` by calling `this.getRadio()`
 - Subclasses automatically inherit these public accessor and mutator methods
- Note that using “double dot” we’ve chained two methods together
 - First, `getRadio` is called, and returns the radio
 - Next, `setFavorite` is called on that radio

```
public class Convertible extends Car {  
    public Convertible() {  
    }  
  
    public void setRadioPresets(){  
        this.getRadio().setFavorite(1, 95.5);  
        this.getRadio().setFavorite(2, 92.3);  
    }  
}
```

Let's step through some code

- Somewhere in our code, a `Convertible` is instantiated

```
//somewhere in the program  
Convertible convertible = new Convertible();  
convertible.setRadioPresets();
```

- The next line of code calls `setRadioPresets()`
- Let's step into `setRadioPresets()`

Let's step through some code

- When someone calls `setRadioPresets()`; first line is `this.getRadio()`
- `getRadio()` returns `_myRadio`
- What is the value of `_myRadio` at this point in the code?
 - Has it been initialized?
 - Nope, assuming that the structure of class `Car` is exactly as shown on right side (i.e. without any constructor), we'll run into a `NullPointerException` here :(

```
public class Convertible extends Car {  
    public Convertible() { //code elided  
    }  
  
    public void setRadioPresets() {  
        this.getRadio().setFavorite(1, 95.5);  
        this.getRadio().setFavorite(2, 92.3);  
    }  
}  
  
public class Car {  
  
    private Radio _myRadio;  
  
    public Radio getRadio() {  
        return _myRadio;  
    }  
}
```

Making Sure Superclass's Instance Variables are Initialized

- `Convertible` may declare its own instance variables, which it initializes in its constructor
- `Car`'s instance variables are initialized in the `Car` constructor
- When we instantiate `Convertible`, how can we make sure `Car`'s instance variables are initialized too?
 - Case-1: Car has a default constructor that instantiate all its fields
 - Case-2: Car has a parameterized constructor for initializing all its fields

super(): Invoking Superclass's Default Constructor (Case 1)

- Let's assume that `Car`'s instance variables (like `_radio`) are initialized in `Car`'s default constructor
- Whenever we instantiates `Convertible`, default constructor of `Car` is called automatically
- To explicitly invoke `Car`'s default constructor, we can call `super()` inside the constructor of `Convertible`
 - **Can only make this call once**, and it must be the very first line in the subclass's constructor

```
public class Convertible extends Car {  
    private ConvertibleTop _top;  
  
    public Convertible() {  
        super();  
        _top = new ConvertibleTop();  
        this.setRadioPresets();  
    }  
  
    public void setRadioPresets(){  
        this.getRadio().setFavorite(1, 95.5);  
        this.getRadio().setFavorite(2, 92.3);  
    }  
}
```

`super()`: Invoking Superclass's Parameterized Constructor (Case 2)

```
public class Car {  
  
    private Racer _driver;  
    public Car(Racer driver) {  
        _driver = driver;  
    }  
    .....  
}
```

```
public class Convertible extends Car {  
  
    private ConvertibleTop _top;  
  
    public Convertible(Racer driver) {  
        super(driver);  
        _top = new ConvertibleTop();  
    }  
    .....  
}
```

- What if the superclass's constructor takes in a parameter?
 - We've modified `Car`'s constructor to take in a `Racer` as a parameter
 - How do we invoke this constructor correctly from the subclass?
- In this case, need the `Convertible`'s constructor to also take in a `Racer`
- The `Racer` is then passed as an argument to `super()` – now `Racer`'s constructor will initialize `_driver` to the instance of `Racer` that was passed to the `Convertible`

What if we don't call `super()`?

- What if we forget to call `super()`?
- If you don't explicitly call `super()` first thing in your constructor, Java automatically calls it for you, passing in no arguments
- But if superclass's constructor requires a parameter, you'll get an error!
- In this case, we get a **compiler error** saying that there is no constructor "public `Car()`", since it was declared with a parameter

```
public class Convertible extends Car {  
    private ConvertibleTop _top;  
  
    public Convertible(Racer driver) {  
        //oops forgot to call super()  
        _top = new ConvertibleTop();  
    }  
  
    .....  
}
```

How to Load Passengers?

- What if we wanted to seat all of the passengers in the car?
- Sedan, Convertible, and Van all have different numbers of seats
 - They will all have different implementations of the same method



Solution-1: Using Constructor Parameters

```
public class Convertible extends Car {  
    private Passenger _p1;  
    public Convertible(Racer driver, Passenger p1) {  
        super(driver);  
        _p1 = p1;  
    }  
    //code with passengers elided  
}
```

```
public class Sedan extends Car {  
    private Passenger _p1, _p2, _p3, _p4;  
    public Sedan(Racer driver, Passenger p1,  
                Passenger p2, Passenger p3, Passenger p4) {  
        super(driver);  
        _p1 = p1;  
        _p2 = p2;  
        _p3 = p4;  
    }  
    //code with passengers elided  
}
```

- Notice how we only need to pass driver to super()
- We can add additional parameters in the constructor that only the subclasses will use
- Note that super() has to be the first statement inside the constructor.

Any drawbacks in Previous Approach?

- How about creating an interface Passengers with a method loadPassenger?
 - Which class should implement that?
 - Superclass (Car) or Subclasses (Convertible, Sedan, and Van) ?
 - Issues
 - Creating an extra interface (possibly a new file)
 - Each subclass should have the declaration in the following form:
 - public class Sedan extends Car implements Passengers { }

abstract Methods and Classes

- We declare a method **abstract** in a **superclass** when the **subclasses** can't really re-use any implementation the superclass might provide
- In this case, we know that all **Cars** should `loadPassengers`, but each **subclass** will `loadPassengers` very differently
- **abstract** method is declared in **superclass**, but not defined – up to **subclasses** farther down hierarchy to provide their own implementations

Solution-2: Using abstract Methods and Classes

- Here, we've modified Car to make it an **abstract** class: a class with preferably an **abstract** method
 - You can avoid abstract method and just mark class as abstract if you don't wish to allow object creation of this class
- We declare both Car and its loadPassengers method **abstract**: if one of a class's methods is **abstract**, the class itself must also be declared **abstract**
- An **abstract** method is only declared by the superclass, not implemented – use semicolon after declaration instead of curly braces

```
public abstract class Car {  
  
    private Racer _driver;  
  
    public Car(Racer driver) {  
        _driver = driver;  
    }  
  
    public abstract void loadPassengers();  
}
```

Solution-2: Using abstract Methods and Classes

```
public class Convertible extends Car{  
    @Override  
    public void loadPassengers(){  
        Passenger p1 = new Passenger();  
        p1.sit();  
    }  
}
```

```
public class Sedan extends Car{  
    @Override  
    public void loadPassengers(){  
        Passenger p1 = new Passenger();  
        p1.sit();  
        .....  
        Passenger p3 = new Passenger();  
        p3.sit();  
    }  
}
```

```
public class Van extends Car{  
    @Override  
    public void loadPassengers(){  
        Passenger p1 = new Passenger();  
        p1.sit();  
        .....  
        .....  
        Passenger p6 = new Passenger();  
        p6.sit();  
    }  
}
```

- All concrete **subclasses** of **Car** override by providing a concrete implementation for **Car's** abstract **loadPassengers()** method
- As usual, method signature must match the one that **Car** declared

abstract Methods and Classes

- abstract classes cannot be instantiated!
 - This makes sense – shouldn't be able to just instantiate a generic `Car`, since it has no code to `loadPassengers()`
 - Instead, provide implementation of `loadPassengers()` in concrete `subclass`, and instantiate `subclass`
- `Subclass` at any level in inheritance hierarchy can make abstract method concrete by providing implementation
- Even though an abstract class can't be instantiated, its constructor must still be invoked via `super()` by a `subclass`
 - because only the superclass knows about (and therefore only it can initialize) its own instance variables

So.. What's the difference?

- You might be wondering: what's the difference between abstract classes and interfaces?
- abstract Classes:
 - Can define instance variables
 - Can define a mix of concrete and abstract methods
 - You can only inherit from one class
- Interfaces:
 - Cannot define any instance variables/concrete methods
 - You can implement multiple interfaces

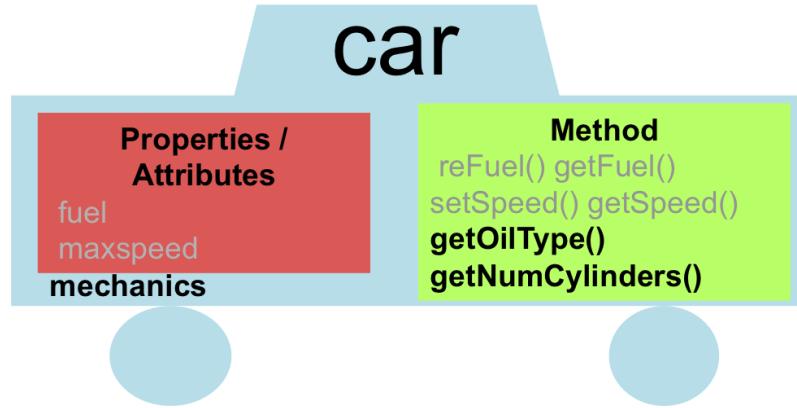
What if the Cars are Getting Modified?



No modifications
should ever be
allowed !!



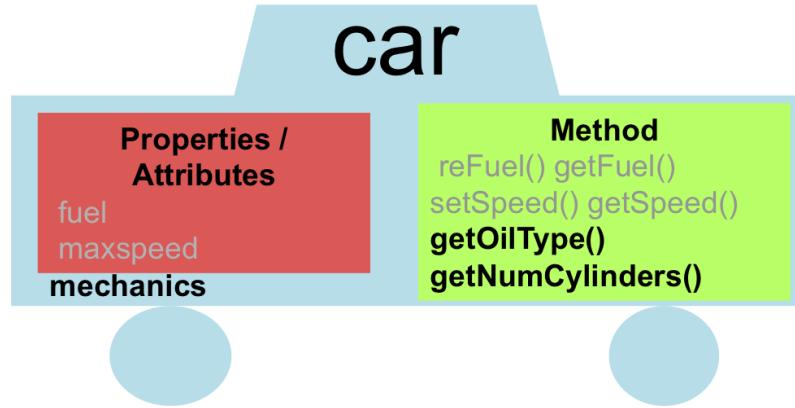
Immutable Classes (1/5)



1. Don't provide any methods that modify the object's state.
2. Make all fields **private**. (ensure encapsulation)
3. Make all fields **final**.

```
public class Mechanics {  
    private final String oilType;  
    private final int numCylinders;  
    public Mechanics(String oil, int cylinders)  
    public String getOilType();  
    public int getNumCylinders();  
}
```

Immutable Classes (2/5)

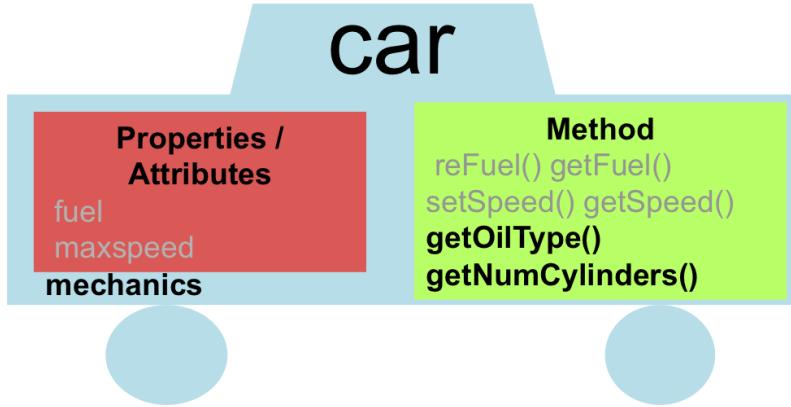


1. Don't provide any methods that modify the object's state.
2. Make all fields `private`. (ensure encapsulation)
3. Make all fields `final`.

```
public class Mechanics {  
    public final Tire tire;  
    .....  
}  
  
// The user can easily do this:  
mechanics.tire.setSize(20);
```

```
public class Tire {  
    private int size;  
    public int getSize();  
    public void setSize(int);  
}
```

Immutable Classes (3/5)



```
public class Mechanics {  
    private final Tire tire;  
    ....  
    public Tire getTire() {return tire;}  
}
```

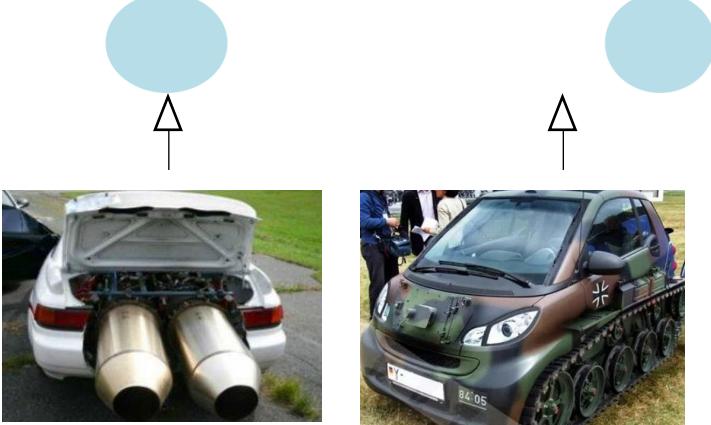
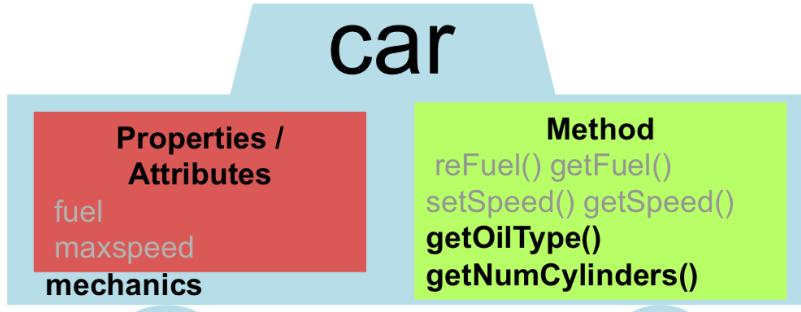
// The user can easily do this:
mechanics.**getTire**().setSize(20);

1. Don't provide any methods that modify the object's state.
2. Make all fields **private**. (ensure encapsulation)
3. Make all fields **final**.

- Setting a reference variable **final** means that it can never be reassigned to refer to a different object.
 - You can't set that reference to refer to another object later (`=`).
 - It does not mean that the object's state can never change!

```
public class Tire {  
    private int size;  
    public int getSize();  
    public void setSize(int);  
}
```

Immutable Classes (4/5)

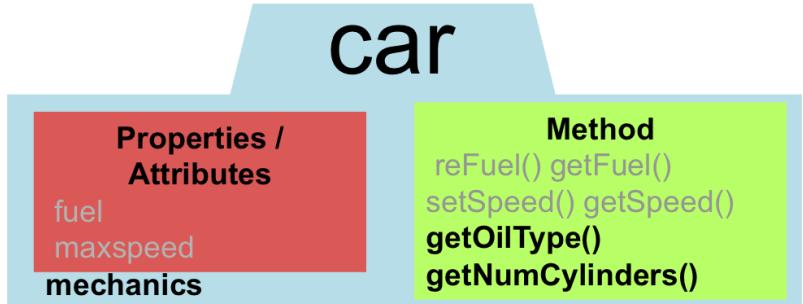


```
public class Mechanics {  
    private final String oilType;  
    private final int numCylinders;  
  
    public Mechanics(String oil, int cylinders)  
    public String getOilType();  
    public int getNumCylinders();  
}
```

```
public class ModifiedMechanics extends Mechanics {  
    .....  
    @Override  
    public String getOilType() {  
        return "Rocket Fuel";  
    }  
    @Override  
    public int getNumCylinders() { return 18; } //Bugatti  
}
```

How to fix these?

Immutable Classes (5/5)



```
public class final Mechanics {  
    private final String oilType;  
    private final int numCylinders;  
  
    public Mechanics(String oil, int cylinders)  
    public String getOilType();  
    public int getNumCylinders();  
}
```

Mechanics cannot be extended
as it is declared as final

```
public class ModifiedMechanics extends Mechanics {  
  
    .....  
    @Override  
    public String getOilType(){  
        return "Rocket Fuel";  
    }  
    @Override  
    public int getNumCylinders(){return 18;} //Bugatti  
}
```

Summary: Making a Class Immutable

1. Don't provide any methods that modify the object's state.
2. Make all fields `private`. (ensure encapsulation)
3. Make all fields `final`.
4. Ensure exclusive access to any mutable object fields.
 - Don't let a client get a reference to a field that is a mutable object (don't allow any mutable representation exposure.)
5. Ensure that the class cannot be *extended*.

CSE201: Monsoon 2024 Advanced Programming

Lecture 07: The Object Class

Dr. Arun Balaji Buduru

Head, Center of Technology in Policing

Founding Head, Usable Security Group (USG)

Associate Professor, Dept. of CSE | HCD

IIT-Delhi, India

This Lecture

- Class Object
 - equals method
 - Comparable and Comparator
 - Clonning

Can You Spot Any Similarities?



- Do you see any similarities between a Cat, Universe, and Furniture?
 - If you just look at their photographs then its hard to guess..

OK, Can You Spot Any Similarities NOW ?

```
public class Cat {  
  
    private String name;  
    private String breed;  
  
    public Cat() { ... }  
    .....  
}
```

```
public class Universe {  
  
    private List<Star> star;  
  
    public Universe(){ ... }  
    .....  
}
```

```
public class Furniture {  
  
    private List<Star> star;  
  
    public Furniture(){ ... }  
    .....  
}
```

- Now we have a class representation of Cat, Universe and Furniture
 - Do you see any similarities now?

They Inherit from Someone!

- What if I tell you that although they look totally unrelated to each other, still they all inherit from a common class, i.e., they have a common parent!

The Class Object in Java

```
public class Object {  
    public Object() { ... }  
    .....  
}
```



```
public class Cat {  
    private String name;  
    private String breed;  
    public Cat() { ... }  
    .....  
}
```



```
public class Universe {  
    private List<Star> star;  
    public Universe(){ ... }  
    .....  
}
```

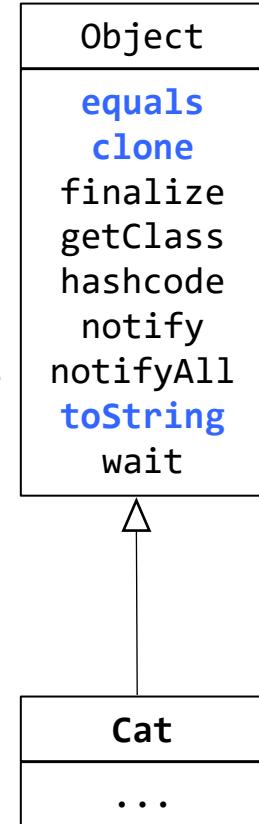


```
public class Furniture {  
    private List<Star> star;  
    public Furniture(){ ... }  
    .....  
}
```

- Every Java class has Object as its superclass and thus inherits the Object methods
 - Due to this, although Cat, Universe and Furniture are totally unrelated, they still inherit from class Object

The Class Object

- The class Object forms the root of the overall inheritance tree of all Java classes.
 - Every class is implicitly a subclass of Object
 - No need to explicitly say “extends Object”
- The Object class defines several methods that become part of every class you write. For example:
 - `public String toString()`
Returns a text representation of the object, usually so that it can be printed.



Object Methods

method	description
protected Object clone ()	creates a copy of the object
public boolean equals (Object o)	returns whether two objects have the same state
protected void finalize ()	called during garbage collection
public Class<?> getClass ()	info about the object's type
public int hashCode ()	a code suitable for putting this object into a hash collection
public String toString ()	text representation of the object
public void notify () public void notifyAll () public void wait () public void wait (...)	methods related to concurrency and locking (seen later)

Using the Object Class

- You can store any object in a variable of type `Object`.

```
Object o1 = new Cat("Meau", "Indian Cat");  
Object o2 = "hello there";
```

Question: `speak()` is a method in `Cat` class, is this correct?

- 1) `o1.speak()`
- 2) `o1.toString()`

- You can write methods that accept an `Object` parameter.

```
public void example(Object o) {  
    if (o != null) {  
        System.out.println("o is " + o.toString());  
    }  
}
```

- You can make arrays or collections of `Objects`.

```
Object[] a = new Object[5];  
a[0] = "hello";  
a[1] = new Cat();  
List<Object> list = new ArrayList<Object>();
```

Equality Test on Objects

```
Point p1 = new Point(5, 3);
```

```
Point p2 = new Point(5, 3);
```

```
Point p3 = p2;
```

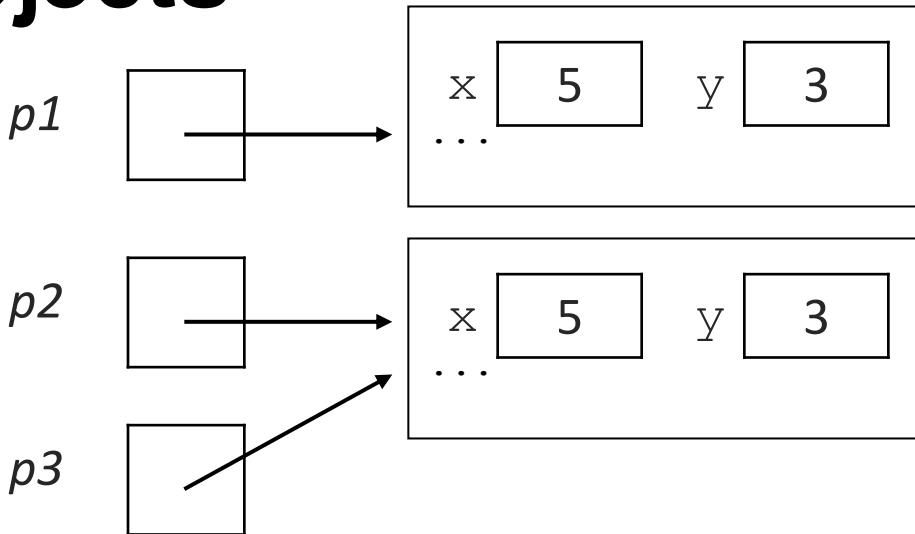
```
// p1 == p2 is false;
```

```
// p1 == p3 is false;
```

```
// p2 == p3 is true
```

```
// p1.equals(p2) ?
```

```
// p2.equals(p3) ?
```



- The `==` operator does not work well with objects.

`==` tests for **referential equality**, not state-based equality.

It produces true only when you compare an object to itself

Default equals Method

- The Object class's equals implementation is very simple:

```
public class Object {  
    ...  
    public boolean equals(Object o) {  
        return this == o;  
    }  
}
```

- The Object class is designed for inheritance.
 - Subclasses can *override* equals to test for equality in their own way

Is this Correctly Implemented NOW ?

```
1. public class Point {  
2.     private int x, y;  
3.     public Point(int _x, int _y) { ... }  
4.     @Override  
5.     public boolean equals(Object o1) {  
6.         Point o = (Point) o1; //type casting  
7.         return (x==o.x && y==o.y);  
8.     }  
9. }  
10.
```

- Still incorrect !
 - Flaw-3
 - It compiles and works fine if Point type objects are passed but fail to compile if non-Point type objects are passed
 - The typecasting will be an issue for following statement

```
Object o1=new Point(1,2);  
Object o2="hello";  
boolean cond=o1.equals(o2);
```
 - The flaw is in line 6 as not every Object will be of Point type:

```
Point o = (Point) o1;  
ClassCastException!!
```

The instanceof Keyword

```
if (variable instanceof type) {  
    statement;  
}
```

- Tests whether **variable** refers to an object of class **type** (or any subclass of **type**)

```
String s = "hello";  
Point p = new Point();
```

expression	result
s instanceof Point	false
s instanceof String	true
p instanceof Point	true
p instanceof String	false
p instanceof Object	true
s instanceof Object	true
null instanceof String	false
null instanceof Object	false

(null is a reference and is not an object)

Is this Correctly Implemented NOW ?

```
1. public class Point {  
2.     private int x, y;  
3.     public Point(int _x, int _y) { ... }  
4.     @Override  
5.     public boolean equals(Object o1) {  
6.         if(o1 instanceof Point) {  
7.             Point o = (Point) o1; //type casting  
8.             return (x==o.x && y==o.y);  
9.         }  
10.        else {  
11.            return false;  
12.        }  
13.    }  
14. }  
15. // subclass of Point  
16. class Point3D extends Point {  
17.     private int z;  
18.     public Point3D(int _x,int _y,int _z) {...}  
19. } .....  
20. }
```

- Still incorrect !

- **Flaw-4**

- The method equals will not behave correctly if Point class is extended

Point3D p1 = new Point3D(1,2,0);
Point3D p2 = new Point3D(1,2,3);
Point p3 = new Point(1,2);
p1.equals(p2); // true 
p2.equals(p3); // true
p3.equals(p1); // true

Is this Correctly Implemented NOW ?

```
1. public class Point {  
2.     private int x, y;  
3.     public Point(int _x, int _y) { ... }  
4.     @Override  
5.     public boolean equals(Object o1) {  
6.         if(o1 instanceof Point) {  
7.             Point o = (Point) o1; //type casting  
8.             return (x==o.x && y==o.y);  
9.         }  
10.        else {  
11.            return false;  
12.        }  
13.    }  
14. }  
15. // subclass of Point  
16. class Point3D extends Point {  
17.     private int z;  
18.     public Point3D(int _x, int _y, int _z) { ... }  
19.     @Override  
20.     public boolean equals(Object o1) {  
8.         if(o1 instanceof Point3D) {  
9.             Point3D o = (Point3D) o1; //type casting  
8.             return (super.equals(o1) && z==o.z);  
9.         }  
10.        else {  
11.            return false;  
12.        }  
13.    }  
14. }
```

- Still incorrect !

- Flaw-5

- It produces *asymmetric* results when Point and Point3D are mixed

```
Point p1 = new Point(1,2);  
Point3D p2 = new Point3D(1,2,3);  
p1.equals(p2); // true  
p2.equals(p1); // false
```



Equality should be symmetric !!

Rules of Equality for Any Two Objects

- Equality is reflexive:
 - `a.equals(a)` is true for every object `a`
- Equality is symmetric:
 - `a.equals(b) ↔ b.equals(a)`
- Equality is transitive:
 - `(a.equals(b) && b.equals(c)) ↔ a.equals(c)`
- No non-null object is equal to null:
 - `a.equals(null)` is false for every object `a`

Finally, the Correct Implementation

```
1. public class Point {  
2.     private int x, y;  
3.     public Point(int _x, int _y) { ... }  
4.     @Override  
5.     public boolean equals(Object o1) {  
6.         if(o1 != null && getClass() == o1.getClass()) {  
7.             Point o = (Point) o1; //type casting  
8.             return (x==o.x && y==o.y);  
9.         }  
10.    else {  
11.        return false;  
12.    }  
13. }  
14. }  
15. // subclass of Point  
16. class Point3D extends Point {  
17.     private int z;  
18.     public Point3D(int _x, int _y, int _z) { ... }  
19.     @Override  
20.     public boolean equals(Object o1) {  
8.         if(o1 != null && getClass() == o1.getClass()) {  
9.             Point3D o = (Point3D) o1; //type casting  
8.             return (super.equals(o1) && z==o.z);  
9.         }  
10.    else {  
11.        return false;  
12.    }  
13. }  
14. }
```

- `getClass` returns information about the type of an object
 - Stricter than `instanceof`; subclasses return different results
- `getClass` should be used when implementing `equals`
 - Instead of `instanceof` to check for same type, use `getClass`
 - This will eliminate subclasses from being considered for equality
 - Caution: Must check for `null` before calling `getClass`

Comparing Objects

Comparing Objects in Java

[Blue].equals([Blue]) = true

[Red].equals([Blue]) = false



Can we use equals to get the above arrangement?

- We have seen how to check equality between two objects:
 - `Obj1 == Obj2`
 - `Obj1.equals(Obj2)`
- But how to check the following:
 - `Obj1 < Obj2`
 - `Obj1 > Obj2`
- Operators like `<` and `>` do not work with objects in Java

Comparing Objects in Java

 .compareTo() < 0

A call of `A.compareTo(B)` should return:

// if A comes "before" B in

// the ordering, a value < 0

// if A comes "after" B in

// the ordering, a value > 0

// or exactly 0 if A and B

// are “equal” in the ordering

 .compareTo() > 0

 .compareTo() = 0

The Comparable Interface

- The standard way for a Java class to define a comparison function for its objects is to implement the Comparable interface.

```
public interface Comparable<T> {  
    public int compareTo(T other);  
}
```

compareTo Example

```
public class Rectangle implements Comparable<Rectangle> {  
    private int sideA, sideB, area;  
    public Rectangle (int _a, int _b) { ... }  
  
    @Override  
    public int compareTo(Rectangle o) {  
        if(area == o.area) return 0;  
        else if(area < o.area) return -1;  
        else return 1;  
    }  
}
```

- In this Rectangle class, the compareTo method compares the Rectangle objects as per their area
- You can choose your own comparison algorithm!

compareTo v/s equals

```
public class Rectangle implements Comparable<Rectangle> {  
    private int sideA, sideB, area;  
    public Rectangle (int _a, int _b) { ... }  
  
    @Override  
    public int compareTo(Rectangle o) {  
        if(area == o.area) return 0;  
        else if(area < o.area) return -1;  
        else return 1;  
    }  
    @Override  
    public boolean equals(Object o1) {  
        if(o1 != null && getClass() == o1.getClass()) {  
            Rectangle o = (Rectangle) o1; //type casting  
            return (sideA==o.sideA && sideB==o.sideB);  
        }  
        else {  
            return false;  
        }  
    }  
}
```

```
// Area1 = 2 x 32 = 64  
Rectangle r1=Rectange(2, 32);  
  
// Area2 = 4 x 16 = 64  
Rectangle r2=Rectange(4, 16);  
  
if(r1.compareTo(r2)==0) {  
    // is this true??  
}  
  
if(r1.equals(r2)) {  
    // is this true??  
}
```

Recall, that two Rectangles with same area could still have different values for sideA and sideB

How to Compare Two Objects in Different Styles ?

- Our Rectangle class can only implement one compareTo method and hence only one comparison algorithm (style)
- We may want to compare two Rectangles differently
 - Based on sides
 - Based on area
 -

Comparator Interface

```
public interface Comparator<T> {  
    public int compare(T first, T second);  
}
```

- Interface Comparator is an external object that specifies a comparison function over some other type of objects.
 - Allows you to define multiple orderings for the same type.
 - Allows you to define a specific ordering for a type even if there is no obvious "natural" ordering for that type

Comparator Example

```
public class RectangleAreaComparator
    implements Comparator<Rectangle> {
    @Override
    public int compare(Rectangle r1, Rectangle r2) {
        return r1.getArea() - r2.getArea();
    }
}
```

```
public class RectangleSidesComparator
    implements Comparator<Rectangle> {
    @Override
    public int compare(Rectangle r1, Rectangle r2) {
        if (r1.getSideA() != r2.getSideA()) {
            return r1.getSideA() - r2.getSideA();
        } else {
            return r1.getSideB() - r2.getSideB();
        }
    }
}
```

- Using Comparators, two objects could be compared in different possible ways
- For creating different comparison, implement different objects of Comparator type

```
Class Main {
    public static void main(String[] args) {
        Rectangle r1=Rectange(2, 32);
        Rectangle r2=Rectange(4, 16);
        RectangleAreaComparator rac = new RectangleAreaComparator();
        RectangleSidesComparator rsc = new RectangleSidesComparator();
        int area_result = rac.compare(r1, r2);
        int sides_result = rsc.compare(r1, r2);
    }
}
```

Benefits of Comparator

- Java Collections class (*covered later*) provide method for sorting elements of collections
`public static <T> void sort(List<T> list, Comparator(? super T> c)`
- You can sort list of Rectangles based on different criteria using the Comparator interface
`Collections.sort(list, new RectangleAreaComparator());`
`Collections.sort(list, new RectangleSidesComparator());`

CSE201: Monsoon 2024

Advanced Programming

Lecture 08: Generic Programming

Dr. Arun Balaji Buduru

Head, Center of Technology in Policing

Founding Head, Usable Security Group (USG)

Associate Professor, Dept. of CSE | HCD

IIT-Delhi, India

Today's Lecture

- Generic programming in Java
 - What?
 - Why?
 - How?
 - What not to do in generic programming?
- Quiz-2

Question

- By using any of the concepts taught till now in this course, how can you store different types of objects in a same datastructure
 - E.g., String, Integer, Float, etc. ?

Approach 1

```
public class MyGenericList {  
    private ArrayList myList;  
    public MyGenericList() {  
        myList = new ArrayList();  
    }  
    public void add(Object o) {  
        myList.add(o);  
    }  
    public Object get(int i) {  
        return myList.get(i);  
    }  
  
    public static void main(String[] args) {  
        MyGenericList generic = new MyGenericList();  
        generic.add("hello");  
        generic.add(10);  
        generic.add(10.23f);  
        .....  
        String str = (String) generic.get(0); // OK  
        String str = (String) generic.get(1); // NOT OK  
    }  
}
```

- Using inheritance we know Object class can hold any type of objects
 - We can create ArrayList of objects
- Problems we face:
 - Mandatory type casting while getting the object from list
 - No error checking while adding objects as we are allowed to add any type of objects
 - Wrong type casting can land you with runtime errors

Approach 2

```
public class MyGenericList {  
    private ArrayList myList;  
    public MyGenericList() {  
        myList = new ArrayList();  
    }  
    public void add(Object o) {  
        myList.add(o);  
    }  
    public Object get(int i) {  
        return myList.get(i);  
    }  
  
    public static void main(String[] args) {  
        MyGenericList generic = new MyGenericList();  
        generic.add("hello");  
        generic.add(10);  
        generic.add(10.23f);  
        .....  
        String str = (String) generic.get(0); // OK  
        if(generic.get(1) instanceof String) {  
            String str = (String) generic.get(1); // OK  
        }  
    }  
}
```

- We can use `instanceof` keyword to verify the type of object retrieved from `get()` function
 - Is this programmer friendly?
 - How many such “if” when you have several different types of objects in the list?

Approach 3

```
public class MyStringList {  
    private ArrayList myList;  
    public MyStringList() {  
        myList = new ArrayList();  
    }  
    public void add(String o) {  
        myList.add(o);  
    }  
    public String get(int i) {  
        return myList.get(i);  
    }  
}
```

```
public class MyIntList {  
    private ArrayList myList;  
    public MyIntList() {  
        myList = new ArrayList();  
    }  
    public void add(Integer o) {  
        myList.add(o);  
    }  
    public Integer get(int i) {  
        return myList.get(i);  
    }  
}
```

```
public class MyTypeXList {  
    private ArrayList myList;  
    public MyTypeXList() {  
        myList = new ArrayList();  
    }  
    public void add(TypeX o) {  
        myList.add(o);  
    }  
    public TypeX get(int i) {  
        return myList.get(i);  
    }  
}
```

```
public class Main {  
    public static void main(String args[]) {  
        MyStringList strList = new MyStringList();  
        MyIntList intList = new MyIntList();  
        MyTypeXList typeXList = new MyTypeXList();  
  
        strList.add("hello");  
        intList.add(1);  
        ...  
    }  
}
```

- We can create one class to hold one type of object
 - How many classes for N types of objects?
 - How many lines of code?

- Is this programmer friendly?
 - NO !!

Solution: Generic Programming



- Our generic cup can hold different types of liquid
- In the notation `Cup<T>`:
 - T = Coffee
 - T = Tea
 - T = Milk
 - T = Soup
 -

Cup == Generic Container

Implementing generics

```
// a parameterized (generic) class
```

```
public class name<Type> {
```

or

```
public class name<Type1, Type2, . . . , TypeN> {
```

- By putting the **Type** in < >, you are demanding that any client that constructs your object must supply a type parameter
 - You can require multiple type parameters separated by commas
- The rest of your class's code can refer to that type by name
- The type parameter is *instantiated* by the client. (e.g. E → String)

Solution to our Problem

```
public class MyGenericList <T> {  
    private ArrayList <T> myList;  
    public MyGenericList() {  
        myList = new ArrayList <T>();  
    }  
    public void add(T o) {  
        myList.add(o);  
    }  
    public T get(int i) {  
        return myList.get(i);  
    }  
}
```

- Using generic programming we don't have to implement different classes for different object types
 - Programmer friendly code!
- We just have to create different instances of MyGenericList for different objects

```
public class Main {  
    public static void main(String args[]) {  
        MyGenericList<String> strList = new MyGenericList<String>();  
        MyGenericList<Integer> intList = new MyGenericList<Integer>();  
  
        strList.add("hello");  
        intList.add(1);  
        ...  
    }  
}
```

A Generic Class with Multiple Fields

- Let's create a class that could contain two different types of field, and type of both the fields are unknown

Generic Class with Two Fields (1/3)

```
public class Pair <T1, T2> {  
    private T1 key;  
    private T2 value;  
    public Pair(T1 _k, T2 _v) {  
        key = _k; value = _v;  
    }  
    public T1 getKey() { return key; }  
    public T2 getValue() { return value; }  
}
```

```
public class Main {  
    public static void main(String args[]) {  
        MyGenericList<Pair> db =  
            new MyGenericList<Pair>();  
        db.add(new Pair<String, Integer>("John", 2343));  
        db.add(new Pair<String, Integer>("Susane", 8908));  
        ...  
    }  
}
```

- Why this code isn't correct?
 - Database class instantiated without specifying the type of its two fields

Generic Class with Two Fields (2/3)

```
public class Pair <T1, T2> {  
    private T1 key;  
    private T2 value;  
    public Pair(T1 _k, T2 _v) {  
        key = _k; value = _v;  
    }  
    public T1 getKey() { return key; }  
    public T2 getValue() { return value; }  
}
```

```
public class Main {  
    public static void main(String args[]) {  
        MyGenericList<Pair<String, Integer>> db =  
            new MyGenericList<Pair>();  
        db.add(new Pair<String, Integer>("John", 2343));  
        db.add(new Pair<String, Integer>("Susane", 8908));  
        ...  
    }  
}
```

- Why this code isn't correct
 - During instantiation we have to declare the type of fields in Database class on both RHS and LHS of statement

Generic Class with Two Fields (3/3)

```
public class Pair <T1, T2> {  
    private T1 key;  
    private T2 value;  
    public Pair(T1 _k, T2 _v) {  
        key = _k; value = _v;  
    }  
    public T1 getKey() { return key; }  
    public T2 getValue() { return value; }  
}
```

- This is the correct implementation and usage of a generic class with multiple fields

```
public class Main {  
    public static void main(String args[]) {  
        MyGenericList<Pair<String, Integer>> db =  
            new MyGenericList<Pair<String, Integer>>();  
        db.add(new Pair<String, Integer>("John", 2343));  
        db.add(new Pair<String, Integer>("Susane", 8908));  
        ...  
    }  
}
```

Goals for Generic Programming

- Writing code that can be reused for objects of many different types
 - Programmer friendly
- For example, you don't want to program separate classes to collect String and Integer objects

Behind the Scene: Generics are Implemented using Type Erasures

```
public class MyGenericList <T> {  
    private ArrayList <T> myList;  
    public MyGenericList() {  
        myList = new ArrayList <T>();  
    }  
    public void add(T o) {  
        myList.add(o);  
    }  
    public T get(int i) {  
        return myList.get(i);  
    }  
}  
  
public class Main {  
    public static void main(String args[]) {  
        MyGenericList<String> strList = new  
            MyGenericList<String>();  
        strList.add("hello");  
        String str = strList.get(0);  
    }  
}
```



Compile
Time

```
public class MyGenericList {  
    private ArrayList myList;  
    public MyGenericList() {  
        myList = new ArrayList ();  
    }  
    public void add(Object o) {  
        myList.add(o);  
    }  
    public Object get(int i) {  
        return myList.get(i);  
    }  
}  
  
public class Main {  
    public static void main(String args[]) {  
        MyGenericList strList = new MyGenericList();  
        strList.add("hello");  
        String str = (String) strList.get(0);  
    }  
}
```

- Compiler erases all parameter type information (type erasure)
- Compiler also ensures proper typecasting

Restrictions (1/6)

- Which of the following is correct?
 1. MyGenericList <double> var = new MyGenericList<Double>();
 2. MyGenericList <Double> var = new MyGenericList<Double>();

Type Parameters Cannot Be Instantiated with Primitive Types !

– No double, only Double

Restrictions (2/6)

```
public class MyGenericClass<T> {  
    .....  
  
    public void doSomething() {  
        T my_var = new T(); // ERROR  
    }  
  
    public static void main(String[] arg){  
        .....  
    }  
}
```

- Instantiating Type variable is not allowed
 - Compile time error
 - Type erasure removes the type information at runtime and hence its impossible to figure out the type at runtime

Restrictions (3/6)

```
public class MyGenericClass<T> {  
    .....  
  
    static <T> void doSomething(List<T> list) {  
        if(list instanceof ArrayList<Integer>) {  
            .....  
        }  
    }  
  
    public static void main(String[] args){  
        .....  
    }  
}
```

- Cannot use casts or instanceof with parameterized types
 - Compile time error
 - Type erasure removes the type information at runtime and hence its impossible to figure out the type at runtime

Restrictions (4/6)

```
public class MyGenericClass<T> {  
    .....  
    private static T field;  
  
    public static void main(String[] arg){  
        MyGenericClass<Integer> c1 = new .....  
        MyGenericClass<String> c2 = new .....  
        MyGenericClass<Double> c3 = new .....  
        // What is the type of "field" now ?  
    }  
}
```

- Cannot declare static fields whose types are Type parameter
 - If it was allowed then in the code shown here what will be the Type of “field” as it’s a static object hence shared by c1, c2 and c3

Restrictions (5/6)

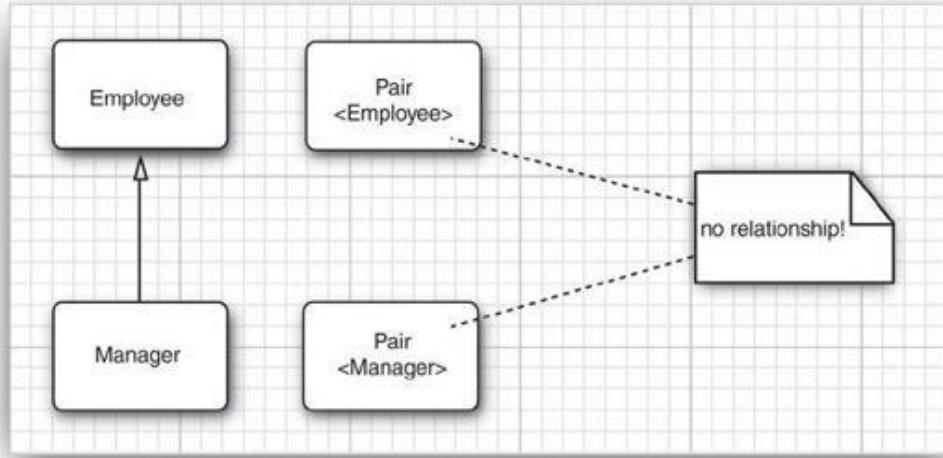


Image Source: Core Java, Volume-1

- Generic does not support sub-typing
 - If a class Employee is the superclass (parent) for a class Manager, then for a generic class Pair<T>, it does not mean Pair<Employee> also becomes the superclass (parent) for Pair<Manager>

Restrictions (6/6)

```
public class MyGenericClass<T> {  
    .....  
  
    public void doSomething() {  
        T[] my_arr = new T[10]; // ERROR  
    }  
  
    public static void main(String[] arg){  
        // ERROR  
        MyGenericClass<String>[] str_array  
            = new  
MyGenericClass<String>[10];  
    }  
}
```

- Generic array creation is not allowed
 - Solution: create array of Object and typecast that array to generic type

Why Generic Array Creation not Allowed ?

```
// Legal statement (arrays are covariant)
Object array[] = new Integer[10];
// Compilation error below (generics are invariant)
List<Object> myList = new ArrayList<Integer>();
```

```
// Below line incorrect but let's assume its
correct
List<Integer> intList[] = new
ArrayList<Integer>[5];
List<String> stringList = new ArratList<String>();

stringList.add("John");

Object objArray[] = intList;
objArray[0] = stringList;

// This will generate ClassCastException
int my_int_number = objArray[0].get(0);
```

- Arrays are covariant
 - Subclass array type can be assigned to superclass array reference
- Generics are invariant
 - Subclass type generic type cannot be assigned to superclass generic reference
- If generic array creation was allowed then compile time strict type checking cannot be enforced
 - Runtime ClassCastException will be generated in the example here

Is there any Problem in Below Code?

```
public class Main {  
    ....  
    public static void print(ArrayList<Object>  
list){  
        for(Object o: list)  
            System.out.println(o);  
    }  
    public static void main(String[] arg){  
        ArrayList<Integer> I = new  
                    ArrayList<Integer>();  
        I.add(1);  
        I.add(2);  
        ArrayList<String> S = new  
                    ArrayList<String>();  
        S.add("Bob");  
        S.add("Paul");  
        print(I);  
        print(S);  
    }  
}
```

- The code won't compile
- Although Object is superclass for Integer and String class, it does not mean that in the print method, `ArrayList<Object>` can hold `ArrayList<Integer>` or `ArrayList<String>`
 - Restriction-5 discussed in this lecture
- How to resolve this issue?

The WildCard to our Rescue !



The WildCard “?” to our Rescue !

```
public class Main {  
    ....  
  
    public static void print(ArrayList<?> list){  
        for(Object o: list)  
            System.out.println(o);  
    }  
    public static void main(String[] arg){  
        ArrayList<Integer> I = new  
                           ArrayList<Integer>();  
        I.add(1);  
        I.add(2);  
        ArrayList<String> S = new  
                           ArrayList<String>();  
        S.add("Bob");  
        S.add("Paul");  
        print(I);  
        print(S);  
    }  
}
```

- We just need **one** change in our code
- Simply use a wildcard character as type variable in the parameter `ArrayList` in `print` method
 -

More Meaningful Example of Wildcard (1/2)

```
public class Main {  
    ....  
    static void print(ArrayList<? extends Car>  
list){  
        .....  
    }  
    public static void main(String[] arg){  
        .....  
    }  
}
```

- Upper bounded wildcard
 - Here the print method will only accept ArrayList of Car type or its subclass type

More Meaningful Example of Wildcard (2/2)

```
public class Main {  
    ....  
    static void print(ArrayList<? super Integer> list){  
        .....  
    }  
    public static void main(String[] arg){  
        .....  
    }  
}
```

- Lower bounded wildcard
 - Here the print method will only accept ArrayList of Integer or any Type that is supertype of Integer
 - Integer
 - Number
 - Object

Next Lecture

- Exception Handling

CSE201: Monsoon 2024 Advanced Programming

Lecture 9: Exception Handling

Dr. Arun Balaji Buduru

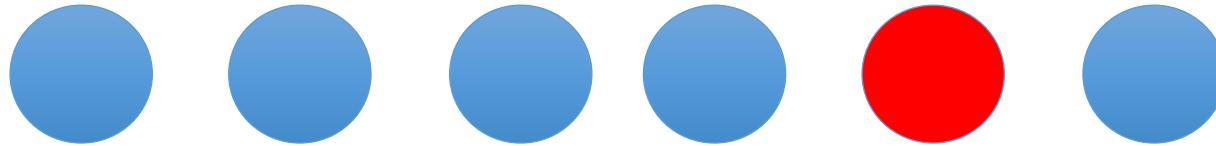
Head, Center of Technology in Policing

Founding Head, Usable Security Group (USG)

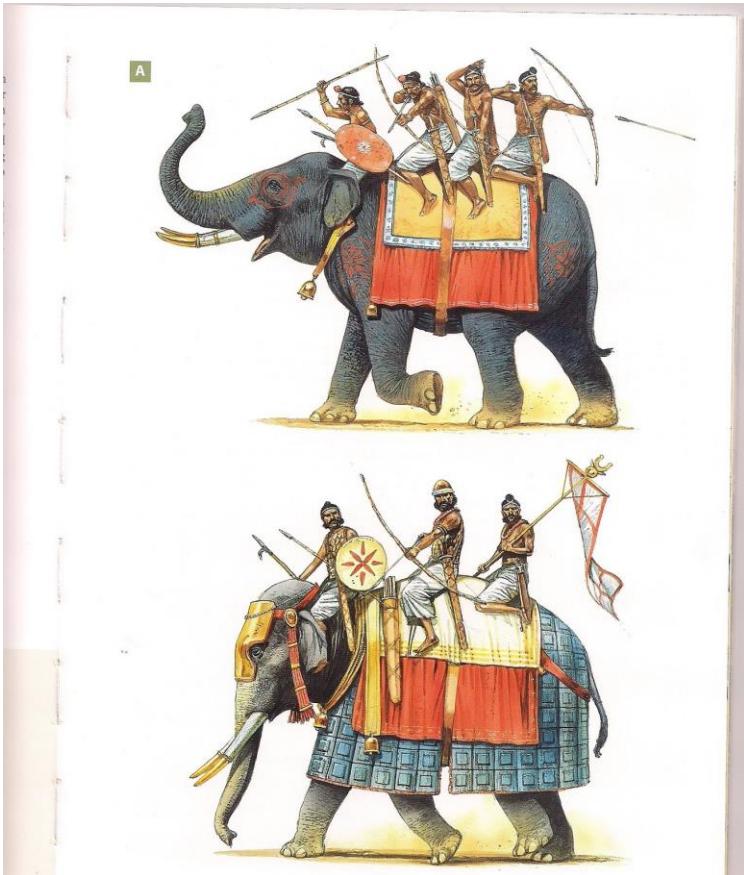
Associate Professor, Dept. of CSE | HCD

IIT-Delhi, India

Today's Lecture: Exceptions



Being Defensive is Important



www.shutterstock.com · 109665452

Defensive Programming

- Murphy's law
 - “Anything that can possibly go wrong, does.”
- Finagle's law
 - “Anything that can go wrong, will – at the worst possible moment.”
- Sod's law
 - “If something can go wrong, it will”

Defensive programming: Hope for the best, expect the worst!

Defensive Programming

- Collection of techniques to reduce the risk of failure at run time
 - An analogy is defensive driving by being never sure how other drivers would be driving
- The technique is in making the software behave in a predictable manner despite unexpected inputs or user actions and internal errors
 - After all debugging takes a lot of time!

Types of Programming Errors

- Syntax errors
 - Compile time errors
 - Easiest to fix
- Logical errors
 - Program runs without crashing but gives incorrect result
 - Most difficult to fix
- Runtime errors
 - Occur while the program is running if the environment detects an operation that is impossible to carry out
 - Could be fixed easily with defensive programming
 - **Exception handling!**

Exception Handling Syntax

- Process for handling exceptions
 - `try` some code, catch exception thrown by tried code, finally, “clean up” if necessary
 - `try`, `catch`, and `finally` are reserved words
- `try` denotes code that may throw an exception
 - place questionable code within a `try` block
 - a `try` block must be immediately followed by a `catch` block unlike an if w/o else
 - thus, `try-catch` blocks always occurs as pairs
- `catch` exception thrown in `try` block and write special code to handle it
 - catch blocks distinguished by type of exception
 - can have several `catch blocks`, each specifying a particular type of exception
 - Once an exception is handled, execution continues after the catch block
- `finally` (optional)
 - special block of code that is executed whether or not an exception is thrown
 - follows `catch block`

Trace a **try/catch** Program Execution (1/3)

```
try {  
    statements;  
}  
catch (TheException ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}  
  
Next statement;
```

Suppose no exceptions in the statements

Trace a **try/catch** Program Execution (2/3)

```
try {  
    statements;  
}  
catch (TheException ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}  
  
Next statement;
```

The final block is always
executed

Trace a **try/catch** Program Execution (3/3)

```
try {  
    statements;  
}  
catch (TheException ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}
```

Next **statement**;

Next statement in the method is
executed

Trace a **try/catch** Program Execution (1/4)

```
try {  
    statement1;  
statement2;  
    statement3;  
}  
catch (Exception1 ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}  
  
Next statement;
```

Suppose an exception of type
Exception1 is thrown in statement2

Trace a **try/catch** Program Execution (2/4)

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch (Exception1 ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}  
  
Next statement;
```

The exception is handled.

Trace a **try/catch** Program Execution (3/4)

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch (Exception1 ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}  
  
Next statement;
```

The final block is always executed.

Trace a **try/catch** Program Execution (4/4)

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch (Exception1 ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}  
  
Next statement;
```

The next statement in the method is now executed.

Is this Defensive Programming ?

```
import java.util.*;
public class Main {

    public static void main(String[] args) {

        System.out.println("Enter Integer Input");

        Scanner sc = new Scanner(System.in);
        int num = sc.nextInt();

    }

}
```

- Is program correct?
 - Yes
 - But, only if the user is paying attention
 - Invalid input ?
 - String as input?

Exception Handling using **try/catch**

```
import java.util.*;
public class Main {

    public static void main(String[] args) {
        boolean done = false;
        while(!done) {
            System.out.println("Enter Integer Input");
            try {
                Scanner sc = new Scanner(System.in);
                int num = sc.nextInt(); //exception
                done = true;
            }
            catch(InputMismatchException inp) {
                System.out.println("Wrong input:");
                System.out.println("Try again");
            }
            finally {
                System.out.println("Always execute");
            }
        }
    }
}
```

point

- This is a foolproof program now!
- Exception handling using **try/catch** block of statements
 - Defensive programming
- `InputMismatchException` is a type of exception provided by the `Scanner` class in Java

Multiple **catch** Blocks

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        String[] s = {"a", "23", null, "4", "P"};
        int sum = 0;
        for(int i=0; i<10; i++) {

            sum += (s[i].length() > 0) ?
                Integer.parseInt(s[i]) : 0;
        }
    }
}
```

Multiple **catch** Blocks

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        String[] s = {"a", "23", null, "4", "p"};
        int sum = 0;
        for(int i=0; i<10; i++) {
            try {
                sum += (s[i].length() > 0) ?
                    Integer.parseInt(s[i]) : 0;
            }
            catch(NumberFormatException e) {
                System.out.println("Not an Integer");
            }
            catch(NullPointerException e) {
                System.out.println("NULL value
found");
            }
            catch(ArrayIndexOutOfBoundsException e) {
                System.out.println("Index not in
range");
            }
        }
    }
}
```

- There could be multiple **catch** for a single **try** block
- They are designed to catch different types of exceptions that could be raised from a single **try** block
- **How the exceptions are generated here?**
 - $i=0$ will raise NumberFormatException
 - $i=2$ will raise NullPointerException
 - $i=4$ will raise NumberFormatException
 - $i>4$ will raise ArrayIndexOutOfBoundsException exception

Question

```
public class Main {  
    public static void main(String[] args) {  
        String s = null;  
        try {  
            int length = s.length();  
        }  
  
        System.out.println("Just before catch block");  
  
        catch(NullPointerException e) {  
            System.out.println("String was null");  
        }  
    }  
}
```

- What is the output of the following program?
- Answer
 - Compilation error!
 - **No statement is allowed between a pair of try and catch**
 - error: 'catch' without 'try'

Nested try/catch Blocks

```
public class Andy {  
    ....  
  
    public void getWater() {  
        try {  
            _water = _wendy.getADrink();  
            int volume = _water.getVolume();  
        }  
        catch(NullPointerException e) {  
            this.fire(_wendy);  
            System.out.println("Wendy is fired!");  
            try {  
                _water = johny.getADrink();  
                int volume = _water.getVolume();  
            }  
            catch(NullPointerException e) {  
                this.fire(johny);  
                System.out.println("Johny is fired!");  
            }  
        }  
    }  
}
```

- try/catch block could be nested!
 - If Andy's call to getADrink from Wendy returns null, he can ask Johny to getADrink

Methods Can **throw** Exception

```
public class Andy {  
    ....  
    public void drinkWater() {  
        try {  
            getWater();  
        }  
        catch(NullPointerException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
    public void getWater() {  
        _water = _wendy.getADrink();  
        if(_water == null) {  
            this.fire(_wendy);  
            System.out.println("Wendy is fired!");  
            throw new NullPointerException("NO Water");  
        }  
    }  
}
```

- If you wish to throw an exception in your code you use the **throw** keyword
- Most common would be for an unmet precondition
- When the program detects an error, the program can create an instance of an appropriate exception type and throw it:

`throw new TheException("Message");`
- In the above constructor call for the exception, the message is optional but it's always good to pass some meaningful message

Re-throwing Exception

```
public class Andy {  
    ....  
    public void drinkWater() {  
        try {  
            getWater();  
        }  
        catch(NullPointerException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
    public void getWater() {  
        try {  
            _water = _wendy.getADrink();  
            int volume = _water.getVolume();  
        }  
        catch(NullPointerException e) {  
            this.fire(_wendy);  
            System.out.println("Wendy is fired!");  
            throw new NullPointerException("NO Water");  
        }  
    }  
}
```

- The caught exceptions can be re-thrown using **throw** keyword
- Re-thrown exception must be handled somewhere in the program, otherwise program will terminate abruptly

Trace a **try/catch** Program Execution (1/4)

```
try {  
    statement1;  
statement2;  
    statement3;  
}  
catch (Exception1 ex) {  
    handling ex;  
}  
catch (Exception2 ex) {  
    handling ex;  
    throw ex;  
}  
finally {  
    finalStatements;  
}  
  
Next statement;
```

statement2 throws an exception of type Exception2.

Trace a **try/catch** Program Execution (2/4)

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch (Exception1 ex) {  
    handling ex;  
}  
catch (Exception2 ex) {  
    handling ex;  
    throw ex;  
}  
finally {  
    finalStatements;  
}  
  
Next statement;
```

Handling exception

Trace a **try/catch** Program Execution (3/4)

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch (Exception1 ex) {  
    handling ex;  
}  
catch (Exception2 ex) {  
    handling ex;  
    throw ex;  
}  
finally {  
    finalStatements;  
}  
  
Next statement;
```

Execute the final block

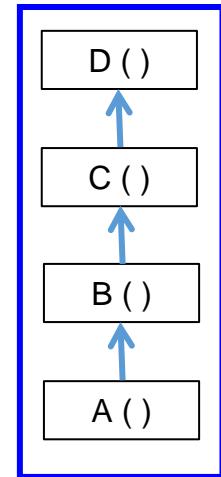
Trace a **try/catch** Program Execution (4/4)

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch (Exception1 ex) {  
    handling ex;  
}  
catch (Exception2 ex) {  
    handling ex;  
    throw ex;  
}  
finally {  
    finalStatements;  
}  
  
Next statement;
```

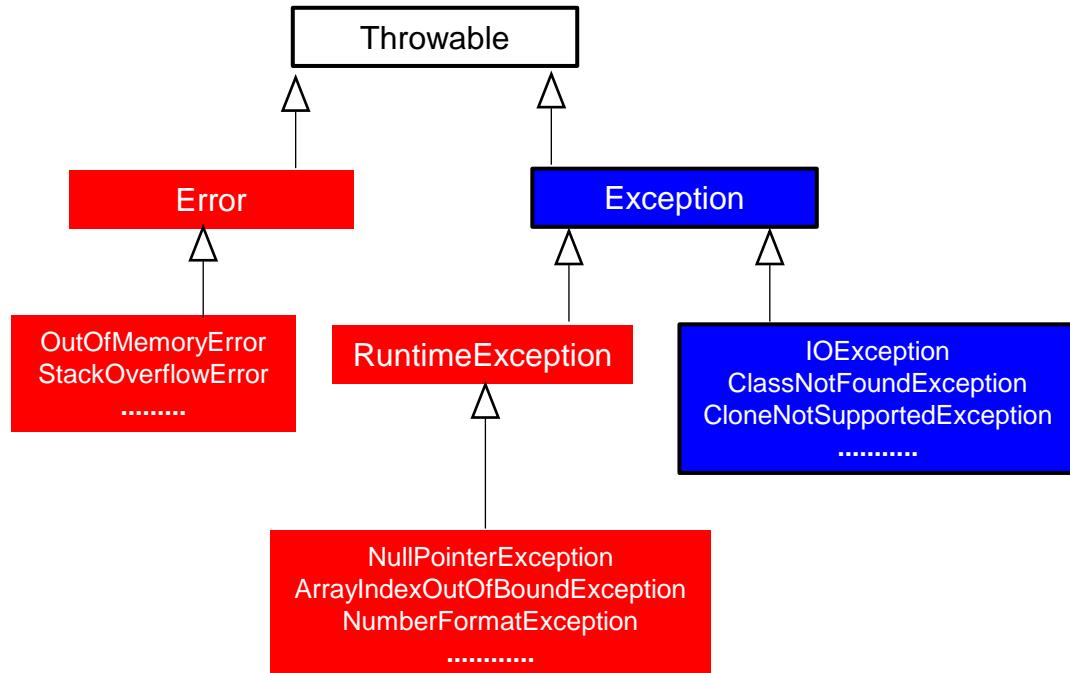
Rethrow the exception and control is transferred to the caller

How Exceptions are Handled by JVM

- Any method invocation is represented as a “**stack frame**” on the Java “**stack**”
 - **Callee-Caller** relationship
 - If method A calls method B then A is **caller** and B is **callee**
 - Each frame stores local variables, input parameters, return values and intermediate calculations
 - In addition, each frame also stores an “**exception table**”
 - This exception table stores information on each try/catch/finally block, i.e. the instruction offset where the catch/finally blocks are defined
 - When an exception is thrown, JVM does the following:
 1. Look for exception handler in current stack frame (method)
 2. If not found, then terminate the execution of current method and go to the callee method and repeat step 1 by looking into callee's exception table
 3. If no matching handler is found in any stack frame, then JVM finally terminates by throwing the stack trace (printStackTrace method)



Exception Hierarchy



- Exceptions are classes that extends `Throwable`
- Come in two types
 - **Checked exceptions**
 - Those that must be handled somehow (we will see soon)
 - E.g., `IOException` – file reading issue
 - **Unchecked exceptions**
 - Those that do not
 - E.g., `RuntimeException` that is caused due to programming errors
 - You should **not** attempt to handle exceptions from subclass of `Error`
 - Rarely occurring exceptions that even if you try to handle, there is little you can do beyond notifying the user and trying to terminate the program gracefully

Handling Checked Exception (1/3)

```
import java.io.FileReader;

public class Tester {
    public int countChars(String fileName) {
        FileReader r = new FileReader(fileName);
        int total = 0;
        while( r.ready() ) {
            r.read();
            total++;
        }
        r.close();
        return total;
    }
}
```

- If we have code that tries to build a FileReader we must deal with the possibility of the exception
 - The code contains a syntax error. "unreported exception
java.io.FileNotFoundException"
 - must be caught or declared to be thrown

Handling Checked Exception (2/3)

```
import java.io.FileReader;

public class Tester {
    public int countChars(String fileName) {
        FileReader r = new FileReader(fileName);
        int total = 0;
        while( r.ready() ) {
            r.read();
            total++;
        }
        r.close();
        return total;
    }
}
```

- Here, there are 4 statements that can generate checked exceptions:
 - The FileReader constructor
 - the ready method
 - the read method
 - the close method
- To deal with the exceptions we can either state this method **"throws"** an Exception of the proper type or handle the exception within the method itself

Handling Checked Exception (3/3)

```
import java.io.FileReader;

public class Tester {
    public int countChars(String fileName) throws
FileNotFoundException, IOException {
        FileReader r = new FileReader(fileName);
        int total = 0;
        while( r.ready() ) {
            r.read();
            total++;
        }
        r.close();
        return total;
    }
}
```

- It may be that we don't know how to deal with an error within the method that can generate it
- In this case we will pass the buck to the method that called us
- The keyword **throws** is used to indicate a method has the possibility of generating an exception of the stated type
- Now any method calling ours, must also throw an exception or handle it

Question

```
public class Main {  
    public static void main(String[] args) {  
        String s = null;  
        try {  
            int length = s.length();  
        }  
  
        catch (Exception e) {  
            System.out.println("Catch block -1");  
        }  
        catch (NullPointerException e) {  
            System.out.println("Catch block -2");  
        }  
    }  
}
```

- What is the output of the following program?
- Answer
 - Compilation error!
 - **Unreachable catch block**
 - **error: exception NullPointerException has already been caught**

Some Important Methods in **Throwable**

String toString()

Returns a short description of the exception

String getMessage()

Returns the detail description of the exception

void printStackTrace() Prints the stacktrace information on the console

```
1. public class Andy {  
2.     public void drinkWater() {  
3.         getWater();  
4.     }  
5.     public void getWater() {  
6.         try {  
7.             _water = _wendy.getADrink(); //null  
8.             int volume = _water.getVolume();  
9.         }  
10.        catch(NullPointerException e) {  
11.            e.printStackTrace();  
12.        }  
13.    }  
14. }
```

- Output:

java.lang.NullPointerException

at Andy.getWater(Andy.java:8)

at

Andy.drinkWater(Andy.java:3)

.....

Overriding Methods Having **throws** (1/3)

```
import java.lang.CloneNotSupportedException;

public class Cloning {

    public void createClone()
        throws CloneNotSupportedException {
        System.out.println("Clone created");
    }

    public class Human extends Cloning {

        @Override
        public void createClone()
        {
            System.out.println("Cloning not allowed");
        }
    }
}
```

- If a method in parent class throws an exception (either checked or unchecked), then overridden implementation of that method in child class is not required to throw that exception
 - Although throwing that **same** exception in overridden method won't hurt

Overriding Methods Having **throws** (2/3)

```
import java.lang.CloneNotSupportedException;

public class Cloning {

    public void createClone()
    {
        System.out.println("Clone created");
    }
}

public class Human extends Cloning {

    @Override
    public void createClone()
        throws CloneNotSupportedException {
        System.out.println("Cloning not allowed");
    }
}
```

- However, the reverse may/may not work
- **Case-1:** Overridden method throws **checked exception** but not the actual method in parent class
 - Compilation error

Overriding Methods Having **throws** (3/3)

```
import java.lang.CloneNotSupportedException;

public class Cloning {

    public void createClone()
    {
        System.out.println("Clone created");
    }
}

public class Human extends Cloning {

    @Override
    public void createClone()
        throws RuntimeException {
        System.out.println("Cloning not allowed");
    }
}
```

- However, the reverse may/may not work
- **Case-2:** Overridden method throws **unchecked exception** but not the actual method in parent class
 - This works fine

Defining Your Own Exception (1/4)

```
public class NoWaterException extends Exception {  
    public NoWaterException(String message) {  
        super(message);  
    }  
}  
  
public class Andy {  
    public void drinkWater() {  
        try {  
            getWater();  
        }  
        catch(NoWaterException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
    public void getWater() throws NoWaterException {  
        _water = _wendy.getADrink();  
        if(_water == null) {  
            this.fire(_wendy);  
            throw new NoWaterException("NO Water");  
        }  
    }  
}
```

- You can define and throw your own specialized exceptions
 - `throw new NoWaterException(...);`
- Useful for responding to special cases, not covered by pre-defined exceptions
- The class `Exception` has a method `getMessage()`. The String passed to `super` is printed to the output window for debugging when `getMessage()` is called by the user

Defining Your Own Exception (2/4)

```
public class NoWaterException extends Exception {  
    public NoWaterException(String message) {  
        super(message);  
    }  
}  
  
public class Andy {  
    public void drinkWater() {  
        try {  
            getWater();  
        }  
        catch(NoWaterException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
    public void getWater() throws NoWaterException {  
        _water = _wendy.getADrink();  
        if(_water == null) {  
            this.fire(_wendy);  
            throw new NoWaterException("NO Water");  
        }  
    }  
}
```

- Every method that throws Exceptions that are not subclasses of RuntimeException must declare what exceptions it throws in method declaration
- getWater() is throwing the exception, hence it must declare that using the “throws” on method declaration

Defining Your Own Exception (3/4)

```
public class NoWaterException extends Exception {  
    public NoWaterException(String message) {  
        super(message);  
    }  
}  
  
public class Andy {  
    public void drinkWater() throws NoWaterException {  
        getWater();  
    }  
    public void getWater() throws NoWaterException {  
        _water = _wendy.getADrink();  
        if(_water == null) {  
            this.fire(_wendy);  
            throw new NoWaterException("NO Water");  
        }  
    }  
    public static void main(String[] args) {  
        Andy obj = new Andy();  
        obj.drinkWater();  
    }  
}
```

- Any method that directly or indirectly calls `getWater()` must declare that it can generate `NoWaterException` using `throws` keyword
 - Not doing this generate compilation error
 - `error: unreported exception
NoWaterException;
must be caught or
declared to be thrown`

Defining Your Own Exception (4/4)

```
1. public class NoWaterException extends Exception {  
2.     public NoWaterException(String message) {  
3.         super(message);  
4.     }  
5. }  
6. public class Andy {  
7.     public void drinkWater() throws NoWaterException {  
8.         getWater();  
9.     }  
10.    public void getWater() throws NoWaterException {  
11.        _water = _wendy.getADrink();  
12.        if(_water == null) {  
13.            this.fire(_wendy);  
14.            throw new NoWaterException("NO Water");  
15.        }  
16.    }  
17.    public static void main(String[] args)  
18.                    throws NoWaterException {  
19.        Andy obj = new Andy();  
20.        obj.drinkWater();  
21.    }  
22.}
```

- This works fine, although we are not catching the NoWaterException anywhere that is again not a defensive programming!

- Running this program with `_water = null`

```
Exception in thread "main"  
NoWaterException: NO Water  
at Andy.getWater(Andy.java:14)  
at Andy.drinkWater(Andy.java:8)  
at Andy.main(Andy.java:20)
```

Pros and Cons of Exception

- Pros
 - Cleaner code: rather than returning a boolean up chain of calls to check for exceptional cases, throw an exception!
 - Use return value for meaningful data, not error checking
 - Factor out error-checking code into one class, so it can be reused
- Cons
 - Throwing exceptions requires extra computation
 - Can become messy if not used economically
 - Can accidentally cover up serious exceptions, such as `NullPointerException` by catching them

Next Lecture

- Assertions
- Java collection framework

CSE201: Monsoon 2024 Advanced Programming

Lecture 10: Collection Framework

Dr. Arun Balaji Buduru

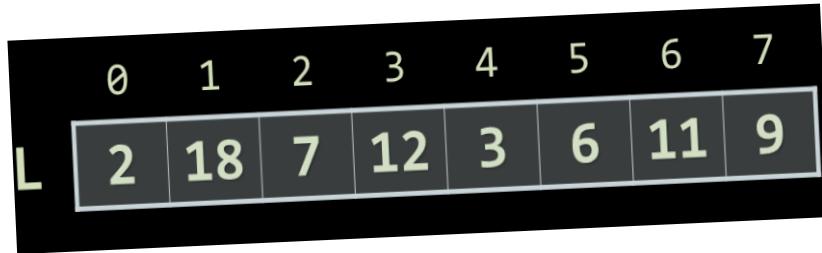
Head, Center of Technology in Policing

Founding Head, Usable Security Group (USG)

Associate Professor, Dept. of CSE | HCD

IIT-Delhi, India

How is your Experience using Arrays?

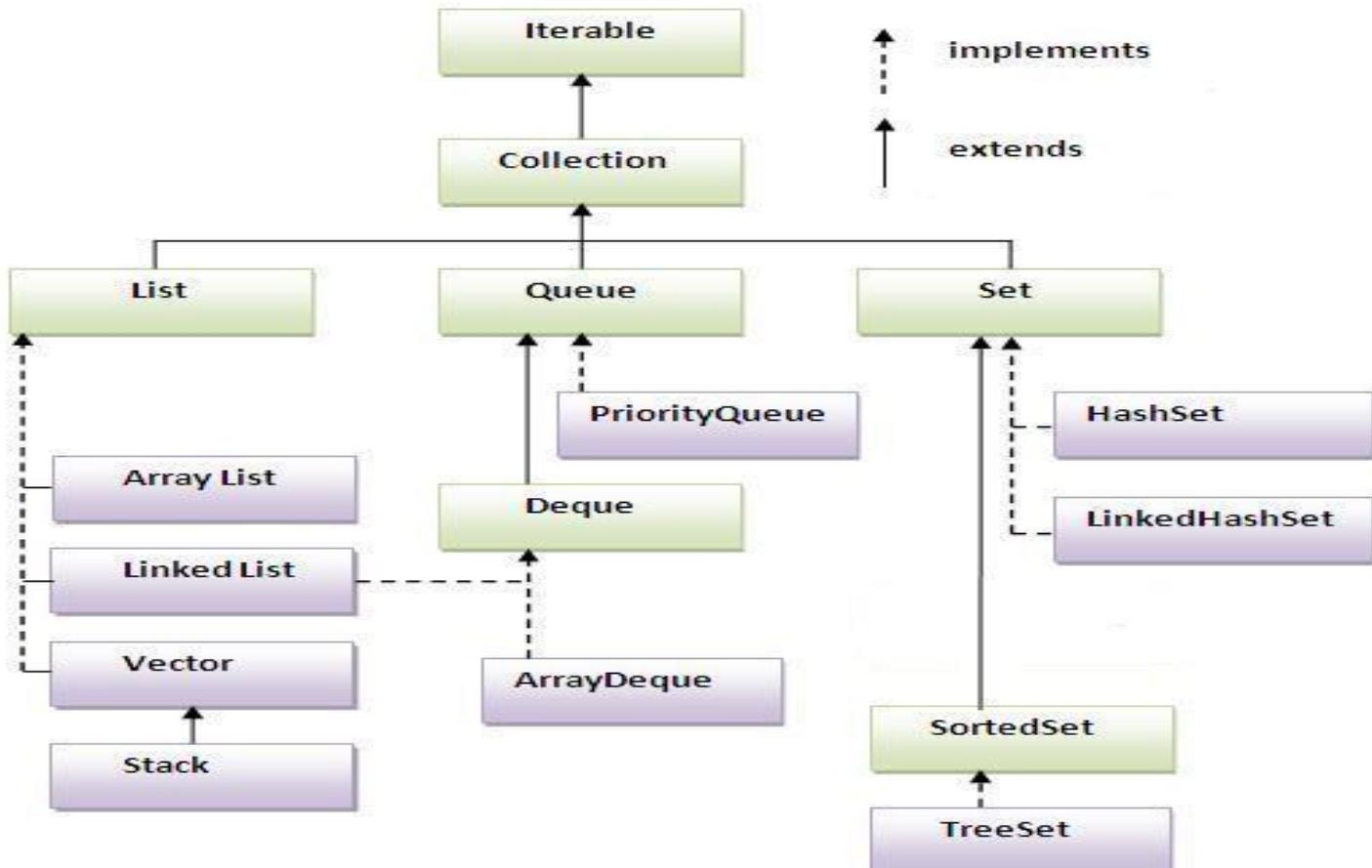


- Has fixed size (`length`)
 - Can you do it programmatically?
 - Memory wastage?
- Deleting an element
 - Can you do it programmatically?
- Comparing two arrays
 - Can you use “==” or `equals()`?
- Can you assign one array to other?
 - `int a[], b[]; a=b`

Java Collection Framework

- Unified architecture for representing and manipulating collections
 - A collection (sometimes called a *container*) is simply an object that groups multiple elements into a single unit
 - Very useful
 - store, retrieve and manipulate data
 - transmit data from one method to another
- Collection framework contains three things
 - Interfaces
 - Implementations
 - Algorithms
- This group of collection classes/interfaces are referred to as Java Collection Framework (JCF)
 - The classes in JCF are found in package “`java.util`”.

Collection Hierarchy



Interface Can Extend Another Interface (1/2)

```
public interface Moveable {  
    public void move_left();  
    public void move_right();  
}
```

```
public interface Flyable {  
    public void fly_up();  
    public void fly_down();  
    public void move_left();  
    public void move_right();  
}
```

```
public class Car  
    implements Moveable {  
    ...  
    public void move_left() {  
        // move left  
    }  
    public void move_right() {  
        // move right  
    }  
    // more methods elided  
}
```

```
public class Airplane  
    implements Flyable {  
    ...  
    public void move_left() {  
        // move left  
    }  
    public void move_right() {  
        // move right  
    }  
    public void fly_up() {  
        // fly up  
    }  
    public void fly_down() {  
        // fly down  
    }  
}
```

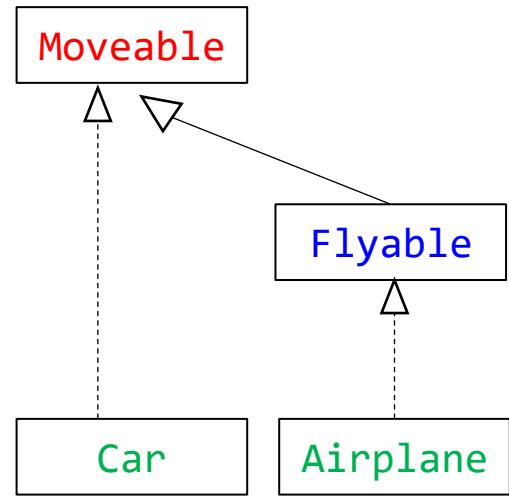
Interface Can Extend Another Interface (2/2)

```
public interface Moveable {  
    public void move_left();  
    public void move_right();  
}
```

```
public interface Flyable  
    extends Moveable {  
    public void fly_up();  
    public void fly_down();  
}
```

```
public class Car  
    implements Moveable {  
    ...  
    public void move_left() {  
        // move left  
    }  
    public void move_right() {  
        // move right  
    }  
    // more methods elided  
}
```

```
public class Airplane  
    implements Flyable {  
    ...  
    public void move_left() {  
        // move left  
    }  
    public void move_right() {  
        // move right  
    }  
    public void fly_up() {  
        // fly up  
    }  
    public void fly_down() {  
        // fly down  
    }  
}
```



Iterable Interface Source Code

```
package java.lang;
public interface Iterable<E> {
    Iterator<E> iterator();
}
```

- Just one method in this interface
- Objects of all classes that implements this interface can be the target of foreach statement
- Iterators allow iterating over the entire collection. It also allows element removal from collection during iteration

Iterator Interface

- Defines three fundamental methods
 - `Object next()`
 - `boolean hasNext()`
 - `void remove()`
- These three methods provide access to the contents of the collection
- An Iterator knows position within collection
- Each call to `next()` “reads” an element from the collection
 - Then you can use it or remove it

Iterator Position

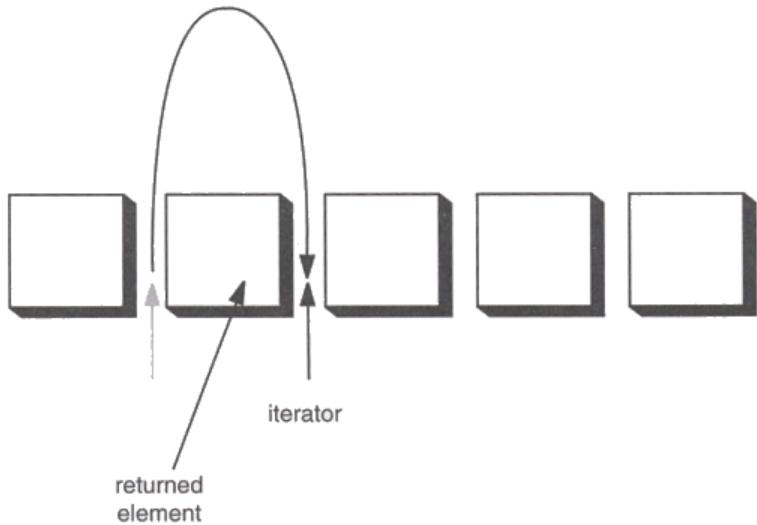


Figure 2–3: Advancing an iterator

Collection Interface Source Code

```
package java.util;
public interface Collection<E> extends Iterable<E>
{
    int size();
    boolean isEmpty();
    contains(Object o);
    boolean add(E e);
    boolean remove(Object o);
    equals(Object o);
    .....
    .....
}
```

- Defines fundamental methods that are enough to define the basic behavior of a collection
- Inherit the method from Iterable interface

Example - SimpleCollection

```
public class SimpleCollection {  
    public static void main(String[] args) {  
        Collection c = new ArrayList();  
        for (int i=0; i < 10; i++) {  
            c.add(i);  
        }  
        Iterator iter = c.iterator();  
        while (iter.hasNext())  
            System.out.println(iter.next());  
    }  
}
```

List Interface

- Recall in Java, arrays have fixed length
 - Cannot add / remove / insert elements
- Lists are like resizable arrays
 - Allow add / remove / insert of elements
- List **interface** is defined through the **ArrayList<E>** class
 - Where **E** is the type of the list, e.g. **String** or **Integer**

List Interface Source Code

```
package java.util;
public interface List<E> extends Collection<E> {
    E get(int index);
    E set(int index);
    void add(int index, E element);
    E remove(int index);
    ListIterator<E> listIterator();
    .....
    .....
}
```

- Observe that List interface has two different iterators
 - `Iterator<E> iterator();`
 - `ListIterator<E> listIterator();`

ListIterator Interface

- Extends the Iterator interface
- Defines three fundamental methods
 - `void add(Object o)` - before current position
 - `boolean hasPrevious()`
 - `Object previous()`
- The addition of these three methods defines the basic behavior of an ordered list
- Iterator v/s ListIterator
 - Unlike Iterator, a ListIterator knows position within list (obtain indexes)
 - Iterator allows traversal only in forward direction but ListIterator allows list traversal in both forward and backward directions
 - ListIterator can be used to traverse a List only

List Implementations

- **ArrayList**
 - low cost random access (at an index)
 - high cost insert and delete
 - array that resizes if need be
- **LinkedList**
 - sequential access but high cost random access (at an index)
 - low cost insert and delete

ArrayList overview

- Constant time positional access (it's an array)
- One tuning parameter, the initial capacity to constructor
- Constructors
 - `ArrayList()`
 - Build an empty ArrayList (of initial size 10)
 - `ArrayList(Collection c)`
 - Build an ArrayList initialized with the elements of the collection c
 - `ArrayList(int initialCapacity)`
 - Build with the specified initial capacity

ArrayList Methods

- The indexed get and set methods of the List interface are appropriate to use since ArrayLists are backed by an array
 - `Object get(int index)`
 - `Object set(int index, Object element)`
 - `May throw IndexOutOfBoundsException`
- Indexed add and remove are provided, but can be costly if used frequently
 - `void add(int index, Object element)`
 - `Object remove(int index)`
 - `May throw IndexOutOfBoundsException`
- May want to resize in one shot if adding many elements
 - `void ensureCapacity(int minCapacity)`
- ArrayList allows adding duplicate elements

How ArrayList Store Objects in Heap?

```
public boolean add(E e) {  
    ensureCapacity(size+1);  
    elementData[size++] = e;  
    return true;  
}
```

```
// Increase the capacity if necessary to ensure that it can  
// hold atleast the minCapacity number of elements  
public void ensureCapacity(int minCapacity) {  
    ....  
    int oldCapacity = elementData.length;  
    if(minCapacity > oldCapacity) {  
        ....  
        int newCapacity = .....  
        elementData = Arrays.copyOf(elementData, newCapacity);  
    }  
}
```

- ArrayList stores objects in an Object array
 - private Object[] elementData;
- Resizable array implementation

LinkedList Overview (1/2)

- Stores each element in a node
- Each node stores a link to the next and previous nodes
 - Doubly linked list
- Insertion and removal are inexpensive
 - just update the links in the surrounding nodes
- Linear traversal is inexpensive
- Random access is expensive
 - Start from beginning or end and traverse each node while counting

LinkedList Overview (2/2)

- Constructors
 - `LinkedList()`
 - Build an empty `LinkedList`
 - `LinkedList(Collection c)`
 - Construct a list containing the elements of the specified collection, in the order they are returned by the collection's iterator

LinkedList Methods

- ListIterator knows about position
 - use `add()` to add at a position
 - use `remove()` to remove at a position
- Few other methods
 - `void addFirst(Object o)`, `void addLast(Object o)`
 - `Object getFirst()`, `Object getLast()`
 - `Object removeFirst()`, `Object removeLast()`

Example: LinkedList

```
import java.util.*;
public class Book {
    private String name;
    private int pages;
    public Book(int p, String s) { ..... }
    @Override
    public String toString() { ..... }
    public static void main(String[] args) {
        List<Book> list = new LinkedList<Book>();

        list.add(new Book(100, "ABC"));
        list.add(new Book(200, "DEF"));
        list.add(new Book(300, "GHI"));

        for(Book b:list) {
            System.out.println(b);
        }
    }
}
```

Sets

- Sets keep unique elements only
 - Like lists but no duplicates
- HashSet<E>
 - Keeps a set of elements in a hash tables
 - The elements are randomly ordered by their hash code
- TreeSet<E>
 - Keeps a set of elements in a red-black ordered search tree
 - The elements are ordered incrementally

Set Interface

- Same methods as Collection
 - different contract - no duplicate entries
 - How?
- Provides an Iterator to step through the elements in the Set
 - No guaranteed order in the basic Set interface

HashSet

- Find and add elements very quickly
 - uses hashing
- Hashing uses an array of linked lists
 - The `hashCode()` is used to index into the array
 - Then `equals()` is used to determine if element is in the (short) list of elements at that index
- No order imposed on elements

TreeSet

- Elements can be inserted in any order
 - The TreeSet stores them in order
- Default order is defined by natural order
 - Objects implement the Comparable interface
 - TreeSet uses `compareTo(Object o)` to sort

Example: TreeSet

```
import java.util.*;
public class Book implements Comparable<Book> {
    private String name;
    private int pages;
    public Book(int p, String s) { ..... }
    @Override
    public String toString() { ..... }
    public int compareTo(Book b) {
        if(this.page>b.getpage()) return 1;
        else if(this.page<b.getpage()) return -1;
        else return 0;
    }
    public static void main(String[] args) {
        Set<Book> set = new TreeSet<Book>();
        set.add(new Book(100, "ABC"));
        set.add(new Book(200, "DEF"));
        for(Book b:set) { // you can also use iterator
            System.out.println(b);
        }
    }
}
```

- The elements in TreeSet must be of Comparable type
- You need to add compareTo in user defined classes

Maps

- Maps keep unique <key, value> pairs
- `HashMap<K, V>`
 - Keeps a map of elements in a hash table
 - The elements are randomly ordered using their hash code
- `TreeMap<K, V>`
 - Keep a set of elements in a red-black ordered search tree
 - The elements are ordered incrementally by their key

Map Interface

- Stores unique key/value pairs
- Maps from the key to the value
- Keys are unique
 - a single key only appears once in the Map
 - a key can map to only one value
- Values do not have to be unique

Example: HashMap

```
import java.util.*;
public class Book {
    private String name;
    private int pages;
    public Book(int p, String s) { ..... }
    @Override
    public String toString() { ..... }
    public static void main(String[] args) {
        Map<Integer, Book> map = new HashMap<Integer, Book>();

        map.add(1, new Book(100, "ABC"));
        map.add(2, new Book(200, "DEF"));
        for(Map.Entry e:map.entrySet()) {
            System.out.println(e.getKey() + ":" + e.getValue());
        }
    }
}
```