Avoid cast as much as possible

**Introduction**

1. Each type of CPU has its own language and machine codes – binary digits (Intel processors cannot be run on AMD processors)
2. Compiler: translate high level program into machine language(.exe) -> executed

Interpreter: translate a smaller part of the program at a time and execute it

Java source code bytecode machine code -> executed

1. Object can represent real-world entities (A class represents a concept, and an object represents the embodiment of that concept)

state: descriptive characteristics -> variables

behavior: what it can do/ be done -> methods

1. \*Class names: title case

Constants: upper case

**Expression**

1. Every character string is an object in Java, defined by the **String** class

Add two string together: +

1. Print: System.out.print()/System.out.println()

\b: backspace \t: tab \n: newline \”: double quote \’: single quote \\: backslash

1. Variable: int total = 30;

Constant: final int TOTAL = 30;

Eight primitive data types: byte, short, int, long, float, double, char, Boolean

1. Data conversion: do not change the type of variable/value stored in it
2. Widening conversion: small data type to a larger one -> safest
3. Narrowing conversion: large data type to a smaller one -> can lose information
4. Reading input: Scanner class (import java.util)

Scanner scan = new Scanner (System.in);

int a = scan.nextInt(); float b = scan.nextDouble(); String c = scan.nextLine();

**Class**

1. Define a class:
2. state (faceValue) & behavior (roll) – data & methods
3. **constructor** (no return type): initialize an instance

public <classname> (<parameter list>) //allow multiple constructors with different parameter list

1. **setValue() & getValue()**
2. **toString** method: called automatically when an object is passed to the println method

public String toString()

{ return Integer.toString(faceValue); }

1. \*Not necessary to define a **main** method

public static void main(String[] args)

**Objects**

1. Primitive variable: int num; -> primitive variable contains value itself

Object reference: Die die1; -> no object created (object variable holds the **address** of an object)

Create object: new <classname> (<parameters>) (e.g. die1 = new Die(5))

An object is an instance of a class

1. **Alias**: Two or more references that refer to the same object

**Garbage**: An object no longer has any valid references to it, it can no longer be accessed by the program

1. **This** reference: an object refers to the object itself, used to distinguish the instance variable of a class from method parameter with the same names.

**Data Visibility**

1. Data declared at the class level can be referenced by all methods in that class

Data declared within a method (local data) can be used only in that method

1. **Instance data**: each instance (object) that is created has its own version

A class declares the type of the data, but it does not reserve any memory space for it. Every time a Die object is created, a new faceValue variable is created as well (each object has its own data space)

1. Encapsulation: an object is an encapsulated entity, providing a set of specific services. These services define the interface to the object.

Any changes to the object's state (its variables) should be made by that object's methods, instead of changing directly.

1. **final** modifier: define constants (e.g. final int MAX = 6)
2. Members of a class that are declared with public visibility can be referenced anywhere

Members of a class that are declared with private visibility can be referenced only within that class

Members declared without a visibility modifier have default visibility and can be referenced by any class in the same package

* Variables:

a. Instance variables should be declared with private visibility

b. It is acceptable to give a constant (all-cap) public visibility

* Methods:

a. provide the services to clients (other classes) 🡪 service methods (public visibility)

b. support other methods in the class (be used by the objects of the same class only) 🡪 support method (private visibility)

1. accessor method (**getX**): return the current value of a variable (designed so that the values of variables can be set only within particular limits)

mutator method (**setX**): change the value of a variable

**Class library**

1. import all classes in a package: import java.lang.\*
2. Class **String**

New string object: String name = new String (“ELEC 2543”); or String name = “ELEC 2543”;

Once a String object has been created, it’s immutable.

Methods return new String objects modified:

str.concat(“abc”) -> to modify: str = str.concat(“abc”)

str.toUpperCase()

str.replace(‘a’, ’b’)

str.substring(3, 10)

str.length()

str.charAt(3) 🡪 return the character at the position(char type)

1. Class **Math** **Static/Class method**: can be invoked through the class name. No object needed.

Math.random() 🡪 return random double number in [0.0, 1.0]

Math.cos(90)

Math.sqrt(9)

1. Class **Random** (in java.util package)

Random generator = new Random();

num1 = generator.nextInt(10) 🡪 random int in [0,9] generator.nextInt() 🡪 random in the range of int

num2 = generator.nextFloat() 🡪 random float in [0,1)

1. Class **NumberFormat** (in java.text package)

Instantiate a NumberFormat object by invoking a class method (no need to new):

NumberFormat fmt1 = NumberFormat.getCurrencyInstance(); //currency format

NumberFormat fmt2 = NumberFormat.getPercentInstance(); //percent format

System.out.println(fmt1.format(tax)) + " at " + fmt2.format(tax\_rate));

Class **DecimalFormat** (in java.text package)

DecimalFormat fmt = new DecimalFormat(“0.###”);

System.out.println(“The circle’s area: “ + fmt.format(area));

**Wrapper class**

1. allow parameters to be object

The java.lang package contains wrapper classes corresponding to each primitive type

Create: Integer age = new Integer(40)

**Autoboxing** (primitive type -> wrapper object): int num = 42; Integer obj = num;

**Unboxing** (wrapper object -> primitive value): Integer obj = new Integer(40); int num = obj;

String🡪interger: int num = Integer.parseInt(str);

max/min value of int: Integer.MAX\_VALUE / Integer.MIN\_VALUE

**Methods**

1. Method calling

If the called method is in the same class, only the method name is needed.

If the called method is in another class, use the dot operator (obj.doIt())

1. **Method overloading**: giving a single method name multiple definitions

The signature (the number, type, and order of parameters) must be unique.

Return type is NOT part of signature.

1. (1) Parameters in a Java method are *passed by value*: A copy of the actual parameter (the value passed in) is stored into the formal parameter (in the method header), similar to assignment statement.

(2) When an object is passed to a method as parameter, the actual parameter and the formal parameter become aliases of each other. (The method can permanently change the object)

**Enumerator types**

1. Define: enum Season {WINTER, SPRING, SUMMER, FALL}

The values are accessed through the name of type: Season time = Season.SPRING;

**ordinal()** method: time.ordinal() 🡪 1

**name()** method: time.name() 🡪 SPRING

1. Enumerator type is a special type of class, we can define constructors, methods and instance variables in an enumerator type:

public enum Rank {

TWO("2"),

THREE("3"),

…

ACE("A");

private String rank;

// Constructor

Rank (String r) {

rank = r; }

}

**values()** method: returns an array of all possible values of the enum type

**Arrays**

1. The array itself is an object that must be instantiated.

The element type can be a primitive type or an object reference.

1. Declaring: int[] scores; or int scores[]; -> don’t have to provide size
2. Instantiate array objects: scores = new int[10]; -> must provide size (Once an array is created, it has a fixed size)

Declare + Instantiate: float[] prices = new float[500]; or

Instantiate with initial values: char[] vowels = {‘A’, ‘E’, ‘I’, ‘O’, ‘U’};

1. Iterator: int[] scores = new int[10];

for (**int score : scores**) -> score: each element in scores

System.out.println (score); -> only be used when processing all array elements

1. Once an array is created, it has a fixed size

Get array length: scores.length

1. String[] words = new String[5]; -> reserve space to store 5 references to String objects; does not create String objects. (At this point System.out.println (words[0]); -> null)

Each object stored must be instantiated separately: words[0] = new String("friend");

1. Two dimensional arrays: int[][] scores = new int[12][50];

The array stored in one row can be specified using one index.

1. Array passed as parameters to a method: reference is passed, making the formal and actual parameters **aliases** of each other -> changing an array element within the method changes the original
2. Main method takes an array of String objects as a parameter, which comes from **command-line arguments** provided when the interpreter is invoked.

public class NameTag{

public static void main (String[] args){

System.out.println ();

System.out.println (args[0]);

System.out.println (args[1]);

}

}

> java NameTag Howdy John

Howdy

John

1. Variable Length Parameter Lists: public double average (int ... list) list -> array name

A method can accept other parameters apart from the variable length parameter lists, but the varying number of parameters must come last in the formal arguments.

A single method cannot accept two sets of varying parameters.

**Arraylist** -in java.util package

An ArrayList: An array that could adjust its capacity during the program execution; stores references to the Object class, which allows it to store any kind of object.

1. Declaration: ArrayList<Integer> intList or ArrayList objList
2. Instantiation: intList = new ArrayList<Integer>();

Declaration+Instantiation: ArrayList<Integer> intList = new ArrayList<Integer>();

1. Add a new element: intList.add(5) –>add in the end or intList.add(1, 5); -> add in position 1

(4) Remove an element:

a. by position: intList.remove(1) ->The element is returned by the remove method

b. by object: intList.remove(**new Integer(15)**) ->The first occurrence of the object is removed; Method returns true if an element is removed from the list

(5) Retrieve an element by position: intList.get(0);

\*(6) Replace an element by position: intList.set(0, new Integer(2));

(7) Get arraylist size: intList.size() -> return the number of elements in arraylist

(8) Get index of a specific element: intList.indexOf(object) ->returns the index of the first occurrence of the specified element, or -1 if this list does not contain the element.

(9) Print out the arraylist: System.out.println(intList) ->[0,1,2,3,4]

**Comparison**

1. Floats/doubles: == may not work due to rounding errors

Characters: use relational operators (>, <, =)

Unicode character set -> Lexicographic Ordering: uppercase < lowercase; short string (the same as the beginning of long string) < long string

1. String object:
2. **equals** method: name1.equals(name2) -> true/false
3. **compareTo** method: name1.compareTo(name2) -> 0:equal; -1:name1<name2; 1:name1>name2
4. Comparing Objects: == operator returns true if the two references are aliases of each other (the address will be compared for object references)

**equals method:** a. defined for all wrapper class to compare the content

b. For self-defined classes, it is the same as the == operator (compare the address) by default, but can be redefined under whatever conditions are appropriate

e.g. public boolean equals(Die3 d) {

return d.faceValue == faceValue;

}

**Static modifier**

1. Static -> the method or variable with the class rather than with an object of that class
2. Static/Class method:
3. Can be invoked without creating any object. e.g. MyMath.isPrime(num)
4. Static methods cannot reference instance variables (instance variables don't exist until an object exists)

But can reference static variables or local variables.

1. Static variables:
2. Only one copy of static variable exists. All objects instantiated from the class share its static variables. Changing the value of a static variable in one object changes it for all others.
3. Memory space for a static variable is created when the class is first referenced. (before instantiated)

**Interface**

1. Interface: a collection of **abstract methods** and constants (no instance variables)

**abstract method**: a method header without a method body public void doThis();

1. A class implements an interface -> Header: public class English **implements** Hello

Should define all methods in the interface

1. a. Use interface as a type to declare a reference for an object that implements that interface:

Hello c1 = new Chinese("Ying", "Zheng");

b. Use interface as a parameter to define an invoke method that takes in objects of similar behavior (that implement the interface) but of different classes

public static void invoke\_method(Hello person) {

person.sayHello();

}

1. A class can implement multiple interfaces:

class ManyThings implements interface1, interface2 { // all methods of both interfaces }

1. **Comparable** interface: definition ->

public interface Comparable {

int **compareTo**(Object other);

}

Implement:

public int compareTo(Object other) {

Salary salary = (Salary) other;

…}

Comparable as parameter: a single method can be used to compare different types of objects->

public static Comparable largest (Comparable c1, Comparable c2, Comparable c3)

**Inheritance**

1. The **child class** inherits the methods (except the constructors) and data defined by the **parent class**
2. Visibility modifiers:

Private: cannot be referenced by name in a child class

Public: allows the child class and every other class to reference

Protected: allows a child class to reference a variable or method directly in the child class

1. Establish an inheritance relationship: class UGStudent extends Student

When the object of child class is created, the no-argument constructor of parent class will be called automatically if not specified which one to be called.

To invoke parent’s constructor with specific argument: super(<parameter list>) -> Must in the first line of child constructor

1. **Multiple inheritance**: allows a class to be derived from two or more classes, inheriting the members of all parents -> resolved by the use of interface in Java
2. **Overriding methods**: A child class can override the definition of an inherited method (have the same overriding) in favor of its own

Overloading: multiple methods with the same name in the same class, but with different signatures

1. Shading variables: variables of child class have the same name as one in the parents

To distinguish: this.a super.a (<parent class name>)this.a -> not recommended

1. **Object** class: all classes are derived from the Object class

Contains methods such as *toString* and *equals*

1. If the *final* modifier is applied to a method, that method cannot be overridden in any descendant classes

If the *final* modifier is applied to an entire class, then that class cannot be used to derive any children at all

1. **Abstract class**: a placeholder, cannot be instantiated -> details may be provided in child class

can contain both abstract and non-abstract methods with full definitions (*abstract* modifier must be applied to each abstract method).

The child of an abstract class must override all abstract methods of the parent.

**Polymorphism**

1. Polymorphic reference: a variable that can refer to different types of objects at different points in time

All object references in Java are potentially polymorphic

1. Assigning a child object to a parent reference (widening conversion): Parent p = new Child();

-> can be compiled and executed

Assigning a parent object to a child reference (narrowing conversion) must be done with a cast:

childRef = (Child) new Parent(); -> can be compiled but will be runtime error

1. When the method is overridden in the decedent class, it is the type of the object being referenced, not the reference type, that determines which method is invoked

To invoke a method in child which is not defined in parent: **Typecast** the ancestor reference

-> ((Clerk)staffList[1]).addHours (40);

**Exceptions**

1. Try statement:

try {…

} catch (FileNotFoundException e) {…

} catch (InputMismatchException e) {…

}

finally {…

}

When an exception occurs, processing continues at the first catch clause that matches the exception type

The statements in the finally clause are always executed: If no exception is generated, it is executed after the statements in the **try block** complete; If an exception is generated, it is executed after the statements in the appropriate **catch clause** complete.

1. Exception Propagation: When an exception is not handled when it occurs, the control is returned back to the method that called the method, propagate up through hierarchy until they are caught and handled or until they reach the level of the main method

Exception message: e.getMessage()

Print exception trace: e.printStackTrace();

1. Exception Class Hierarchy: Throwable -> Error, Exception -> IOException, RuntimeException -> ArithmeticException, IndexOutOfBoundsException
2. (1) A **checked exception** (e.g. IOException) either must be caught by a method, or must be listed in the throws clause of any method that may throw or propagate it, otherwise cannot be compiled

e.g. public static void main (String args[]) throws FileNotFoundException

(2) An unchecked exception (only RuntimeException and its descendants) does not require explicit handling (catch or throw)

5. Define a new exception (see MakeSureInputCorrect.java)

**Data Structure**

1. Recursive method: must be structured to handle both the base case and the recursive case
2. Running Time Analysis

Asymptotic analysis: Look at growth of T(n) (running time of input n) as n→∞. Ignore low-order terms and leading constants

1. Big-O notation

if constants c and n0 such that 0 ≤ O(g(n)) ≤ c\*g(n) n0 ≤ n

Always use the tightest one -> 2n2 = O(n3) or 2n2 = O(n4) but not 2n2 = O(n)

1. merge sort: O(n log n) Insertion sort: (n2)
2. Inner Class: Class within another class

Outer class and Inner class can access the instance variables of each other directly.

**Linked list** (Dynamic data type)

public class MyList {

private MyNode list; //instance variable for the reference to the first node

// (Inner class) hidden from outside MyList

private class MyNode {

Object obj; //instance variable for the object it carries

MyNode next; // instance variable for the reference to the next node

MyNode(Object item) {

obj = item;

next = null;

}

}

//add a node at the front

public void addtoFront(MyNode newNode){

newNode.next = list;

list = newNode;}

//Find an object

public boolean contains(Object item) {

MyNode current = list;

while (current != null) {

if (current.obj.equals(item))

return true;

current = current.next;

}

return false;}

//Remove the front node

public void removeHead() {

if (list == null)

return;

list = list.next;

}

**Doubly linked list**

//remove one node in the doubly linked list

//the previous node of the first node and the next node of the last node is null

public void remove(MyNode node){

if (node.prev == null){

list = node.next;

node.next.prev = null;

}

else if (node.next == null)

node.prev.next = null;

else{

node.prev.next = node.next;

node.next.prev = node.prev;

}

**Queues**

(FIFO): adds items only to the rear of the list and removes them only from the front

The items can be kept in a linked list(more efficient) or an arraylist.

Operations:

enqueue - add an item to the rear of the queue

dequeue - remove an item from the front of the queue

empty - returns true if the queue is empty

size – returns the size of the queue

public class MyQueue {

private ArrayList<Object> queue;

public MyQueue () {

queue = new ArrayList<Object>();

}

public boolean empty() {

return queue.isEmpty();

}

public void enqueue(Object item) {

queue.add(item);

}

public Object dequeue () {

if (queue.isEmpty()) return null;

return queue.remove(0);

}

}

**Stacks**

(LIFO): Items are added and removed from only one end of a stack

Operations:

push - add an item to the top of the stack

pop - remove an item from the top of the stack

peek/top - retrieves the top item without removing it

empty - returns true if the stack is empty

public class MyStack {

private ArrayList<Object> stack;

public MyStack () {

stack = new ArrayList<Object>();

}

public boolean empty(){

return (stack.isEmpty());

}

public void push(Object item){

stack.add(0,item);

}

public Object pop(){

if (stack.isEmpty()) return null;

return stack.remove(0);

}

}

**Trees**:

Non-linear data structure

Exactly one path connecting two nodes together (N nodes -> N-1 links)

1. Binary trees:
2. **Siblings**: Nodes having the same parent
3. **Leaf**: A node that has no child <-> **internal node**: A node that has one or two children
4. **Subtree**: The node and all nodes descending from it
5. **Ancestor**: nodes lie on the path leading to the root <-> descendant: nodes lie in the subtree
6. **Height of Tree**: the number of edges on the longest path from the root to a leaf

//establish a binary tree

public class BinTree {

private BinTreeNode root;

public BinTree() {

root = null;

}

}

public class BinTreeNode {

private BinTreeNode parent, left, right;

private int item;

public BinTreeNode(int item) {

this.item = item;

parent = left = right = null;

}

public boolean isRoot(){

return (parent == null);

}

public boolean isLeftChild(){

if (parent == null) //always check NullPointerException

return false;

return (this == parent.left);

}

// return the distance from root to a node

public int distFromRoot(){

if (parent == null){

return 0;

else

return parent.distFromRoot()+1;

}

}

//remove a node

public void removeMyself(){

//is a leaf

if (left == null && right == null){

if (parent == null)

root = null;

else

parent.left = parent.right == null;

//only have left child

else if (left != null && right == null){

if (parent == null){

left.parent = null;

root = left;

}

else{

left.parent = parent;

if (isLeftChild())

parent.left = left;

else

parent.right = left;

}

else{

MyNode node = right;

//find the leftmost node of right subtree

while(node.left != null)

node = node.left;

if (parent == null){

root = node;

node.parent = null;

node.left = left;

node.right = right;

}

else{

if (node.right != null){

node.right.parent = node.parent;

if node.isLeftChild()

node.parent.left = node.right;

else

node.parent.right = node.right;

}

node.parent = parent;

node.left = left;

node.right = right;

if (isLeftChild())

parent.left = node;

else

parent.right = node;

}

}

}

//return the size of binary tree -> size(node) = size(left\_subtree)+size(right\_subtree)+1

public int size(){

int leftSize = 0;

int rightSize = 0;

if (left != null)

leftSize = left.size();

if (right != null)

rightSize = right.size();

return leftSize+rightSize+1;

}

1. Tree traversal :
2. Pre-order: root-left-right;
3. In-order: left-root-right;
4. Post-order: left-right-root

Public void inOrderTraversal(){

if (left != null)

left.inOrderTraversal();

System.out.print(item + ” “);

if (right != null)

right.inOrderTraversal();

}

1. Ordered-tree: in-order traversal presents the ascending order of the objects
2. Insertion(can use recursion):
3. if the root is null, make the new element to be the root
4. If the item being added is less than or equal to the root, add on the left subtree (If there is no left subtree, item is added as the left child)
5. If the item being added is larger than the root, add on the right subtree (If there is no right subtree, item is added as the right child)

Deletion:

1. If the node is a leaf, simply remove it
2. If the node has a single child, promote the child
3. If the node has two children, find the leftmost child on the right subtree to replace it
4. **Heap**: complete binary tree: each level is filled except possibly the bottommost level, which must be packed to the left if not completely filled.

The value stored at a node ≤ the value(s) stored at the child(ren)

The root of a heap contains the minimum number of the whole heap

1. Accessing the minimum number: constant time
2. Removing the minimum number: O(log2n) -> can only remove the smallest one
3. copy the last node to the root
4. Re-heapify the resultant tree: keep swapping the root with the smaller child until the parent is smaller than both children
5. Adding a new number:
6. Add the new number in the bottommost level
7. Re-heapify the resultant tree: keep swapping if parent is larger (poping up)
8. Heap sort: O(nlog2n)
9. Insert each number one by one to the heap: O(nlog2n)
10. Remove the minimum number from the heap one by one until the heap is empty: O(nlog2n)
11. Array implementation: (elements are stored level-by-level)

For element arr[i], left child -> arr[2\*i+1], right child -> arr[2\*i+2], parent -> arr[(i-1)/2].

1. **Huffman Encoding**: Ensures that the minimum number of bits is used to encode a passage using a binary code (string of 0’s and 1’s)

-> Based on frequency: the number of occurrence of each character is known

-> The larger the frequency, the shorter the code

(1) Process:

a. In the beginning, each node that represents a certain character forms its own subtree. The subtrees are sorted according to their frequencies carried in the root in ascending order.

b. Pick the two subtrees with the smallest frequencies and put them as the left child and right child of a new subtree. Keeps on going until all the subtrees are connected to form a single tree.

c. Either all edges leading to a left child are assigned to 1 or 0.

d. The code of the character is the bit pattern obtained by traversing the edges from the root to the leaf node representing the character.

(2) decode: traverse the tree to identify the characters

1. **AVL tree**: heights of the left and right subtrees differ by at most 1

The height of an empty tree is -1

1. Insert a new node: put it in the right position according to the order -> if the AVL tree property is violated, rebalance the tree through rotations:

(heights become imbalanced at node α)

1. left subtree of the left child too tall -> Single rotation at α (promote left child as parent)
2. right subtree of the right child too tall -> Single rotation at α (promote right child as parent)
3. left subtree of the right child is too tall -> double rotation (right child then α)
4. right subtree of the left child is too tall -> double rotation (left child then α)

类中的方法只针对类的实例变量