**Inheritance**

* Develop a new class from an existing class
* Existing class is called the parent class, super class, or base class
* New class is called derived class, subclass, or child class
* The child class inherits the methods and data define for parent class
* Software reuse is the main benefit of inheritance

**Inheritance Implementation**

bbbbb

* Inheritance creates an is-a relationship meaning the child class is a parent class.
* Deriving the subclass

Class Car extends Vehicle{}

* Access Modifiers
  + Visibility modifiers (private, protected, public) determines which class members are inherited and which are not
  + Public modifier methods or data are inherited
  + Private modifier methods and data are not inherited
  + Protected modifier method or data are inherited but the object of the subclass cannot access the protected methods and modifiers
* UML diagram

|  |
| --- |
| Book |
| * Author: string |
| # pages: int |
| + display: void |

Class name: Book  
- private # protected + public

**Single Inheritance**

* A subclass can be derived from one parent class
* No multiple inheritance in Java  
  A subclass cannot have more than one parent class

**Concrete class**

* Can instantiate concrete class

**Abstract class**

* Use as a base class in inheritance
* Cannot instantiate
* Must have at least 1 abstract method
  + - Declare in abstract class
    - Ex: public abstract double calculateArea();
    - Defined in subclass which is derived from the abstract class
    - Ex: public double calculateArea() // no keyword abstract

{

…

}

**Inheritance design example**

public abstract class Shape

{

private double xPos;

private double yPos;

Shape()

{

xPos = 0;

yPos = 0;

}

Shape(double x, double y)

{

xPos = x;

yPos = y;

}

public final double get xPos() // final means cannot be overrided in subclass

{

return xPos;

}

public final double get yPos()

{

return yPos;

}

public void moveTo(double x, double y)

{

xPos = x;

yPos = y;

}

public String toString()

{

return “x=” + xPos + “y=” + yPos;  
}

public abstract double computeArea();

} // end class Shape

public class Circle extends Shape

{

Private double radius; // can also be protected so subclass can inherit it

// Constructor is NOT inherited to the subclass

Circle()

{

Shape(); // cannot do this

super(); // calls Shape(). Must be in the first line of the subclass constructor

radius = 0.0;

}

Circle(double x, double y, double r)

{

super(x, y);

radius = r;

}

public double getRadius()

{

return radius;

}

public double computeArea()

{

return radius \* radius \* Math.PI;

}

public String toString() //Overidding the method toString from the class Shape

{return super.toString() + “radius =” + radius; // super.toString calls the method toString from the //class Shape

}

} //end class Circle

**Overriding method**

Return type, argument, etc. are the same but implemented differently

**Overloading method**

Argument is different but implementation is same

A class is derived from the class Object by default. The class Object has many public methods like toString().

public class Cylinder extends Circle

{

// x, y, z, radius, height

// Only z and height must be defined in Cylinder

// x and y are from Shape, radius is from the class circle

private double zPos;

private double height;

Cylinder()

{

super(); // calls circle, whose own super() calls shape

//and then goes back to circle and then cylinder

zPos = 0.0;

height = 0.0;

}

Cylinder(double x, double y, double z, double r, double h)

{

super(x, y, r);

height = h;

zPos = z;

}

// override moveTo

public void moveTo(double x, double y, double z)

{

super.moveTo(x, y);

zPos = z;

}

public double getHeight()

{

return height;  
}

public double computeArea() // surface area

{

// (2 \* circle area) + (circumference of circle \* height)

Return (2 \* super.computeArea() ) + (2\* Math.PI \* getRadius() \* height);

// use getRadius() accessor instead of radius because radius is private in the class Circle

}

public double computeVolume()

{

return super.computeArea() \* height;

}

} // end class Cylinder

public class Rectangle extends Shape

{

private double length;

private double width;

// etc….

}

**Polymorphism**

* In inheritance structure
* Same thing but behaves differently

**Implement test driver**

main()

{

Shape [] s = new Shape[6]; // an array of references

//this method…

Circle c1 = new Circle(\_);

s[0] = c1;

s[1] = new Circle(\_);

Cylinder cl1 = new Cylinder(\_);

s[2] = cl1;

s [3] = new Rectangle(\_);

}

// compute the area of all shapes

double totalArea = 0.0;

for (int i = 0; i < s.length; i++)

totalArea +=s[i].computeArea();

// compute area of all circles

double totalAreaCircle = 0.0;

For (int i = 0; i < s.length; i++)

If (s[i] instanceOf Circle) // Java operator instanceOf

totalAreaCircle += s[i].computeArea();

// compute total volume of all cylinders

Double totalVolume = 0.0;

For (int i = 0; i < s.length; i++)

If (s[i] instanceOf Cylinder)

totalVolume = totalVolume +( (Cylinder) s[i]). computVolume()

**Sorting an Array**

String fruits[ ]= {“Pineapple”, “Orange”, “Apple”}

**Arrays.sort**(fruits);

for(string temp: fruits)

System.out.println(temp);

**Sorting an Arraylist**

List <String> fruits = new ArrayList<String>();

fruits.add(“Pineapple”);

fruits.add(“Orange”);

fruits.add(“Apple”);

**Collection.sort**(fruits);

//Use same for loop to display the result\*

**Sorting an object with Comparable Interface**

* Comparable interface

public int compareTo(Object o)

{

}

* Use the interface Comparable  
  class Fruit

{

private String fruitName;

private int quantity;

Define methods mutators and accessors

}

**Problem**

In main

Fruit[ ] fruits = new Fruit[3];

Fruits[0] = new Fruit(“Pineapple”, 10);

Fruit apple = new Fruit(“Apple”, 5);

Fruits[1] = apple;

Fruits[2] = new Fruit(“Orange”, 12);

Arrays.sort(fruits); //sort by fruit name.  
Solution: Modify the class Fruit to implement the comparable interface

public class Fruit implements Comparable<Fruit> //Comparable interface is generic

{ //Define the method compareTo

public int compareTo(Fruit o)

{

String fruitNameO = o.getFruitName();

Return fruitName.compareTo(fruitNameO);//switch fruitName & fruitNameO and it’ll sort in descending order

}//Can only use the interface Comparable ONCE, so if you want to sort in ascending/descending order of quantity you’d need to use the Comparator interface

} //Comparable has method compareTo, Comparator has method compare

**Using the Comparator interface**

* public interface Comparator<Object o1, Object o2> //Passing two objects

{

public compare(Object o1, Object o2)

}

* In main

{

Arrays.sort(fruits, Fruit.FruitQuantityAscending)//Sort by quantity

}

public class Fruit implement Comparable<Fruit>

{ //create a static method to compare the quantity in ascending order

public static Comparator<Fruit> FruitQuantityAscending = new Comparator <Fruit>()

{

public int compare(Fruit o1, Fruit o2)

{

return o1.quantity – o2.quantity //If you reverse o1 & o2 it sorts in descending

}

};

}

//You might want to implement Comparator in another class

public class FruitQuantityAscending implement Comparator<Fruit>

{

public int compare(Fruit o1, Fruit o2)

{

return o1.quantity – o2.quantity;

}//If you are in a diff class and want to reference a private int use “get” to access it

}

**In main**

List<Fruit> fruitList = new ArrayList<Fruit>();

fruitlist.add(“Pineapple”);   
………………  
Collection.sort(fruitList, new FruitQuantityAscending());

**Java Interface**

* public interface interfaceName

{

constant variables

Abstract public method declaration (no keyword abstract)

}

* No multiple inheritance (B and C from A)
* It’s ok to have multiple interface

public class D extends A implements B, C //B and C are interfaces

{

}

**Composition**

* Composition is a has-a relationship
* Relationship between Car and Engine? Composition, Car has an engine so making engine a subclass of car wouldn’t make sense.

**Implement a Composition**

public class Book

{

private String code;

private String title;

double price;

public book(c, t, p)

{  
 ………

}

mutators and accessors

}

public class BookOrder

{

private Book book; //Composition

private double quantity;

private double total;

public BookOrder(c , t, p, q, to)

{

book = new Book(c, t, p);

quantity = q;

total = to;

}

public Book getBook()

{

return book;

}

public void setBook(Book b)

{

book = b;

}

public void setTotal()

{

total = quantity\*b.getPrice();

}

}

**Clone Object**

* Copy an object
* C2 = C1; //No Clone

**Copy Constructor**

public Circle(Circle C1)

{

radius = C1.radius;

}

In Main

Circle ca = new Circle(\_\_\_);

Circle cb = new Circle(ca) //You cannot declare the same object more than once (within the scope)

**Clone Method**

* Shallow copy – No Composition
* Deep copy – Composition

**Clone Book Object**

* Book b1 = new Book();

Book b2 = (Book)b1.clone(); //Java method from the interface Cloneable, clone always returns //type Object, need to cast it with (Book)

**Modify the class Book**

* Public class Book implements Cloneable

{

…

public Object clone() throws CloneNotSupportedException //Define the clone method

{

return super.clone(); //Shallow Copy, shares same memory location

}

}

**Clone BookOrder Object**

* In Main

BookOrder b01 = new BookOrder(\_\_\_);

BookOrder b02 = (BookOrder)b01.clone();

**Implement a deep copy on the class BookOrder**

* Public class BookOrder implements Cloneable

{

…

//Shallow copy

public Object clone() throws CloneNotSupportedException

{

return super.clone();

}

//Deep Copy

public Object clone() throws CloneNotSupportedException

{

//Clone primitive type members

BookOrder bO = (BookOrder)super.clone();

//Clone the object

Book b = (Book)book.clone();

bO.setBook(b); //from class BookOrder

return bO;

}

}

**Upcasting and Downcasting**

Suppose that we have the following class hierarchy:

**Mammal > Animal > Dog, Cat**

Mammal is the super interface

public interface Mammal {

    public void eat();

    public void move();

    public void sleep();

}

Animal is the abstract class:

public abstract class Animal implements Mammal {

    public void eat() {

        System.out.println("Eating...");

    }

    public void move() {

        System.out.println("Moving...");

    }

    public void sleep() {

        System.out.println("Sleeping...");

    }

}

Dog and Cat are the two concrete sub classes:

    public void bark() {

        System.out.println("Gow gow!");

    }

    public void eat() {

        System.out.println("Dog is eating...");

    }

}

public class Cat extends Animal {

    public void meow() {

        System.out.println("Meow Meow!");

    }

}

**What is Upcasting?**

***Upcasting***is casting a subtype to a supertype, upward to the inheritance tree. Let’s see an example:

Dog dog = new Dog();

Animal anim = (Animal) dog;

anim.eat();

Here, we cast the Dog type to the Animal type. Because Animal is the supertype of Dog, this casting is called upcasting.

Note that the actual object type does not change because of casting. The Dog object is still a Dog object. Only the reference type gets changed. Hence the above code produces the following output:

Dog is eating…

Upcasting is always safe, as we treat a type to a more general one. In the above example, an Animal has all behaviors of a Dog.

This is also another example of upcasting:

Mammal mam = new Cat();

Animal anim = new Dog();

**Why is Upcasting?**

Generally, upcasting is not necessary. However, we need upcasting when we want to write general code that deals with only the supertype. Consider the following class:

public class AnimalTrainer {

    public void teach(Animal anim) {

        anim.move();

        anim.eat();

    }

}

Here, the teach() method can accept any object which is subtype of Animal. So objects of type Dog and Cat will be upcasted to Animal when they are passed into this method:

Dog dog = new Dog();

Cat cat = new Cat();

AnimalTrainer trainer = new AnimalTrainer();

trainer.teach(dog);

trainer.teach(cat);

**What is Downcasting?**

***Downcasting***is casting to a subtype, downward to the inheritance tree. Let’s see an example:

|  |  |
| --- | --- |
|  | Animal anim = new Cat();  Cat cat = (Cat) anim; |

Here, we cast the Animal type to the Cat type. As Cat is subclass of Animal, this casting is called downcasting.

Unlike upcasting, downcasting can fail if the actual object type is not the target object type. For example:

|  |  |
| --- | --- |
|  | Animal anim = new Cat();  Dog dog = (Dog) anim; |

This will throw a ClassCastException because the actual object type is Cat. And a Cat is not a Dog so we cannot cast it to a Dog.

The Java language provides the **instanceof** keyword to check type of an object before casting. For example:

|  |  |
| --- | --- |
|  | if (anim instanceof Cat) {      Cat cat = (Cat) anim;      cat.meow();  } else if (anim instanceof Dog) {      Dog dog = (Dog) anim;      dog.bark();  } |

So if you are not sure about the original object type, use the instanceof operator to check the type before casting. This eliminates the risk of a ClassCastException thrown.

**Why is Downcasting?**

Downcasting is used more frequently than upcasting. Use downcasting when we want to access specific behaviors of a subtype.

Consider the following example:

|  |  |
| --- | --- |
|  | public class AnimalTrainer {      public void teach(Animal anim) {          // do animal-things          anim.move();          anim.eat();            // if there's a dog, tell it barks          if (anim instanceof Dog) {              Dog dog = (Dog) anim;              dog.bark();          }      }  } |

Here, in the teach() method, we check if there is an instance of a Dog object passed in, downcast it to the Dog type and invoke its specific method, bark().

Okay, so far you have got the nuts and bolts of upcasting and downcasting in Java. Remember:

* Casting does not change the actual object type. Only the reference type gets changed.
* Upcasting is always safe and never fails.
* Downcasting can risk throwing a ClassCastException, so the instanceof operator is used to check type before casting.

**Exception**

* an object that represents an error or condition that prevents execution from proceeding normally
* if not handled, the program will terminate abnormally

**Exception Handling Overview**

Scanner input = \_\_\_\_

int x = input.nextInt();

int y = input.nextInt();

System.out.println(x/y);

In this situation, an exception may occur if:

y has a value of 0 (System throws an ArithmeticException object, and program terminates)

**How do we handle an exception?**

* First method // not very efficient, and doesn’t tell us much information

if (y != 0)

System.out.println(x/y);

else

System.out.println(“y cannot be 0!”);

* **Second method** // try and catch block. A try ALWAYS needs a catch

try // monitor the system and try to find an error

{

if (y ==0)

throw new ArithmeticException(“Divisor cannot be zero.”);

System.out.println(x/y);

}

catch (ArithmeticException e)

{

System.out.print(e.message); // return the error message

}

System.out.println(“Execution continue!”);

**Example**

public class Quotient

{

public static int quotient(int x, int y)

{

If (y ==0)

Throw new ArithmeticException(“…”); // client left to deal with exception

return x/y;

}

}

public class Test

{

main

{

// input x and y here

try // monitor exceptions in this portion of the program

{

int r = Quotient.quotient(x,y);

}

catch (ArithmeticException e)

{

System.out.print(e.message);

}

System.out.println(“Execution continues…”);

}

}

**FileNotFoundException**

Scanner input = \_\_\_;

String filename = input.nextLine();

try

{

Scanner inputFile = new Scanner(new File(filename)); // might throw FileNotFileException

// processing file…

}

Catch (FileNotFoundException e)

{

… // whatever you want

}

**InputMismatchException** // ex: if program asks users for int, but they input String

Scanner input = \_\_\_;

Boolean continueInput = true;

do

{

try

{

System.out.println(“Enter an integer”);

int n = input.nextInt(); // might throw InputMisMatchException

System.out.println(n);

continueInput = false;

}

Catch (InputMismatchException e)

{

System.out.println(“Incorrect input: must be an integer”);

Input.nextLine(); // make sure to discard input

}

} while (continueInput);

**Multiple Exceptions**

Catch the more specific exceptions (subclasses) before the general ones.

**Exception Type**

Object 🡨 Throwable 🡨 Error

🡨 Exception

**Three major types of exceptions**

1. **System errors**
   1. Thrown by JVM (Java Virtual Machine)
   2. Represented in error class
   3. Rarely occurs, may happen if computer runs out of resources
   4. Not much you can do to fix it
2. **Exceptions**
   1. Represented in exception class
   2. Describes errors caused by your program and by external circumstances
   3. Ex: Exception 🡨 ClassNotFoundException or IOException
   4. Can be caught and handled by your program
3. **Runtime Exception**
   1. Generally thrown by JVM
   2. Represented in RunTimeException class
   3. Describes programming errors, like bad casting, accessing an out-of-bounds array, and numeric errors
   4. Exception 🡨 RunTimeException 🡨 IllegalArgumentException or IndexOutOfBoundException

* RuntimeException, Error, and their subclasses are known as **unchecked exceptions**.
  + In most cases, unchecked exceptions reflect logic errors in programming and are unrecoverable.
  + Ex: IndexOutOfBoundException, NullPointerException
* All other exceptions are known as **checked exceptions**.
* Java does not mandate that you write code to catch or declare unchecked exceptions, but you must write code to catch or declare checked exceptions.

**Catch or declare checked exceptions**

void p1() // compiler error

{

p2(); // may throw checked exception (Ex: IOException)

// The checked exception must be handled.

}

**Solutions:**

1. **Catch exception**  
   void p1()

{

try

{

p2();

}

catch (IOException e)

{

…

}

}

1. void p1() throws IOException // However, you should try catching exceptions  
   {

p2();

}

**Catching Exceptions**

Exception 🡨 Runtime Exception 🡨 etc…

Main

{

try

{

Invoke method 1;

Statement 1;

}

catch (exception1 e)

{ … }

Statement 2;

}

Method 1

{

try

{

Invoke method 2;

Statement 3;

}

catch (exception2 e)

{ … }

Statement 4;

}

Method 2

{

try

{

Invoke method 3; // exception thrown in method 3

Statement 5;

}

catch (exception3 e)

{ … }

Statement 6;

}

* if the exception type is exception 3

execute catch (Exception3 e)

{…}

Statement 6;

* if exception type is exception 2

method 2 is aborted

execute catch (exception2 e)

{…}

Statement 4;

* if exception type is exception 1

method 1 is aborted

execute catch (exception1 e)

{…}

Statement 2;

* if exceptions not caught, whole program is terminated

**Getting information from exceptions**

Exception class

* String getMessage() – returns message of Exception object
  + Ex: throw new IOException(“Invalid I/O”);  
    catch (IOException e)

{

System.out.println(e.getMessage); // Invalid I/O  
}

* String toString() – return a String
  + Full name of Exception class + getMessage()
* getStackTrace() – returns an array of stack trace elements

**Declaring, throwing, and catching exceptions in your own classes**

public class Circle

{

private double radius;

public circle (double r)

{

setRadius(r);

}

public void setRadius(double nradius) throws IllegalArgumentException

{

if (nradius >= 0)

radius = nradius;

else

throw new IllegalArgumentException (“Radius >= 0”);

}

}

Main()

{

try

{

Circle c1 = new Circle(-1.0);

}

Catch (IllegalArgumentException e)

{…}

}

**finally Clause**

try {…}

catch {…}

finally {…} // The code in the finally clause is executed whether an exception occurs in the try block //or not.

Example:

try

{

File inputFile = new File(“data.txt”);

}

catch (IOException e)

{

…

}

finally

{

if (inputFile != null)

inputFile.close();

}

**File Processing**

import java.io.\*;

import java.nio.file;  
  
// create file object

File inputFile = new File(“input.txt”); // or specify file path: File(“C:\\cecs277\\lab3\\input.txt”);

// create scanner object

Scanner input = new Scanner (inputFile);

Scanner methods: Next(), Nextline(), nextDouble(), hasNextLine()

Path path = null; // return filepath;

File file = null;

Path = Paths.get(“Project.txt”);

File = path.toFile();

Reading data from a file

try

{

If (!Files.exist(path))

Files.createFile(path);

}

BufferedReader in = new BufferedReader(new FileReader(file)); // choose a delimiter

String line = in.readLine();

while (line != null)

{

String tokenizer t = new StringTokenizer(line, “\t”); // delimiter

String lastName = t.nextToken();

String firstName = t.nextToken();

String sHourlyRate = t.nextToken(); // must convert to double

// Convert sHourlyRate to double

double hourlyRate = Double.parseDouble(sHourlyRate);

Line = in.readLine();

// etc… Add these to a List collection using static method

}

* catch exceptions in the main method, like FileNotFound, etc.

**Text File output**

PrintWriter out = new PrintWriter(“out.dat”); // if file does not exist, it creates file. If file // exists, file //is cleared

PrintWriter methods: print(), println(), printf() format output

Input.close(); // make sure to close file so that it can be accessed by another program

***Classifying characters***

char c;

Character.isDigit(c);

Character.isLetter(c);

String s = input.nextLine();

int i = 0;

while (!Character.isDigit(s.charAt(i)))

{

i++;

}