CSE201: Monsoon 2023
CSE Section Advanced Programming

Lecture 01: Introduction to OOP

Raghava Mutharaju
Computer Science and Engineering
IIIT-Delhi
raghava.mutharaju@iiitd.ac.in

Why Object Oriented Programming?

Procedure Oriented Programming

Object Oriented Programming

© Vivek Kumar

What is OOP?

It is a programming paradigm based on the concept of "objects", which may contain data in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods
(Wikipedia)

© Vivek Kumar

What is OOP?

© Vivek Kumar

Advantages of OOP

- Code reuse and recycling
  - Objects can easily be reused
- Design benefits
  - Extensive planning phase results in better design and lesser flaws
- Software maintenance
  - Easy to incorporate changes in legacy code (e.g., supporting a new hardware)
- Simplicity

© Vivek Kumar

OOP Features

- **Encapsulation**
- Method overloading
- Inheritance
- Abstraction
- Method overriding
- Polymorphism

© Vivek Kumar

Encapsulation

The main thing is How to drive a car ……

How the car is moving and how the engine is working, this information is hidden.

(**Encapsulation**)

© Vivek Kumar

Encapsulation

- An encapsulated object can be thought of as a *black box* -- its inner workings are hidden from the client
- The client invokes the interface methods of the object, which manages the instance data

© 2004 Pearson Addison-Wesley. All rights reserved

Class Without Encapsulation

Class Without Encapsulation

Class Supporting Encapsulation

Visibility Modifier

	`public`	`private`
Variables	**Violate encapsulation**	**Enforce encapsulation**
Methods	**Provide services to clients**	**Support other methods in the class**

© 2004 Pearson Addison-Wesley. All rights reserved

Accessors and Mutators

- Because instance data is private, a class usually provides services to access and modify data values
- An *accessor method* returns the current value of a variable
- A *mutator method* changes the value of a variable
- The names of accessor and mutator methods take the form `getX` and `setX`, respectively, where `X` is the name of the value
- They are sometimes called “getters” and “setters”

Wait, but why do we need “setter” when we are talking about restricting accesses to fields from outside world ?

© 2004 Pearson Addison-Wesley. All rights reserved

Mutator Restrictions

- The use of mutators gives the class designer the ability to restrict a client's options to modify an object's state
- A mutator is often designed so that the values of variables can be set only within particular limits

© 2004 Pearson Addison-Wesley. All rights reserved

Procedural v/s OOP

PROCEDURAL PROGRAMMING	OBJECT ORIENTED PROGRAMMING
11.Relationship of data and function in procedural	11.Relationship of data and function in OOP.

A Sample Problem

- Write a method that will throw 2 Dice with varying number of sides, a specified amount of times, and reports how many times we got a snake eyes (both dice showing 1)

- For example numSnakeEyes(6, 13, 100) should return the number of snake eyes after throwing a 6 sided Dice and 13 sided Dice 100 times

© 2004 Pearson Addison-Wesley. All rights reserved

Procedural (Structured) Programming Approach

```
static Random rand = new Random();

static int roll(int numFaces) {
    return 1 + rand.nextInt(numFaces);
}

static int numSnakeEyes(int sides1, int sides2, int numThrows) {
    int count = 0;
    for(int i = 0; i < numThrows; i++) {
        int face1 = roll(sides1);
        int face2 = roll(sides2);
        if (face1 == 1 && face2 == 1)
            count++;
    }

    return count;
}
```


© 2004 Pearson Addison-Wesley. All rights reserved

OOP Approach

- In OOP, we first focus on the **main actors**, not how things are done.
- The main actors here are Dice objects. We need to define a Dice class that captures the *state* and *behavior of a Dice.*
- We can then instantiate as many dice objects as we need for any particular programs

© 2004 Pearson Addison-Wesley. All rights reserved

Classes (Recap)

- A class can contain data declarations and method declarations

© 2004 Pearson Addison-Wesley. All rights reserved

Dice Class

© 2004 Pearson Addison-Wesley. All rights reserved

OOP Approach

```
public class Dice {
    private final int numFaces;  //maximum face value
    private int faceValue;  //current value showing on the dice

    //  Constructor: Sets the initial face value.
    public Dice(int _numFaces)  {
    numFaces = _numFaces;
        roll();
    }

    //  Rolls the dice
    public void roll() {
        faceValue = 1 + rand.nextInt(numFaces);
    }

    //  Face value setter/mutator.
    public void setFaceValue (int value)  {
    if (value <= numFaces)
            faceValue = value;
    }
```

© 2004 Pearson Addison-Wesley. All rights reserved

OOP Approach

```
    //  Face value getter/setter.
    public int getFaceValue() {
        return faceValue;
    }
    //  Face value getter/setter.
    public int getNumFaces() {
        return numFaces;
    }
    //  Returns a string representation of this dice
    public String toString() {
        return “number of Faces “ + numFaces +
    “current face value “ + faceValue);
    }
} // End of Dice class
```

© 2004 Pearson Addison-Wesley. All rights reserved

OOP Approach

The new version

```
static int numSnakeEyes(int sides1, int sides2, int
   numThrows) {
   Dice die1 = new Dice(sides1);
   Dice die2 = new Dice(sides2);

   int count = 0;
   for(int i = 0; i < numThrows; i++) {
   die1.roll();
   die2.roll();
   if (die1.getFaceValue == 1 &&
   die2.getFaceValue == 1 )
   count++;
   }
   return count;
}
```

© 2004 Pearson Addison-Wesley. All rights reserved

OOP Approach

Using Dice class in general

```
Dice dice1, dice2;
int sum;

dice1 = new Dice(7);
dice2 = new Dice(34);

dice1.roll();
dice2.roll();
System.out.println ("Dice One: " + dice1 + ", Dice Two: " +
   dice2);

dice1.roll();
dice2.setFaceValue(4);
System.out.println ("Dice One: " + dice1 + ", Dice Two: " +
   dice2);

sum = dice1.getFaceValue() + dice2.getFaceValue();
System.out.println ("Sum: " + sum);

sum = dice1.roll() + dice2.roll();
System.out.println ("Dice One: " + dice1 + ", Dice Two: " +
   dice2);
System.out.println ("New sum: " + sum);
```

© 2004 Pearson Addison-Wesley. All rights reserved

Instance Data

- We can depict the two `Dice` objects from the `RollingDice` program as follows:

Each object maintains its own `faceValue` and `numFaces` variable, and thus its own state

The toString Method

- All classes that represent objects should define a `toString` method
- The `toString` method returns a character string that represents the object in some way
- It is called automatically when an object is concatenated to a string or when it is passed to the `println` method

© 2004 Pearson Addison-Wesley. All rights reserved

Another Sample Problem

- Coin example
  - Write a program that flips two coins until one of them comes up with heads three times in a row, and report the winner

© 2004 Pearson Addison-Wesley. All rights reserved

Coin Class

```
public class Coin
{
   private final int HEADS = 0;
   private final int TAILS = 1;

   private int face;

  public Coin () {
     flip();
   }
  public void flip () {
     face = (int) (Math.random() * 2);
   }
  public boolean isHeads () {
     return (face == HEADS);
  }
  public String toString() {
     String faceName;
     if (face == HEADS)
       faceName = "Heads";
     else
       faceName = "Tails";
     return faceName;
   }
} // end of class Coin
```

© 2004 Pearson Addison-Wesley. All rights reserved

FlipRace

```
//  Flips two coins until one of them comes up
// heads three times  in a row.
public static void main (String[] args) {
     final int GOAL = 3;
     int count1 = 0, count2 = 0;

     // Create two separate coin objects
     Coin coin1 = new Coin();
     Coin coin2 = new Coin();

     while (count1 < GOAL && count2 < GOAL)
     {
        coin1.flip();
        coin2.flip();

        // Print the flip results (uses Coin's toString method)
        System.out.print ("Coin 1: " + coin1);
        System.out.println ("   Coin 2: " + coin2);

        // Increment or reset the counters
        count1 = (coin1.isHeads()) ? count1+1 : 0;
        count2 = (coin2.isHeads()) ? count2+1 : 0;
     }
     // Determine the winner
     if (count1 < GOAL)
        System.out.println ("Coin 2 Wins!");
     else
        if (count2 < GOAL)
           System.out.println ("Coin 1 Wins!");
        else
           System.out.println ("It's a TIE!");
 } // end of main()
```

© 2004 Pearson Addison-Wesley. All rights reserved

Summary

- What is OOP?
- Encapsulation
  - Visibility modifiers
  - Accessors and mutators
- Simple examples to understand the above concepts

About this Course

- **Quizzes**
  - Roughly every two weeks; two will be announced beforehand
  - 30mins duration
  - During lecture hours
  - Best of N-1 (4 are planned)
- **Labs / Assignments**
  - Take home type programming assignments (3 are planned)
  - You **must** maintain github repositories
  - **Rigor will be lot more than Java Refresher Module Assignments**
- **Projects**
  - Will commence/announced right after mid semester exam
  - Based on pair programming (two students in each group)
  - Topic(s) will be provided by the instructor
  - Will run until end of semester with three deadlines altogether

We are very strict with plagiarism policy. No excuses for violations!

Google Classroom

qrfi3cj

Next Class

- How to identify classes and objects in OOP

CSE201: Monsoon 2023
Advanced Programming

Lecture 02: Classes and Objects

Raghava Mutharaju

Computer Science and Engineering

IIIT Delhi

raghava.mutharaju@iiitd.ac.in

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

© Vivek Kumar

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

© Vivek Kumar

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

© Vivek Kumar

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

© Vivek Kumar

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*

© Vivek Kumar

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
Customer
HomePage
LoginPage
EmailAccount

Methods
clickLogin
display
enterCredentials
clickOK
validate

Sequence Diagram

© Vivek Kumar

Sequence Diagram

© Vivek Kumar

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

Andries van Dam © 2016 09/20/16

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

Andries van Dam © 2016 09/20/16

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
  }
}
```


Andries van Dam © 2016 09/20/16

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
  }
}
```


Andries van Dam © 2016 09/20/16

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`

Andries van Dam © 2016 09/20/16

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;
```

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

- 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"

(somewhere else in memory)

Andries van Dam © 2016 09/20/16

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 seems 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

Andries van Dam © 2016 09/20/16

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!
  }
                 type          name
  public void groom(Dog shaggyDog)
  {
      // code that grooms shaggyDog
  }
}
```


Andries van Dam © 2016 09/20/16

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);
  }

}
```


Andries van Dam © 2016 09/20/16

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);
  }

}
```


Andries van Dam © 2016 09/20/16

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...

Andries van Dam © 2016 09/20/16

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.

Andries van Dam © 2016 09/20/16

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`.

Andries van Dam © 2016 09/20/16

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`.

Andries van Dam © 2016 09/20/16

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`.

Andries van Dam © 2016 09/20/16

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`.

Andries van Dam © 2016 09/20/16

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 `Dog`s 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);
  }
}
```


Andries van Dam © 2016 09/20/16

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);
  }
}
```


Andries van Dam © 2016 09/20/16

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);
   }
```

Andries van Dam © 2016 09/20/16

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);
   }
```

Andries van Dam © 2016 09/20/16

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);
   }
```

Andries van Dam © 2016 09/20/16

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);
   }
```

Andries van Dam © 2016 09/20/16

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);
    }
```

Andries van Dam © 2016 09/20/16

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

Andries van Dam © 2016 09/20/16

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, arg2, arg3);
        groomer.test2();
    }

}
```

local variables

Andries van Dam © 2016 09/20/16

Accessing Local Variables

- If you try to access a local variable outside of it's method, you'll receive a "cannot find symbol" compilation error.

```
public class PetShop {

   private DogGroomer _groomer;

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


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

}
```

In Terminal:

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


Andries van Dam © 2016 09/20/16

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**

Andries van Dam © 2016 09/20/16

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)

Andries van Dam © 2016 09/20/16

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 {

  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);
  }
}
```

declaration

assignment

Andries van Dam © 2016 09/20/16

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

Andries van Dam © 2016 09/20/16

CSE201: Monsoon 2023
Advanced Programming

Lecture 03: Class Relationships

Raghava Mutharaju (Section-A)

CSE Department, IIIT-Delhi

raghava.mutharaju@iiitd.ac.in

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.

```java
public class PetShop {

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

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

```java
public class DogGroomer {

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

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

Somewhere in memory...

```java
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);
  }
```

```java
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);
  }

}
```

declaration

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)

© Vivek Kumar

A Sample UML Class Diagram

© Vivek Kumar

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?

© Vivek Kumar

Most Common Class Relationships

- **Composition**
  - A “contains” B
- **Association**
  - A “knows-about” B
- **Dependency**
  - A “depends on” B
- Inheritance
  - HarleyDavidson “is-a” Bike

© Vivek Kumar

Composition Relationship

- Class A **contains** object of class B
  - A **instantiate** 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

© Vivek Kumar

Composition in UML

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

© Vivek Kumar

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();
  }
}
```

© Vivek Kumar

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! B “doesn’t knows about” A
- **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

© Vivek Kumar

Association Example (1/4)

- A contractor's project keep's 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;
  }
}
```

© Vivek Kumar

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);
  }
}
```


Andries van Dam © 2016 9/22/16

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);
  }
}
```


Andries van Dam © 2016 9/22/16

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);
  }
}
```


Andries van Dam © 2016 9/22/16

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);
  }
}
```


Andries van Dam © 2016 9/22/16

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);
  }
}
```


Andries van Dam © 2016 9/22/16

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 is 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
}
```


Andries van Dam © 2016 9/22/16

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);
  }
}
```


Andries van Dam © 2016 9/22/16

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);
   }
}
```


Andries van Dam © 2016 9/22/16

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...

Andries van Dam © 2016 9/22/16

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)

Andries van Dam © 2016 9/22/16

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?)

Andries van Dam © 2016 9/22/16

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.

Andries van Dam © 2016 9/22/16

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...

Andries van Dam © 2016 9/22/16

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 */
}
```


Andries van Dam © 2016 9/22/16

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 */
}
```


Andries van Dam © 2016 9/22/16

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 */
}
```


Andries van Dam © 2016 9/22/16

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!

  }
}
```


Andries van Dam © 2016 9/22/16

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 `TA`s, we pass in `_akanksha`, `_akash`, `_alind` and `_abhiprayah`

```
public class Course {

  private Professor _vivek;
  private TA _akanksha;
  private TA _akash;
  private TA _alind;
  private TA _abhiprayah;

  public Course() {
      _akanksha = new TA();
      _akash = new TA();
      _alind = new TA();
      _abhiprayah = new TA();
      _vivek = new Professor(_akanksha,
                  _akash , _alind ,
  _abhiprayah);
  }
}
```


Andries van Dam © 2016 9/22/16

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 _vivek;
  private TA _akanksha;
  private TA _akash;
  private TA _alind;
  private TA _abhiprayah;

  public Course() {
      _akanksha = new TA();
      _akash = new TA();
      _alind = new TA();
      _abhiprayah = new TA();
      _vivek = new Professor(_akanksha,
               _akash , _alind ,
  _abhiprayah);
  }
}
```


Andries van Dam © 2016 9/22/16

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 _vivek;
   private TA _akanksha;
   private TA _akash;
   private TA _alind;
   private TA _abhiprayah;

   public Course() {
       _akanksha = new TA();
       _akash = new TA();
       _alind = new TA();
       _abhiprayah = new TA();
       _vivek = new Professor(_akanksha,
                _akash , _alind ,
   _abhiprayah);
   }
}
```


Andries van Dam © 2016 9/22/16

Associations Example (4/4)

- This doesn't work: when we instantiate `_akanksha`, `_akash`, `_alind` and `_abhiprayah`, we would like to pass them an argument, `_vivek`
- But `_vivek` hasn't been instantiated yet! And can't initialize `_vivek` first because the TAs haven't been created yet…
- What can we try instead?

```
public class Course {

  private Professor _vivek;
  private TA _akanksha;
  private TA _akash;
  private TA _alind;
  private TA _abhiprayah;

  public Course() {
      _akanksha = new TA();
      _akash = new TA();
      _alind = new TA();
      _abhiprayah = new TA();
      _vivek = new Professor(_akanksha,
               _akash , _alind ,
  _abhiprayah);
  }
}
```


Andries van Dam © 2016 9/22/16

Associations Example (4/4)

- Need a way to pass `_vivek` to `_akanksha`, `_akash`, `_alind` and `_abhiprayah` **after** we instantiate `_vivek`
- Use a new method, `setProf`, and pass each TA `_vivek`

```
public class Course {

  private Professor _vivek;
  private TA _akanksha;
  private TA _akash;
  private TA _alind;
  private TA _abhiprayah;

  public Course() {
      _akanksha = new TA();
      _akash = new TA();
      _alind = new TA();
      _abhiprayah = new TA();
      _vivek = new Professor(_akanksha,
              _akash , _alind ,
  _abhiprayah);

  }
}
```


Andries van Dam © 2016 9/22/16

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 `_vivek`!

```
public class Course {

  private Professor _vivek;
  private TA _akanksha;
  private TA _akash;
  private TA _alind;
  private TA _abhiprayah;

  public Course() {
      _akanksha = new TA();
      _akash = new TA();
      _alind = new TA();
      _abhiprayah = new TA();
      _vivek = new Professor(_akanksha,
               _akash , _alind ,
  _abhiprayah);

      _akanksha.setProf(_vivek);
      _akash.setProf(_vivek);
      _alind.setProf(_vivek);
      _abhiprayah.setProf(_vivek);
  }
}
```

Question

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

Andries van Dam © 2016 9/22/16

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

© Vivek Kumar

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

© Vivek Kumar

Dependency Example (1/3)

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

class Player {
  public void takeTurn(Die die) {
      die.roll();
  }
}
```

© Vivek Kumar

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();
  }
}
```

© Vivek Kumar

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();
  }
  .....
}
```

© Vivek Kumar

Next Lecture

- Inheritance in Java

CSE201: Monsoon 2023
Advanced Programming

Lecture 04:Interfaces in Java

Raghava Mutharaju (Section-A)
CSE, IIIT-Delhi
raghava.mutharaju@iiitd.ac.in

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 **instantiate** 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
    }
}
```


Andries van Dam © 2016 9/22/16

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`

Andries van Dam © 2016 9/22/16

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`

Andries van Dam © 2016 9/22/16

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

Andries van Dam © 2016 9/22/16

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
}
```


Andries van Dam © 2016 9/22/16

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

Andries van Dam © 2016 9/22/16

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
}
```


Andries van Dam © 2016 9/22/16

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
```


Andries van Dam © 2016 9/22/16

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();
    }
}
```


Andries van Dam © 2016 9/22/16

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 race to the city center!

Andries van Dam © 2016 9/22/16

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()`

Andries van Dam © 2016 9/22/16

Can we do better?

Andries van Dam © 2016 9/22/16

Things to think about

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

Andries van Dam © 2016 9/22/16

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?

Andries van Dam © 2016 9/22/16

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…
}
```


Andries van Dam © 2016 9/22/16

Is there another solution?

```
Racer
useCar(Car car)
useBike(Bike bike)
useHoverBoard(HoverBoard
   hoverboard)
useHorse(Horse horse)
useScooter(Scooter scooter)
useMotorcycle(Motorcycle
   motorcycle)
usePogoStick(PogoStick pogo)
```

→

```
Racer
useTransportation()
```

- Can we go from left to right?

Andries van Dam © 2016 9/22/16

Interfaces: Spot the Similarities

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

Andries van Dam © 2016 9/22/16

Cars vs. Bikes

Cars

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

- Move

Bikes

- Drop kickstand
- Change gears

Andries van Dam © 2016 9/22/16

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

Andries van Dam © 2016 9/22/16

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()`

Andries van Dam © 2016 9/22/16

Introducing Interfaces

- Interfaces group similar capabilities/function of different classes together
- Model “acts-as” relationship
- `Car`s and `Bike`s 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

Andries van Dam © 2016 9/22/16

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?**

Andries van Dam © 2016 9/22/16

Let’s break that down

1) Models Similarities

2) Ensures Consistency

Andries van Dam © 2016 9/22/16

Models Similarities While **Ensuring Consistency**

- How does this help our program?
- We know `Car`s and `Bike`s 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

Andries van Dam © 2016 9/22/16

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`

Andries van Dam © 2016 9/22/16

Declaring an Interface (2/4)

What does this look like?

```
public interface Transporter {

    public void move();

}
```

- Declare it as interface rather than class

Andries van Dam © 2016 9/22/16

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

Andries van Dam © 2016 9/22/16

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!

Andries van Dam © 2016 9/22/16

Questions

Which line(s) of this program is incorrect?

```
A. public interface Colorable {
      public Color getColor() {
            B.  return Color.WHITE;
      }
}
```

```
C. public class Rectangle implements Colorable {
      //constructor elided
      public Color getColor() {
            D. return Color.PURPLE;
      }
}
```


Andries van Dam © 2016 9/22/16

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?

Andries van Dam © 2016 9/22/16

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.

Andries van Dam © 2016 9/22/16

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

Andries van Dam © 2016 9/22/16

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

Andries van Dam © 2016 9/22/16

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
    }

}
```


Andries van Dam © 2016 9/22/16

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  }
}
```


Andries van Dam © 2016 9/22/16

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`

Andries van Dam © 2016 9/22/16

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);
}
```

B.

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

C.

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

D.

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


Andries van Dam © 2016 9/22/16

Next Lecture

- Interface and polymorphism

CSE201: Monsoon 2023
Advanced Programming

Lecture 05: Interfaces and Polymorphism

V. Raghava Mutharaju (Sections - A, B)

CSE, IIIT-Delhi

raghava.mutharaju@iiitd.ac.in

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`

Do we need two different Racer classes??

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();

}
```

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
}
```

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

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
}
```


Andries van Dam © 2016 9/22/16

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?

Andries van Dam © 2016 9/22/16

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?

Andries van Dam © 2016 9/22/16

What would this look like in code?

```
public class Racer {

    //previous code elided
    public void useTransportation(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

Andries van Dam © 2016 9/22/16

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

Andries van Dam © 2016 9/22/16

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?

Andries van Dam © 2016 9/22/16

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

Andries van Dam © 2016 9/22/16

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.
```


Andries van Dam © 2016 9/22/16

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`.

Andries van Dam © 2016 9/22/16

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()`

Andries van Dam © 2016 9/22/16

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`
- `transportation` can only call methods declared in `Transporter`

Andries van Dam © 2016 9/22/16

Is this legal?

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

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

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

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

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

Andries van Dam © 2016 9/22/16

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();
    }

}
```


Andries van Dam © 2016 9/22/16

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`

Andries van Dam © 2016 9/22/16

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?

Andries van Dam © 2016 9/22/16

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?

Andries van Dam © 2016 9/22/16

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

Andries van Dam © 2016 9/22/16

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());
    }
}
```


Andries van Dam © 2016 9/22/16

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

Andries van Dam © 2016 9/22/16

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();
```

B.
```
Clickable macBook = new Clickable();
macBook.type();
```

C.
```
Typeable macBook= new Laptop();
macBook.click();
```

D.
```
Clickable macBook = new Laptop();
macBook.click();
```


Andries van Dam © 2016 9/22/16

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

Andries van Dam © 2016 9/22/16

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

Andries van Dam © 2016 9/22/16

When to use polymorphism?

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

Andries van Dam © 2016 9/22/16

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

Andries van Dam © 2016 9/22/16

Why is this important?

- With 2 modes of transportation!
- Old Design:
  - need more classes and 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`

Andries van Dam © 2016 9/22/16

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();
   }
}
```


Andries van Dam © 2016 9/22/16

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

Andries van Dam © 2016 9/22/16

Next Lecture

- Inheritance and polymorphism

CSE201: Monsoon 2023
Advanced Programming

Lecture 06: Inheritance and Polymorphism

Raghava Mutharaju (Section-A)
CSE, IIIT Delhi
raghava.mutharaju@iiitd.ac.in

Last Lecture

- Polymorphism in Java
  - A way of coding generically
    - way of referencing many related objects as one generic type

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

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

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

```
    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 class Car implements Transporter
{
   public void move() { this.drive(); }
   .....
}

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

This Lecture

- Inheritance and Polymorphism

Slide acknowledgements: CS15, Brown University

Andries van Dam © 2016 9/27/16

Spot the Similarities

- What are the similarities between a convertible and a sedan?
- What are the differences?

Andries van Dam © 2016 9/27/16

Convertibles vs. Sedans

Andries van Dam © 2016 9/27/16

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()`

Andries van Dam © 2016 9/27/16

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?**

Andries van Dam © 2016 9/27/16

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

Animalia

Chordata

Mammalia

⋮

Canis

Andries van Dam © 2016 9/27/16

Modeling Inheritance (1/2)

- This is an inheritance diagram
  - Each box represents a class
- A Poodle “is-a” Dog, a Dog “is-a” Mammal
  - Transitively, a Poodle is a Mammal
- “Inherits from” = “is-a”
  - Poodle inherits from Dog
  - Dog inherits from Mammal
- This relationship is not bidirectional
  - A Poodle is a Dog, but not every Dog is a Poodle (could be a Labrador, a German Shepard, etc)

Andries van Dam © 2016 9/27/16

Modeling Inheritance (2/2)

- **Superclass/parent/base**: A class that is inherited from
- **Subclass/child/derived**: A class that inherits from another
- "A Poodle is a Dog"
  - Poodle is the subclass
  - Dog is the superclass
- A class can be both a superclass and a subclass
  - Ex. Dog
- In Java you can only inherit from one superclass (no multiple inheritance)
  - Other languages, like C++, allow for multiple inheritance, but too easy to mess up

Andries van Dam © 2016 9/27/16

Motivations for Inheritance

- A subclass inherits all of its parent's **public** and **protected** capabilities
  - If `Car` defines `drive()`, `Convertible` inherits `drive()` from `Car` and drives the same way. This holds true for all of `Convertible`'s subclasses as well
- Inheritance and Interfaces both legislate class's behavior, although in very different ways
  - Interfaces allow the compiler to enforce method implementation
    - An implementing class will have all capabilities outlined in an interface
  - Inheritance assures the compiler that all **subclasses** of a **superclass** will have the superclass's public capabilities without having to respecify code – methods are inherited
    - A `Convertible` knows how to drive and drives the same way as `Car` because of inherited code
- Benefit of inheritance
  - Code reuse
    - If `drive()` is defined in `Car`, `Convertible` doesn't need to redefine it! Code is inherited
    - Only need to implement what is different, i.e. what makes `Convertible` special

Andries van Dam © 2016 9/27/16

Superclasses vs Subclasses

- A superclass factors out commonalities among its subclasses
  - describes everything that all subclasses have in common
  - `Dog` defines things common to all `Dog`s
- A subclass differentiates/specializes its superclass by:
  - **adding new methods:**
    - the subclass should define specialized methods. All `Animal`s cannot swim, but `Fish` can
  - **overriding inherited methods:** (more on this after few slides!)
    - a `Bear` class might override its inherited sleep method so that it hibernates rather than sleeping as most other `Animals` do
  - **defining "abstract" methods:** (next lecture!)
    - the superclass declares but does not define

Andries van Dam © 2016 9/27/16

Let’s examine inheritance further

1. Model inheritance relationship
2. Adding new methods
3. Overriding methods

Andries van Dam © 2016 9/27/16

Modeling Inheritance

- Let's model a `Van`, a `Sedan`, and a `Convertible` class with inheritance!

Andries van Dam © 2016 9/27/16

Step 1: Define the superclass

- Defining `Car` is just like defining any other class

```
public class Car {
    private Engine _engine;
    //other variables elided
    public Car(){
        _engine = new Engine();
    }
    public void turnOnEngine() {
        _engine.start();
    }
    public void turnOffEngine() {
        _engine.shutOff();
    }
    public void cleanEngine() {
        _engine.steamClean();
    }
    public void drive() {
        //code elided
    }
    //more methods elided
}
```

Step 2: Define a subclass

- Notice the `extends` keyword
  - `extends` means “is a subclass of” or “inheriting from”
  - `extends` lets the compiler know that `Convertible` is inheriting from `Car`
  - Whenever you create a class that inherits from a superclass, must include “`extends <superclass name>`” in class declaration

```
public class Convertible extends Car
{
    //code elided for now
}
```


Andries van Dam © 2016 9/27/16

Model Inheritance

- You can create any number of subclasses
  - `Sedan`, `Van`, `Convertible`, `SUV`...could all extend from `Car`
  - These classes will inherit public capabilities from `Car`
- Each subclass can only inherit from one superclass
  - `Convertible` cannot extend `Car`, `FourWheeledTransportation`, and `GasFueledTransportation`
  - Contrast with interfaces: you can implement as many interfaces as you want

Andries van Dam © 2016 9/27/16

Let’s examine inheritance further

1. Model inheritance relationship
2. Adding new methods
3. Overriding methods

Andries van Dam © 2016 9/27/16

Adding new methods (1/2)

- Let's make a `Sedan` class that inherits from `Car`
- Let's make Convertible class that inherits from Car
- Can `Sedan` use `putTopDown()`?
  - Nope. That method is defined in `Convertible`, so only `Convertible` and `Convertible`'s subclasses can use it

```
public class Sedan extends Car {

    public Sedan (){

    }
    //other methods elided

}
```

```
public class Convertible extends Car {

    public Convertible(){

    }

    public void putTopDown(){
        //code elided
    }
    }
```


Andries van Dam © 2016 9/27/16

Adding new methods (2/2)

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

Andries van Dam © 2016 9/27/16

What can subclasses access? (1/2)

- Remember: a subclass inherits any **public or protected** methods and variables from its superclass. Subclass cannot access any **private** field/method from superclass
- Before adding any code to `Convertible` class, what does `Convertible` already know how to do?
  - It can do anything a `Car` can do!
    - `turnOnEngine()`
    - `turnOffEngine()`
    - `drive()`

Note that we don't list the parent's public methods again here – they are implicitly inherited!

What can subclasses access? (2/2)

```
public class Car {
    private Engine _engine;
    //other variables elided
    public Car(){
        _engine = new Engine();
    }
    public void turnOnEngine() {
        _engine.start();
    }
    public void turnOffEngine() {
        _engine.shutOff();
    }
    public void drive() {
        //code elided
    }
    protected void cleanEngine() { ... }
}
```

```
public class Convertible extends
Car {
    //constructor elided
    public void cleanCar() {
        _engine.steamClean();
    }
}
```

```
public class Convertible extends Car {
    //constructor elided
    public void cleanCar() {
        this.cleanEngine();
    }
}
```

This makes use of *the parent's* inherited `cleanEngine` method, hence our use of **`this`**

- Will `Convertible` have access to `_engine`?
- Subclasses **cannot directly inherit private** variables / methods from parent
  - But you can can use methods defined in your parent, which have access to the variable

Andries van Dam © 2016 9/27/16

Question

Which of the following is a superclass/parent of the rest?

A. Lions
B. Tigers
C. Cats
D. Leopards

Andries van Dam © 2016 9/27/16

Question

All of the following are appropriate ways to model superclasses and subclasses EXCEPT:

A.

B.

Andries van Dam © 2016 9/27/16

Let’s examine inheritance further

1. Model inheritance relationship
2. Adding new methods
3. Overriding methods

Andries van Dam © 2016 9/27/16

Overriding methods (1/3)

- A `Convertible` may decide `Car`'s `drive()` method just doesn't cut it
  - A `Convertible` drives much faster than a regular car
- Can **override** a parent class's method and redefine it

```
public class Car {

    private Engine _engine;
    //other variables elided

    public Car() {
        _engine = new Engine();
    }
    public void drive() {
        this.goFortyMPH();
    }
    public void goFortyMPH() {
        //code elided
    }
    //more methods elided
}
```


Andries van Dam © 2016 9/27/16

Overriding methods (2/3)

- `@Override` is an annotation--signals to compiler (and to anyone reading your code) that you're overriding a method of the superclass
  - We include `@Override` right before we declare method we mean to override

```
public class Convertible extends Car {

    public Convertible() {

    }

    @Override
    public void drive(){
        this.goSixtyMPH();
    }

    public void goSixtyMPH(){
        //code elided
    }
}
```


Andries van Dam © 2016 9/27/16

Overriding methods (3/3)

- Here's where we re-declare method we want to override
  - Be careful – method signature must match that of the superclass's method exactly else Java will create a new additional method instead of overriding !
- `drive()` is the ***method signature***, indicating that name of method is drive and it takes in no parameters
  - When a `Convertible` is told to drive, it will execute this code instead of the code in its superclass's drive method

```
public class Convertible extends Car {

    public Convertible() {

    }

    @Override
    public void drive(){
        this.goSixtyMPH();
    }

    public void goSixtyMPH(){
        //code elided
    }
}
```


Andries van Dam © 2016 9/27/16

Partially overriding methods

- Keyword `super` used to invoke original inherited method from parent: in this case, drive as implemented in parent `Car`
- While you can use `super` to call other methods in the parent class, it's strongly discouraged
  - Use the **`this`** keyword instead
  - *Except* when you are calling the parent's method within the child's method of the same name
    - This is **partial overriding**
    - What would happen if we said `this.drive()` instead of `super.drive()`?

```
public class Sedan extends Car {

    public Sedan () {
     //code elided
    }

    @Override
    public void drive(){
        this.turnOnEngine();
        super.drive(); // super == parent
                class
        this.addPinToMap();
        super.drive();
        super.drive();
        this.addPinToMap();
    }

}
```

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!

Andries van Dam © 2016 9/27/16

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

Andries van Dam © 2016 9/27/16

Inheritance and Polymorphism (1/3)

- Let's borrow the Racer class from the example we discussed in lecture on interfaces
- However, we change the parameter type in method `useTransportation()` from `Transporter` to `Car`
- What would happen?
  - **We can only pass in `Car` and subclasses of `Car`**

```
public class Racer {

    //previous code elided

    public void useTransportation(Car myCar) {
        //code elided
    }

}
```


Andries van Dam © 2016 9/27/16

Inheritance and Polymorphism (2/3)

- Let's define `useTransportation()`
- What method should we call on `myCar`?
  - Every `Car` knows how to drive, which means we can guarantee that every subclass of `Car` also knows how to drive

```
public class Racer {

    //previous code elided

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

}
```


Andries van Dam © 2016 9/27/16

Is this legal?

```
Car convertible = new Convertible();
_sophia.useTransportation(convertible);
```

✓

```
Car sedan = new Sedan();
_sophia.useTransportation(sedan);
```

✓

```
Car bike = new Bike();
_sophia.useTransportation(bike);
```

✗

`Bike` is not a subclass of `Car`, so you cannot treat an instance of `Bike` as a `Car`.

Andries van Dam © 2016 9/27/16

Inheritance and Polymorphism (3/3)

- That's all we needed to do!
- Our inheritance structure looks really similar to our interfaces structure
  - Therefore, we only need to change 2 lines in `Racer` in order to use any of our new cars!
  - But remember: what's happening behind the curtain is very different: method resolution "climbs up the hierarchy" for inheritance
- Polymorphism is an incredibly powerful tool
  - Allows for generic programming
  - Treat multiple classes as their generic type while still allowing specific method implementations to be executed
- Polymorphism+Inheritance is strong generic coding

Andries van Dam © 2016 9/27/16

Question

In the following code, the `Elephant` subclass extends the `Animal` superclass, both of which contain and define an `eat()` method:

```
Animal horton = new Elephant();
horton.eat();
```

Whose eat method is being called?

A. Animal
B. Elephant
C. Sedan
D. None of the above

Andries van Dam © 2016 9/27/16

Next Lecture

- Inheritance and polymorphism (continued)
- Immutable classes
- Abstract classes

Andries van Dam © 2016 9/27/16

CSE201: Monsoon 2023
Advanced Programming

Lecture 07: Abstract Class and Immutable Class

Raghava Mutharaju (Section-A)
CSE, IIIT-Delhi
raghava.mutharaju@iiitd.ac.in

Last Lecture: Inheritance & Polymorphism

Method resolution

30

```
public class Car {

    private Engine _engine;
    //other variables elided

    public Car() {
        _engine = new Engine();
    }
    public void drive() {
        this.goFortyMPH();
    }
    public void goFortyMPH() {
        //code elided
    }
    protected void cleanEngine()
{ ... }
}
```

```
public class Convertible extends Car {

    public Convertible(){

    }

    public void putTopDown(){
        //code elided
    }
}
```

18

```
public class Convertible extends Car {
    //constructor elided
    public void cleanCar() {
        _engine.steamClean();
    }
}
```

```
public class Sedan extends Car {

    public Sedan () {
        //code elided
    }

    @Override
    public void drive(){
        this.turnOnEngine();
        super.drive(); // super == parent class

        this.addPinToMap();
        super.drive();
        super.drive();
        this.addPinToMap();
    }
}
```

```
public class Racer {

    //previous code elided

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

}
```

- Adding new methods
- Accessing superclass fields/methods
- Overriding superclass methods
- Polymorphism
- Method resolution

This Lecture

- Inheritance and Polymorphism (continued from last lecture)
- Abstract class and abstract methods
- Immutable classes

Slide acknowledgements: CS15, Brown University

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?

Andries van Dam © 2016 9/22/16

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);
   }
}
```


Andries van Dam © 2016 9/22/16

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()`

Andries van Dam © 2016 9/22/16

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;
   }
}
```


Andries van Dam © 2016 9/22/16

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

Andries van Dam © 2016 9/22/16

`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 instantiate `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);
   }
}
```


Andries van Dam © 2016 9/22/16

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`

Andries van Dam © 2016 9/22/16

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();
    }

    ......
}
```


Andries van Dam © 2016 9/22/16

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

Andries van Dam © 2016 9/22/16

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.

Andries van Dam © 2016 9/22/16

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 { …. }

© Vivek Kumar

`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 `Car`s 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

Andries van Dam © 2016 9/22/16

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();

}
```


Andries van Dam © 2016 9/22/16

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

Andries van Dam © 2016 9/22/16

`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

Andries van Dam © 2016 9/22/16

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 only instance variables/concrete methods
  - You can implement multiple interfaces

Note: Java, like most programming languages, is evolving. In Java 8, interfaces and `abstract` classes are even closer in that you can have concrete methods in interfaces. We will not make use of this in CSE201.

Andries van Dam © 2016 9/22/16

What if the Cars are Getting Modified?

No modifications should ever be allowed !!

© Vivek Kumar

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();
}
```

© Vivek Kumar

Question

- Immutable classes have their fields marked as *final*. Then, why can’t we make those fields as *public* and let clients access them without any getter methods ?

© Vivek Kumar

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);
}
```

© Vivek Kumar

Immutable Classes (3/5)

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

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

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

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

© Vivek Kumar

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?

© Vivek Kumar

Immutable Classes (5/5)

```
public final class 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
}
```

© Vivek Kumar

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*.

© Vivek Kumar

Next Lecture

- Class Object

CSE201: Monsoon 2023
Advanced Programming

Lecture 08: The Object Class

Raghava Mutharaju (Section-A)
CSE, IIIT-Delhi
raghava.mutharaju@iiitd.ac.in

This Lecture

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

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..

© Vivek Kumar

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 int numLegs;

    public Furniture(){ ...
}
    .......
}
```

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

© Vivek Kumar

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!

© Vivek Kumar

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`

© Vivek Kumar

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.

© Vivek Kumar

Object Methods

method	description
`protected Object clone()`	creates a copy of the object
`public boolean equals(Object o)`	returns whether or not two objects are same
`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>();
```


© Vivek Kumar

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

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

- Wrong Implementation !
  - **Flaw-1**
    - Signature of `equals` method doesn't matches with that in class `Object`
    - Compilation error as we are not overriding!
  - The parameter to `equals` method is not of type `Object` but is of type `Point`
    - This is method overloading and not overriding

© Vivek Kumar

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 o) {
6.       return (x==o.x && y==o.y);
7.    }
8. }
9.
```

- Still incorrect !
  - **Flaw-2**
    - Compilation error as the parameter to `equals` is of `Object` type but then x and y is not defined in class `Object`
  - Can we can do the following:

```
Object o1 = new Point(1, 2);
// Type casting below
Point p = (Point) o1;
```


© Vivek Kumar

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!!
```


© Vivek Kumar

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
```

© Vivek Kumar

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 !!

© Vivek Kumar

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`

© Vivek Kumar

Comparing Objects

© Vivek Kumar

Comparing Objects in Java

`.equals(  ) = true`

`.equals(  ) = 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

© Vivek Kumar

Comparing Objects in Java

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

© Vivek Kumar

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);
}
```


© Vivek Kumar

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!

© Vivek Kumar

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 `Rectangle`s with same `area` could still have different values for `sideA` and `sideB`

© Vivek Kumar

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
  - ……

© Vivek Kumar

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

© Vivek Kumar

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);

}

}
```

© Vivek Kumar

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());
```


© Vivek Kumar

Copying Objects

© Vivek Kumar

Copying objects

- In other languages (common in C++), to enable clients to easily make copies of an object, you can supply a *copy constructor* :

```
// in client code
Point p1 = new Point(-3, 5);
Point p2 = new Point(p1);     // make p2 a copy of p1


// in Point.java
public Point(Point blueprint) {   // copy constructor
    this.x = blueprint.x;
    this.y = blueprint.y;
}
```

  - Java has some copy constructors but also has a different way...

Object clone method

```
protected Object clone()
        throws CloneNotSupportedException
```

- Creates and returns a copy of this object. General intent:
  - `x.clone() != x`
  - `x.clone().equals(x)`
  - `x.clone().getClass() == x.getClass()`
    - (though none of the above are absolute requirements)

clone() must be Implemented

- If we want to clone **Point** type objects, **Point** class must implement **clone**() method

```
Point.java:11: error: incompatible types: Object cannot be converted to Point
        Point p2 = p.clone();
                            ^
1 error
```

- You must also make your class implement the `Cloneable` interface to signify that it is allowed to be cloned

The Cloneable interface

```
public interface Cloneable {}
```

- Why would there ever be an interface with no methods?
  - Another example: `Set` interface, a sub-interface of `Collection`

- **Tagging/marker interface**: One that does not contain/add any methods, but is meant to mark a class as having a certain quality or ability.
  - Generally a wart in the Java language; a misuse of interfaces.
  - Now largely unnecessary thanks to *annotations* (seen later).
  - But we still must interact with a few tagging interfaces, like this one.

- Let's implement clone for a `Point` class…

What's Wrong with the Below Method?

```
public class Point implements Cloneable {
    private int x, y;
    ...
    public Point clone() {
        Point copy = new Point(this.x, this.y);
        return copy;
    }
}
```

The flaw

```
// also implements Cloneable and inherits clone()
public class Point3D extends Point {
    private int z;
    ...
}
```

- The above `Point3D` class's `clone` method produces a `Point`!
  - This is undesirable and unexpected behavior.
  - The only way to ensure that the clone will have exactly the same type as the original object (even in the presence of inheritance) is to call the `clone` method from class `Object` with `super.clone()`.

Proper clone method (1/2)

```
public class Point implements Cloneable {
    private int x, y;
    ...
    public Point clone() {
        try {
            Point copy = (Point) super.clone();
            return copy;
        } catch (CloneNotSupportedException e) {
            // this will never happen
            return null;
        }
    }
}
```

- To call `Object`'s `clone` method, you must use `try/catch`.
  - But if you implement `Cloneable`, the exception will not be thrown.

Proper clone method (2/2)

```
public class Point implements Cloneable {
    private int x, y;
    ...
    public Point clone() {
        try {
            Point copy = (Point) super.clone();
            return copy;
        } catch (CloneNotSupportedException e) {
            // this will never happen
            return null;
        }
    }
}
```

```
public class Point3D extends Point {
    int z;
    ...
    public Point3D clone() {
        Point3D p = (Point3D)
super.clone();
        p.z = this.z;
        return p;
    }
}
```

- Every subclass should re-implement clone and must call super.clone() internally
- Only the topmost class in parent-child hierarchy should call super.clone() inside try/catch exception handling block

super.clone()

super.clone() does the following.

- Checks whether the class has implemented the Cloneable interface. Throws a CloneNotSupportedException otherwise.
- Creates a new instance of the class.
- Performs a “shallow copy” of the fields in the class.

- https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/lang/Object.html#clone()

What's Still Wrong with the Below Method?

```
public class BankAccount implements Cloneable {
    private String name;
    private List<String> transactions;
    ...
    public BankAccount clone() {
        try {
            BankAccount copy = (BankAccount) super.clone();
            return copy;
        } catch (CloneNotSupportedException e) {
            return null;   // won't ever happen
        }
    }
}
```

Shallow vs. deep copy

- **shallow copy**: Duplicates an object without duplicating any other objects to which it refers.

- **deep copy**: Duplicates an object's entire *reference graph*: copies itself and deep copies any other objects to which it refers.

Proper clone method 2

```
public class BankAccount implements Cloneable {
   private String name;
   private List<String> transactions;
   ...
   public BankAccount clone() {
      try {                // deep copy
         BankAccount copy = (BankAccount) super.clone();
         copy.transactions = new ArrayList<String>(transactions);
         return copy;
      } catch (CloneNotSupportedException e) {
         return null;   // won't ever happen
      }
   }
}
```

- Copying the list of transactions (and any other modifiable reference fields) produces a deep copy that is independent of the original

Next Lecture

- Generic programming

CSE201: Monsoon 2023
Advanced Programming

Lecture 09: Generic Programming

Raghava Mutharaju (Section-A)
CSE, IIIT-Delhi
raghava.mutharaju@iiitd.ac.in

Last Lecture

- Class Object
  - Correct implementation of equals method
  - Comparing objects
    - Comparable interface
    - Comparator interface
  - Copying objects

```
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;
    }
}
```

```
public class BankAccount implements Cloneable {
   private String name;
   private List<String> transactions;
   ...
   public BankAccount clone() {
      try {                // deep copy
         BankAccount copy = (BankAccount) super.clone();
         copy.transactions = new
ArrayList<String>(transactions);
         return copy;
      } catch (CloneNotSupportedException e) {
         // this will never happen
         return null;
      }
   }
}
```

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

```
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 _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.  }
```

© Vivek Kumar

Today’s Lecture

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

© Vivek Kumar

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. ?

© Vivek Kumar

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

© Vivek Kumar

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?

© Vivek Kumar

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(Sting 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 !!

© Vivek Kumar

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

© Vivek Kumar

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)

© Vivek Kumar

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(Sting args[]) {
        MyGenericList<String> strList = new
MyGenericList<String>();
        MyGenericList<Integer> intList = new
MyGenericList<Integer>();

        strList.add("hello");
        intList.add(1);
        ...
    }
```

© Vivek Kumar

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

© Vivek Kumar

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(Sting 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?
  - MyGenericList class instantiated without specifying the type of its two fields

© Vivek Kumar

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(Sting 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 MyGenericList class on both RHS and LHS of statement

© Vivek Kumar

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; }
}
```

```
public class Main {
    public static void main(Sting 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));
        ...
    }
}
```

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

© Vivek Kumar

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

© Vivek Kumar

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(Sting 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(Sting 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

© Vivek Kumar

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

© Vivek Kumar

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

© Vivek Kumar

Restrictions (3/6)

```
public class MyGenericClass<T> {
    ......

    static <T> void doSomething(List<T> list)
{
        if(list instanceof
ArrayList<Integer>) {

        }
    }

    public static void main(String[] arg){
        .....
    }
}
```

- 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

© Vivek Kumar

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

© Vivek Kumar

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>

© Vivek Kumar

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

© Vivek Kumar

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<Object> intList[] = new ArrayList<Integer>[5];
List<String> stringList = new ArrayList<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

© Vivek Kumar

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?

© Vivek Kumar

The WildCard to our Rescue !

© Vivek Kumar

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

© Vivek Kumar

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

© Vivek Kumar

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

© Vivek Kumar

Next Lecture

- Exception Handling

CSE201: Monsoon 2023
Advanced Programming

Lecture 10: Exception Handling

Raghava Mutharaju (Section-A)
CSE, IIIT-Delhi
raghava.mutharaju@iiitd.ac.in

© Vivek Kumar

Last Lecture

- Generic programming in Java
  - Using generic programming we don't have to implement different classes for different object types
    - Programmer friendly code!
  - Generics are implemented using type erasures
    - Compiler erases all parameter type information
  - Restrictions
    - Type parameters cannot be instantiated with primate types
    - Instantiating type variables is not allowed
    - Generic array creation is not allowed
    - Type variables are not valid as static field of a generic class
    - Generic does not supports sub typing

```
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(Sting 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));
        ...
    }
}
```

© Vivek Kumar

Today’s Lecture: Exceptions

Being Defensive is Important

© Vivek Kumar

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!

© Vivek Kumar

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!

© Vivek Kumar

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!**

Andries van Dam © 2016 9/22/16

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*

Andries van Dam © 2016 9/22/16

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?

© Vivek Kumar

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
point          done = true;
      }
      catch(InputMismatchException inp) {
          System.out.println("Wrong input:");
          System.out.println("Try again");
      }
      finally {
          System.out.println("Always execute");
            }
        }
    }
}
```

- 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

© Vivek Kumar

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;
```

```
        }
```

© Vivek Kumar

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 ArrayIndexOutOfBounds exception

© Vivek Kumar

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'`

© Vivek Kumar

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`

© Vivek Kumar

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

© Vivek Kumar

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 some where in the program, otherwise program will terminate abruptly

© Vivek Kumar

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 caller method and repeat step 1 by looking into caller'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)

© Vivek Kumar

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., `RuntimeExceptions` 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

© Vivek Kumar

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

© Vivek Kumar

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

© Vivek Kumar

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

© Vivek Kumar

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`

© Vivek Kumar

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.
2. 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)
 ......
```


© Vivek Kumar

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

© Vivek Kumar

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

© Vivek Kumar

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

© Vivek Kumar

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

© Vivek Kumar

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

© Vivek Kumar

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`

© Vivek Kumar

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)
```


© Vivek Kumar

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

© Vivek Kumar

CSE201: Monsoon 2023
Advanced Programming

Lecture 14: Collection Framework

Raghava Mutharaju

CSE, IIIT-Delhi

raghava.mutharaju@iiitd.ac.in

© Vivek Kumar

Today’s Lecture

- Assertions (Defensive Programming)
- Collections framework in Java

© Vivek Kumar

Assertions

- **assertion**: A statement that is either true or false

  Examples:
  - Java was created in 1995.
  - The sky is purple.
  - 23 is a prime number.
  - 10 is greater than 20.
  - x divided by 2 equals 7. *(depends on the value of x)*

- An assertion might be false ("The sky is purple" above), but it is still an assertion because it is a true/false statement

Copyright 2008 by Pearson Education

Declaring Assertions

An *assertion* is declared using the Java keyword assert as follows:

assert *assertion*; or
assert *assertion* : *detailMessage*;

where **assertion** is a Boolean expression and ***detailMessage*** is a primitive-type or an Object value

Executing Assertion (1/3)

```
public class AssertionDemo {
  public static void main(String[] args) {
    int i; int sum = 0;
    for (i = 0; i < 10; i++) {
      sum += i;
    }
    assert i == 10;
    assert sum > 10 && sum < 5 * 10 : "sum is " +
sum;
  }
}
```

- When an assertion statement is executed, Java evaluates the assertion. If it is false, an `AssertionError` will be thrown

Liang, Introduction to Java Programming, Tenth Edition, (c) 2015 Pearson Education, Inc. All rights reserved

Executing Assertion (2/3)

```
public class AssertionDemo {
  public static void main(String[] args) {
    int i; int sum = 0;
    for (i = 0; i < 10; i++) {
      sum += i;
    }
    assert i == 10;
    assert sum > 10 && sum < 5 * 10 : "sum is " +
sum;
  }
}
```

- By default, the assertions are disabled at runtime as they are costly
  - Constant check of the condition inside assert statement
- To enable use the following command line switch

  `java –ea AssertionDemo`

  OR

  `java –enableassertions AssertionDemo`

Liang, Introduction to Java Programming, Tenth Edition, (c) 2015 Pearson Education, Inc. All rights reserved

Executing Assertion (3/3)

```
public class AssertionDemo {
  public static void main(String[] args) {
    int i; int sum = 0;
    for (i = 0; i < 10; i++) {
      sum += i;
    }
    // deliberately changed to generate assertion
failure
    assert i != 10;
    assert sum > 10 && sum < 5 * 10 : "sum is " +
sum;
  }
}
```

- Let's try to generate the assertion failure in this program
  - Change "==" to "!="
  - Output:

```
Exception in thread "main" java.lang.AssertionError
at AssertionDemo.main(AssertionDemo.java:7)
```

- `AssertionError` extends `Error` and you cannot write a `try/catch` block to catch this. The program will definitely terminate with the complete stack dump

Liang, Introduction to Java Programming, Tenth Edition, (c) 2015 Pearson Education, Inc. All rights reserved

Assertions or Exception Handling?

- Assertion should not be used to replace exception handling
  - Exception handling deals with unusual circumstances whereas assertions are to assure the correctness of the program
  - Exception handling addresses robustness and assertion addresses correctness
- Similar to exceptions, assertions are also checked at runtime but unlike exceptions it can be turned on or off (for entire execution)
- Use assertions to reaffirm assumptions to assure correctness of the program

Liang, Introduction to Java Programming, Tenth Edition, (c) 2015 Pearson Education, Inc. All rights reserved

Let’s change gears…

Collection Framework

© Vivek Kumar

Note

- Remaining slides will use all the concepts that you have learned so far in this course

© Vivek Kumar

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

© Vivek Kumar

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`".

© Vivek Kumar

Collection Hierarchy

© Vivek Kumar

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
    }
}
```


© Vivek Kumar

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
    }
}
```

© Vivek Kumar

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

© Vivek Kumar

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

© Vivek Kumar

Iterator Position

Figure 2–3: Advancing an iterator

© Vivek Kumar

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

© Vivek Kumar

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());
  }
}
```


© Vivek Kumar

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`**

© Vivek Kumar

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();`

© Vivek Kumar

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

© Vivek Kumar

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

© Vivek Kumar

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

© Vivek Kumar

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

© Vivek Kumar

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

© Vivek Kumar

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

© Vivek Kumar

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

© Vivek Kumar

LinkedList Methods

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

© Vivek Kumar

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);
        }
    }
}
```


© Vivek Kumar

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 (In Ascending Order)

© Vivek Kumar

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

© Vivek Kumar

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

© Vivek Kumar

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

In Java, a `TreeSet` is a collection class that implements the `Set` interface and is part of the Java Collections Framework. It is a NavigableSet that uses a Red-Black tree as its underlying data structure for storage. The primary characteristics of a `TreeSet` are as follows:

1. **Sorted Order:** A `TreeSet` maintains elements in sorted (ascending) order based on their natural ordering or a custom comparator provided during creation.
2. **No Duplicates:** It does not allow duplicate elements. If you attempt to add an element that is already present, the operation will have no effect.
3. **Fast Retrieval:** Searching for elements, including insertion and deletion, is very efficient (O(log n)) because of the underlying Red-Black tree structure.
4. **Null Elements:** A `TreeSet` does not allow null elements. If you try to insert a null element, it will throw a `NullPointerException`.

Example: TreeSet

```
import java.util.*;
public class Book implement 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

© Vivek Kumar

Red-Black trees all properties

A Red-Black Tree is a type of self-balancing binary search tree. It has several properties that ensure it remains balanced and efficient for common operations. Here are the key properties of a Red-Black Tree:

1. **Binary Search Tree Property:** A Red-Black Tree is a binary search tree where each node has at most two children. Nodes to the left are smaller, and nodes to the right are greater than their parent node.
2. **Color Property:** Each node in the tree is colored either red or black. This color property ensures that no two adjacent red nodes are allowed on any path in the tree.
3. **Root Property:** The root node is black. This is a requirement to maintain the black height property.
4. **Red Property:** Red nodes cannot have red children. In other words, there are no two consecutive red nodes along any path in the tree. This property helps in balancing the tree during insertions and deletions.
5. **Black Height Property:** The number of black nodes on any path from a node to a leaf (excluding the node itself) must be the same for all paths in the tree. This property ensures that the tree remains balanced.
6. **Leaf Nodes:** Leaf nodes (null or NIL nodes) are considered black.
7. **Balancing Operations:** When a Red-Black Tree is modified by insertions or deletions, it may require balancing operations (recoloring and rotations) to maintain the Red-Black properties. These operations ensure that the tree remains approximately balanced, with a height of O(log n), where n is the number of nodes in the tree.
8. **Ordered Structure:** Due to the binary search tree property, Red-Black Trees can efficiently support search, insert, and delete operations in O(log n) time on average.

Red-Black Trees are commonly used in various applications where self-balancing binary search trees are needed, such as in many programming language libraries for implementing data structures like sets and maps. They provide a good balance between efficient operations and maintaining a relatively balanced tree structure.

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

© Vivek Kumar

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

© Vivek Kumar

Example: HashMap

```
import java.util.HashMap;
import java.util.Map;

public class Book {
    private String name;
    private int pages;

    public Book(int p, String s) {
      pages = p;
      name = s;
    }

    @Override
    public String toString() {
      return "Book name: " + name + "; pages = " + pages;
    }

    public static void main(String[] args) {
        Map<Integer, Book> map = new HashMap<Integer, Book>();

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


© Vivek Kumar

Next Lecture

- I/O Streams

© Vivek Kumar

CSE201: Monsoon 2023
Advanced Programming

Lecture 15: I/O Streams

Raghava Mutharaju (Section-A)
CSE, IIIT-Delhi
raghava.mutharaju@iiitd.ac.in

Today’s Lecture

- I/O Streams
- Object serialization and deserialization

Acknowledgements: Oracle Java doc + javatpoint.com

I/O Streams

- Stream is a sequence of data
  - Flows in/out the program to/from an external source such as file, network, console, etc.
- Similar to a stream of flowing water…
- Program uses **input stream** to read data from a source, one at a time
- Program uses **output stream** to write data to a destination, one at a time

© Vivek Kumar

Streams v/s File Handling

- Stream is a continuous flow of data
  - Streams don’t allow you to move back and forth unlike File
- Streams allows you handle the data the same way irrespective of the location of data (e.g., hard disk, network etc.)
  - You can have the same code to “stream” the data from a file and from the network!

© Vivek Kumar

Types of Streams

- Two types of streams
  - Byte stream
  - Character stream
- Byte stream
  - Operates upon stream of "byte" (8-bit)
- Character stream
  - Operates upon stream of "character" Unicode (16-bit)
  - *Unicode* is a computing industry standard designed to consistently and uniquely encode characters used in written languages throughout the world
  - The Unicode standard uses hexadecimal to express a character
    - JVM is platform independent!

`java.io` Package

- Reading

  open a stream

  while more information

  read information

  close the stream

- Writing

  open a stream

  while more information

  write information

  close the stream

Byte Stream Hierarchy

- OutputStream
  - This is the abstract class
  - Parent class of all classes representing an output stream of bytes
  - An output stream accepts output bytes and sends them to some sink
- InputStream
  - This is the abstract class
  - Parent class of all classes representing an input stream of bytes

© Vivek Kumar

Byte Streams in System Class

```
public final class System {

    public static final InputStream in;

    public static final PrintStream out;

    public static final PrintStream err;

    ...........

}
```

```
public static void main(String args[]) {
    Scanner in = new Scanner(System.in); //java.lang
    // Scanner class implements iterator
    while (in.hasNext()) {
        System.out.println(in.next());
    }
    in.close();
}
```

- In java, 3 streams are created for us automatically. All these streams are attached with console
  - **System.out:** standard output stream
  - **System.in:** standard input stream
  - **System.err:** standard error stream

© Vivek Kumar

Byte Stream Example

```
public static void main(String args[])
            throws IOException
{
    FileInputStream in = null;
    FileOutputStream out = null;
    try {
        // both constr. throws FileNotFoundException
        in = new FileInputStream("input.txt");
        out = new FileOutputStream("output.txt");
        int c;
        while ((c = in.read()) != -1) { //
IOException
            out.write(c);        // IOException
        }
    } finally {
        if (in != null)
            in.close();          // IOException
        if (out != null)
            out.close();         // IOException
        }
}
```

- InputStream
  - **`read()`** – read the next byte of data from the input stream
  - **`close()`** – close input stream
- OutputStream
  - **`write(int)`** – write a byte to current output stream
  - **`close()`** – close output stream
- Byte stream is used for low-level I/O, e.g., processing binary files

© Vivek Kumar

Character Stream Hierarchy

- All character stream classes are subclasses of `Reader` and `Writer` class
- Used for processing text files (character by character)

Character Stream Example

```
public static void main(String args[])
            throws IOException
{
    FileReader in = null;
    FileWriter out = null;
    try {
        // both constr. throws FileNotFoundException
        in = new FileReader("input.txt");
        // throws IOException
        out = new FileWriter("output.txt");
        int c;
        while ((c = in.read()) != -1) { //
IOException
            out.write(c);        // IOException
        }
    }finally {
        if (in != null)
            in.close();                // IOException
        if (out != null)
            out.close();         // IOException
        }
}
```

- This example is very similar to the byte stream I/O
- In terms of coding, the difference is in using FileReader and FileWriter for input and output
- Note that “int” type variable is used in both these examples to read and write. Although internally they are working differently:
  - In byte stream example, the “int” variable holds a byte value in last 8 bits
  - In this example, the “int” variable holds character value in its last 16 bits

© Vivek Kumar

Buffered Streams (1/2)

```
public static void main(String args[])
            throws IOException
{
    BufferedReader in = null;
    PrintWriter out = null;
    try {
        in = new BufferedReader( new
                FileReader("input.txt"));
        out = new PrintWriter( new
                FileWriter("output.txt"));
        String l;
        while ((l = in.readLine()) != null){
//IOException
            out.println(l); // does not throw IOException
        }
    }finally {
        if (in != null)
            in.close();                // IOException
        if (out != null)
            out.close();         // IOException
        }
}
```

- Combine streams into chains to achieve more advanced input and output operations
- Reading character by character from a file is slow
- Faster to read a larger block of data from the disk and then iterate through that block byte by byte afterwards
- The code on the left does input and output one line at a time
  - Unlike **BufferedWriter**, **PrintWriter** swallows exceptions and provide methods such as println(), etc.

© Vivek Kumar

Buffered Streams (2/2)

```
public static void main(String args[])
          throws IOException
{
    Scanner in = null;
    PrintWriter out = null;
    try {
        in = new Scanner( new BufferedReader( new
               FileReader("input.txt")));
        out = new PrintWriter( new
               FileWriter("output.txt"));
        while (in.hasNext()) {
            out.println(in.next());
        }
    }finally {
        if (in != null)
            in.close();
        if (out != null)
            out.close();
        }
}
```

- Here we are combining three classes for breaking input into tokens:
  - Scanner
  - BufferedReader
  - FileReader
- BufferedReader will read one line at a time and Scanner will be able to parse this line by white space separated tokens

© Vivek Kumar

Serialization and Deserialization

- Serialization in Java is a mechanism of writing the state of an **object** into a **byte stream**
  - **Note:** it's the object state that is recorded but not the actual class definition ("class file")
- The reverse operation is called deserialization
- Some usage
  - Storing live objects in a file
  - Hibernating applications
  - Moving object state over the network (marshaling)

© Vivek Kumar

`java.io.Serializable` Interface

```
package java.io;
public interface Serializable { /*empty*/ }
```

- Must be implemented by the class to be serialized
- This is a tag/marker interface similar to Cloneable interface
  - Hint to JVM !

© Vivek Kumar

Example: Serializing and Deserializing

```
1. import java.io.*;
2.
3. class Manager implements Serializable {
4.     private String name;
5.     public Manager(String n) { ..... }
6. }

7. public class Client {
8.    public static void serialize()
9.                          throws IOException {
10.        Manager s1 = new Manager("Amy");
11.        ObjectOutputStream out = null;
12.        try {
13.           out = new ObjectOutputStream (
14.                 new
FileOutputStream("out.txt"));
15.           out.writeObject(s1);
16.        } finally {
17.           out.close();
18.        }
19.    }
20.
/* Continued on RHS window */
```

```
    /* Continued from LHS window */
21.    public static void deserialize()
22.       throws IOException, ClassNotFoundException {
23.        ObjectInputStream in = null;
24.        try {
25.            in =  new ObjectInputStream (
26.                   new FileInputStream("out.txt"));
27.            Manager s1 = (Manager) in.readObject();

29.        } finally {
30.            in.close();
31.        }
32.    }
33.
34.    public static void main(String[] args)
35.        throws IOException,ClassNotFoundException {
36.
37.       serialize();
39.       deserialize();
40.    }
41. } /* End of Main class */
```

Suppose you have a Client.java that only has the above deserializes() method. Compilation of Client.java will generate two class files Client.class and Manager.class. If you try running "java Client" without Manager.class in its classpath then ClassNotFoundException will be thrown at Line 27 above.

© Vivek Kumar

Rules for Serializing (1/3)

```
import java.io.*;
class Address {
    private String city;
    public Address(String c) { ...... }
}
class Manager implements Serializable {
     private String name;
     private Address addr;
     public Manager(String n, String city) { ..... }
}
public class Main {
  public static void serialize() throws IOException {
        Manager s1 = new Manager("Amy", “Delhi”);
        ObjectOutputStream out = null;
        try {
           out = new ObjectOutputStream (
                 new FileOutputStream("out.txt"));
           out.writeObject(s1);
        } finally {
           out.close();
        }
    }
    ......
}
```

- This program compiles fine but will generate `NotSerializableException`
- All fields of Manager class should either be primitive type or serializable objects
  - Address class should also implement Serializable interface

© Vivek Kumar

Rules for Serializing (2/3)

```
import java.io.*;
class Employee {
    private String address;
    public Employee(String a) { ..... }
}
class Manager extends Employee
                        implements Serializable {
    private String name;
    public Manager(String n, String city) { ..... }
}
public class Main {
  public static void serialize() throws IOException
{
        Manager s1 = new Manager("Amy", “Delhi”);
        ObjectOutputStream out = null;
        try {
           out = new ObjectOutputStream (
                 new FileOutputStream("out.txt"));
           out.writeObject(s1);
        } finally {
           out.close();
        }
    }
    ......
}
```

- This program compiles fine but will generate **`InvalidClassException`** while typecasting the object to Manager type during deserialization
- Two ways to fix this issue:
  1. Provide a default constructor in Employee
  2. Or, implement Serializable in Employee class (superclass)
     - This is obviously the safer and easier choice
     - With this change, its not required in the Manager (or any subclass) to mention implements Serializable

© Vivek Kumar

Rules for Serializing (3/3)

- There is no point in serializing static field members in a class
  - Static fields do not represents object state but they represent class state
    - Recall, static variables are accessed using class name and not with objects of the class
  - There will not be any compilation/runtime issue, although the value serialized will not make any sense as it can always be updated later in the class
- `transient` keyword in Java
  - If you don't want any field to be serialized then mark that as "`transient`"

© Vivek Kumar

`serialVersionUID` in Serialization (1/2)

- What happens when we compile the programs in previous slides by enabling warning?

$ javac –Xlint Main.java

Main.java:3: warning: [serial] serializable class Manager has no definition of serialVersionUID

© Vivek Kumar

`serialVersionUID` in Serialization (2/2)

- JVM generates a unique `serialVersionUID` to each class implementing Serializable interface

```
ANY-ACCESS-MODIFIER static final long serialVersionUID = 42L;
```

- Its always advisable to declare this variable in each serializable class with your own number of choice
- Helpful in verifying if the object being deserialized is of the same type of the specified class
  - The class declaration might have got updated (e.g. added new fields) after serialization and now deserializing the object will generate `InvalidClassException`

© Vivek Kumar

Where are we as of now

- **CSE201 Post Conditions**
  1. Students are able to demonstrate the knowledge of basic principles of Object Oriented Programming such as encapsulation (classes and objects), interfaces, polymorphism and inheritance; by implementing programs ranging over few hundreds lines of code
  2. Implement basic event driven programming, exception handling, and threading
     - Already covered little bit of event driven programming in refresher module (Day 3) but we will see more
  3. Students are able to analyze the problem in terms of use cases and create object oriented design for it. Students are able to present the design in UML
     - Already covered little bit of UML but we will see more
  4. Students are able to select and use a few key design pattern to solve a given problem in hand
  5. Students are able to use common tools for testing (e.g., JUnit), debugging, and source code control as an integral part of program development
     - Will turn green by end of this week

Next Lecture

- Unit testing using JUnit
- Inner classes

Advanced Programming

Lecture 16: Inner Class & JUnit

Phone Batteries are Becoming Non-Removable

Non Removable Battery Vs. Removable Battery

- Nowadays most of the phones are coming with non removable batteries
- Earlier, when there were removable batteries, we could easily keep a spare battery and replace when the primary one drained!
  - *Yes, I know we can use a power bank today!*

© Vivek Kumar

Inner Class

```
public class SamsungGalaxy {

    private FixedBattery myBattery;

    public SamsungGalaxy() {
        myBattery = new FixedBattery();
    }


    private class FixedBattery {
        ....
    }
}
```


© Vivek Kumar

Inner Class

- Description
  - Class defined in scope of another class
- Property
  - Can directly access all variables & methods of enclosing class (including private fields & methods)
- Why inner class?
  - Logical grouping of functionality
  - Increases encapsulation
    - Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world
  - More readability and maintainable code

© Vivek Kumar

Inner Class Instance Inside a Method of Outer Class

```
public class SamsungGalaxy {

    private FixedBattery myBattery;

    public SamsungGalaxy() {
        myBattery = new FixedBattery();
    }


    private class FixedBattery {
        private boolean runDiagnosis() { ..... }
        ....
    }

    public static void main(String[] args) {
        SamsungGalaxy sg = new SamsungGalaxy();
        SamsungGalaxy.FixedBattery sgb
           = sg.new FixedBattery();
        boolean test = sgb.runDiagnosis();
    }
}
```

- To instantiate the inner class, first we have to instantiate the outer class
- Thereafter, using the object of the outer class, we can instantiate the inner class
  - Note the usage of “new” keyword

© Vivek Kumar

Inner Class Instance Outside Outer Class

```
public class SamsungGalaxy {

    private FixedBattery myBattery;

    public SamsungGalaxy() {
        myBattery = new FixedBattery();
    }


    private class FixedBattery {
        private boolean runDiagnosis() { ..... }
        ....
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        SamsungGalaxy sg = new SamsungGalaxy();
        SamsungGalaxy.FixedBattery sgb
           = sg.new FixedBattery();
        boolean test = sgb.runDiagnosis();
    }
}
```

- Is this code correct?
  - NO, compilation error!
- Inner class FixedBattery is private and hence cannot be accessed in another class
  - Rules of “private” modifier
  - Making the inner class and its method as public will correct this code

© Vivek Kumar

```
class SamsungGalaxy {
    private int version=2;
    private FixedBattery myBattery;

    public SamsungGalaxy() {
        myBattery = new FixedBattery();
    }

    public class FixedBattery {
        private int version=3;
        public void print() {
            int version = 4;
            System.out.println(version);
            System.out.println(this.version);
            System.out.println(SamsungGalaxy.this.version);
        }
    }
}

class Main {
    public static void main(String[] args) {
SamsungGalaxy sg = new SamsungGalaxy();
SamsungGalaxy.FixedBattery sgb = sg.new FixedBattery();
        sgb.print();
    }
}
```

Output = 4, 3, 2

Bugs and Testing

- **Software reliability**: Probability that a software system will not cause failure under specified conditions.

- **Bugs** are inevitable in any complex software system.
  - Industry estimates: 10-50 bugs per 1000 lines of code.
  - A bug can be visible or can hide in your code until much later.

- **Testing**: A systematic attempt to reveal errors.
  - Failed test: an error was demonstrated.
  - Passed test: no error was found (for this particular situation)
  - It plays a critical role in helping a software company deliver quality products to its customers.

© Vivek Kumar

Manual Testing v/s Automated Testing

Manual Testing	Automated Testing
Executing test cases manually without any tool support is known as manual testing	Taking tool support and executing the test cases by using an automation tool is known as automation testing
Time-consuming and tedious – Since test cases are executed by human resources, it is very slow and tedious	**Fast** – Automation runs test cases significantly faster than human resources
Huge investment in human resources – As test cases need to be executed manually, more testers are required in manual testing	**Less investment in human resources** – Test cases are executed using automation tools, so less number of testers are required in automation testing
Less reliable – Manual testing is less reliable, as it has to account for human errors	**More reliable** – Automation tests are precise and reliable.
Non-programmable – No programming can be done to write sophisticated tests to fetch hidden information	**Programmable** – Testers can program sophisticated tests to bring out hidden information

Source: https://www.tutorialspoint.com/junit/junit_overview.htm

JUnit: Java Unit Testing Framework

- The Java library **JUnit** helps us to easily perform automated unit testing
- The basic idea:
  - For a given class `Foo`, create another class `FooTest` to test it, containing various "test case" methods to run.
  - Each method looks for particular results and passes / fails
- JUnit provides "**assert**" commands to help us write tests.
  - The idea: Put assertion calls in your test methods to check things you expect to be true. If they aren't, the test will fail

© Vivek Kumar

Sample JUnit Test

```
/* The class method to be tested */
public class Sum {
    private int var1, var2;
    public Sum(int v1, int v2) {var1=v1; var2=v2;}
    public int sum () {
        return var1 + var2;
    }
}
```

```
/* Junit test class */

import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class SumTest {

    @Test
    public void testSum() {
        Sum mySum = new Sum(1, 1);
        int sum = mySum.sum();
        assertEquals(2, sum);
    }
}
```

```
/* Junit test runner class */

import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result=
JUnitCore.runClasses(SumTest.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

@Test annotation specifies that method is the test method.

@Test(timeout=1000) annotation specifies that method will be failed if it takes longer than 1000 milliseconds (1 second).

@BeforeClass annotation specifies that method will be invoked only once, before starting all the tests.

@Before annotation specifies that method will be invoked before each test.

@After annotation specifies that method will be invoked after each test.

@AfterClass annotation specifies that method will be invoked only once, after finishing all the tests.

© Vivek Kumar

JUnit Assertion Methods

`assertTrue(`**test**`)`	fails if the boolean test is `false`
`assertFalse(`**test**`)`	fails if the boolean test is `true`
`assertEquals(`**expected, actual**`)`	fails if the values are not equal
`assertSame(`**expected, actual**`)`	fails if the values are not the same (by ==)
`assertNotSame(`**expected, actual**`)`	fails if the values *are* the same (by ==)
`assertNull(`**value**`)`	fails if the given value is *not* `null`
`assertNotNull(`**value**`)`	fails if the given value is `null`
`fail()`	causes current test to immediately fail

- Each method can also be passed a string to display if it fails:
  - e.g. assertEquals("**message**", **expected**, **actual**)
- Detailed description: https://junit.org/junit4/javadoc/4.8/org/junit/Assert.html

What is Wrong?

```java
/* The class method to be tested */
public class Sum {
    private int var1, var2;
    public Sum(int v1, int v2) {var1=v1; var2=v2;}
    public void incr () {
        var1++; var2++;
    }
}
```

```java
/* Junit test class */
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class MyTest {

    @Test
    public void testIncr() {
        Sum mySum = new Sum(1, 1);
        mySum.incr();
        Sum expected = new Sum(2, 2);
        assertEquals(expected, mySum);
    }
}
```

- We are passing two objects of `Sum` type into the `testIncr()` method where the `assertEquals` checks for equality
  - Missing `equals()` method in Sum !
  - No compilation/runtime error but test will fail

© Vivek Kumar

What's Still Wrong?

```java
/* The class method to be tested */
public class Sum {
    private int var1, var2;
    public Sum(int v1, int v2) {var1=v1; var2=v2;}
    public void incr () {
        var1++; var2++;
    }

    @Override
    public boolean equals(Object o) {
        if(o!=null && getClass()==o.getClass()) {
            Sum s = (Sum) o;
            return ((var1==s.var1)&&(var2==s.var2));
        }
        return false;
    }
}
```

```java
/* Junit test class */
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class MyTest {

    @Test
    public void testIncr() {
        Sum mySum = new Sum(1, 1);
        mySum.incr();
        Sum expected = new Sum(3, 3);
        assertEquals(expected, mySum); //should fail
    }
}
```

testIncr(MyTest): expected:<Sum@2e817b38> but was:<Sum@c4437c4>

- Missing `toString()` method!!

© Vivek Kumar

The Correct Version

```
/* The class method to be tested */
public class Sum {
    private int var1, var2;
    public Sum(int v1, int v2) {var1=v1; var2=v2;}
    public void incr () {
        var1++; var2++;
    }

    @Override
    public boolean equals(Object o) {
        if(o!=null && getClass()==o.getClass()) {
            Sum s = (Sum) o;
            return ((var1==s.var1)&&(var2==s.var2));
        }
        return false;
    }

    @Override
    public String toString() {
       return "("+Integer.toString(var1)+",”
                      +Integer.toString(var2)+")";
    }
}
```

```
/* Junit test class */
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class MyTest {

    @Test
    public void testIncr() {
        Sum mySum = new Sum(1, 1);
        mySum.incr();
        Sum expected = new Sum(3, 3);
        assertEquals(expected, mySum); //should fail
    }
}
```

testIncr(MyTest):
expected:<(3,3)> but was:<(2,2)>

© Vivek Kumar

Tests With a Timeout

```
@Test(timeout = 5000)
public void name() { ... }
```

- The above method will be considered a failure if it doesn't finish running within 5000 ms

```
private static final int TIMEOUT = 2000;
...
@Test(timeout = TIMEOUT)
public void name() { ... }
```

- Times out / fails after 2000 ms

Testing for Exceptions

```
@Test(expected = ExceptionType.class)
 public void name() {
     ...
 }
```

- Will pass if it *does* throw the given exception.
  - If the exception is *not* thrown, the test fails
  - Use this to test for expected errors

```
@Test(expected = ArrayIndexOutOfBoundsException.class)
public void testBadIndex() {
    ArrayIntList list = new ArrayIntList();
    list.get(4);   // should fail
}
```

Setup and Teardown

```
@Before
public void name() { ... }
@After
public void name() { ... }
```

- Methods to run before/after **each test case** method is called.
- You need to define it only once, but they will be called along with every test case method.

```
@BeforeClass
public static void name() { ... }
@AfterClass
public static void name() { ... }
```

- Methods to **run once** before/after the entire test class runs

Tips for Testing

- You cannot test every possible input, parameter value, etc.
  - So you must think of a limited set of tests likely to expose bugs.
- Think about boundary cases
  - positive; zero; negative numbers
  - right at the edge of an array or collection's size
- Think about empty cases and error cases
  - 0, -1, null; an empty list or array
- test behavior in combination
  - maybe add() usually works, but fails after you call remove()
  - make multiple calls; maybe size() fails the second time only

Trustworthy Tests

- Test one thing at a time per test method
  - 10 small tests are much better than 1 test 10x as large

- Each test method should have few (likely 1) assert statements
  - If you assert many things, the first that fails stops the test
  - You won't know whether a later assertion would have failed

- Tests should avoid logic.
  - minimize if/else, loops, switch, etc
  - avoid try/catch
    - If it's supposed to throw, use expected= ... if not, let JUnit catch it

CSE201: Monsoon 2023
Advanced Programming

Lecture 17: Unified Modeling Language

Raghava Mutharaju (Section-A)

CSE, IIIT-Delhi

raghava.mutharaju@iiitd.ac.in

Today's Lecture

- Introduction to UML
  - We already covered UML in bits and pieces in prior lectures
    - Sequence diagram (Lecture 2)
    - Representing class relationships (Lectures 3–6)
- Relationships in use case diagrams
- Goal of this lecture is to give you more familiarity with UML
  - You can model 80% of problems by using about 20% UML
  - We will only cover less than 20% here
    - Not possible to teach everything…

© Vivek Kumar

What is UML?

- UML stands for Unified Modeling Language
- It's a widely used modeling language in the field of software engineering
- It's used to analyze, design, and implement software-based systems
- Pretty pictures (diagrams)

© Vivek Kumar

Motivations for UML

- We need a modeling language to:
  - help develop efficient, effective and correct designs, particularly Object Oriented designs
  - communicate clearly with project stakeholders (concerned parties: developers, customer, etc)
  - give us the “big picture” view of the project

© Vivek Kumar

UML Diagrams

Three types of UML diagrams that we will cover:

1. **Class diagrams:** Represents static structure
2. **Use case diagrams:** Sequence of actions a system performs to yield an observable result to an actor
3. **Sequence diagrams:** Shows how groups of objects interact in some behavior
   - Already covered in Lecture 02

© Vivek Kumar

UML Diagrams: Class Diagrams

- Better name: “Static structure diagram”
  - Doesn‘t describe temporal aspects
  - Doesn‘t describe individual objects: Only the overall structure of the system
- There are “object diagrams” where the boxes represent instances
  - Rarely used and not covered in this course

© Vivek Kumar

UML Class Notation

- A class is a rectangle divided into three parts
  - Class name
  - Class attributes (i.e. data members, variables)
  - Class operations (i.e. methods)

- Modifiers
  - Private: -
  - Public: +
  - Protected: #
  - Static: Underlined

Employee

-Name: String
+ID: long
#Salary: double

+getName: String
+setName()
-calcInternalStuff(in x : byte, in y : decimal)

- Abstract class/methods
  - Name in italics

© Vivek Kumar

Different Levels of Specifying Classes

Use this for your project

© Vivek Kumar

Class Relationships

- UML diagrams for these class relationships are already covered before (Lectures 04, 05 and 08)
  - Association
  - Composition
  - Dependency
  - Inheritance
- We will only cover binary association relationship here

Class Relationship: Binary Association

Both entities “Knows About” each other (two-way association)

UML Multiplicities

Links on associations to specify more details about the relationship

Multiplicities	Meaning
0..1	zero or one instance. The notation "***n . . M***" indicates ***n*** to ***m*** instances.
0..* *or* *	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance

Exceptions

Interfaces

How is this diagram different from that of a class ?
<<interface>>
Owner
+acquire(property)
+dispose(property)
Person
-real
-tangible
-intangible
Corporation
-current
-fixed
-longTerm
-intangible

Sample Class Diagram (1/2)

In your UML diagrams, these “+”, “-”, etc, should be inside the rectangle.

Sample Class Diagram (2/2)

UML Diagrams: Use Cases

- Means of capturing requirements
  - Used at a very early phase of software development for requirement gathering (analysis phase)
  - Provides a high level overview of the system
  - Class diagrams are created after generating use case diagrams
- Document interactions between user(s) and the system
  - User (actor) is not part of the system itself
  - But an actor can be *another* system
- A scenario based technique in UML
- **Use case diagrams** describe what a system does from the standpoint of an external observer. The emphasis is on *what* a system does rather than *how*

© Vivek Kumar

Actors in Use Case

- What is an Actor?
  - A user or outside system that interacts with the system being designed in order to obtain some value from that interaction
  - It can be a:
    - Human
    - Peripheral device (hardware)
    - External system or subsystem
    - Time or time-based event
  - Labelled using a descriptive noun or phrase
  - Represented by stick figure

Use Case Analysis (1/4)

- Sample scenario
  - *“A patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot"*

- We want to write a use case for this scenario

Source: http://www.cs.fsu.edu/~baker/swe1/restricted/notes/ppt/UseCaseDiagrams.ppt

Use Case Analysis (2/4)

- Sample scenario
  - "A *patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot"*

- Who is the actor?
  - The actor is a "Patient" here

Source: http://www.cs.fsu.edu/~baker/swe1/restricted/notes/ppt/UseCaseDiagrams.ppt

Use Case Analysis (3/4)

- Sample scenario
  - "A *patient calls the clinic to make an appointment for a yearly checkup. The receptionist finds the nearest empty time slot in the appointment book and schedules the appointment for that time slot"*

- A **use case** is a summary of scenarios for a single task or goal
  - So, what is the use case here?
  - The use case is "Make Appointment"

Source: http://www.cs.fsu.edu/~baker/swe1/restricted/notes/ppt/UseCaseDiagrams.ppt

Use Case Analysis (4/4)

- The picture below is a **Make Appointment** use case for the medical clinic.
- The actor is a **Patient**. The connection between actor and use case is a **communication**
- Actors are stick figures
- Use cases are ovals
  - Labelled using a descriptive verb-noun phrase
- Communications are lines that link actors to use cases
- Boundary rectangle is placed around the perimeter of the system to show how the actors communicate with the system

Source: http://www.cs.fsu.edu/~baker/swe1/restricted/notes/ppt/UseCaseDiagrams.ppt

Use Case Diagram

- A use case diagram is a collection of actors, use cases, and their communications

Source: http://www.cs.fsu.edu/~baker/swe1/restricted/notes/ppt/UseCaseDiagrams.ppt

Relationships for Use Cases

- Association
- Generalization
- Extend
- Include

© Vivek Kumar

Association Relationship

- Exists only between an actor and a use case
  - Indicates that an actor can use certain functionality of the system
- Represented by a sold line without arrowhead
  - Most commonly used representation
  - Uncommon to show one-way association
- The association between an actor and a use case can also show multiplicity at each end

© Vivek Kumar

Generalization Relationship

- Could exit between two actors or between two use cases
  - Indicates parent/child relationship
- Represented by a solid line with a triangular and hollow arrowhead
  - From child to parent

© Vivek Kumar

Extend Relationship “<<extend>”

- Exists only between use cases
  - This relationships represent optional or seldom invoked cases
  - Indicates that although one use case is a variation of another but it is invoked rarely
    - Lot of shared code between these use cases **(not to be confused with inheritance)**
- Represented using a dashed arrow with an arrowhead. The notation “<< extend >>” is also mentioned above the arrow
  - The direction of the arrow is toward the extended use cases

© Vivek Kumar

Include Relationship “<<include>”

- Exists only between use cases
  - Represents behavior that is factored out of the use case
  - Doesn’t mean that the factored out use case is an optional or seldom invoked cases
- Represented using a dashed arrow with an arrowhead. The notation “<< include>>” is also mentioned above the arrow
  - The direction of the arrow is toward the included use case

© Vivek Kumar

Sample Use Case

Next Lecture

- Event driven programming using JavaFX

TA
TF
A
B
C
D
E
F
Evaluate Quiz
Evaluate Bonus
Evaluate Lab
Update Marksheet

CSE201: Monsoon 2023
Advanced Programming

Lecture 18: Event Driven Programming using JavaFX

Raghava Mutharaju (Section A)
Computer Science and Engineering
IIIT-Delhi
raghava.mutharaju@iiitd.ac.in

Last Lecture

- Unified Modeling Language (UML)
  - It's used to analyze, design, and implement software-based systems using diagrams
  - Give us the "big picture" view of the project
  - Types of UML diagrams
    - Class diagrams
      - Static structure diagram
      - Relationship between classes
    - Use case diagrams
      - A sequence of action a systems performs that yields a valuable result for an individual user

Today’s Lecture

- Introduction to JavaFX
- Event driven programming
- *Note that JavaFX is vast and we are only covering very basic concepts in this lecture. For your project you might require some advanced features in JavaFX*

JavaFX slides Acknowledgements: CSE114, Stony Brook University

(http://www3.cs.stonybrook.edu/~pfodor/courses/cse114.html)

+ Oracle online documentation

GUI

- Graphical User Interface
  - Provides user-friendly human interaction
- History of GUI programming
  - Abstract Window Toolkit
  - Swings
  - JavaFX script
  - JavaFX library

© Vivek Kumar

GUI Examples

© Vivek Kumar

How do GUI Works?

- They loop and respond to events

© Vivek Kumar

How does GUI Framework Help?

- Provides ready made visible, interactive, customizable components
  - you wouldn’t want to have to code your own window

© Vivek Kumar

JavaFX: Simplifies Application Development

Prism
Web
JavaFX
Java
Rich
media
Components
Graphics
Accelerated
GUI
development
API
tools
developers
rich
design
desktop
deployed
native
implementation
source
component
based
plugins
runtime
system
improved
Swing
Embed
standard
conversion
binding
footprint
applets
data-driven
Developers
simple
level
pipeline
preserve
simplify
APIs
allow
seamlessly
available
visual
usage
developed
plugin
Java-based
programming
performance
open
WebView
Low
designers
RESTful
Transform
Producer
multi-threading
GPUs
Checkout
hardware-accelerated
Plug-in
Loading
Debug
Create
Crash
Script
Playback
Media
Drawing
Grids
Embedding
Debugging
Layouts
Tables
Illustrators
Illustrator
Mobile
Tablet
Laptop
Desktop
Workstation
Cross-platform
Open-source
Open
Cross-browser
Cross-device
Cross-platform
Cross-browser
Cross-device
Cross-platform
Cross-browser
Cross-device
Cross-platform

Basic Structure of JavaFX

- Class `javafx.stage.Stage` is the top level JavaFX container
- Class `javafx.scene.Scene` class is the container for all content in a scene graph
- Abstract class `javafx.application.Application` is the entry point for JavaFX applications
  - Executes the user application and processes input events
  - User just need to Override the `start` method!
- Components can be created/added programmatically

```
Parent p;
Node n;
p.getChildren().add(n)
```

© Vivek Kumar

JavaFX: Hello World

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    //Override the start method in the Application
class
    @Override
    public void start(Stage primaryStage) {
        // Set the stage title
        primaryStage.setTitle("MyJavaFX");
        // Create a button and place it in the
scene
        Button btn = new Button(”Hello World");
        Scene scene = new Scene(btn, 200, 250);
        // Place the scene in the stage
        primaryStage.setScene(scene);
        // Display the stage
        primaryStage.show();
    }

}
```

- The main class for a JavaFX application extends the `javafx.application.Application` abstract class
  - The start() method is the main entry point for all JavaFX applications

© Vivek Kumar

Application’s Life Cycle

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    //Override the start method in the Application
class
    @Override
    public void start(Stage primaryStage) {
        // Set the stage title
        primaryStage.setTitle("MyJavaFX");
        // Create a button and place it in the
scene
        Button btn = new Button(”Hello World");
        Scene scene = new Scene(btn, 200, 250);
        // Place the scene in the stage
        primaryStage.setScene(scene);
        // Display the stage
        primaryStage.show();
    }

}
```

1. Constructs an instance of the specified Application class
2. Calls the concrete method init()
3. Calls start(javafx.stage.Stage) method (must be Overridden)
4. Waits for the application to finish
5. Calls the concrete method stop()

© Vivek Kumar

JavaFX: Button in a Pane

```
public class ButtonInPane extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        // Set the stage title
        primaryStage.setTitle(”Button in a Pane");
        // Create a button and place it in the
scene
        Button btn = new Button(”OK");
        // Create a pane and place a button in the
pane
        StackPane pane = new StackPane();
        pane.getChildren().add(btn);
        // Create scene with a pane inside it
        Scene scene = new Scene(pane, 200, 50);
        // Place the scene in the stage
        primaryStage.setScene(scene);
        // Display the stage
        primaryStage.show();
    }
}
```

© Vivek Kumar

Alternatively: Scene Builder + FXML

- Scene Builder provides a graphical interface for designing and constructing user interfaces
- Scene Builder allows for components to be created, placed, and for many of their properties to be modified
- Saves your layout in an FXML file, which could be read in the Java file to create the GUI

Let’s Compare: JavaFX 2.0

```
public class JavaFXTest extends Application {
  @Override public void start(Stage stage) {
    stage.setTitle(“FXML Example”);
    Group root = new Group();
    Scene scene = new Scene(root,100,100);
    stage.setScene(scene);

    Circle c1 =
      new Circle(50.0f, 50.0f, 50.0f, Color.RED);

    root.getChildren().add(c1);
    stage.setVisible(true);
    stage.show();
  }

  public static void main(String[] args) {
     launch(args);
  }
}
```

Let’s Compare: FXML

```
<BorderPane>
  <center>
    <Circle radius=“50” centerX=“50” centerY=“50”/>
  </center>
</BorderPane>


public class JavaFXTest extends Application {
  @Override public void start(Stage stage) {
    stage.setTitle(“FXML Example”);
    Parent root = FXMLLoader.load(getClass().getResource(“example.fxml"),
        ResourceBundle.getBundle(“r.fxml_example"));
    stage.setScene(new Scene(root));
    stage.show();

  }
  public static void main(String[] args) {
     launch(args);
  }
}
```

JavaFX UI Controls

Event Programming

- Procedural programming is executed in procedural/statement order
- In event-driven programming, code is executed upon activation of events
- Operating Systems constantly monitor events
  - Ex: keystrokes, mouse clicks, etc…
- The OS:
  - sorts out these events
  - reports them to the appropriate programs

© Vivek Kumar

How to do Event Programming?

- For each control (button, combo box, etc.)
  - define an event handler
  - construct an instance of event handler
  - tell the control who its event handler is
- Event Handler?
  - code with response to event
  - a.k.a. event listener

© Vivek Kumar

Java's Event Handling

- An event source is a GUI control
  - JavaFX: Button, ChoiceBox, etc.
- Different types of sources:
  - can detect different types of events
  - can register different types of listeners (handlers)

Event Creation

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) { // entry
point
        primaryStage.setTitle("Hello World!");
        Button btn = new Button("Say Hello World");


        btn.setOnAction(new HelloEvent());
        StackPane pane = new StackPane();
        pane.getChildren().add(btn);
        Scene scene = new Scene(pane, 200, 50);
        // Place the scene in the stage
        primaryStage.setScene(scene);
        // Display the stage
        primaryStage.show();
    }
}

.........
```

- When the user interacts with a control (source):
  - an event object is constructed
    - Contain information about the event
    - Like what?
      - location of mouse click
      - event source that was interacted with, etc.
  - the event object is sent to all registered listener objects
  - the listener object (handler) responds as you defined it to

© Vivek Kumar

Event Listeners

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) { // entry point
        primaryStage.setTitle("Hello World!");
        Button btn = new Button("Say Hello World");

        btn.setOnAction(new HelloEvent());
        StackPane pane = new StackPane();
        pane.getChildren().add(btn);
        Scene scene = new Scene(pane, 200, 50);
        // Place the scene in the stage
        primaryStage.setScene(scene);
        // Display the stage
        primaryStage.show();
    }
}

class HelloEvent implements EventHandler<ActionEvent> {
     @Override
     public void handle(ActionEvent event) {
         System.out.println("Hello World!");
     }
}
```

- Event listeners (event handler)
  - Defined by you, the application programmer
    - you customize the response
    - How?
      - Inheritance & Polymorphism
  - You define your own listener class
    - implement the appropriate interface
    - define responses in all necessary methods

© Vivek Kumar

Summary: How to Handle GUI Events

- Source object: button
  - An event is generated by external user actions such as mouse movements, mouse clicks, or keystrokes
- An event can be defined as a type of signal to the program that something has happened
- Listener object contains a method for processing the event.

Working of Our Hello World GUI

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) { // entry point
        primaryStage.setTitle("Hello World!");
        Button btn = new Button("Say Hello World");

        btn.setOnAction(new HelloEvent());
        StackPane pane = new StackPane();
        pane.getChildren().add(btn);
        Scene scene = new Scene(pane, 200, 50);
        // Place the scene in the stage
        primaryStage.setScene(scene);
        // Display the stage
        primaryStage.show();
    }
}

class HelloEvent implements EventHandler<ActionEvent> {
     @Override
     public void handle(ActionEvent event) {
         System.out.println("Hello World!");
     }
}
```

© Vivek Kumar

Productivity in Event Programming

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) { // entry point
        primaryStage.setTitle("Hello World!");
        Button btn = new Button("Say Hello World");

        btn.setOnAction(new HelloEvent());
        StackPane pane = new StackPane();
        pane.getChildren().add(btn);
        Scene scene = new Scene(pane, 200, 50);
        // Place the scene in the stage
        primaryStage.setScene(scene);
        // Display the stage
        primaryStage.show();
    }
}

class HelloEvent implements EventHandler<ActionEvent> {
     @Override
     public void handle(ActionEvent event) {
         System.out.println("Hello World!");
     }
}
```

- Can we write this code in much better way?

© Vivek Kumar

Productivity in Event Programming (1/3)

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) { // entry point
        primaryStage.setTitle("Hello World!");
        Button btn = new Button("Say Hello World");

        btn.setOnAction(new HelloEvent());
        StackPane pane = new StackPane();
        pane.getChildren().add(btn);
        Scene scene = new Scene(pane, 200, 50);
        // Place the scene in the stage
        primaryStage.setScene(scene);
        // Display the stage
        primaryStage.show();
    }

    class HelloEvent implements EventHandler<ActionEvent> {
         @Override
         public void handle(ActionEvent event) {
             System.out.println("Hello World!");
         }
    }
}
```

- Using **inner** classes for creating listener objects

© Vivek Kumar

Productivity in Event Programming (2/3)

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) { // entry point
        primaryStage.setTitle("Hello World!");
        Button btn = new Button("Say Hello World");

        btn.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });
        StackPane pane = new StackPane();
        pane.getChildren().add(btn);
        Scene scene = new Scene(pane, 200, 50);
        // Place the scene in the stage
        primaryStage.setScene(scene);
        // Display the stage
        primaryStage.show();
    }
}
```

- Using **anonymous** inner classes for creating listener objects
  - It combines declaring an inner class and creating an instance of the class in one step
  - An anonymous inner class must always extend a superclass or implement an interface, but it cannot have an explicit extends or implements clause
    - An anonymous inner class must implement all the abstract methods in the superclass or in the interface
  - An anonymous inner class always uses the no-arg constructor from its superclass to create an instance

© Vivek Kumar

Productivity in Event Programming (3/3)

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) { // entry
point
        primaryStage.setTitle("Hello World!");
        Button btn = new Button("Say Hello World");


        btn.setOnAction(e -> {
            System.out.println("Hello World!");
        });
        StackPane pane = new StackPane();
        pane.getChildren().add(btn);
        Scene scene = new Scene(pane, 200, 50);
        // Place the scene in the stage
        primaryStage.setScene(scene);
        // Display the stage
        primaryStage.show();
    }
}
```

What is a Lambda?

- **Using Java 8 lambda expressions to simplify event handling**
  - Lambda expressions can be viewed as an anonymous method with a concise syntax
  - The statements in the lambda expression is all for that method
  - If it contains multiple methods, the compiler will not be able to compile the lambda expression
  - So, for the compiler to understand lambda expressions, the interface must contain exactly one method

© Vivek Kumar

Collection Elements (1/4)

```
public class Test {
    Map<String, Integer> items =
          new HashMap<String, Integer>();

    public void addElements() { ..... }

    public void print() {
        for(Map.Entry<String, Integer> entry
          : items.entrySet()) {
            System.out.println(entry.getKey()
          + “, “ + entry.getValue());
        }
    }

}
```

- Till now we know only this way to iterate over a collection (e.g., Map)
- Drawback
  - Slightly inconvenient coding

© Vivek Kumar

Collection Elements (2/4)

```
public class Test {
    Map<String, Integer> items =
          new HashMap<String, Integer>();

    public void addElements() { ..... }

    public void print() {
        items.forEach( (k, v) -> {
            System.out.println(k + “, “ + v);
        });
    }

}
```

- Java 8 introduces **forEach** statement to ease iterating over the collection elements
- With **lambda expressions** in Java 8 this code becomes very compact now!

© Vivek Kumar

Collection Elements (3/4)

```
public class Test {
    Map<String, Integer> items =
          new HashMap<String, Integer>();

    public void addElements() { ..... }

    public void print() {
        items.forEach( (k, v) -> {
            if("ABC".equals(k)) {
                System.out.println("Hello
ABC!");
            }
            System.out.println(k + ", " + v);
        });
    }

}
```

- With lambda expressions in Java 8 this code becomes very compact now!
- You can do some more stuff inside that lambda function!

© Vivek Kumar

Collection Elements (4/4)

```
public class Test {
    Map<String, Integer> items =
          new HashMap<String, Integer>();

    public void addElements() { ..... }

    public void print() {
        items.forEach( (String k, Integer v)
-> {
            if("ABC".equals(k)) {
                System.out.println("Hello
ABC!");
            }
            System.out.println(k + ", " + v);
        });
    }

}
```

- With lambda expressions in Java 8 this code becomes very compact now!
- You can do some more stuff inside that lambda function!
- You can even declare type of variables in lambda function

© Vivek Kumar

Productivity in Event Programming (3/3)

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) { // entry
point
        primaryStage.setTitle("Hello World!");
        Button btn = new Button("Say Hello World");


        btn.setOnAction(e -> {
            System.out.println("Hello World!");
        });
        StackPane pane = new StackPane();
        pane.getChildren().add(btn);
        Scene scene = new Scene(pane, 200, 50);
        // Place the scene in the stage
        primaryStage.setScene(scene);
        // Display the stage
        primaryStage.show();
    }
}
```

- Using Java 8 **lambda** expressions to simplify event handling
  - Lambda expressions can be viewed as an anonymous method with a concise syntax
  - The statements in the lambda expression is all for that method
  - If it contains multiple methods, the compiler will not be able to compile the lambda expression
  - So, for the compiler to understand lambda expressions, the interface must contain exactly one method

© Vivek Kumar

Productivity in Event Programming (3/3)

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) { // entry
point
        primaryStage.setTitle("Hello World!");
        Button btn = new Button("Say Hello World");


        btn.setOnAction(e -> {
            System.out.println("Hello World!");
        });
        StackPane pane = new StackPane();
        pane.getChildren().add(btn);
        Scene scene = new Scene(pane, 200, 50);
        // Place the scene in the stage
        primaryStage.setScene(scene);
        // Display the stage
        primaryStage.show();
    }
}
```

- Using Java 8 **lambda** expressions to simplify event handling
  - Lambda expressions can be viewed as an anonymous method with a concise syntax
  - The statements in the lambda expression is all for that method
  - If it contains multiple methods, the compiler will not be able to compile the lambda expression
  - So, for the compiler to understand lambda expressions, the interface must contain exactly one method

© Vivek Kumar

Next Lecture

- Introduction to process and threads

CSE201: Monsoon 2023
Advanced Programming

Lecture 19(a): Intro to Multithreading

Raghava Mutharaju (Section-A)
CSE, IIIT-Delhi
raghava.mutharaju@iiitd.ac.in

Your Laptop is Multitasking (1/2)

- Multitasking allows many more tasks to be run than there are CPUs
- In the case of a computer with a single CPU, only one task is said to be running at any point in time, meaning that the CPU is actively executing instructions for that task
- Multitasking solves the problem by scheduling which task may be the one running at any given time, and when another waiting task gets a turn

© Vivek Kumar

Your Laptop is Multitasking (1/2)

- In case of multicore processor each core will be running one task
- Here too OS will be multitasking by deciding which task runs on which core

© Vivek Kumar

Process

- Process is a program in execution
  - i.e., an icon of M.S. Word on your desktop isn’t a process. It would become a process once you have launched it
- Each process has its own address/memory space
- Separate processes don’t have access to each other’s memory space
  - O.S. can help in creating a communication channel between processes if there be any need

© Vivek Kumar

Process Structure

- Process contains:
  - Code
    - Program instructions
  - Data
    - Global variables in program
  - Program counter (PC)
    - Address of currently executing program instruction
  - Registers
    - CPU registers
  - Stack
    - Local variables
    - caller-callee relationship between function

© Vivek Kumar

Thread – A Lightweight Process

- Processes are heavyweight
  - Personal address space (allocated memory)
  - Communication across process always requires help from Operating System
- Threads are lightweight
  - Share resources inside the parent process (code, data and files)
    - Easy to communicate across sibling threads!
  - They have their own personal stack (local variables, caller-callee relationship between function)
    - Each thread is assigned a different job in the program
- A process can have one or more threads

© Vivek Kumar

Uses of Threads in Java

- Video game application (one process)
  - one thread for graphics
  - one thread for user interaction
  - one thread for networking with peers on internet
- Parallel programming
  - Speeding up the program execution by using multiple threads
  - A program to find the array sum can be made faster by subdividing the array among **`N`** threads (total indices at each thread = **`array.length/N`**). Each thread will find its local sum and later these local sums will be combined to find the total sum
- Producer-consumer type applications
  - News reporter (producer) sends report to news agency and one or more office clerks (consumers) will process each items and upload to Twitter, Facebook, Website, or telecast on TV
- Client-server type applications
  - Server could create a new thread for listening to a new client

© Vivek Kumar

Advantages of Multithreading

- Responsiveness
  - Even if part of program is blocked or performing lengthy operation, multithreading allows the program to continue
- Economical resource sharing
  - Threads share memory and resources of their parent process which allows multiple tasks to be performed simultaneously inside the process
- Utilization of multicores
  - Easily scale on modern multicore processors

© Vivek Kumar

Class Thread in Java

Constructors of Thread class

1. **Thread** ( )
2. **Thread** (*String str*)
3. **Thread** (*Runnable r*)
4. **Thread** (*Runnable r*, *String str*)

You can create new thread, either by extending Thread class or by implementing Runnable interface.

Thread class also defines many methods for managing threads. Some of them are,

Method	Description
setName()	to give thread a name
getName()	return thread's name
getPriority()	return thread's priority
isAlive()	checks if thread is still running or not
join()	Wait for a thread to end
run()	Entry point for a thread
sleep()	suspend thread for a specified time
start()	start a thread by calling run() method

Alive

Thread State Diagram (Lifecycle)

© Vivek Kumar

Thread Priority

- Every thread has a priority (or seniority)
- The highest priority runnable thread is always selected by JVM for execution above lower priority threads
- The priority values range from 1 to 10, in increasing priority
- When a thread is created, it inherits the priority of the thread that created it
- The priority can be adjusted subsequently using the **setPriority()** method
- The priority of a thread may be obtained using **getPriority()**
- Pre-defined priority constants in Thread class:
  - MIN_PRIORITY=1
  - MAX_PRIORITY=10
  - NORM_PRIORITY=5

© Vivek Kumar

Java Application Thread

- When we execute an application:
  - The JVM creates a Thread object whose task is defined by the **main()** method
  - It starts the thread with NORM_PRIORITY
  - The thread executes the statements of the program one by one until the method returns and the thread dies

© Vivek Kumar

Daemon Threads

- Daemon threads are “background” threads, that provide services to other threads, e.g., the garbage collection thread
- JVM will not exit if non-Daemon threads are executing
- JVM will exit if only Daemon threads are executing
- Daemon threads die when the JVM exits

© Vivek Kumar

Next Lecture

- Programming with threads in Java
  - Threads in Action !
  - A.K.A., Parallel Programming!

CSE201: Monsoon 2023
Advanced Programming

Lecture 19(b): Thread Creation

Raghava Mutharaju (Section-A)
CSE, IIIT-Delhi
raghava.mutharaju@iiitd.ac.in

© Vivek Kumar

Creating Threads in Java

```
public class MyThread implements
java.lang.Runnable {
    ...........
    @Override
    public void run() { ..... }
}
```

```
public class MyThread extends java.lang.Thread {
    ...........
    @Override
    public void run() { ..... }
}
```

- There are two ways to create your own **Thread** object
  - Implementing the **Runnable** interface
  - Subclassing the **Thread** class and instantiating a new object of that class
- In both cases the **run()** method should be implemented

© Vivek Kumar

Sequential Array Sum Implementation

```
public class ArraySum {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }

    //assume array.length%2=0
    public void calculate() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }

    public int getResult() { return sum; }

    public static void main(String[] args)
    {
      int size; int[] array; //allocated (size) &
initialized
      ArraySum asum = new ArraySum(array, 0, size);
      asum.calculate();
      int result = asum.getResult();
    }
}
```

- This is a sequential code to find the sum of elements in an array
- Can we use multithreading here?
  - Which part of the code we can parallelize?
  - As the length of array grows huge, the execution time will start increasing

© Vivek Kumar

Parallel Array Sum Implementation (1/6)

```
public class ArraySum implements Runnable {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void calculate() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
    {
      int size; int[] array; //allocated (size) &
initialized
      ArraySum asum = new ArraySum(array, 0, size);
      asum.calculate();
      int result = asum.getResult();
    }
}
```

- Lets parallelize the execution of “calculate” method by implementing `Runnable` interface
  - This method is the performance bottleneck as array length grows huge
- Step-1
  - Implement `java.lang.Runnable` interface

© Vivek Kumar

Parallel Array Sum Implementation (2/6)

```
public class ArraySum implements Runnable {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
    {
      int size; int[] array; //allocated (size) &
initialized
      ArraySum asum = new ArraySum(array, 0, size);
      asum.calculate();
      int result = asum.getResult();
    }
}
```

- Step-2
  - Implement the method "public void run()"
  - This abstract method is in Runnable interface (no other methods there)
  - For simplicity, we will rename "calculate" method in this example to "run"
    - Note that run() method is of void type
    - In next lecture we will see how to return results (or objects) from Threads

© Vivek Kumar

Parallel Array Sum Implementation (3/6)

```
public class ArraySum implements Runnable {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
    {
      int size; int[] array; //allocated (size) &
initialized
      ArraySum left = new ArraySum(array, 0, size/2);
      ArraySum right = new ArraySum(array, size/2, size);
      Thread t1 = new Thread(left);
      Thread t2 = new Thread(right);


    }
}
```

- Step-3
  - Create two threads (t1 & t2)
  - `java.lang.Thread` class
  - t1 will calculate the sum of left half of the array and t2 will calculate the sum of right half of array
    - Before creating t1 and t2 we must create objects of Runnable type that should be passed to the Thread constructor

© Vivek Kumar

Parallel Array Sum Implementation (4/6)

```
public class ArraySum implements Runnable {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
    {
      int size; int[] array; //allocated (size) &
initialized
      ArraySum left = new ArraySum(array, 0, size/2);
      ArraySum right = new ArraySum(array, size/2, size);
      Thread t1 = new Thread(left);
      Thread t2 = new Thread(right);
      t1.start(); t2.start();

    }
}
```

- Step-4
  - Start both the threads by calling the start() method in Thread class
  - JVM now allows this thread to begin its execution
  - JVM calls the run() method of this thread
    - Thread class also implements Runnable interface but has empty bodied run()
    - When a Thread is created using a Runnable object (as in this example), then run() implementation of that Runnable object is called

© Vivek Kumar

Parallel Array Sum Implementation (5/6)

```
public class ArraySum implements Runnable {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
                                throws InterruptedException {
      int size; int[] array; //allocated (size) &
initialized
      ArraySum left = new ArraySum(array, 0, size/2);
      ArraySum right = new ArraySum(array, size/2, size);
      Thread t1 = new Thread(left);
      Thread t2 = new Thread(right);
      t1.start(); t2.start();
      t1.join(); t2.join();

    }
}
```

- Step-5
  - Wait for both the threads to complete their execution (i.e. wait for them to finish execution of run method)
    - join() method from Thread class is used for this purpose
    - join() method throws checked exception and hence main() must declare that

© Vivek Kumar

Parallel Array Sum Implementation (6/6)

```
public class ArraySum implements Runnable {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
                                throws InterruptedException {
      int size; int[] array; //allocated (size) &
initialized
      ArraySum left = new ArraySum(array, 0, size/2);
      ArraySum right = new ArraySum(array, size/2, size);
      Thread t1 = new Thread(left);
      Thread t2 = new Thread(right);
      t1.start(); t2.start();
      t1.join(); t2.join();
      int result = left.getResult() + right.getResult();
    }
}
```

- Step-6
  - Sum the partial results from each threads to get the final results
- What would happen if you call t1.start() followed by t1.join() and then similarly for thread t2?
  - Although there are two threads, still the program is sequential!
- Can you write this same program with more than two threads?

© Vivek Kumar

Parallel Array Sum By Subclassing Thread

```
public class ArraySum extends Thread {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    @Override
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
                                throws InterruptedException {
      int size; int[] array; //allocated (size) &
initialized
      ArraySum t1 = new ArraySum(array, 0, size/2);
      ArraySum t2 = new ArraySum(array, size/2, size);
      t1.start(); t2.start();
      t1.join(); t2.join();
      int result = t1.getResult() + t2.getResult();
    }
}
```

- Only three changes are required
  1. Instead of implementing Runnable, now the ArraySum class will extend Thread class
  2. Override the run() method as Thread class also has empty-body implementation of run()
  3. ArraySum objects are themselves Thread objects and hence now no need to explicitly call constructor of Thread class

© Vivek Kumar

Runnable v/s Subclassing Thread

- **Multiple inheritance is not allowed in Java** hence if our ArraySum class extends Thread then it cannot extend any other class. By implementing Runnable our ArraySum can easily extend any other class
- **Subclassing is used in OOP to add additional feature**, modifying or improving behavior. If no modifications are being made to Thread class then use Runnable interface
- **Thread can only be started once**. Runnable is better as same object could be passed to different threads
- If just run() method has to be provided then **extending Thread class is an overhead for JVM**

© Vivek Kumar

Question: Any Issues Below?

```
.......

class MyClass1 implements Runnable {
    .......
}
class MyClass2 extends Thread {
 .......
}

.......
MyClass1 MyClass1Object = new MyClass1();
Thread t1 = new Thread(MyClass1Object);
t1.run()

MyClass2 t2 = new MyClass2();
t2.run();
```

- What would happen if we directly call run() method from Runnable or Thread object instead of start() and join()?
  - Neither a compilation or runtime error
  - No thread is created by JVM!
  - Sequential execution
  - Calling start() method is mandatory !

© Vivek Kumar

Question: Any Issues Below?

```
.......

class MyClass1 implements Runnable {
    .......
}
class MyClass2 extends Thread {
 .......
}

.......
MyClass1 MyClass1Object = new MyClass1();
Thread t1 = new Thread(MyClass1Object);
t1.start();
t1.start();

MyClass2 t2 = new MyClass2();
t2.start();
t2.start();
```

- start() method cannot be invoked more than once
  - A thread can't be restarted
  - Exception generated at runtime
    - IllegalThreadStateException
- Although we can create several threads with the same runnable type object
  - Advantage of implementing Runnable over extending Thread

© Vivek Kumar

Fibonacci Number Calculation

```java
// Sequential Implementation of Fibonacci
public class Fibonacci {
    int result, n;
    public Fibonacci(int n) { this.n = n; }
    public int fib(int n) {
        if(n<2) return n;
        else return fib(n-1) + fib(n-2);
    }
    public void calculate() {
        result = fib(n);
    }
    public int getResult() { return
result; }
    public static void main(String[] args) {
      int n = 40;
      Fibonacci fib = new Fibonacci(n);
      int result = fib.getResult();
    }
}
```

Is this an efficient implementation of parallel Fibonacci ??

```java
// Parallel Implementation of Fibonacci
public class Fibonacci implements Runnable {
    int result, n;
    public Fibonacci(int n) { this.n = n; }
    public static int fib(int n) {
        if(n<2) return n;
        else return fib(n-1) + fib(n-2);
    }
    public void run() {
        result = fib(n);
    }
    public int getResult() { return result; }
    public static void main(String[] args)
                                throws InterruptedException {
      int n = 40;
      Fibonacci left = new Fibonacci(n-1);
      Fibonacci right = new Fibonacci(n-2);
      Thread t1 = new Thread(left);
      Thread t2 = new Thread(right);
      t1.start(); t2.start();
      t1.join(); t2.join();
      int result = left.getResult() + right.getResult();
    }
}
```


© Vivek Kumar

Multithreading in Socket Programming

- Sockets provide the communication mechanism between two computers that are connected using a network
  - A two-way communication protocol
  - Communication between two processes
- A client program creates a socket on its end of the communication and attempts to connect that socket to a server
- When the connection is made, the server creates a socket object on its end of the communication
- The client and the server can now communicate by writing to and reading from the socket

© Vivek Kumar

Multithreaded Server Application

```
import java.io.*;
import java.net.*;
public class Server  {

  public static void main(String args[ ])
      throws IOException {
    /* create a server socket
       bound to the specified port 1234 */
    ServerSocket me = new ServerSocket(1234);
    /* Server is now listening
       for incoming client's request */
    while (true) {
      /* Connection is established */
       Socket connection = me.accept();
       System.out.println("Connected");
       /* Spawn a thread for every
          connecting client */
     Thread t=new Thread(new
Handler(connection));
       t.start();
    }
  }
}
```

```
class Handler implements Runnable {
  Socket connection;
  Handler(Socket connection) {
    this.connection = connection;
  }
  public void run() throws IOException {
    DataOutputStream out = null;
    try {
     out=new
DataOutputStream(connection.getOutputStream());
     out.writeUTF("Hello Client!!");
    } finally {
      out.close();
      connection.close();
    }
  }
}
```


© Vivek Kumar

Client Application

```
import java.io.*;
import java.net.*;

public class Client {

    public static void main(String args[ ])
                  throws IOException {
        String serverName = "localhost"; //or remote IP
Address
        int port = 1234; // should be same as used in server
        /* Connect to server that is already listening */
        Socket server = new Socket(serverName, port);
        System.out.println("Just connected to " +
                      server.getRemoteSocketAddress());
        DataInputStream in = new

DataInputStream(server.getInputStream());
        System.out.println("Server says " + in.readUTF());
        in.close();
        /* close connection with server */
        server.close();
    }
}
```

- Why our server application was missing the join() for the threads it spawned for every new client connection ?
  - Will the server be able to serve multiple clients in parallel?

© Vivek Kumar

Some Other Methods in Thread

- `static Thread currentThread()`
  - Returns a reference to the currently executing thread object
- `long getId()`
  - Returns the identifier of this thread
- `static void sleep(long millisec)`
  - Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds

© Vivek Kumar

Parallel Array Sum Implementation (6/6)

```
public class ArraySum implements Runnable {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
                                throws InterruptedException {
      int size; int[] array; //allocated (size) &
initialized
      ArraySum left = new ArraySum(array, 0, size/2);
      ArraySum right = new ArraySum(array, size/2, size);
      Thread t1 = new Thread(left);
      Thread t2 = new Thread(right);
      t1.start(); t2.start();
      t1.join(); t2.join();
      int result = left.getResult() + right.getResult();
    }
}
```

- Step-6
  - Sum the partial results from each threads to get the final results
- What would happen if you call t1.start() followed by t1.join() and then similarly for thread t2?
  - Although there are two threads, still the program is sequential!
- Can you write this same program with more than two threads?

© Vivek Kumar

CSE201: Monsoon 2023
Advanced Programming

Lecture 20: Introduction to Design Patterns

Raghava Mutharaju (Section A)

CSE, IIIT-Delhi

raghava.mutharaju@iiitd.ac.in

© Vivek Kumar

What is Design Pattern

- It is a solution for a repeatable problem in the software design
- This is not a complete design for a software system that can be directly transformed into code
- It is a description or template for how to solve the problem that can be used in many different situations

Why Study Patterns

- Reuse tried, proven solutions
  - Provides a head start
  - Avoids gotchas later (unanticipated things)
  - No need to reinvent the wheel
- Establish common terminology
  - Design patterns provide a common point of reference
  - Easier to say, “We could use Strategy here.”
- Provide a higher level prerspective
  - Frees us from dealing with the details too early

“GoF” (Gang of Four) patterns

- **Creational Patterns** *(abstracting the object-instantiation process)*
  - Factory Method Abstract Factory Singleton
  - Builder Prototype
- **Structural Patterns** *(how objects/classes can be combined)*
  - Adapter Bridge Composite
  - Decorator Facade Flyweight
  - Proxy
- **Behavioral Patterns** *(communication between objects)*
  - Command Interpreter Iterator
  - Mediator Observer State
  - Strategy Chain of Responsibility Visitor
  - Template Method

In 1990 a group called the Gang of Four or "GoF" (Gamma, Helm, Johnson, Vlissides) compile a catalog of design patterns in the book “Design Patterns: Elements of Reusable Object-Oriented Software”

Pattern: Iterator

objects that traverse collections

Pattern: Iterator

- Recurring Problem
  - How can you loop over all objects in any collection. You don’t want to change client code when the collection changes. Want the same methods
- Solution
  1. Provide a standard *iterator* object supplied by all data structures
  2. The implementation performs traversals, does bookkeeping
  3. The implementation has knowledge about the representation
  4. Results are communicated to clients via a standard interface
- Consequences
  - Can change collection class details without changing code to traverse the collection

Java Iterator Recap

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

© Vivek Kumar

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

© Vivek Kumar

Iterator Position

Figure 2–3: Advancing an iterator

© Vivek Kumar

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

© Vivek Kumar

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());
  }
}
```


© Vivek Kumar

Pattern: Singleton

A class that has only a single instance

Pattern: Singleton

- Recurring problem
  - Sometimes we only ever need one instance of a particular class
  - It should be illegal to have another instance of the same class
- Solution
  - Singleton pattern – ensuring that a class has at most one instance
  - Providing global access to that instance

Fig. source: https://rajneekanth.wordpress.com/2014/04/11/what-are-design-patterns/

Implementing Singleton

1. Make constructor private so that no client is able to call it from outside
2. Declare a single private static instance of the class
3. Write a getInstance() method (or similar) that allows access to the single instance
   - Ensure thread safety in case multiple threads can access this method

Singleton Example

```
public class RandomGenerator {
    private static RandomGenerator gen = null;
    public static RandomGenerator getInstance()
    {
        if (gen == null) {
            gen = new RandomGenerator();
        }
        return gen;
    }
    private RandomGenerator() {}
    ...
}
```

- Creates a new random generator
- Clients will not use the constructor directly but will instead call getInstance to obtain a RandomGenerator obect that is shared by all classes in the application
- Lazy initialization
  - Can wait until client asks for the instance to create it
  - How to ensure thread safety?

© Vivek Kumar

Singleton Comparator

```
public class LengthComparator
                implements Comparator<String> {

    private static LengthComparator comp = null;

    public static LengthComparator getInstance()
    {
        if (comp == null) {
            comp = new LengthComparator();
        }
        return comp;
    }

    private LengthComparator() {}

    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

- Comparators make great singletons because they have no state
- Saves memory by not allowing the creation of more than one object

© Vivek Kumar

Pattern: Flyweight

a class that has only one instance for each unique state

Pattern: Flyweight

- Problem
  - **Redundant objects** can bog down the system
    - Many objects have the same state
  - Example: `File` objects that represent the same file on disk
    - `new File("chatlog.txt")`
    - `new File("chatlog.txt")`
    - `new File("chatlog.txt")`
      `...`
    - `new File("notes.txt")`
  - Example: Date objects that represent the same date of the year
    - `new Date(4, 18)`
    - `new Date(4, 18)`

Pattern: Flyweight

- An assurance that no more than one instance of a class will have identical state
  - Achieved by caching identical instances of objects.
  - Similar to singleton, but one instance for each unique object state
  - Useful when there are many instances, but many are equivalent

Fig source: http://www.c-sharpcorner.com/UploadFile/SukeshMarla/learn-design-pattern-flyweight-pattern/

Implementing a Flyweight (1/2)

```
public class Flyweighted {
    private static Map<KeyType, Flyweighted> instances
            = new HashMap<KeyType, Flyweighted>();

    private Flyweighted(...) { ... }

    public static Flyweighted getInstance(KeyType key) {
        if (!instances.contains(key)) {
            instances.put(key, new Flyweighted(key));
        }
        return instances.get(key);
    }
}
```

Implementing a Flyweight (2/2)

```java
public class Point {
    private int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }
    public int getY() { return y; }

    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}
```

```java
public class Point {

    private static Map<String, Point> instances =
        new HashMap<String, Point>();

    public static Point getInstance(int x, int y)
    {
        String key = x + ", " + y;
        if (!instances.containsKey(key)) {
            instances.put(key, new Point(x, y));
        }
        return instances.get(key);
    }

    private final int x, y;  // immutable

    private Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() { return x; }
    public int getY() { return y; }
    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}
```


© Vivek Kumar

Flyweighting in String by JVM

- The possible combinations for Strings is close to infinite, hence JVM maintains a cache for strings, called the **string constant pool**
  - It is empty at startup and is filled constantly during the lifecycle of the JVM
- Java String objects are automatically flyweighted by the JVM **whenever** possible
  - If you declare two string variables that point to the same literal.
  - If you concatenate two string literals to match another literal

```
String a = "neat";
String b = "neat";
String c = "n" +
"eat";
```

Next Lecture

- More design patterns

CSE201: Monsoon 2023
Advanced Programming

Lecture 20: Adapter and Strategy Design Pattern

Raghava Mutharaju (Section A)
CSE, IIIT-Delhi
raghava.mutharaju@iiitd.ac.in

Today’s Lecture

- Adapter design pattern (DP # 4)
- Strategy design pattern (DP # 5)

Pattern: Adapter

an object that fits another object into a given interface

Pattern: Adapter

- Recurring problem
  - We have an object that contains the functionality we need, but not in the way we want to use it
- Solution
  - Create an **adapter object** that bridges the provided and desired functionality

Fig source: http://javarevisited.blogspot.in/2016/08/adapter-design-pattern-in-java-example.html

Adapter Pattern Example (1/2)

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

public class Car implements Movable {
    public void move() {
        System.out.println("Car is moving");
    }
}

public class Bike implements Movable {
    public void move() {
        System.out.println("Bike is moving");
    }
}
```

```
public class Vehicle {
  public static void main(String[] args) {
    List<Movable> mylist = new ArrayList<Movable>();

    mylist.add(new Car());
    mylist.add(new Bike());


    for(Movable obj: mylist) {
        obj.move();
    }
  }

}
```

```
public interface Flyable {
   public void fly();
}

public class Airplane implements Flyable {
   public void fly() {
      System.out.println("Airplane is flying");
   }
}

public class Drone implements Flyable {
   public void fly() {
      System.out.println("Drone is flying");
   }
}
```

- The **adaptee** interface "Flyable" only implements fly() method, although it is similar to move() in Movable inteface
- Client class, Vehicle, doesn't understand Flyable and only use Movable
  - How to add Flyable type objects inside Movable type list in Vehicle?
  - We will code an adaptor that can serve this client by using this adaptee without any modifications

© Vivek Kumar

Adapter Pattern Example (2/2)

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

public class Car implements Movable {
    public void move() {
        System.out.println("Car is moving");
    }
}

public class Bike implements Movable {
    public void move() {
        System.out.println("Bike is moving");
    }
}
```

```
public interface Flyable {
   public void fly();
}

public class Airplane implements Flyable {
   public void fly() {
      System.out.println("Airplane is flying");
   }
}

public class Drone implements Flyable {
   public void fly() {
      System.out.println("Drone is flying");
   }
}
```

```
public class Vehicle {
  public static void main(String[] args) {
    List<Movable> mylist = new ArrayList<Movable>();

    mylist.add(new Car());
    mylist.add(new Bike());

    mylist.add(new FlyableAdapter(new Airplane()));
    mylist.add(new FlyableAdapter(new Drone()));

    for(Movable obj: mylist) {
        obj.move();
    }
  }

}
```

```
public class FlyableAdapter implements Movable {
    Flyable type;
    public FlyableAdapter(Flyable type) {
        this.type = type;
    }
    public void move() {
        type.fly();
    }
}
```


© Vivek Kumar

Pattern: Strategy

objects that hold different algorithms to solve a problem

The Ducks File Structure for Redux – S ...
medium.com

Wood Duck Identification, All Abo
allaboutbirds.org

Duck test - Wikipedia
en.wikipedia.org

Ever Wanted to Know About Ducks
thoughtco.com

Duck

Bufflehead

(*Bucephala albeola*)

Scientific classification

Kingdom:	Animalia
Phylum:	Chordata
Class:	Aves
Order:	Anseriformes
Superfamily:	Anatoidea
Family:	**Anatidae**

Subfamilies

see text

Amazon.com: Giant Duck F...
amazon.com

Different Kind of Ducks | D'Artagnan
dartagnan.com

Caring for Ducks in Winter | Modern ...
oysfarm.com

Different Types of Ducks With Examples
thespruce.com

Let’s Build a Duck Simulator!

- Concepts we will revisit
  - Inheritance
  - Interfaces
  - Polymorphism

© Vivek Kumar

What are their Characteristics?

- I’m Dabbler duck
- I can quack
- I can swim
- I can fly
- My home is on ground

- I’m Wood duck
- I can quack
- I can swim
- I can fly
- My home is on trees

© Vivek Kumar

How to Code a Duck Simulator?

- I'm Dabbler duck
- I can quack
- I can swim
- I can fly
- My home is on ground

- I'm Wood duck
- I can quack
- I can swim
- I can fly
- My home is on trees

Inheritance?

© Vivek Kumar

Lets See the Code

```
public abstract class Duck {
    private String name;
    public Duck(String n) { this.name = n; }

    public void type() {
        System.out.println("I am "+ name+" Duck");
    }
    public void speak() {
        System.out.println("I can quack");
    }
    public void swim() {
        System.out.println("I can swim");
    }
    public void fly() {
        System.out.println("I can fly");
    }
    public abstract void home();
    public void display() {
        this.type();
        this.speak();
        this.swim();
        this.fly();
        this.home();
    }
}
```

```
public class Dabbler extends Duck {
    public Dabbler() { super("Dabbler"); }

    public void home() {
        System.out.println("My home is on ground");
    }
}
```

```
public class Wood extends Duck {
    public Wood() { super("Wood"); }

    public void home() {
        System.out.println("My home is on trees");
    }
}
```

```
// Calling display on above two Duck type
objects
I am Wood Duck
I can quack
I can swim
I can fly
My home is on trees
I am Dabbler Duck
I can quack
I can swim
I can fly
My home is on ground
```


© Vivek Kumar

Any Problems?

```
public abstract class Duck {
    private String name;
    public Duck(String n) { this.name = n; }

    public void type() {
        System.out.println("I am "+ name+"
Duck");
    }
    public void speak() {
        System.out.println("I can quack");
    }
    public void swim() {
        System.out.println("I can swim");
    }
    public void fly() {
        System.out.println("I can fly");
    }
    public abstract void home();
    public void display() {
        this.type();
        this.speak();
        this.swim();
        this.fly();
        this.home();
    }
}
```

```
public class Dabbler extends Duck {
    public Dabbler() { super("Dabbler"); }

    public void home() {
        System.out.println("My home is on
ground");
    }
}
```

```
public class Wood extends Duck {
    public Wood() { super("Wood"); }

    public void home() {
        System.out.println("My home is on
trees");
    }
}
```

Please code me too

- I'm Rubber duck
- **I can squeak**
- I can swim
- **I don't fly**
- Your home is my home

© Vivek Kumar

What are the Issues?

- Applying inheritance for code reuse sometimes backfires
- Poor solution for maintenance
  - Our assumption that all Ducks can Fly is incorrect
  - Our assumption that all Ducks make quack-quack sound is incorrect
- How to fix this issue?
  - Overriding both the methods fly() and speak() in subclass Rubber Duck

© Vivek Kumar

Let’s Implement the Fix

```
public abstract class Duck {
    private String name;
    public Duck(String n) { this.name = n; }

    public void type() {
        System.out.println(“I am “+ name+”
Duck”);
    }
    public void speak() {
        System.out.println(“I can quack”);
    }
    public void swim() {
        System.out.println(“I can swim”);
    }
    public void fly() {
        System.out.println(“I can fly”);
    }
    public abstract void home();
    public void display() {
        this.type();
        this.speak();
        this.swim();
        this.fly();
        this.home();
    }
}
```

```
public class Rubber extends Duck {
    public Rubber() { super(“Rubber”); }

    @Override
    public void speak() {
        System.out.println(“I can Squeak”);
    }
    @Override
    public void fly() {
        System.out.println(“I don’t Fly”);
    }
    public void home() {
        System.out.println(“Your home is my
home”);
    }
}
```

```
// Calling display on Rubber Duck type object
I am Rubber Duck
I can Squeak
I can swim
I don’t Fly
Your home is my home
```


© Vivek Kumar

Wait.. What if we get other non-flyable Duck?

```
public abstract class Duck {
    private String name;
    public Duck(String n) { this.name = n; }

    public void type() {
        System.out.println("I am "+ name+"
Duck");
    }
    public void speak() {
        System.out.println("I can quack");
    }
    public void swim() {
        System.out.println("I can swim");
    }
    public void fly() {
        System.out.println("I can fly");
    }
    public abstract void home();
    public void display() {
        this.type();
        this.speak();
        this.swim();
        this.fly();
        this.home();
    }
}
```

```
public class Rubber extends Duck {
    public Rubber() { super("Rubber"); }

    @Override
    public void speak() {
        System.out.println("I can Squeak");
    }
    @Override
    public void fly() {
        System.out.println("I don't Fly");
    }
    public void home() {
        System.out.println("Your home is my
home");
    }
}
```

- If we have to code a **Domestic Duck** then they too don't fly
  - This means we need to Override the fly() method even inside Domestic Duck class

© Vivek Kumar

What are the Issues?

- Another Duck type could speak in a language other than “Quack” and “Squeak”
  - Examples:
    - **Decoy** Duck can’t speak
    - **Whistling** Duck make whistles
  - As there are **several possible** ways to speak, we don’t have any choice other than **Overriding** the speak() method
- However, the flying capability could be either true or false only. As the options for flying capability is limited, can we write a better code?
  - How about using an interface called Flyable that has fly() method?
    - Again there will be lot of duplicate code as each Duck type will have to implement this interface to show their flying capability

© Vivek Kumar

Recap: Design Principals

- Program to a supertype and not for an implementation
  - We used Duck as superclass in past
- Identify the aspects of the implementation that differs and separate them out from what stays the same
  - We took out similar functionality inside the superclass Duck and left the specialized implementation inside subclass

© Vivek Kumar

Using Strategy Pattern for Final Fix

1. We will still use **Flyable** interface BUT will limit its implementation in only **two** classes
2. Create a field of **Flyable** type in supertype (Duck)
3. Each subclass will simply instantiate this field inside their constructor with correct flying ability. The flying capability are defined inside the two classes mentioned in Step-1
4. display() method in Duck will use polymorphism to show the correct flying capability

© Vivek Kumar

Applying Strategy Pattern: The Final Fix!

```
public interface Flyable {
    public void fly();
}
```

```
public abstract class Duck {
    private String name;
    private Flyable flyStatus;
    public Duck(String n, Flyable f) {
        this.name = n;
        this.flyStatus = f;
    }
    ........
    ........
    public void tryFlying() {
        flyStatus.fly();
    }
    public void display() {
        this.type();
        this.speak();
        this.swim();
        this.tryFlying();
        this.home();
    }
}
```

```
public class CannotFly implements Flyable {
    public void fly() {
        System.out.println("I don't Fly");
    }
}
```

```
public class CanFly implements Flyable {
    public void fly() {
        System.out.println("I can Fly");
    }
}
```

```
public class Dabbler extends Duck {
    public Dabbler() {
        super("Dabbler", new CanFly());
    }
    ........
}
```

```
public class Rubber extends Duck {
    public Rubber() {
        super("Rubber", new CannotFly());
    }
    @Override
    public void speak() {
        System.out.println("I can Squeak");
    }
    public void home() {
        System.out.println("Your home is my home");
    }
}
```

© Vivek Kumar

Summary: Strategy Pattern

- In Strategy pattern, a class behavior (or its algorithm) can be changed at run time
- In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object
- The strategy object changes the executing algorithm of the context object
- This type of design pattern comes under behavior pattern

© Vivek Kumar

CSE201: Monsoon 2023
Advanced Programming

Lecture 21: Thread Pool

Raghava Mutharaju
Computer Science and Engineering
IIIT Delhi
raghava.mutharaju@iiitd.ac.in

© Vivek Kumar

Recap

- There are two ways to create your own **Thread** object
  - Implementing the **Runnable** interface
  - Subclassing the **Thread** class and instantiating a new object of that class

```java
public class ArraySum extends Thread {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    @Override
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
                                throws InterruptedException {
      int size; int[] array; //allocated (size) & initialized
      ArraySum t1 = new ArraySum(array, 0, size/2);
      ArraySum t2 = new ArraySum(array, size/2, size);
      t1.start(); t2.start();
      t1.join(); t2.join();
      int result = t1.getResult() + t2.getResult();
    }
}
```

```java
public class ArraySum implements Runnable {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
                                throws InterruptedException {
      int size; int[] array; //allocated (size) & initialized
      ArraySum left = new ArraySum(array, 0, size/2);
      ArraySum right = new ArraySum(array, size/2, size);
      Thread t1 = new Thread(left);
      Thread t2 = new Thread(right);
      t1.start(); t2.start();
      t1.join(); t2.join();
      int result = left.getResult() + right.getResult();
    }
}
```

- **Multiple inheritance is not allowed in Java** hence if our ArraySum class extends Thread then it cannot extend any other class. By implementing Runnable our ArraySum can easily extend any other class
- **Subclassing is used in OOP to add additional feature**, modifying or improving behavior. If no modifications are being made to Thread class then use Runnable interface
- **Thread can only be started once**. Runnable is better as same object could be passed to different threads
- If just run() method has to be provided then **extending Thread class is an overhead for JVM**

© Vivek Kumar

Think Tasks, not Threads

- Tasks are logic unit of work
- Threads are mechanism by which tasks can run asynchronously
- E.g., for calculating Fibonacci number (Lecture 19), each node in this tree represents one task

© Vivek Kumar

Mapping Tasks to Cores

- Generally
  - # of tasks $\geq$ threads available
  - parallel algorithm must map tasks to threads
  - schedule independent tasks on separate threads (consider computation graph)
  - threads should have minimum interaction with one another

© Vivek Kumar

Thread Pool

- Thread-pool consists of a fixed number of threads
  - Provided by the Java runtime
- User application creates “task” rather than threads
- These tasks are added to a task-pool
- Free threads from thread-pool takes out a task from task-pool and execute it

Fig. Source: http://www.geek-programmer.com/what-are-thread-pools-in-java/

© Vivek Kumar

Package java.util.concurrent

- Framework for concurrent programming
- In this course we will only introduce a few basic features of this framework

`ExecutorService` Interface

- An `ExecutorService` is a group of thread objects (thread pool), each running some variant of the following
  - `while (....) { get work and run it; }`
- `ExecutorService` take responsibility for the threads they create
  - User starts and shuts down `ExecutorService`
  - `ExecutorService` starts and shut down threads
- Method `execute(Runnable object)`
  - Accepts task as a Runnable type object that is executed by a thread in thread pool
- Method `shutdown()`
  - Thread pool terminates once all pre-submitted tasks are executed

© Vivek Kumar

`Executors` Class

- Provides factory and utility methods for ExecutorService
- Static method `newFixedThreadPool(int num_threads)`
  - Creates a thread pool that reuses a fixed number of threads for task execution

© Vivek Kumar

Let’s Revisit Our Parallel Array Sum

```
public class ArraySum implements Runnable {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
                                throws InterruptedException {
      int size; int[] array; //allocated (size) & initialized
      ExecutorService exec = Executors.newFixedThreadPool(2);
      ArraySum left = new ArraySum(array, 0, size/2);
      ArraySum right = new ArraySum(array, size/2, size);
      exec.execute(left); exec.execute(right);
      if(!exec.isTerminated()) {
          exec.shutdown();
          exec.awaitTermination(5L, TimeUnit.SECONDS);
      }
      int result = left.getResult() + right.getResult();
    }
}
```

- `ExecutorService` methods:
  - `isTerminated()`
    - Returns true if all tasks are terminated following the shutdown
  - `awaitTermination(long timeout, TimeUnit unit) throws InterruptedExecption`
    - Blocks until all tasks have completed execution after a shutdown request
- Important that you wait for all tasks to terminate after a shutdown request

© Vivek Kumar

Let’s Revisit Our Multithreaded Server

```
import java.io.*; import java.net.*;
import java.util.concurrent.*;
public class Server  {
    public static void main(String args[ ])
                  throws IOException {
        /* create a server socket
           bound to the specified port 1234 */
        ServerSocket me = new ServerSocket(1234);
        /* Server is now listening
           for incoming client’s request */
        ExecutorService exec =
Executors.newFixedThreadPool(2);
        while (true) {
           /* Connection is established */
           Socket connection = me.accept();
           System.out.println("Connected");
           Runnable task = new
ConnectionHandler(connection);
           /* new Thread(task).start(); */
           exec.execute(task);
        }
    }
}
```

- Rather than creating a new thread for every incoming client connection, we will instead create a new task and submit it to thread pool
  - No other changes to Server.java or Client.java
- Now our server will not go crazy even if several clients are lined up simultaneously

© Vivek Kumar

How to Improve Parallel Fibonacci?

- We know that there is a lot of parallelism and hence its not efficient to just create two tasks, i.e., one task for fib(n-1) and another task for fib(n-2)
- Every node in this tree can be computed in parallel
- Recursive divide and conquer application!

© Vivek Kumar

ForkJoinPool

- Designed to support a common need
  - Recursive divide and conquer pattern
  - For small problems (below cutoff threshold), execute sequentially
  - For larger problems
    - Define a task for each subproblem
    - Library provides
      - A Thread manager, called a ForkJoinPool
      - Methods to send your subtask objects to the pool to be run, and your call waits until they are done
      - The pool handles the multithreading well
- The “thread manager”
  - Used when calls are made to RecursiveTask’s methods fork(), invokeAll(), etc.

© Vivek Kumar

Parallel Fibonacci Using ForkJoinPool

```
import java.util.concurrent.*;

public class Fibonacci extends RecursiveAction {
    int n, result;
    public Fibonacci(int _n) { n=_n; }

}
```

- Step-1
  - `Fibonacci` class should extend the class `RecursiveAction`
  - `RecursiveAction` represents a task that doesn't return any result

© Vivek Kumar

Parallel Fibonacci Using ForkJoinPool

```
import java.util.concurrent.*;

public class Fibonacci extends RecursiveAction {
    int n, result;
    public Fibonacci(int _n) { n=_n; }

    public void compute() {
        if(n<2) {
            this.result = n;
            return;
        }
        Fibonacci left = new Fibonacci(this.n-1);
        Fibonacci right = new Fibonacci(this.n-2);




    }





}
```

- Step-2
  - Implement the method "`public void compute()`"
    - Similar to run() method
  - Computes the recursive divide and conquer task
  - Similar to Runnable implementation of Fibonacci, create the two tasks. One for calculating fib(n-1) while the other for calculating fib(n-2)

© Vivek Kumar

Parallel Fibonacci Using ForkJoinPool

```
import java.util.concurrent.*;

public class Fibonacci extends RecursiveAction {
    int n, result;
    public Fibonacci(int _n) { n=_n; }

    public void compute() {
        if(n<2) {
            this.result = n;
            return;
        }
        Fibonacci left = new Fibonacci(this.n-1);
        Fibonacci right = new Fibonacci(this.n-2);
        left.fork();



    }



}
```

- Step-3
  - Start the first task (“left”) **asynchronously**
  - Calling the fork() method on one of the task is similar to calling start() on a thread.
  - However, fork() does not start any new thread but rather adds this task to the task pool
    - Similar to calling execute() from ExecuterService

© Vivek Kumar

Parallel Fibonacci Using ForkJoinPool

```
import java.util.concurrent.*;

public class Fibonacci extends RecursiveAction {
    int n, result;
    public Fibonacci(int _n) { n=_n; }

    public void compute() {
        if(n<2) {
            this.result = n;
            return;
        }
        Fibonacci left = new Fibonacci(this.n-1);
        Fibonacci right = new Fibonacci(this.n-2);
        left.fork();
        right.compute();


    }





}
```

- Step-4
  - Start the second task (“right”) **sequentially**, i.e. on the current thread
  - Why not start this also with fork() ?
    - Not an error and you can definitely do so
    - However, the current thread is already done with current task (the compute() method) hence it can be reused to directly compute the “right” task

© Vivek Kumar

Parallel Fibonacci Using ForkJoinPool

```
import java.util.concurrent.*;

public class Fibonacci extends RecursiveAction {
    int n, result;
    public Fibonacci(int _n) { n=_n; }

    public void compute() {
        if(n<2) {
            this.result = n;
            return;
        }
        Fibonacci left = new Fibonacci(this.n-1);
        Fibonacci right = new Fibonacci(this.n-2);
        left.fork();
        right.compute();
        left.join();
        // add right.join() here if right.fork() is used
        // instead of right.compute()
        this.result = left.result + right.result;
    }



}
```

- Step-5
  - Once the “right” task completes, compute method should wait for all the asynchronous tasks spawned inside it (i.e. “left” task)
  - left.join() is a blocking operation and will return only when “left” has terminated
    - Similar to thread.join() but this waits for a “task” to terminate rather than a “thread”
  - Sum the partial results

© Vivek Kumar

Parallel Fibonacci Using ForkJoinPool

```
import java.util.concurrent.*;

public class Fibonacci extends RecursiveAction {
    int n, result;
    public Fibonacci(int _n) { n=_n; }

    public void compute() {
        if(n<2) {
            this.result = n;
            return;
        }
        Fibonacci left = new Fibonacci(this.n-1);
        Fibonacci right = new Fibonacci(this.n-2);
        left.fork();
        right.compute();
        left.join();
        // add right.join() here if right.fork() is used
        // instead of right.compute()
        this.result = left.result + right.result;
    }
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool(2);
        Fibonacci task = new Fibonacci(40);
        pool.invoke(task);
        int result = task.result;
    }}
```

- Step-6
  - Create a ForkJoinPool type thread pool with fixed number of threads
  - Create the root task (see the binary tree representation for Fibonacci)
  - Add this root task in the **task pool**
    - `pool.invoke`
    - Blocking operation and doesn't return until all tasks are terminated
  - A free thread from thread pool will execute this task and recursively create new tasks that will in turn be added to the task pool

© Vivek Kumar

Using RecursiveTask<T> to Return Value

```
import java.util.concurrent.*;

public class Fibonacci extends RecursiveTask<Integer> {
    int n;
    public Fibonacci(int _n) { n=_n; }

    public Integer compute() {
        if(n<2) return n;

        Fibonacci left = new Fibonacci(this.n-1);
        Fibonacci right = new Fibonacci(this.n-2);
        left.fork();
        return right.compute() + left.join();
    }
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool(2);
        Fibonacci task = new Fibonacci(40);
        int result = pool.invoke(task);
    }
}
```

- RecursiveTask<T> is better suited in scenarios where there is a need to return results from each task (same return type for all tasks)
- Very minimal changes required to our Fibonacci program to use this feature

© Vivek Kumar

Performance of Our Parallel Fibonacci

Total Threads in ForkJoinPool

- Increasing the thread pool size decreases the execution time
  - 4 core processor

© Vivek Kumar

Too Many Tasks Hamper Performance

```
import java.util.concurrent.*;

public class Fibonacci extends RecursiveTask<Integer> {
    int n;
    static int threshold = 10;
    public Fibonacci(int _n) { n=_n; }
    private int sequential(int n) {
        if(n<2) return n;
        else return sequential(n-1) + sequential(n-2);
    }
    public Integer compute() {
        if(n<threshold) return sequential(n);

        Fibonacci left = new Fibonacci(this.n-1);
        Fibonacci right = new Fibonacci(this.n-2);
        left.fork();
        return right.compute() + left.join();
    }
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool(2);
        Fibonacci task = new Fibonacci(40);
        int result = pool.invoke(task);
    }
}
```

- Although, tasks are lightweight than threads, too many tasks can also hamper the performance
- Use some cut off in your application to stop creation of tasks beyond certain threshold
  - When computation become too small, stop creation of any new task
- Fibonacci on left even with a single thread will run significantly faster than the Fibonacci shown on slide-16

© Vivek Kumar

Thread Pool Shutdown

```
import java.util.concurrent.*;

public class Search extends RecursiveAction<...> {
    ......
    public void compute() {
        if(this.searchItemIsFound()) {
            pool.shutdownNow();
        }

        Search left = new Search(...);
        Search right = new Search(...);
        left.fork();
        return right.compute() + left.join();
    }
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool(2);
        Search task = new Search(..., pool);
        try {
            pool.invoke(task);
        }
        catch(CancellationException e) {
            System.out.println("Goal is found, pool
aborted");
        }
    }
}
```

- For some type of parallel applications (e.g., searching element in a huge array) you would like to stop creating tasks once the goal is found
- *public void shutdownNow()*
  - Stops everything, i.e., creation of new tasks, all running tasks and previously submitted tasks
  - Throws an unchecked exception CancellationException upon cancellation

© Vivek Kumar

Measures of parallel performance

- Speedup = $T_{serial}/T_{parallel}$

Fig. source: http://www.drdobbs.com/cpp/going-superlinear/206100542

Amdahl's Law

If 50% of your application is parallel and 50% is serial, you can't get more than a factor of 2 speedup, no matter how many processors it runs on.

Gene M. Amdahl

Amdahl's Law

T_Parallel = Tseq + T'par

With infinite processors, T'par ~ 0 (theoretically) => T_Parallel = Tseq

CSE201: Monsoon 2023
Advanced Programming

Design Patterns-2

Pattern: Facade

Facade Pattern

- **Facade:** a structural design pattern used to identifying a simple way to realize relationships between entities
- Provide a unified “interface” to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use

© Vivek Kumar

The Tale of a Call Center

```
class CallCenter {
    public void handleNetwork() { / *Some code */ }
    public void handleBilling() { /* Some code */ }
    public void handleRoaming() { /* Some code */ }
    public void handleAccount() { /* Some code */ }
    ......
}
```

```
public class Client {
    public static void main(String[] args) {
        CallCenter c = new CallCenter();
        c.handleNetwork();
        c.handleBilling();
        c.handleRoaming();
        c.handleAccount();
    }
}
```

- Call center wants cost cutting and employes only one agent for handling all customer issues
  - Result?
    - Overloaded employee and bad customer satisfaction!

© Vivek Kumar

A Better Call Center Using Facade

```
class CallCenter {
    NetworkTeam net;
    BillingTeam bill;
    RoamingTeam roam;
    AccountTeam account;
    public CallCenter() { /* initializations */ }
    public void handleCalls(int option) {
        switch(option) {
            case 1:
                net.handleNetwork();
                break;
            case 2:
                bill.handleBilling();
                break;
            .......
        }
    }
}
```

```
public class Client {
    public static void main(String[] args) {
        CallCenter c = new CallCenter();
        c.handleCalls(1);
        ......
    }
}
```

- Facade design to the rescue
  - Hiding the complexities of a large body of code by providing a simplified interface

© Vivek Kumar

Pattern: Template

Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses

Acknowledgements for slides on template pattern: Informatics 122 Software Design II, Bren School (UCI)

Let’s Build a Café Simulator

- Coffee
  - Boil Water
  - Brew Coffee in boiling water
  - Pour in cup
  - Add sugar and milk

Inheritance?

- Tea
  - Boil Water
  - Steep tea in boiling water
  - Pour in cup
  - Add sugar and lemon

© Vivek Kumar

Let’s See the Code

```
public abstract class Cafe {
    public void boilWater() {
        System.out.println(“Boil Water”);
    }
    public void pourInCup() {
        System.out.println(“Pour in Cup”);
    }
    public abstract void prepare();
}
```

```
public class Coffee extends Cafe {
    public void prepare() {
        boilWater();
        brewCoffee();
        pourInCup();
        addSugarAndMilk();
    }
    private void brewCoffee() {
        System.out.println(“Brew Coffee”);
    }
    private void addSugarAndMilk() {
        System.out.println(“Add Sugar and Milk”);
    }
}
```

```
public class Tea extends Cafe {
    public void prepare() {
        boilWater();
        steepTeaBag();
        pourInCup();
        addSugarAndLemon();
    }
    private void steepTeaBag() {
        System.out.println(“Steep Tea Bag”);
    }
    private void addSugarAndLemon() {
        System.out.println(“Add Sugar and Lemon”);
    }
}
```

- Do you see any issues here?
  - Similar algorithms in `prepare` !!
    - How about doing the following?
      - Replace brewCoffee() and steepTeaBag() with brew()
      - Replace addSugarAndMilk() and addSugarAndLemon() with addCondiments()

© Vivek Kumar

Template Pattern

- The Template Method pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses
- Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure
- Usage
  1. Define the algorithm in superclass and ensure that subclasses cannot change the structure of this algorithm
  2. Each step of the algorithm is represented by a method
  3. Steps (methods) handled by subclasses are declared abstract
  4. Shared steps (concrete methods) are placed in the superclass

The Fixed Code

```java
public abstract class Cafe {
    public void boilWater() {
        System.out.println("Boil Water");
    }
    public void pourInCup() {
        System.out.println("Pour in Cup");
    }
    // "final" ensures that the person preparing
    // the beverage sticks to the recipe of this
    // Café instead of generating his own
    public final void prepare() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }
    public abstract void brew();
    public abstract void addCondiments();
}
```

```java
public class Coffee extends Cafe {
    private void brew() {
        System.out.println("Brew Coffee");
    }
    private void addCondiments() {
        System.out.println("Add Sugar and Milk");
    }
}
```

```java
public class Tea extends Cafe {
    private void brew() {
        System.out.println("Steep Tea Bag");
    }
    private void addCondiments() {
        System.out.println("Add Sugar and Lemon");
    }
}
```


© Vivek Kumar

FACTORY METHOD

Factory Pattern

- **Factory**: A method or object whose primary purpose is to manage the creation of other objects (usually of a different type)
- **Problem:** Object creation is cumbersome or heavily coupled for a given client. Client needs to create but doesn't want the details.
- **Solution:** A helper method that creates and returns the object(s)
- In Factory pattern, we create objects without exposing the creation logic to the client. This logic is taken care of by the factory method, where we refer to the objects via their common interface’s variable.
- The factory method allows us to select the object creation code as per the information received.

It’s very useful in expansion

Let’s consider a logistics app that has been dealing with only trucks so far.

Adding a new class to the program isn’t that simple if the rest of the code is already coupled to existing classes.

You want to include ships now

Factory design pattern is very handy in such cases.

```java
interface Shape {//the common interface
    void draw();
}

class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw() method.");
    }
}

class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Square::draw() method.");
    }
}

class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Circle::draw() method.");
    }
}

class Triangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Triangle::draw() method.");
    }
}
```

```java
class ShapeFactory {
    //Factory Method: use getShape method to create the required object are to
refer to it as shape
    public Shape getShape(String shapeType){//factory methods
        if(shapeType == null){
            return null;
        } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
            return new Rectangle();
        } else if(shapeType.equalsIgnoreCase("SQUARE")){
            return new Square();
        } else if(shapeType.equalsIgnoreCase("CIRCLE")){
            return new Circle();
        } else if(shapeType.equalsIgnoreCase("TRIANGLE")){
            return new Triangle();
        }
        return null;
    }
}
```

```java
class Client {
    public static void main(String[] args) {
        // Client Code
        ShapeFactory shapeFactory = new ShapeFactory();
        //get an object of Triangle by passing the information "Triangle"
        Shape shape1 = shapeFactory.getShape("TRIANGLE");
        //call draw method of the shape created
        shape1.draw();
    }
}
```

ABSTRACT FACTORY

Let’s consider you have variants in your products

Abstract Factory Pattern

- Creating a super-factory that selects sub-factories.
- The idea is to create separate factories for Art Deco, Victorian and Modern for selecting their products, and then create one more factory for selecting the factories.

```java
interface Shape3 {
   void draw();}

class RoundRectangle implements Shape3 {
   @Override
   public void draw() {
      System.out.println("Inside RoundRectangle::draw() method."); }}

class RoundSquare implements Shape3 {
   @Override
   public void draw() {
      System.out.println("Inside RoundSquare::draw() method."); }}

class RoundTriangle implements Shape3 {
   @Override
   public void draw() {
      System.out.println("Inside RoundTriangle::draw() method."); }}

class SharpRectangle implements Shape3 {
   @Override
   public void draw() {
      System.out.println("Inside SharpRectangle::draw() method."); }}

class SharpSquare implements Shape3 {
   @Override
   public void draw() {
      System.out.println("Inside SharpSquare::draw() method."); }}

class SharpTriangle implements Shape3 {
   @Override
   public void draw() {
      System.out.println("Inside SharpTriangle::draw() method."); }}

class FlatRectangle implements Shape3 {
   @Override
   public void draw() {
      System.out.println("Inside FlatRectangle::draw() method."); }}

class FlatSquare implements Shape3 {
   @Override
   public void draw() {
      System.out.println("Inside FlatSquare::draw() method."); }}

class FlatTriangle implements Shape3 {
   @Override
   public void draw() {
      System.out.println("Inside FlatTriangle::draw() method."); }}
```

```java
interface AFx{
   public Shape3 FM(String s);
}

class FM4Round implements AFx{
   @Override
   public Shape3 FM(String shapeType){
      if(shapeType == null){
         return null;
      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new RoundRectangle();
      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new RoundSquare();
      } else if(shapeType.equalsIgnoreCase("TRIANGLE")){
         return new RoundTriangle();
      }
      return null;
   }

}

class FM4Sharp implements AFx{
   @Override
   public Shape3 FM(String shapeType){
      if(shapeType == null){
         return null;
      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new SharpRectangle();
      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new SharpSquare();
      }else if(shapeType.equalsIgnoreCase("TRIANGLE")){
         return new SharpTriangle();
      }
      return null;
   }

}

class FM4Flat implements AFx{
   @Override
   public Shape3 FM(String shapeType){
      if(shapeType == null){
         return null;
      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new FlatRectangle();
      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new FlatSquare();
      }else if(shapeType.equalsIgnoreCase("TRIANGLE")){
         return new FlatTriangle();
      }
      return null;
   }
}
```

3 Sub-factories

Super-factory

```
class FactorySelector{
   public AFx FM2(String s){
      if(s == null){
         return null;
      } else if(s.equalsIgnoreCase("ROUND")){
         return new FM4Round();
      } else if(s.equalsIgnoreCase("SHARP")){
         return new FM4Sharp();
      } else if(s.equalsIgnoreCase("FLAT")){
         return new FM4Flat();
      }
      return null;
   }

}
```

```
class Client {

   public static void main(String[] args) {
      // write your code here
      FactorySelector fs = new FactorySelector();
      AFx af2=fs.FM2("FLAT");
      Shape3 shape1=af2.FM("TRIANGLE");

      //call draw method of Circle
      shape1.draw();

   }
}
```

Pattern: Decorator

objects that wrap around other objects to add useful features

Source of this slide: CSE331, Washington University

Remember this from IO Streams?

```
public static void main(String args[])
                          throws IOException
{
    Scanner in = null;
    PrintWriter out = null;
    try {
        in = new Scanner( new BufferedReader( new
                              FileReader("input.txt")));
        out = new PrintWriter( new
                              FileWriter("output.txt"));
        while (in.hasNext()) {
            out.println(in.next());
        }
    } finally {
        if (in != null)
            in.close();
        if (out != null)
            out.close();
        }
}
```

- We saw this example in Lecture on I/O Streams of combining three classes for breaking input into tokens:
  - Scanner
  - BufferedReader
  - FileReader
- Normal InputStream class has only public int read() method to read one letter at a time
- BufferedReader or Scanner add additional functionality to read the stream more easily
  - Here, BufferedReader and Scanner are examples of Decorator objects

© Vivek Kumar

Decorator pattern

- **Decorator**: an object that modifies behavior of, or adds features to, another object
  - Helps in adding features to an existing simple object without needing to disrupt the interface that client code expects when using the simple object

Decorator Pattern: Vehicle Paint Shop

```
interface Vehicle {
    public void paint();
}
```

```
class Bike implements Vehicle {
    public void paint() {
        System.out.println("White color Bike");
    }
}
```

```
class Car implements Vehicle {
    public void paint() {
        System.out.println("White color Car");
    }
}
```

```
//Abstract to disallow clients to instantiate it
abstract class VehicleDecorator implements Vehicle {
    private Vehicle decoratedVehicle;
    public VehicleDecorator(Vehicle v) {
        this.decoratedVehicle = v;
    }
    public void paint() {
        decoratedVehicle.paint();
    }
}
```

```
class BlueVehicleDecorator extends VehicleDecorator {
    public BlueVehicleDecorator(Vehicle v) {
        super(v);
    }
    public void paint() {
        super.paint();
        System.out.println("Now painted in Blue color");
    }
}
```

```
public class Client {
    public static void main(String[] args) {
        Vehicle c1 = new Car();
        c1.paint(); // default white paint
        Vehicle c2 = new BlueVehicleDecorator(new Car));
        c2.paint(); // painted in blue color
        ......
    }
}
```

© Vivek Kumar

Pattern: Composite

objects that can contain their own type

Source of this slide: CSE331, Washington University

Composite Pattern

- An object that can be either an individual item or a collection of many items
  - Can be composed of individual items or other composites
  - Recursive definition: Objects that can hold themselves

Employee Hierarchy

```
interface Employee {
    public void print();
}
```

```
class Manager implements Employee {
    List<Employee> emp = new ArrayList<Employee>();
    public void add(Employee e) { emp.add(e); };
    public void remove(Employee e) { emp.remove(e); }
    public void print() {
        System.out.println("Manager");
        for(Employee e : emp) {
            e.print();
        }
    }
}
```

```
class Developer implements Employee {
    public void print() {
        System.out.println("Employee");
    }
}
```

```
public class Client {
    public static void main(String[] args) {
        Employee gm = new Manager();
        Employee emp1 = new Developer();
        Employee manager = new Manager();
        Employee emp2 = new Developer();
        Employee emp3 = new Developer();
        gm.add(emp1); gm.add(manager);
        manager.add(emp2); manager.add(emp3);
        gm.print(); // print all nodes in tree above
    }
}
```

- Composite pattern helps client to ignore the difference between individual objects and allow him to treat all objects in the composite structure uniformly

Source of this figure: https://dzone.com/articles/composite-design-pattern-java-0

© Vivek Kumar