Printing output

* System.out.print(“Hello”); → Next print starts in the same line
* System.out.println(“Hello”); → Next print starts in another line
* System.out.print(“Hello\n”); → Next print starts in another line
* System.out.println(“Hello\n”); → Next print after two lines
* Can’t use ‘’ with print, Always use “”

Primitive data types

* Integers - byte, short, int, long
* Floating point numbers - float, double
* Characters - char
* Boolean - boolean
* Literals enclosed with double quotes “” are strings, whereas single quoted literals ‘’ are chars. You can’t interchange chars and strings
* 1 byte = 8 bits

Declaring a variable

* int a;
* int a = 10;
* int a, b;
* int a = 10, b = 20;
* Variable names can’t start with a number
* int weight1 = 6;

System.out.println(“Weight of 1 = ” + weight1); → Weight of 1 = 6

Array

* int weight[] = new int[4]; → Size of the array = 4

weight [0] = 6;

weight[1] = 3;

weight[2] = 2;

weight[3] = 5;

System.out.println(“Weight of 1 = ” + weight[0]); → Weight of 1 = 6

* int weight[] = {6, 3, 2, 5};

System.out.println(“Weight of 1 = ” + weight[0]); → Weight of 1 = 6

* int weight[] = {6, 3, 2, 5};

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

System.out.println(weight[i]); → 6 3 2 5

}

* int weight[] = {6, 3, 2, 5};

int total = 0;

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

total = total + weight[i];

}

System.out.println(“Total = ” + total); → Total = 16

Character Array

* Use ‘ instead of “
* char name[] = {‘D’, ‘i’, ‘l’, ‘s’, ‘h’};

System.out.println(name); → Dilsh

System.out.println(name.length); → 5

System.out.println(name[2]); → l

System.out.println(“I am ” + new String(name)); → I am Dilsh

String Data Type

* Remember to use String instead of string
* Use “” instead of ‘’
* String name = “Dilshan”;

System.out.println(name); → Dilshan

System.out.println(“I am ” + name); → I am Dilshan

System.out.println(name.length()); → 7

System.out.println(name.charAt(2)); → l

Two Dimensional Array

* Chess board is a two dimensional array
* int name[][] = new int[4][4]; → Creates 4x4 two dimensional array (Matrix)

name[1][1] = 1;

name[1][2] = 2;

name[1][3] = 3;

System.out.println(name[1][2]); → Shows the 3rd row 2nd column item → 2

Three Dimensional Array

* Rubik cube is a three dimensional array
* int name[][][] = new int[3][3][3] → Creates 3x3 three dimensional array

Operators

* Arithmetic operators (+, -, \*, /, %, ++, --, +=, -=, /=, \*=, %=)
* Assignment operators (=)
* Relational operators (==, !=, >, <, <=, >=)
* Boolean logical operators (!, &&, ||)
* Ternary (conditional) operator (?)

Arithmetic operators

* int x = 6;

int y = 2;

int z = 2;

int result = x - y/z;

System.out.println(result); → Division happens first → 5

* int x = 5;

int y = 2;

int result = x/y;

System.out.println(result); → Integer division → 2

result = x%y;

System.out.println(result); → Outputs the remainder → 1

* int x = 5;

int y = 0;

x = x + 1;

System.out.println(x); → 6

x++;

System.out.println(x); → 7

y = x++;

System.out.println(x); → 9

System.out.println(y); → 8

y = ++x;

System.out.println(x); → 10

System.out.println(y); → 10

Relational operators

* int x = 4;

int y = 5;

boolean compare = (x == y);

System.out.println(compare); → false

System.out.println(x == y); → false

System.out.println(x != y); → true

Boolean operators

* Use && and || instead & and |. They are used as bitwise operators while && and || are boolean operators
* boolean x = true;

boolean y = false;

boolean compare;

compare = x && y;

System.out.println(compare); → false

Ternary operator

* Condition ? Expression1 : Expression2
* int age = 20;

boolean canVote;

canVote = (age >= 18) ? true : false;

System.out.println(canVote); → true

* int age = 20;

String canVote;

canVote = (age >= 18) ? “Yeah” : ”No”;

System.out.println(canVote); → Yeah

Pseudocode

* Begin

Assign age to variable “x”

If “x” is greater than or equal to 18,

Print “Voter”

Otherwise,

Print “Nonvoter”

End

Control statements

* Selection
  + If
    - Ternary operator substitute
    - int age = 20

boolean canVote;

canVote = (age >= 18) ? true : false;

System.out.println(canVote); → true

if (age >= 18)

canVote = true;

else

canVote = false;

System.out.println(canVote); → true

* + - Parantheses in ternary operator is optional while in if statement it’s mandatory
    - Use curly braces when multiple statements are included inside the if statement. Better to use them as a habit all the time
    - int x = 23;

if (x >= 18) {

System.out.println(“Voter”);

} else {

System.out.println(“Nonvoter”);

}

* int month = 1;

String name;

if (month == 1) {

name = “January”;

}

else if (month == 2) {

name = “February”;

}

else {

name = “Not January or February”;

}

* + Switch
    - Remember to use break with every case. With ‘default’ it’s optional
    - int month = 1;

String name;

switch (month) {

case 1:

name = “January”;

break;

case 2:

name = “February”;

break;

default:

name = “Not Jan or Feb”

}

System.out.println(name);

* Iteration/loop
  + For loop
    - int weight[] = {6, 3, 2, 5, 6, 8, 7, 1};

int total = 0;

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

total += weight[i]

}

System.out.println(“Total = ” + total); → Outputs the total → 38

* While loop
  + int i = 1;

while (i <= 5) {

System.out.println(i); → 1 2 3 4 5

i++;

}

* + int f=1, i=2;

while(++i<5)

f\*=i;

System.out.println(f); → Prints “12”

* Do-while loop
  + Here, even the initial condition is false, the statements will be executed at least once
  + Remember to add semicolon (;) after while condition
  + Initialization and iteration has to be done separately unlike for loop
  + int i = 1;

do {

System.out.println(i);

i++;

}

while (i <= 5);

* Nested loop
  + Loops inside loops
  + int n = 5;

for (int i = 0; i < n; i++) {

for (int j = 0; j <= i; j++) {

System.outprint(“\* ”);

}

System.out.print(“\n”); → System.out.println(“”); also can be used

}

* Enhanced for loop
  + int[] nums = {11, 4, 18, 73, 65}; → int nums[] or int []nums can be used

int sum = 0;

for (int n: nums) {

sum += n;

}

System.out.println(sum);

* Jump
* ‘main’ method in the program should always be ‘static’

Switch statement without break

* Here, break statement is not used
* int day = 2;

String dayType = switch(day) {

case 1, 2, 3, 4, 5 -> “Working day”;

case 6, 7 -> “Weekend”;

default -> “Invalid day”;

} ; → Remember this semicolon

System.out.println(dayType); → Working day

Inputing data to Java code

* import java.util.Scanner;

public class Main {

public static void main(String[] args) {

Scanner inputData = new Scanner(System.in);

int number = inputData.nextInt();

}

}

* When ‘nextLine()’ method is used after ‘nextInt()’ method, remember that you need to put ‘.nextLine()’ to avoid Scanner taking new line “\n” as an input
* import.java.util.\*;

public class Main {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int a = scanner.nextInt();

scanner.nextLine(); → Use this when ‘String’ is being read after ‘int’ for

exception handling

String b = scanner.nextLine();

}

}

Iteration/loops

* break can be used to break the loop
* continue can be used to skip to next iteration

For loop

* It’s best to know start and end numbers when for loop is used
* for(int x = 0; x <= 10; x = x+2) {

System.out.println(x); → Prints even numbers between 0 and 10

}

Do-while loop

* At least run one time
* int x =1;

do {

System.out.println(x);

x++;

}

while (x < 5);

Enhanced for loop/for each loop

* Used to traverse elements in arrays
* int prime[] = {2, 3, 5, 7};

for (int t: primes) {

System.out.println(t); → Prints elements of the array → 2 3 5 7

}

Two dimensional array

* First index represents rows while the other one represents columns. These are called row index and column index
* int sample[][] = {{1, 2, 3}, {4, 5, 6}};
* All array members must be of the same data type
* Arrays can be nested within arrays to as many levels as your program needs

Object oriented programming (OOP)

* Each object is independent unit with a unique identity
* Four principles of OOP
  + Encapsulation
  + Abstraction
  + Inheritance
  + Polymorphism
* Characteristics of objects are called attributes
* Each object has 3 dimensions
  + Identity
  + Attributes - Current state of the object
  + Behavior - Capability of the object
* Class describe what the object will be - blueprint of object. Same class can be used as blueprint for multiple objects
* Object is an instance of a class
* Methods define behavior - Method is like function in Python
  + class MyClass {

static void sayHello() { → This is the sayHello() method

System.out.println(“Hello World!”);

}

public static void main(String[] args) {

sayHello(); → sayHello() method is called in main(String[] args)

}

}

* Method parameters
  + class Main {

static void sayHello(String name) {

System.out.println(“Hello ” + name);

}

public static void main(String args[]) {

sayHello(“David”);

sayHello(“Amy”);

}

}

* Method return type
  + When void is used return is null
  + static int sum(int val1, int val2) { → Type of the method is decided initially

return val1 + val2;

}

Creating classes and objects

* public class Animal {

void bark() {

System.out.println(“Woof-Woof”);

}

}

class MyClass {

public static void main(String args[]) {

Animal dog = new Animal(); → ‘dog’ is object of type ‘Animal’

dog.bark(); → ‘.’ notation is used to access ‘dog’ object’s attributes

}

}

Defining attributes and creating multiple objects

* public class Vehicle { → Vehicle is a class

int maxSpeed;

int wheels;

String color;

double fuelCapacity; → maxSpeed, wheels, color, fuelCapacity are attributes

void horn() { → horn() is a method

System.out.println(“Beep!”);

}

}

class MyClass {

public static void main(String args[]) {

Vehicle v1 = new Vehicle(); → v1 is a new object from class Vehicle

Vehicle v2 = new Vehicle();

v1.color = “red”; → This is an attribute

v2.horn(); → This is calling horn method

}

}

Access Modifiers

* public is an access modifier
* Used to set the level of access
* Can be used for classes, attributes and methods
* For classes,
  + public → Accessible by any other class
  + default (left blank) → Accessible by classes in the same package
* For attributes and methods,
  + public → Accessible by any other class
  + default (left blank) → Accessible by classes in the same package
  + private → Accessible only within the declared class itself
  + protected → Same as the default and subclasses can access protected methods and variable of the superclass
  + public class Vehicle {

private int maxSpeed;

private int wheels;

private String color;

private double fuelCapacity;

public void horn() {

System.out.println(“Beep!”);

}

}

* Better to keep variables in a private class

Getters and Setters

* Getters and Setters are used to protect your data
* get method → Returns the variable’s value, get and first letter capital for the class variable → getVariable
* set method → Sets the variable’s value, set and first letter capital for the class variable → setVariable
* public class Vehicle {

private String color;

public String getColor() {

return color; → Getter

}

public void setColor(String c) {

this.color = c; → Setter

}

}

* public class Vehicle {

private String color;

public String getColor() {

return color;

}

public void setColor(String c) {

this.color = c;

}

}

class Program {

public static void main(String args[]) {

Vehicle v1 = new Vehicle();

v1.setColor(“Red”);

System.out.println(v1.getColor());

}

}

Constructors

* A constructor name must be same as its class name
* A constructor must have no explicit return type
* Used to initialize objects
* public class Vehicle {

private String color;

Vehicle() { → This Vehicle() method is the constructor of our class

color = “Red”’

}

}

* In the above example, whenever an object of Vehicle class is created, the color will be set to “Red”
* Constructor can also take parameters to initialize attributes
* public class Vehicle {

private String color;

Vehicle(String c) {

color = c;

}

}

* Constructors are like methods that will set up the class by default → To stop repeat the same code every time
* The constructor is called when you create an object using ‘new’ keyword
* public class MyClass {

public static void main(String[] args) {

Vehicle v = new Vehicle(“Blue”); → Color attribute will be set to “Blue”

}

}

* Single class can have multiple constructors with different numbers of parameters
* Setter methods inside the constructors can be used to set attribute values
* public class Vehicle {

private String color;

Vehicle() {

this.setColor(“Red”); → ‘this’ keyword refers to the current object in a

method or constructor

}

Vehicle(String c) {

this.setColor(c);

}

public void setColor(String c) { → Setter

this.color = c;

}

}

* Above example has 2 constructors. One of them takes parameters and assigns it to the attribute while the other one sets the attribute to default “Red”
* Java automatically provides a default constructor (Even if it’s not defined)
* public class Vehicle {

private String color;

Vehicle() {

this.setColor(“Red”);

}

Vehicle(String c) {

this.setColor(c);

}

public void setColor(String c) {

this.color = c;

}

public String getColor() {

return color;

}

}

public class Program {

public static void main(String[] args) {

Vehicle v1 = new Vehicle();

Vehicle v2 = new Vehicle(“Green”);

System.out.println(v1.setColor());

System.out.println(v2.getColor()); → This is used because ‘.color’ can’t be

used (color is a Private attribute in

Vehicle class)

}

}

Movie Ticket Example

* import java.util.Scanner;

class Main {

public static void main(String[] args) {

Scanner read = new Scanner(System.in);

String movie = read.nextLine();

int row = read.nextInt();

int seat = read.nextInt();

Ticket ticket = new Ticket(movie, row, seat);

System.out.println("Movie: " + ticket.getMovie());

System.out.println("Row: " + ticket.getRow());

System.out.println("Seat: " + ticket.getSeat());

}

}

class Ticket {

private String movie;

private int row;

private int seat;

//complete the constructor

public Ticket(String movie, int row, int seat) {

this.movie = movie;

this.row = row;

this.seat = seat;

}

public String getMovie() {

return movie;

}

public int getRow() {

return row;

}

public int getSeat() {

return seat;

}

}

Value Types

* byte, short, int, long, float, double, boolean, char
* public class Myclass {

public static void main(String[] args) {

int x = 5;

addOneTo(x);

System.out.println(x); → Prints “5”. Value of x doesn’t change

}

static void addOneTo(int num) {

num = num + 1;

System.out.println(num); → New value of ‘num’ is 6

}

}

Reference Type

* Stores a reference (address0 to the memory location
* Arrays and Strings are also reference data types
* When you create an object using the constructor, you create a reference variable
* public class MyClass {

public static void main(String[] args) {

Person j;

j = new Person(“John”); → Similar to Person j = new Person(“john”);

j.setAge(20);

celebrateBirthday(j);

System.out.println(j.getAge());

}

static void celebrateBirthday(Person p) { → celebrateBirthday takes a Person

object as its parameter and increases

it’s attribute. Since j is a reference

type actual value of its attribute

changes

p.setAge(p.getAge() + 1)

}

}

public class Person {

private String name;

private int age;

Person(String n) {

this.name = n;

}

public int getAge() {

return age;

}

public void setAge(int a); {

this.age = a;

}

}

* public static void main(String[ ] args) {

Person p = new Person();

p.setAge(25);

change(p);

System.out.println(p.getAge()); → Prints new age value 10 (Not 25)

}

static void change(Person p) {

p.setAge(10);

}

The Math Class

* Contains predefined methods for mathematical operations
* No need to create object of ‘Math’ class to use it
* int a = Math.abs(10); → Absolute value of 10 → 10

int b = Math.abs(-20); → Absolute value of -20 → 20

double c = Math.ceil(7.342); → Rounds a floating value up to nearest integer (Outputs a

double) → 8.0

double d = Math.floor(7.342); → Rounds a floating value up to nearest integer (Outputs

a double) → 7.0

int e = Math.max(10, 20); → Returns the maximum → 20

int f = Math.min(10, 20); → Returns the minimum → 10

double g = Math.pow(2, 3); → Returns 23 (Returns a double) → 8.0

* You can also use sqrt(), sin(), cos(), etc

Static

* public class Counter {

public static int COUNT = 0;

Counter() {

COUNT++; → ‘COUNT variable will be shared by all objects of the class

due to ‘static’

}

}

public class MyClass {

public static void main(String[] args) {

Counter c1 = new Counter();

Counter c2 = new Counter();

System.out.println(Counter.count); → value of ‘COUNT static variable is

2, since each time new object of

‘Counter’ class is created value of

‘COUNT is incremented

}

}

* For static variables upper case names are used in practice
* Same concept is applied for ‘static’ methods
* public class Vehicle {

public static void horn() {

System.out.println(“Beep”);

}

}

public class Main {

public static void main(String[] args) {

Vehicle.horn(); → since ‘horn’ is a static method it can be called without

creating an object → Prints “Beep”

}

}

* ‘Math’ class is an example for static method where ‘Math’ objects are not created at calling
* ‘main’ method must always be ‘static’
* class Person {

public static int pCount;

public static void main(String[] args) {

Person.pCount = 1;

Person.pCount++;

System.out.println(Person.pCount); → pCount can be used without

creating object → Prints “2”

}

}

final

* ‘final’ keyword is used to mark a variable constant. It can be assigned only once
* class MyClass {

public static final double PI = 3.14; → PI is now a constant, Assigning value will

cause an error

public static void main(String[] args) {

System.out.println(PI);

}

}

* Methods and classes can also be marked ‘final’
* ‘final’ marked restrict methods can’t be overwritten
* ‘final’ marked classes can’t be subclassed

Packages

* Package consists of similar type of classes and sub packages
* To avoid name conflicts and control access to classes
* import samples.Vehicle; → ‘Vehicle’ class from ‘samples’ package is imported to ‘main’

to used them

class Main {

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

Vehicle v1 = new Vehicle();

v1.horn();

}

}

* The name of the package becomes part of the class name
* The package name must match the directory structure where the corresponding class file resides
* import samples.\* → Import all classes in the ‘samples’ package

Binary converter

* import java.util.Scanner;

public class Converter {

public static String toBinary(int num) {

String binary = "";

while (num > 0) {

binary = (num%2) + binary;

num /= 2;

}

return binary;

}

}

public class Program {

public static void main(String[ ] args) {

Scanner sc = new Scanner(System.in);

int x = sc.nextInt();

System.out.print(Converter.toBinary(x)); → Converter.toBinary()

can be called without

creating object since

it’s a static method

}

}

Core concepts in OOP

* Encapsulation
* Inheritance
* Polymorphism
* Abstraction

Encapsulation

* To ensure that implementation details are not visible to users
* Data hiding - Variables of one class will be hidden from other classes. Accessible only through the methods of the current class
* For encapsulation, declare class’ variables as ‘private’ and provide ‘public’ ‘setter’ and ‘getter’ methods to modify and view the variables’ values
* class BankAccount {

private double balance = 0;

public void deposit(double x) {

if (x > 0) {

balance += x; → This implementation hides ‘balance’ variable. Can

get access to ‘balance’ variable only through

‘deposit()’ method.

}

}

}

Inheritance

* Enables one class to acquire properties (methods and variables) of another class
* Superclass/base class/parent class - the class whose properties are inherited
* Subclass/derived class/child class - the class inheriting the properties of another
* Use ‘extends’ keyword to inherit properties from a class
* class Dog extends Animal { → ‘Dog’ class inherits properties from ‘Animal’ class. ‘Dog’ is

the subclass, and ‘Animal’ is the superclass

//some code

}

* At inheritance, subclass inherits all non-private variables and methods from the superclass
* class Animal {

protected int legs; → In protected modifier, members only visible to subclasses

or any class in the package

public void eat() {

System.out.println(“Animal eats”);

}

}

class Dog extends Animal {

Dog() {

legs = 4;

}

}

class Main {

public static void main(String[] args) {

Dog d = new Dog();

d.eat();

}

}

* Since constructors are not member methods they are not inherited, but the constructor of the superclass is called when the subclass is instantiated
* class A {

public A() {

System.out.println(“New A”);

}

}

class B extends A {

public B() {

System.out.println(“New B”);

}

}

class Program {

public static void main(String[] args) {

B obj = new B(); → Prints “New A” and “New B” since superclass

constructed is also called

}

}

* Superclass can be accessed from the subclass using ‘super’ keyword
* super.var → Accesses the ‘var’ member of the superclass

Polymorphism

* class Animal {

public void makeSound() {

System.out.println(“Grr…”);

}

}

class Cat extends Animal {

public void makeSound() {

System.out.println(“Meow”) {

}

}

}

class Dog extends Animal {

public void makeSound() {

System.out.println(“Woof”);

}

}

class Program {

public static void main(String[] args) {

Animal a = new Dog();

Animal b = new Cat();

a.makeSound();

b.makeSound();

}

}

* You can use ‘Animal’ variable without knowing that it contains an object of the subclass
* This is useful when you have multiple subclasses of the superclass
* One method is used for different implementations
* class Vehicle {  
   public void start(){  
   System.out.println("Starting");  
   }  
   public void resource(){  
   System.out.println("I use petrol");  
   }  
  }

class ElectricVehicle extends Vehicle{  
 /\*reimplement resource() method   
 to output "I use electricity"\*/  
 public void resource(){  
 System.out.println("I use electricity");  
 }  
  
}

class HybridVehicle extends Vehicle{  
 /\*reimplement resource() method   
 to output "I use both petrol and electricity"\*/  
 public void resource(){  
 System.out.println("I use both petrol and electricity");

}  
  
}

class Main {  
 public static void main(String[] args) {  
  
 Vehicle vehicle = new Vehicle();  
 Vehicle electric = new ElectricVehicle();  
 Vehicle hybrid = new HybridVehicle();  
  
 //calls  
 vehicle.start();  
 vehicle.resource();  
 electric.start();  
 electric.resource();  
 hybrid.start();  
 hybrid.resource(); → All attributes use similar methods but implementation is

different  
 }  
}

Method overriding/runtime polymorphism

* Subclass implementing a parent method based on its requirement is called ‘overriding’
* class Animal {

public void makeSound() {

System.out.println(“Grr…”);

}

}

class Cat extends Animal {

public void makeSound() {

System.out.println(“Meow”);

}

}

class Program {

public static void main(String[] args) {

Cat c = new Cat();

c.makeSound(); → Prints “Meow”. parent method is overridden by

subclass implementation

}

}

* For overriding,
  + Should have the same return type and arguments
  + Access level can’t be more restrictive than overridden method’s access level (If superclass method is public, overridden method in subclass can’t be private or protected)
  + ‘static’ or ‘final’ method can’t be overridden
  + If a method can’t be inherited, it can’t be overridden
  + Constructors can’t be overridden

Method overloading/compile-time polymorphism

* When methods have same name but different parameters, it’s called ‘method overloading’
* int Program {

static double max(double a, double b) {

if (a > b) {

return a;

}

else {

return b;

}

}

static int max(int a, int b) {

if (a > b) {

return a;

}

else {

return b;

}

}

public static void main(String[] args) {

System.out.println(max(8,17)); → Prints “17”

System.out.println(max(3.14, 7.68)); → Prints “7.68”

}

Abstraction

* Provides only essential information without including implementation details
* Abstraction is achieved using ‘abstract classes’ and ‘interfaces’
* Abstract class is defined by ‘abstract’ keyword
* Abstract classes can’t be instantiated (Can’t create objects of that type)
* To use an abstract class, you have to inherit it from another class
* Any class that has ‘abstract’ method is defined as abstract class
* Abstract method is a method declared without implementation (No braces {}, and followed by a semicolon) → abstract void walk();
* abstract class Animal {

int legs = 0;

abstract void makeSound(); → makeSound method is also abstract. It has no

implementation in the superclass. We can inherit

from the Animal class and define makeSound()

method for subclass

}

class Cat extends Animal {

public void makeSound() {

System.out.println(“Meow”);

}

}

public class Program {

public static void main(String[] args) {

Cat c = new Cat();

c.makeSound(); → Prints “Meow”

}

}

* Abstraction is used when there’s no meaningful definition for the method in superclass. We leave the implementation to the subclasses.
* abstract class Game {

abstract String getName();

abstract void play();

}

class Monopoly extends Game {

//give "Monopoly" name to game

String getName() {

String name = "Monopoly";

return name;

}

// play method should print "Buy all property."

void play() {

System.out.println("Buy all property.");

}

}

class Chess extends Game {

//give "Chess" name to game

String getName() {

String name = "Chess";

return name;

}

// play method should print "Kill the enemy king."

void play() {

System.out.println("Kill the enemy king.");

}

}

class Battleships extends Game {

//give "Battleships" name to game

String getName() {

String name = "Battleships";

return name;

}

// play method should print "Sink all ships."

void play() {

System.out.println("Sink all ships.");

}

}

class Main {

public static void main(String[] args) {

//do not touch this code

Monopoly monopoly = new Monopoly();

Chess chess = new Chess();

Battleships battleships = new Battleships();

monopoly.play();

chess.play();

battleships.play();

}

}

Interfaces

* Interface is a completely abstract class that has only abstract methods
* Defined using the ‘interface’ keyword
* This may contain only static final variables
* Since interfaces can’t be instantiated, they can’t contain a constructor
* Interfaces can extend other interfaces
* A class can implement any number of interfaces
* interface Animal {

public void eat();

public void makeSound();

}

* Since interface is implicitly abstract, you don’t need to use the ‘abstract’ keyword while declaring an interface. This is same for methods in interface
* Methods in an interface are public
* A class can inherit from one superclass, but can implement multiple interfaces
* Use ‘implements’ keyword to use an interface with your class
* interface Animal {

public void eat();

public void makeSound();

}

class Cat implements Animal {

public void makeSound() {

System.out.println(“Meow”);

}

public void eat() {

System.out.println(“omnomnom”);

}

}

public class Program {

public static void main(String[] args) {

Cat c = new Cat();

c.eat(); → Prints “omnomnom”

c.makeSound(); → Prints “Meow”

}

}

* When you implement an interface, you need to override all of its methods.
* In this case, u have to override makeSound and eat methods both. Otherwise it will give an error
* class Main {

   public static void main(String[] args) {

       Animal dog = new Dog();

       Animal cat = new Cat();

       dog.swim();

       dog.play();

       cat.swim();

       cat.play();

   }

}

interface Swimmer {

   void swim();

}

interface Player {

   void play();

}

class Dog extends Animal {

    void swim() {

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

    }

    void play() {

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

    }

}

abstract class Animal {

    void swim() {

        System.out.println("A");

    }

    void play() {

        System.out.println("B");

    }

}

class Cat extends Animal {

    void swim() {

        System.out.println("Cat is swimming");

    }

    void play() {

        System.out.println("Cat is playing");

    }

}

Type casting

* Assigning a value of one type to a variable of another type
* public class Program {

public static void main(String[] args) {

int a = (int) 3.14;

System.out.println(a); → Print “3”

}

}

* public class Program {

public static void main(String[] args) {

double a = 42.571;

int b = (int) a;

System.out.println(b); → Print “42”

}

}

* Java automatically type cast integers to floating points (Since there’s no loss of precision)
* Type casting is mandatory when assigning floating points to integer variables
* import java.util.Scanner;

class Main {

public static void main(String[] args) {

Scanner read = new Scanner(System.in);

char a = read.next().charAt(0);

int b = (char) a;

System.out.println(b); → Returns ASCII value of “a” → Prints “97”

}

}

* public class Main {

public static void main(String[] args) {

double x = 1.5;

double y = 2.65;

sum((int)x,(int)y); → 1 + 2 → 3

}

static void sum(int x, int y) {

System.out.println(x + y);

}

}

* There are 2 types of casting
  + Upcasting - Casting instance of subclass to its superclass
    - Upcasting is automatic. Upcasting can never fail
    - Assuming Cat is a subclass of Animal
    - Animal a = new Cat();
    - Java automatically upcasted the Cat type variable to Animal type
  + Downcasting - Casting an object of a superclass to its subclass
    - Downcasting is manual. Downcasting could fail sometimes
    - Animal a = new Cat();

((Cat)a).makeSound(); → Casting the variable ‘a’ to ‘Cat’ type and call its

‘makeSound()’ method

* class A {

public void print() {

System.out.println("A");

}

}

class B extends A {

public void print() {

System.out.println("B");

}

public static void main(String[ ] args) {

A object = new B();

B b = (B) object;

b.print();

}

}

Anonymous classes

* Anonymous classes are used to extend the existing classes on the fly
* class Machine {

public void start() {

System.out.println(“Starting…”);

}

}

class Program {

public static void main(String[] args) {

Machine m = new Machine() {

@Override public void start() {

System.out.println(“Wooooo”);

}

};

m.start(); → Prints “Woooo”

}

}

* The modification is applicable only to current object (Not the class itself)
* If we create another object of that class, the start method’s implementation will be the one defined in the class
* class Machine {

public void start() {

System.out.println(“Starting…”);

}

}

class Program {

public static void main(String[] args) {

Machine m1 = new Machine() {

@Override public void start() {

System.out.println(“Wooooo”);

}

};

Machine m2 = new Machine();

m1.start(); → Print “Wooooo”

m2.start(); → Print “Starting…”

}

}

* class Purchase {

int price;

public int totalAmount(int price) {

return price - (price\*10)/100;

}

}

public class Main {

public static void main(String[] args) {

Purchase customer = new Purchase();

Purchase specialCustomer = new Purchase() {

@Override public int totalAmount(int price) {

return price - (price\*20)/100;

}

};

System.out.println(customer.totalAmount(1000));

System.out.println(specialCustomer.totalAmount(10000));

}

}

Inner classes

* Java supports nesting classes - class within classes
* Unlike a class, an inner class can be private - can’t be accessed from an object outside the class
* class Robot { → ‘Robot’ class has inner class ‘Brain’

int id;

Robot(int i) {

id = i;

Brain b = new Brain();

b.think();

}

private class Brain { → Inner class ‘Brain’ can access member variables and

methods of ‘Robot’, but it can’t access from outside class

public void think() {

System.out.println(id + “ is thinking”);

}

}

}

public class Program {

public static void main(String[] args) {

Robot r = new Robot(1); → Print “1 is thinking”

}

}

Comparing objects

* When objects are created, the variables store references to the objects
* class Animal {

String name;

Animal(String n) {

name = n;

}

}

class MyClass {

public static void main(String[] args) {

Animal a1 = new Animal(“Robby”);

Animal a2 = new Animal(“Robby”);

System.out.println(a1 == a2); → Prints “false” since this actually

compares the references not the object

values

}

}

equals()

* Each object has predefined equals() method that’s used for semantical equality testing
* To make it work for our classes, we need to override it and check the conditions we need
* Simple, fast way of generating equals() method,
* Right click in the class → Source → Generate hashCode() and equals()…
* In intelliJ IDEA, use Alt + Insert
* This will create the necessary methods
* class Animal {

String name;

Animal (String n) {

name = n;

}

@Override

public int hashCode() {

final int prime = 31;

int result = 1;

result = prime\*result + ((name == null) ? 0: name.hashCode());

return result;

}

@Override

public boolean equals(Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (getClass() != obj.getClass())

return false;

Animal other = (Animal) obj;

if (name == null) {

if (other.name != null)

return false;

}

else if (!name.equals(other.name))

return false;

return true;

}

}

class Program {

public static void main(String[ ] args) {

Animal a1 = new Animal("Robby");

Animal a2 = new Animal("Robby");

System.out.println(a1.equals(a2)); → Print “true”

}

}

* Automatically generated ‘hashCode()’ method is used to determine where to store the object internally
* Whenever you implement ‘equals’, you must also implement ‘hashCode’
* The same menu can be used to generate other methods such as ‘getters’ and ‘setters’
* class A {

private int x;

public boolean equals(Object o) {

return ((A)o).x == this.x;

}

public static void main(String[ ] args) {

A a = new A();

a.x = 9;

A b = new A();

b.x = 5;

System.out.println(a.equals(b)); → Print “false”

}

}

Enums

* class Main {

    public static void main(String[] args) {

        Player player1 = new Player(Difficulty.EASY);

        Player player2 = new Player(Difficulty.MEDIUM);

        Player player3 = new Player(Difficulty.HARD);

   }

}

enum Difficulty {

    EASY,

    MEDIUM,

    HARD

}

public class Player{

    Player(Difficulty diff){

        //your code goes here

        switch (diff) {

            case EASY:

                System.out.println("You have 3000 bullets");

                break;

            case MEDIUM:

                System.out.println("You have 2000 bullets");

                break;

            default:

                System.out.println("You have 1000 bullets");

        }

    }

}

* enum Color {

RED, BLUE, GREEN;

}

class PrintColor {

public static void main(String[ ] args) {

Color color = Color.RED;

switch(color) {

case BLUE:

System.out.println("1");

break;

case GREEN:

System.out.println("2");

break;

default:

System.out.println("0"); → Prints “0” since color is “RED”

break;

}

}

}

Java API

* Collection of classes and interfaces that are already written
* Java API with all the API is available here → <http://docs.oracle.com/javase/7/docs/api/>
* import java.awt.\*;
* The ‘awt’ package contains all the classes for creating user interfaces and painting graphics and images
* Wildcard character ‘\*’ is used to import all classes in the API package

Module 5 Quiz

* class A {

private void print() {

System.out.println(''a'');

}

private void print(String str) {

System.out.println(''b'');

}

private void print(int x) {

System.out.println(''c'');

}

public static void main(String[ ] args) {

A object = new A();

object.print(12); → Prints “c”

}

}

Shapes

* import java.util.Scanner;

abstract class Shape {

    int width;

    abstract void area();

}

class Square extends Shape {

    public Square(int x) {

        width = x;

    }

    public void area() {

        int area = width\*width;

        System.out.println(area);

    }

}

class Circle extends Shape {

    public Circle(int y) {

        width = y;

    }

    public void area() {

        double area = Math.PI\*width\*width;

        System.out.println(area);

    }

}

public class Program {

    public static void main(String[ ] args) {

        Scanner sc = new Scanner(System.in);

        int x = sc.nextInt();

        int y = sc.nextInt();

        Square a = new Square(x);

        Circle b = new Circle(y);

        a.area();

        b.area();

    }

}

Input formatting

* import java.io.\*;

import java.util.\*;

public class Solution {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

String a = scanner.next();

int b = scanner.nextInt();

scanner.nextLine();

String c = scanner.next();

int d = scanner.nextInt();

scanner.nextLine();

String e = scanner.next();

int f = scanner.nextInt();

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

System.out.printf("%-15s%03d%n",a,b);

System.out.printf("%-15s%03d%n",c,d);

System.out.printf("%-15s%03d%n",e,f);

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

}

}

* Here, ‘%-6s’ prints a String and left aligns for 2 characters. If the string is only 4 letters, then the rest of 4 characters will be filled with spaces. For example, if the String is “Java”, then the output will be “Java ” with total 6 characters
* Here, ‘%03d’ prints a number to 3 characters. If the number is only 1 character, then rest 2 characters will be filled with “0”. For example, if the number is “6”, then the output will be “006”
* ‘%n’ produces a new line

Exceptions

* Problems occurring at execution
* Exception handling - handling runtime errors
* Reasons for exception,
  + User entered invalid data
  + A file needed to open is not found
  + Network connection loss in the middle of communications
  + Insufficient memory and issues related to physical resources
* ‘try’ and ‘catch’ are used for exception handling
* try {

//some code

} catch (Exception e) {

//some code to handle errors

}

* Type of exception must be listed in ‘catch’ block
* public class MyClass {

public static void main(String[] args) {

try {

int a[] = new int[2];

System.out.println(a[5]);

}

catch (Exception e) { → ‘Exception e’ catches all exceptions

System.out.println(“An error ocurred”); → Prints “An error

occurred”

}

}

}

* import java.util.Scanner;  
    
  public class Main  
  {  
   public static void main(String[] args) {  
    
   Scanner scanner = new Scanner(System.in);  
   int choice = scanner.nextInt();  
    
   String[] categories = {"PCs", "Notebooks", "Tablets", "Phones", "Аccessories"};  
   try {  
   System.out.println(categories[choice]);  
   } catch(Exception e) {  
   System.out.println("Wrong Option"); → Prints “Wrong Option”  
   }  
   }  
  }

throw

* Allows to manually generate exceptions from your methods
* Some available exception types,
  + IndexOutOfBoundsException
  + IllegalArgumentException
  + InputMismatchException → Can be used when input type is different

from what user defined

* + IOException
  + ArithmeticException → Can be used when parameter is “0”
* public class Program {

static int div(int a, int b) throws ArithmeticException {

if (b == 0) {

throw new ArithmeticException(“Division by Zero”);

}

else {

return a/b;

}

}

public static void main(String[] args) {

System.out.println(div(42,0)); → Print “Division by Zero”

}

}

* Using ‘throws’ we can define the types of exceptions to handle
* Using ‘throw’ we can decide what message to Print. For example, “Division by Zero”

Multiple exceptions

* Multiple exceptions can be defined in ‘throws’ statement using comma-separated list
* Single ‘try’ block can contain multiple ‘catch’ blocks that handle different exceptions separately
* try {

//some code

}

catch (ExceptionType1 e1) {

//catch block

}

catch (ExceptionType2 e2) {

//catch block

}

catch (ExceptionType3 e3) {

//catch block

}

* All catch blocks must be ordered from most specific to most general. Otherwise, those specific exceptions will be handled by the general catch block
* import java.util.Scanner;

import java.util.InputMismatchException;

public class Main {

public static void main(String[] args) {

try {

Scanner scanner = new Scanner(System.in);

int num1 = scanner.nextInt();

int num2 = scanner.nextInt();

System.out.println(num1/num2);

}

catch(ArithmeticException e) {

System.out.println("Error: division by zero");

}

catch(InputMismatchException e){

System.out.println("Error: wrong value type");

}

}

}

Threads

* Java is a multi-threaded programming language
* Two ways to create a thread,
  + Extend the thread class
    - Inherit from the ‘Thread’ class, override ‘run()’ method, and write the functionality of the thread in the ‘run()’ method
    - Then, create a new object of your class and call it’s start method to run the thread
    - class Loader extends Thread {

public void run() {

System.out.println(“Hello”);

}

}

class Program {

public static void main(String[] args) {

Loader obj = new Loader();

obj.start(); → Prints “Hello”

}

}

* + - When we create ‘obj’ object and call its ‘start()’ method, the ‘run()’ method statements execute on a different thread
* Every java thread is prioritized to help the os determine the order in which to schedule threads
* The priorities range from 1 to 10, defaulting priority 5
* You can set prority with ‘setPriority()’ method
* Priority depends on the priority number. For example, priority number 9 is preferred over priority number 3
  + Implementing the ‘Runnable’ interface
    - Implement ‘run()’ method
    - Then, create a new Thread object, pass the Runnable class to its conductor, and start the Thread by calling ‘start()’ method
    - class Loader implements Runnable {

public void run() {

System.out.println(“Hello”);

}

}

class Program {

public static void main(String[] args) {

Thread obj = new Thread(new Loader());

obj.start(); → Prints “Hello”

}

}

* + - ‘Thread.sleep()’ method pauses a Thread for a specified period of time
    - For example, ‘Thread.sleep(1000);’ pauses the thread for 1 second
    - Remember, ‘Thread.sleep()’ throws an ‘InterruptedException’. Surround it with ‘try-catch’ block
    - Implementing the Runnable interface is the preferred way to start a Thread, since it enables you to extend from another class as well
    - class B implements Runnable {

public void run() {

System.out.println("B");

}

}

class A extends Thread {

public void run() {

System.out.println("A");

Thread t = new Thread(new B());

t.start();

}

public static void main(String[ ] args) {

A object = new A();

object.start();

}

}

* + - Above code prints “A” and “B”
    - class Name extends Thread {

public void run() {

System.out.println("Please enter your name");

}

}

class Welcome extends Thread {

public void run() {

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

}

}

class Main {

public static void main(String[ ] args) {

Name name = new Name();

name.setPriority(1);

Welcome welcome = new Welcome();

welcome.setPriority(10);

name.start();

welcome.start();

}

}

* + - Here, ‘welcome’ is executed before ‘name’, since ‘welcome’ has a higher priority number

Runtime vs checked exceptions

* + - Two exception types
      * checked - ‘checked’ exceptions are checked when compiled
      * unchecked/runtime - ‘unchecked’ exceptions are checked at runtime
    - ‘Thread.sleep()’ throwing an ‘InterruptedException’ is an example of ‘checked’ exception
    - Your code won’t compile until you handle the exception
    - public class Main {

public static main(String[] args) {

try {

Thread.sleep(1000);

}

catch (InterruptedException e) {

//some code

}

}

}

* + - Dividing by zero is checked at runtime. Hence, it’s an unchecked exception
    - public class Main {

public static void main(String[] args) {

int value = 7;

value = value/0;

}

}

ArrayList

* ‘ArrayList’ is a class in Java API
* Standard Java arrays have a fixed length (can’t change length after creation)
* ArrayLists are created with an initial size, but when this size is exceeded the collection is automatically enlarged
* When objects are removed, the ‘ArrayList’ will shrink in size
* Since ‘ArrayList’ is in ‘java.util’ package, import ‘java.util.ArrayList’ or ‘java.util.\*’ before use
* Optional - You can specify a capacity and type of objects the ‘ArrayList’ will hold
* ArrayList<String> colors = new ArrayList<String>(10);
* Above code defines an ‘ArrayList’ of Strings with initial size 10
* ‘ArrayLists’ store objects. Thus, type specified must be a class type
* For example, ‘Integer’ for ‘int’, ‘Double’ for ‘double’, etc
* ArraList<Integer> ar = new ArrayList<Integer>(9)
* Above code defines an ‘ArrayList’ of Integers with initial size 9
* Useful methods in ‘ArrayList’
  + ‘add()’ method → adds new objects to the ‘ArrayList’
  + ‘remove()’ method → removes objects from the ‘ArrayList’
  + ‘contains()’ method → returns ‘true’ if the list contains specified element
  + ‘get(int index)’ method → returns the element at the specified position in the list
  + ‘size()’ method → returns numbers of elements in the list
  + ‘clear()’ method → removes all of the elements from the list
* Indexing starts with ‘0’
* import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<String> colors = new ArrayList<String>();

colors.add(“Red”);

colors.add(“Blue”);

colors.add(“Green”);

colors.remove(“Blue”);

System.out.println(colors);

}

}

* import java.util.ArrayList;

import java.util.Scanner;

public class Main {

public static void main(String[ ] args) {

Scanner scanner = new Scanner(System.in);

int sum = 0;

ArrayList<Integer> evennums = new ArrayList<Integer>();

while(evennums.size()<3){

int num = scanner.nextInt();

sum += num;

evennums.add(num);

}

System.out.println(sum/3);

}

}

LinkedList

* ‘LinkedList’ is similar to ‘ArrayList’, but you can’t specify initial capacity for ‘LinkedList’
* ‘ArrayList’ can be changed to ‘LinkedList’ by changing the object type
* import java.util.LinkedList;

public class Main {

public static void main(String[] args) {

LinkedList<String> c = new LinkedList<String>();

c.add(“Red”);

c.add(“Blue”);

c.add(“Green”);

c.remove(“Blue”);

System.out.println(c);

System.out.println(c.get(0)) → Print ‘0’ index element → Print

“Red”

}

}

LinkedList vs ArrayList

* ArrayList is better for storing and accessing data, since it’s similar to a normal array
* LinkedList is better for manipulating data, such as making numerous inserts and deletes
* LinkedList store an object and memory addresse(link) of the element that follow it
* Thus, each element has a link to the neighboring element
* Use ‘enhanced for loop’ to loop over its elements
* import java.util.LinkedList;

public class Main {

public static void main(String[] args) {

LinkedList<String> c = new LinkedList<String>();

c.add(“Red”);

c.add(“Blue”);

c.add(“Green”);

c.remove(“Blue”);

for(String s:c) {

System.out.println(s); → Prints ‘c’ array elements

}

}

}

* import java.util.LinkedList;

import java.util.Scanner;

public class Main {

public static void main(String[ ] args) {

Scanner scanner = new Scanner(System.in);

LinkedList<String> words = new LinkedList<String>();

while(words.size()<5){

String word = scanner.nextLine();

words.add(word);

}

for (String i:words) {

if (i.length() > 4) {

System.out.println(i); → Prints elements of ‘word’

LinkedList that has more than 4

characters

}

}

}

}

HashMap

* Arrays and Lists store elements with each element given an integer index
* HashMap is used to store data collections as key and value pairs
* One object is used as a key (index) to another object (the value)
* HashMap methods
  + put() - add values to the HashMap
  + remove() - delete values from the HashMap
  + get() - access values from the HashMap
* import java.util.HashMap;

public class Main {

public static void main(String[] args) {

HashMap<String, Integer> points = new HashMap<String, Integer>();

points.put(“Amy”, 154);

points.put(“Dave”, 42);

points.put(“Rob”,7 33);

System.out.println(points.get(“Dave”)); → Prints “42”

}

}

* In the above example, Strings are the keys and Integers are the values. Thus, key “Dave” is used with ‘get’ to access value “42”
* HashMap can’t have duplicate keys. If you add a new item with an existing key it will overwrite the old element
* HashMap class provides ‘containsKey’ and ‘containsValue’ methods to determine the presense of a specified key or value
* If you try to get a value that’s not present in the HashMap, it returns ‘null’ which represents absence of a value
* import java.util.HashMap;

import java.util.Scanner;

public class Main {

public static void main(String[ ] args) {

Scanner scanner = new Scanner(System.in);

HashMap<String, Integer> ages = new HashMap<String, Integer>();

ages.put("David", 22);

ages.put("Tom", 23);

ages.put("Robert", 32);

ages.put("Alice", 21);

ages.put("Sophie", 19);

ages.put("Maria", 24);

ages.put("John", 28);

String[] nameArr = new String[ages.size()];

nameArr = ages.keySet().toArray(nameArr);

int ageLimit = scanner.nextInt();

for (String emp : nameArr){

if (ages.get(emp) < ageLimit) {

ages.remove(emp);

}

}

System.out.println(ages);

}

}

Sets

* ‘Set’ is a collection that can’t contain duplicate elements
* It models the mathematical set abstraction
* One of the implementations of the ‘Set’ is the ‘HashSet’ class
* import java.util.HashSet;

public class Main {

public static void main(String[] args) {

HashSet<String> set = new HashSet<String>();

set.add(“A”);

set.add(“B”);

set.add(“C”);

System.out.println(set); → Prints “[A, B, C]”

}

}

* ‘size()’ method is used to get the number of elements in the HashSet

LinkedHashSet

* HashSet class doesn’t automatically retain the order of the elements as they are added
* To order the elements, use a ‘LinkedHashSet’, which maintains a linked list of the set’s elements in the order in which they were inserted

Hashing

* Hash table stores information through a mechanism called ‘hashing’, in which a key’s informational content is used to determine a unique value called a ‘hash code’
* Each element in the ‘HashSet’ is associated with its unique hash code

Sorting lists

* For manipulation of data in different collection types, Java API provides a ‘Collections’ class which must be imported from java.util.pacakage
* ‘Collections’ class methods
  + sort() - sorts elements of the collection type
  + max(Collection c) - returns maximum element in ‘c’
  + min(Collection c) - returns minimum element in ‘c’
  + reverse(List list) - reverse the sequence in list
  + shuffle(List list) - shuffles the elements in list (randomly)
* Methods in the ‘Collections’ class are static - don’t need a ‘Collections’ object to call them
* import java.util.ArrayList;

import java.util.Collections;

public class Main {

public static void main(String[] args) {

ArrayList<String> animals = new ArrayList<String>();

animals.add(“tiger”);

animals.add(“cat”);

animals.add(“snake”);

animals.add(“dog”);

Collections.sort(animals); → Elements are sorted

alphabetically

System.out.println(animals);

}

}

* import java.util.ArrayList;

import java.util.Collections;

public class Main {

public static void main(String[] args) {

ArrayList<Integer> nums = new ArrayList<Integer>();

nums.add(3);

nums.add(36);

nums.add(73);

nums.add(40);

nums.add(1);

Collections.sort(nums); → Elements are sorted from small to

large

System.out.println(nums);

}

}

* import java.util.Scanner;

import java.util.ArrayList;

import java.util.Collections;

public class Main {

public static void main(String[ ] args) {

ArrayList<Integer> nums = new ArrayList<Integer>();

Scanner scanner = new Scanner(System.in);

while(nums.size()<5){

int num = scanner.nextInt();

nums.add(num);

}

System.out.println(Collections.max(nums));

System.out.println(Collections.min(nums));

}

}

Iterators

* ‘Iterator’ is an object that enables to cycle through a collection, obtain or remove elements
* Before accessing a collection through an iterator, you must obtain one
* Each of the collection classes provide ‘iterator()’ method that returns an iterator to the start of the collection
* Iterator class provides these methods
  + hasNext() - returns true if there are elements or at least one element, otherwise returns false
  + next() - returns the next object and advances the iterator
  + remove() - removes the last object that was returned by next() from the collection
  + import java.util.Iterator;

import java.util.LinkedList;

public class Main {

public static void main(String[] args) {

LinkedList<String> animals = new LinkedList<String>();

animals.add(“fox”);

animals.add(“cat”);

animals.add(“dog”);

animals.add(“rabbit”);

Iterator<String> it = new animals.iterator();

String value = it.next();

System.out.println(value); → Print “fox”

System.out.println(it.next()); → Print “cat”

}

}

* + Each time you call ‘next()’ iterator returns the value and moves to the next element
  + Typically, iterators are used in loops
  + import java.util.Iterator;

import java.util.LinkedList;

public class Main {

public static void main(String[] args) {

LinkedList<String> animals = new LinkedList<String>();

animals.add(“fox”);

animals.add(“cat”);

animals.add(“dog”);

animals.add(“rabbit”);

Iterator<String> it = new animals.iterator();

while (it.hasNext()) {

String value = it.next();

System.out.println(value);

}

}

}

* import java.util.\*;

public class Main {

public static void main(String[] args) {

ArrayList<Integer> list = new ArrayList<Integer>();

for (int i = 0; i < 6; i++) {

list.add(i);

}

int x = 0;

Iterator<Integer> it = list.iterator();

while (it.hasNext()) {

x+= it.next();

}

System.out.println(x); → Prints “15”

}

}

* import java.util.Scanner;

import java.util.Iterator;

import java.util.LinkedList;

public class Main {

public static void main(String[ ] args) {

Scanner scanner = new Scanner(System.in);

LinkedList<Integer> nums = new LinkedList<Integer>();

while(nums.size()<5){

int num = scanner.nextInt();

nums.add(num);

}

int sum = 0;

for (int i: nums) {

sum += i;

}

System.out.println(sum);

}

}

Working with files

* ‘java,io’ package has a ‘File’ class that allows you to work with files
* Create a file object and specify the file path in the constructor
* import java.io.File;

File file = new File(“D:\\Job Hunt\\Programming\\Codes\\Java.txt”)

* Note that we use “\\” in the file path instead of “\”
* Use ‘exists()’ method to determine if a file exists
* import java.io.File;

public class Main {

public static void main(String[] args) {

File txt = new File(“D:\\Job Hunt\\Programming\\Codes\\Java.txt”);

if (txt.exists()) {

System.out.println(x.getName() + “exists!”);

}

else {

System.out.println(“The file does not exist”);

}

}

}

* ‘getName()’ returns the name of the file

Reading a file

* Files are useful for storing and retrieving data
* Use the constructor of the ‘Scanner’ class to take ‘File’ object as input
* try {

File doc = new File(“D:\\Job Hunt\\Programming\\Codes\\Java.txt”);

Scanner scanner = new Scanner(doc);

}

catch (FileNotFoundException e) {

}

* ‘Scanner’ class inherits from the ‘Iterator’
* We can use Scanner object’s ‘next()’ method to read the file’s contents
* try {

File file = new File(“D:\\Job Hunt\\Programming\\Codes\\Java.txt”);

Scanner scanner = new Scanner(file);

while (scanner.hasNext()) {

System.out.println(scanner.next());

}

scanner.close();

}

catch (FileNotFoundException e) {

System.out.println(“Error”);

}

* The file content is printed word by word since the ‘next()’ method returns each word separately
* Use ‘close()’ method to close a file

Creating and writing files

* ‘Formatter’ class is in the java.util.package is used to create content and write them to files
* import java.util.Formatter;

public class Main {

public static void main(String[] args) {

try {

Formatter f = new Formatter(“D:\\Job Hunt\\Java.txt”);

}

catch (Exception e) {

System.out.println(“Error”);

}

}

}

* This will create a file at the file path. If the file already exists this will overwrite it
* After the file is created, content can be written using ‘format()’ method
* import java.io.File;

import java.util.Scanner;

import java.util.Formatter;

public class Main {

public static void main(String[] args) {

try {

Formatter f = new Formatter(“D:\\Job Hunt\\Java.txt”);

f.format(“%s %s %s”, “1”, “John”, “Smith \r\n”);

f.format(“%s %s %s”, “2”, “Amy”, “Brown”);

f.close();

File x = new File(“D:\\Job Hunt\\Java.txt”);

Scanner sc = new Scanner(x);

while (sc.hasNext()) {

System.out.println(sc.next());

}

sc.close();

}

catch (Exception e) {

System.out.println(“Error”);

}

}

}

* “\r\n” is the new line symbol in windows
* After writing close the file
* import java.io.File;

import java.util.Scanner;

import java.util.Formatter;

public class Main {

public static void main(String[ ] args) {

Scanner input = new Scanner(System.in);

try {

Formatter f = new Formatter("tasks.txt");

int count = 0;

while(count < 3) {

f.format("%s", input.next());

count++;

}

f.close();

}

catch (Exception e) {

System.out.println("Error");

}

readFile();

}

public static void readFile() {

try {

File x = new File("tasks.txt");

Scanner sc = new Scanner(x);

while(sc.hasNext()) {

System.out.println(sc.next());

}

sc.close();

}

catch (Exception e) {

System.out.println("Error");

}

}

}

Final problem - Bowling game

* import java.util.\*;

public class Program {

public static void main(String[ ] args) {

Bowling game = new Bowling();

Scanner sc = new Scanner(System.in);

for(int i=0;i<3;i++) {

String input = sc.nextLine();

String[] values = input.split(" ");

String name = values[0];

int points = Integer.parseInt(values[1]);

game.addPlayer(name, points);

}

game.getWinner();

}

}

public class Bowling {

HashMap<String, Integer> players;

Bowling() {

players = new HashMap<String, Integer>();

}

public void addPlayer(String name, int p) {

players.put(name, p);

}

public void getWinner() {

String winner = "";

int max = 0;

for(String i:players.keySet()) {

if(players.get(i) > max) {

winner = i;

max = players.get(i);

}

}

System.out.println(winner);

}

}