COMP2021 Review

Lecture 01 Intro to OOP and Java

Object : name + properties + receiving message

↳ directive to perform an action

↳ request to change of its properties

Simple Java Program

In d:\comp2021\Welcome.java

```
//This program prints "Welcome to Java!"
Public class Welcome {
   public static void main(String[] args) {
      System.out.println("Welcome to Java!");
   }
}
```

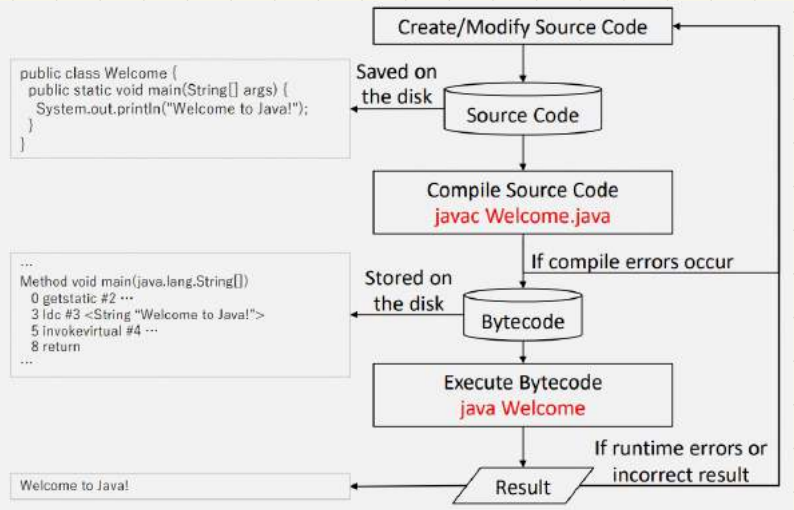
comments

class named "Welcome"

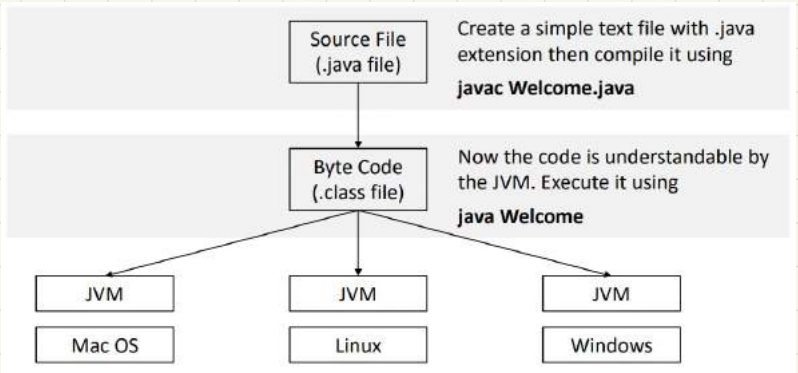
main method

statement : print a line

How to develop a Java Program



Write once, Run Anywhere



Lecture02 Java Basics

Identifier

start with letter, "_", "$" x reserve word any length

Name

variable and method name — camelCase

class name — CamelCase

constants name — UPPER_CASE

Data Type

Type	Contains	Default	Size	Range
boolean	true or false	false	1 bit	NA
char	Unicode character (Unsigned integer)	¥u0000	16 bits	¥u0000 to ¥uFFFF
byte	Signed integer	0	8 bits	-128 to 127
short	Signed integer	0	16 bits	-32768 to 32767
int	Signed integer	0	32 bits	-2147483648 to 2147483647
long	Signed integer	0	64 bits	-9223372036854775808 to 9223372036854775807
float	floating point	0.0	32 bits	±1.4E-45 to ±3.4028235E+38
double	floating point	0.0	64 bits	±4.9E-324 to ±1.7976931348623157E+308

Integer values Real values

Number System - Integer Type

byte / char / short / int / long Signed / unsigned

- unsigned : only positive value (magnitude)
- signed : positive / negative
  - signed magnitude
  - two's compliment → positive & 0 原码+最高位0
    → negative 取反+1
    - one representation of 0
    - subtraction by addition
- Conversion of integer type
  - unsigned and signed
  - widening : sign-extend
  - narrowing : discard high-order bits

Number System - Floating-Point Types

float and double
Scientific notation
IEEE standards

Number - Reading from Keyboard

```
Scanner input = new Scanner(System.in);      23<Enter>
int value = input.nextInt(); nextDouble() nextFloat()
                  nextByte() nextShort() nextLong()
String str = input.nextLine(); // first read input as string, then parse
```

Number - Numeric Operator

+	-	*	/	%	++var	--var
+=	-=	*=	/=	%=	var++	var--

Number - Numeric Type Conversion

Implicit — `double d = 3 //type widening`
Explicit (casting) — `int i = (int) 3.5 //type narrowing)`

precision vs. Range

```
int x = 12345678;
float y = x;
int z = (int) y;
```

Augmented assignment

```
short x = 2;
x = x + 1.1; //error
x += 1.1;    // OK!
//x = (short)(x + 1.1)
```

Error - integer overflow
- round-off error
- unintended integer division

Number - Boolean Type & Operator

True or False

```
boolean b = (1 > 2);
```

<	>	==
<=	>=	!=

! Not
^ Exclusive-or
&& And
|| Or

*Short circuit evaluation

```
int x=1, y=2; boolean b = x<0 && y++>2
System.out.println(x+","+y);
```

1,2

false — will not reach

Number - Bitwise & Bit Shift Operator

~ bitwise complement
& bitwise and
| bitwise or

* Eagle Logical Operators

>> signed right shift
<< signed right shift
>>> unsigned right shift

```
int x=1, y=2; boolean b = x<0 & y++>2
System.out.println(x+","+y);
```

1,3

false

Conditional Expressions

boolean-expression ? expl : exp2.
true false

```
if (num % 2 == 0)
   System.out.println(num + "is even");
else
   System.out.println(num + "is odd");

// is equivalent to the following
System.out.println(
      (num % 2 == 0)? num + "is even" : num + "is odd");
```

Java Operator Precedence Table

Operator	Description	Level	Associativity
[] . () ++ --	access array element access object member invoke a method post-increment post-decrement	1	left to right
++ -- + - ! ~	pre-increment pre-decrement unary plus unary minus logical NOT bitwise NOT	2	right to left
() new	cast object creation	3	right to left
* / %	multiplicative	4	left to right

Java Operator Precedence Table (Cont'd)

Operator	Description	Level	Associativity
+ - +	additive string concatenation	5	left to right
<< >> >>>	shift	6	left to right
< <= > >= instanceof	relational type comparison	7	left to right
== !=	equality	8	left to right
&	bitwise AND	9	left to right
^	bitwise XOR	10	left to right
|	bitwise OR	11	left to right
&&	conditional AND	12	left to right
||	conditional OR	13	left to right
?:	conditional	14	right to left
= += -= *= /= %= &= ^= |= <<= >>= >>>=	assignment	15	right to left

30

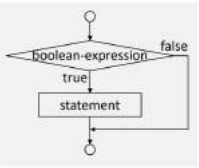
* Evaluation order: precedence / Associativity

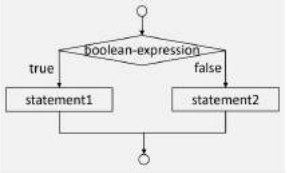
Control Structure

if

```
if(boolean-expression)
   statement;
```

```
if(boolean-expression)
   statement1;
else
   statement2;
```






```
if(condition1)
   statement1;
else
   if(condition2)
      statement2;
   else
      if(condition3)
         statement3;
      else
         statement4;
```

is equivalent to

```
if(condition1)
   statement1;
else if(condition2)
   statement2;
else if(condition3)
   statement3;
else
   statement4;
```

```
if(number % 2 == 0)
   isEven = true;
else
   isEven = false;
```

is equivalent to

```
boolean isEven
      = number % 2 == 0;
```

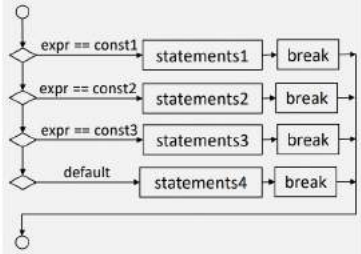
```
if(even == true){
   statement(s);
}
```

is equivalent to

```
if(even){
   statement(s);
}
```

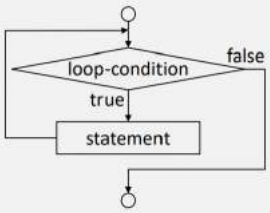
switch

```
switch(expr){
 case const1: statements1;
              break;
 case const2: statements2;
              break;
 case const3: statements3;
              break;
 default:     statements4;
              break;
}
```

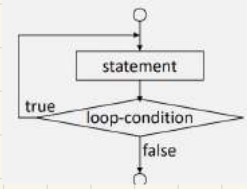


while

```
while(loop-condition)
   // loop body
   statement;
```

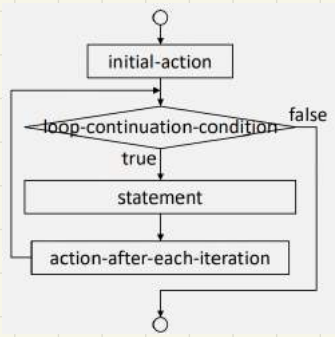



```
do
   // loop body
   statement;
while(loop-condition)
```



for

```
for(initial-action;
    loop-continuation-condition;
    action-after-each-iteration)
  // loop body
  statement;
```



* break, continue

Lecture 04 Object-based Programming

Object and Classes

object — Java class defines data types

field/instant variable — defines info/state an object contains

method — what object can do

```
class Hero{
    String name;      //Field
    int health;       //Field
    void initHero(){...}          //Method
    void increaseHealth(int amount){...} //Method
}
```

Reference

object contain all fields defined in its class

reference only contain null/address of its object

```
Hero heroA, heroB, heroC; // three references
```

Primitive Type & Reference Type

		assign (x=y)	comparision (x==y)
primitive type:	boolean, byte, short, char, int, long primitive variable store primitive value	value copied	value compared
reference type:	all other type reference variable store address	address copied	address compared

Accessing member of an Object

member access operator(.) → access a member field — `heroA.health += 2;`

→ calling a method — `heroA.increaseHealth(2);`

Member Method Declaration & Invocation

```
class Hero{
  String name; int health;
  void increasingHealth(int amount){...} // Declaration
} Hero heroA = ...;
  heroA.increaseHealth(2); //Invocation
```

Compile ⟹

```
void increaseHealth(Hero this, int amount){...}
increaseHealth(heroA, 2);
```

Member Method Implementation

```
void increaseHealth(int amount){
    this.health += amount; //health += amount }
heroA.increaseHealth(2); ⟶ if(heroA==null) //error
                             increaseHealth(heroA, 2);
```

Variable Scope

from statement, to end of block

explicit member access — `this.health` in line 7

*mistake: declare before for loop & use it after loop

```
1  class Hero{
2    int health;
3
4    void heal(int amount){
5      int j = 5;
6      for(int i = 0; i < j; i++){
7        this.health += amount;
8      }
9    }
10
11   int gold;
12 }
```

Evaluation of Argument

temp variables

Method Invocation *

Argument Passing pass-by-value

* example swap(int a, int b)
swap(Hero x, Hero y)//swap health

Object Creation

* create on the heap X stack

Stack and Heap

stack : store temporary information efficient / limit storage / strict order in (de)allocation

heap: object stored in section of memory slower / larger storage / more flexible

Default Values for Object Field

set to 0's

Constructors

some name as class name

no return type

may some argument

constructor

```
class Hero{
  int health;

    Hero(int h){
      health = h;
    }
}
```

```
Hero heroA = new Hero(3);
```

default one

```
class Hero{
  int health;
}
```

```
Hero heroA = new Hero(); // OK
// What's the value of
//   heroA.health?
```

normal one

```
class Hero{
  int health;

  Hero(int h){
    health = h;
  }
}
```

```
Hero heroA = new Hero(3); // OK
Hero heroB = new Hero();  // Error
```

Default constructor

no-arg constructor

Overload Methods

more than one constructor

different list of parameter type

call one constructor from another

first statement

form: this(...)

method overloading

System.out.

overload

```
class Hero{
  int health;

  Hero(){ // requires no explicit parameter
    health = 5;
  }

  Hero(int h){ // requires one explicit parameter
    health = h;
  }
}
```

```
Hero heroA = new Hero(3); // OK
Hero heroB = new Hero();  // OK
```

call one constructor from another

```
class Hero{
  String name;
  int health;
  Hero(String a){
    name = a;
  }

  Hero(String a, int h){
    this(a);
    health = h;
  }
}
```

```
Hero heroA = new Hero("Tom");
Hero heroB = new Hero("Tom", 5);
```

System.out is of type PrintStream

```
class PrintStream{
  ...
  void println(boolean b){...}
  void println(char c){...}
  void println(double d){...}
  void println(float f){...}
  ...
}
```

Hero

```
class Hero{
  ...
  void fight(Daemon daemon){ ... }
  void fight(Monster monster){ ... }
  void fight(Zombie zombie){ ... }
  ...
}
```

Static Member

full

Lecture 04 Object-based Programming

Information Hiding (Encapsulation)

- no access control
  - risk corrupting data
  - change implementation
- access control (accessible)
  - public - all other codes
  - private - in same class
  - package - in same package

*Modifier

* public
* private
* no keyword needed

Field vs. Method

Field
- internal implementation
- volatile
- strong bond

Method
- external behavior
- stable
- week bond

```
// Hero.java
class Hero{
  private String name;

  public void setName(String name){
    if(name == null || name.isEmpty()){
        // report the error!
    }
    this.name = name;  // in class Hero
  }

  public String getName(){
    return name;       // in class Hero
  }

  ...
}
```

```
// HeroTest.java
Hero heroA = new Hero();
heroA.name = "M. J.";    // error!
heroA.setName(null);     // error!
heroA.setName("M. J."); // OK!
```

No unauthorized or uncontrolled access!

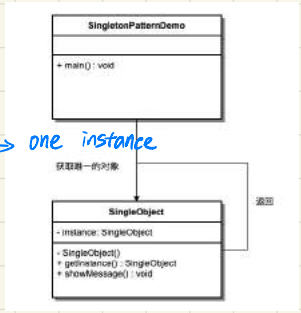
setter & getter

Controlling Access to Classes

Modifier	Class Itself	Other Class (same pkg)	Subclass (same pkg)	Subclass (diff pkg)	World
public	Y	Y	Y	Y	Y
(no modifier)	Y	Y	Y	N	N
protected	Y	Y	Y	Y	N
private	Y	N	N	N	N

Singleton

one instance




```
private Logger instance
public Logger getInstance()
main(...){Logger.logger1 = Logger.getInstance()}
```

Constants

initialized cannot be changed

- Class variables

```
public static final int MAX_HEALTH = 100;
```

```
public static final int MAX_HEALTH;
...
static { MAX_HEALTH = 100; }
```

- Instance variables

```
class Hero{
  private final int ID = 1;
  ...
}
```

✓

```
class Hero{
  private final int ID;
  public Hero(){ ID = nbrCreated++; }
  ...
}
```

✓

```
class Hero{
  private final int ID;
  { ID = nbrCreated++; }
  ...
}
```

✓

```
class Hero{
  private final int ID = 1;
  { ID = nbrCreated++; }
  ...
}
```

✗

Java Packages

- organized codes
- avoid name collision
- provide access protection
- package naming convention
  - lower-case, hierarchy seperated by "."

creating package
- first non-comment statement.
- fully qualified name
- if no, default package

Classpath

```
…\jdk\src;…\mylib\src; // On Windows.
.../jdk/src:…/mylib/src // On MacOS.
```

Can be set in an environment variable or passed to Java tools as an argument

```
set classpath =…\jdk\lib;…\mylib1;
```

```
java –classpath …\jdk\lib;…\mylib2 <other parameters>
```

fully-qualified name

refering to Java class in Program

import statement

```
import hk.edu.polyu.comp.comp2021.quiz.Example;
// Refer to the class using its simple name "Example"
```

```
import hk.edu.polyu.comp.comp2021.quiz.*;
// Refer to all classes from the package using simple names
```

name ambiguity

refer by its qualified name

```
import hk.edu.polyu.comp.comp2021.quiz.*;
import hk.edu.polyu.comp.comp2021.assignment.*;

...

hk.edu.polyu.comp.comp2021.assignment.Example e
  = new hk.edu.polyu.comp.comp2021.assignment.Example();
```

Apparent hierarchy of package

* Class1 in a.b.c Class2 in a.b

```
import a.b.*; ...
Class1 c1 = ...; //error
Class2 c2 = ...; //OK!
```

```
import a.b.*; cannot reach subpackage
import a.b.c.*; ...
Class1 c1 = ...; //OK!
Class2 c2 = ...; //OK!
```

* GC algorithm

Garbage collection

- Instantiating an object – new operator
  - Allocate memory
  - Reset memory to 0's
  - Initialize object
  - Return a reference
- Destroying an object? No explicit way to do that!

```
void fight(Hero other){
  Hero tmpHero = new Hero(); // creates an object
  tmpHero = other;           // the object becomes inaccessible
  ...
}
```

> Garbage collection (GC) is a form of automatic memory management. The garbage collector attempts to reclaim garbage, or memory occupied by objects that are no longer in use by the program.
>
> -- Wikipedia

- Pros
  - Eliminates the need for explicit memory deallocation
  - Simplifies program writing
  - Helps ensure program integrity
  - Reduces heap fragmentation
- Cons
  - Extra overhead in time and space
  - Less control over task scheduling
    - When to run GC is decided by the JVM
    - A program can request, but no response is guaranteed

Lecture 05 Object-based Programming

Character

16 bit unicode \u followed by 4 hexadecimal number
ASCII Character Set
Escape Sequence \n \b \t \\ \"

Operation
- Character.isDigit(c)
- ....isLetter(c)
- ....isLetterOrDigit(c)
- ....isLowerCase(c)
- ....isUpperCase(c)
- ....toLowerCase(c)
- ....toUpperCase(c)

String

String of characters
immutable

Method	Description	Result
s1.length()	Returns the number of characters in s1	5
s1.charAt(0)	Returns the character at the specified index from s1	'J'
s1.concat(s2)	Returns a new string that concatenates s1 with s2 Note: the same as s1 + s2	"Java !"
s1.toUpperCase()	Returns a new string with all letters in uppercase	"JAVA "
s1.toLowerCase()	Returns a new string with all letters in lowercase	"java "
s1.trim()	Returns a new string with whitespace characters trimmed on both sides	"Java"

Method	Description	Result
s1.equals(s2)	Returns true if s1 and s2 have the same content Note s1 == s2 tests something different	false
s1.startsWith(s2)	Returns true if s1 starts with s2	false
s1.endsWith(s2)	Returns true if s1 ends with s2	true
s1.substring(1, 3)	Returns the substring of s1 starting from position 1 and ending at position 2 (= 3 – 1) Note s1.charAt(3) is not included in the substring	"av"
s1.indexOf('a', 0)	Returns the first index in s1 starting from 0 where 'a' appears	1
s1.indexOf(s2, 0)	Returns the first index in s1 starting from 0 where s2 appears	2
s1.lastIndexOf('v', 3)	Returns the last index in s1 before position 3 where 'a' appears	2
s1.lastIndexOf(s2, 3)	Returns the last index in s1 before position 3 where s2 appears	2

String → number Integer.parseInt(intString);
Double.parseDouble(doubleString);

Number → String String.valueOf(number)
+ operator if one operand is string → string concatenation

StringBuilder

mutable string

```
StringBuilder builder = new StringBuilder("a");
builder.append("b").append("c").append("d");
String a = builder.toString();       // Convert to String
```

Format Output

System.printf(format, items);

Specifier	Output	Example
%b	A Boolean value	true or false
%c	A character	'a'
%d	A decimal integer	30
%f	A floating point number	45.460000
%s	A string	"Java is cool"

```
String name = "Jack";
int count = 5;
float amount = 3.5f;
System.out.printf("Hi %s, count is %d, amount is %f!",
                  name, count, amount);
// output: Hi Jack, count is 5, amount is 3.500000!
```

Array

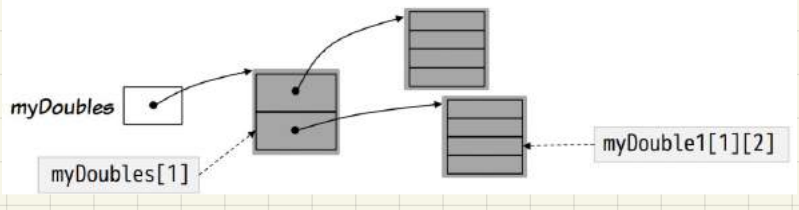
Variable declaration double[] myDoubles;
Array creation myDoubles = new double[10];
Equivalence (==) (==) compare reference (=) assign reference
length int len = myDouble.length;
Initializer

```
double[] myDoubles = {1.9, 2, 3.5, 9.1}; // all-in-one

double[] myDoubles;
myDoubles = {1.9, 2, 3.5, 9.1}; // error! not in assignment
```

Multi-Dimension Array

```
double[][] myDoubles;
myDoubles = new double[2][4];
```



* Ragged Arrays

```
int[][] myIntegers = {
  {1, 2, 3, 4, 5},
  {2, 3, 4, 5},
  {3, 4, 5},
  {4, 5},
  {5}
};
```

```
int[][] myIntegers = new int[5][];
myIntegers[0] = new int[5];
myIntegers[1] = new int[4];
myIntegers[2] = new int[3];
myIntegers[3] = new int[2];
myIntegers[4] = new int[1];
```

```
for(int i = 0; i < myIntegers.length; i++){
  int[] ele = myIntegers[i];
  for(int j = 0; j < ele.length; j++){
    ...
  }
}
```

Enums

constant in all caps
enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES}
classes
fix number of instance
field, methods, constructors
as type
compare by ==, compareTo(...)
switch statement

* adv. type safety
  name space
  robustness 稳健性
  informative printout — convert String
  add field, method.

Wrapper Classes

constructor with argument

immutable

```
+ valueOf(s: String): Integer
+ valueOf(s: String, radix: int): Integer
+ parseInt(s: String): int
+ parseInt(s: String, radix: int): int
```

Similar Structor

Autoboxing and Unboxing

primitive typed value → (box) appropriate wrapper class
← (unbox) wrapper class

BigInteger & BigDecimal

package java.math, immutable

Unit test with JUnit

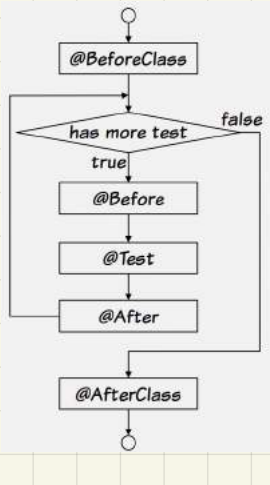
```
import org.junit.Test;
       Assert,...;
public class ...Test{
  @Test
  public void test...(){
    assertEqual("...", 0, ...());
  }
  ...
}
```

independent test

```
assertEquals()
assertTrue()
assertFalse()
assertNotNull()
assertNull()
assertSame()
assertNotSame()
assertArrayEquals()
fail()
```

@Test @Timeout @Ignore

@BeforeClass } static method
@AfterClass }
@Before } instance method
@After }



Lecture 06 Object-oriented Programming

Inheritance reusability

* tree structure
  base class vs derived class

inheriting member from existing class
→ adding new member
→ replace existing member
→ new class more specific

default class can't extend from default class
superclass's field & method inherited (member)
× inherited × constructor, static initializer, instance initializer
member: public & protected
private (setter, getter)

			p.A	p.B	p.C extends p.A	q.D extends p.A	q.E
p.A	Field (static or not)	public	Y	Y	Y	Y	Y
		protected	Y	Y	Y	Y	N
		(default)	Y	Y	Y	N	N
		private	Y	N	N	N	N
	Method (static or not)	public	Y	Y	Y	Y	Y
		protected	Y	Y	Y	Y	N
		(default)	Y	Y	Y	N	N
		private	Y	N	N	N	N

- p, q: packages
- A, B, C, D, E: classes

Constructor

chaining

```
class A{
  public A(){System.out.print("1");}
  public A(int x){System.out.print("2");}
}
class B extends A{
  public B(){
    this(1);
    System.out.print("3");
  }
  public B(float x){ System.out.print("4"); }
}
class C extends B{
  public C(){ System.out.print("5"); }
}
```

```
// What's the output?
C cObj = new C();
```

```
1435
```

new class instance
class initialization
static member

from top superclass to bottom subclass

Define Subclass

new field, new method, override method from superclass

Override

modify implementation of method it inherits from superclass

```
public class Hero{
  public void run(int x){ ... }
  ...
}
public class Wizard extends Hero{
  public void run(int x){ ... } // overriding
  ...
}
public class Knight extends Hero{
  public void run(float x){ ... } // overloading
  ...
}
```

Polymorphism (many-form)

```
public class Hero{
  private String name;
  private int health;

  public Hero(){...}
  public void fight(){...}
  ...
}
```

```
public class Monster extends Hero{
  private boolean isAngry;

  public Monster(){...}

  ...
}
```

```
Hero h;
h = new Hero();
...
h = new Monster();
...
```

IS-A relation: A monster is also a hero, as it is composed of the same information (fields) and has the same behavior (public methods)

Type and SubType

type decided by constructor

type defined by superclass → super-type
type defined by subclass → subtype

Static and dynamic Type

```
Hero h;          // static type of h is always Hero
h = new Hero();  // dynamic type of h is Hero
h = new Monster(); // dynamic type of h is Monster
```

Casting Objects

from subtype type to supertype

```
Hero h;
h = new Monster(); // implicit casting
```

```
Hero h1 = new Hero(), h2 = new Monster();
Monster m;

if(h1 instanceof Monster){ // Check at runtime if h1 refers
 m = (Monster) h1;     // to an object of type Monster
}                      // or a subtype of Monster

if(null instanceof Monster){ // Always return false if the
 ...                         // left operand is null
}
```

```
Potato p = new Potato();

if(p instanceof Monster){ // Compilation error! Because
 ...                      // Potato and Monster are in
}                         // different class hierarchies
```

Instanceof Operator

test whether object is an instance of a class

Binding

dynamic binding depend on receiver

```
public class Hero{
 private String name;
 private int health;

 public Hero(){...}
 public void fight(){...}
 ...
}
```

```
public class Monster extends Hero{
 private boolean isAngry;

 public Monster(){...}
 public void fight(){...}
 ...
}
```

```
Hero h;
h = new Hero();
h.fight(); // calling fight() defined in Hero
h = new Monster();
h.fight(); // calling fight() defined in Monster
```

Static binding static member
on this or super → static

dynamic binding override

Polymorphic reference & call

```
Hero[] heroes = ...;
...
for(int i=0; i<heroes.length; i++){
 Hero hero = heroes[i];
 if(hero instanceOf Hero){
  hero.fight ();
 }
 else if(hero instanceOf Monster){
  Monster monster=(Monster)hero;
  monster fight ();
 }
 ...
}
```

```
Hero[] heroes = ...;// polymorphic refs
...
for(int i=0; i < heroes.length; i++){
 // polymorphic call:
 // selects the right fight() to call
 // based on heroes[i]'s dynamic type
 heroes[i].fight ();
}
```

method overloading vs. method overriding

Overriding	Overloading
Method names should be the same	
Overrides a method from superclass	Overloads a *member method*
With the same signature	With a different signature
Is a runtime concept	Is a compile-time concept
Return type must be the same or a subtype	Return type does not matter
Accessibility must be more permissive	Accessibility does not matter

Lecture 07 Object-oriented Programming

Class Object extends java.lang.Object class

public String toString()

public boolean equals(Object other)

public int hashcode() →

```
contains(o): buckets[o.hashCode() % buckets.length].contains(o)
add(o): buckets[o.hashCode() % buckets.length].add(o)              if(!contains(o))
remove(o): buckets[o.hashCode() % buckets.length].remove(o)        if(contains(o))
```

public Object clone()

Abstract class & Abstract methods

with one abstract method

Current class does not provide any implementation.

Inheriting from Abstract Class

subclass of abstract class is also abstract

* override all abstract method in superclass

△ No Instantation

Abstract Class & Concrete Class

no abstract method can be abstract class → prevent creating instance.

subclass override superclass (concrete class → abstract).

Interface group of related method with empty bodies

instance member

no field allowed

method: public & abstract

static member

all field: public, static, final

all method: public, concrete

implement

```
public interface Shape{
  double area();
  double CENTIMETERS_PER_INCH = 2.54;
  static double inchToCM (double inch){···}
}
public class Circle implements Shape{
  public double area() { return Math.PI * radius * radius;}
  ...
}
```

Static (access)

implement abstract.

extend

use interface as a type

sharing a reference

* Clonable interface

```
package java.lang;
public interface Cloneable { }
```

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

// clone()

- Calling clone() on a non-Cloneable object will trigger a CloneNotSupportedException

shallow & deep copy

shallow copy: change old object also change cloned object

deep copy: public Object clone() {...}

clone × object creation problem: duplicate code from constructor
limit capability to clone field

solution: 1. copy constructor
2. clone + copy constructor

multiple inheritance

characterized in different way.

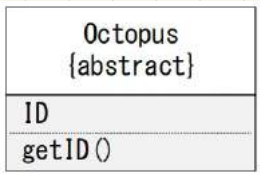
problem: name ambiguity
impact on substitutability.
× extend multiple super-class
√ implement interface

Lecture 08 UML & Exception Handling

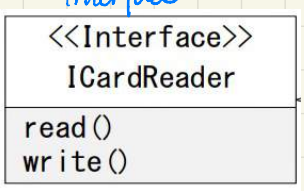
UML/class diagram

Class

abstract



interface



Relationships

dependency - - - - - - - - ->

association ——— do ———▶

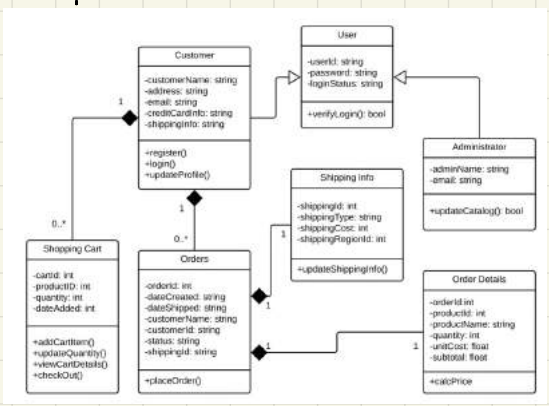
* multiplicity

has-a (aggregation) ◇———

(composition) ◆———

(strong aggregation)

example



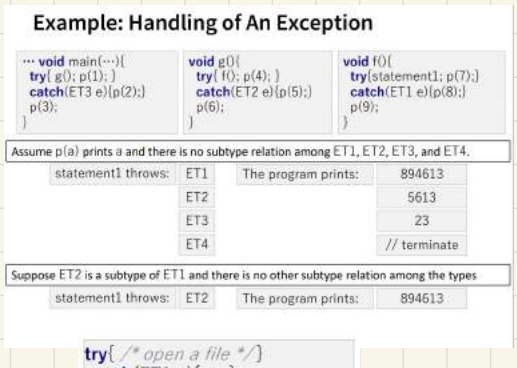
exception handling

exception handler: throw, try, catch

try

↓

catch : specified type of exception

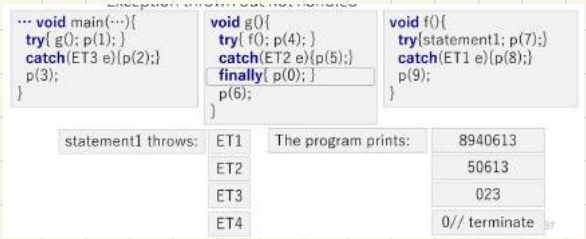



```
try{ /* open a file */}
catch(ET1 e){ ··· }
finally{/*always close file*/}
```

simple: catch (ET1 e) { ... }

multi: catch (ET1 e|ET2 e|ET3 e){ ... }

finally block



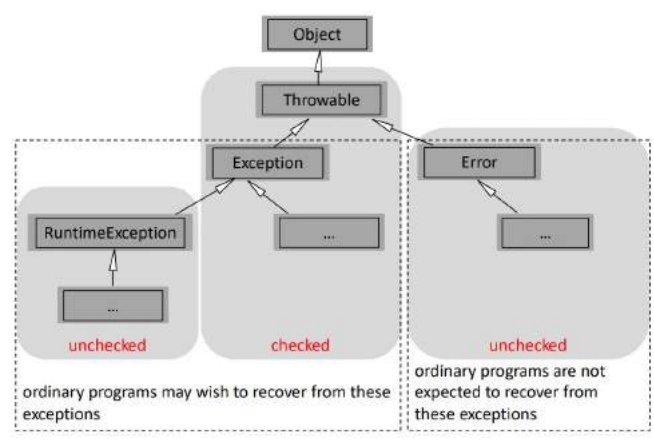
catch → rethrow it/wrap with another exception

try in finally block/catch

*try with resource

exception type: uncheck exception

checked exception



Lecture 09 Generics & Concurrency

Generic method & Classes

Container → fixed-size object array
→ more flexible

Stack → copy and modify push() pop()

Generic class → parameterized
raw type (without any type argument)

Method: Scope of type parameter limited

Generic types and inheritance.

bounded type Inheritance: extends (composite type)

* multiple bounds first class

wild cards ?

Multiple Threads

Concurrency
process → running program, independent address

thread → share address

Thread public class Thread extends Object implements Runnable {...}

extend class Thread
- override run() method

```
public class DumbThread extends Thread {

  private Hero hero;

  public DumbThread(Hero hero) {
    this.hero = hero;  // now the thread shares a Hero object
  }                    // with its creator

  public void run() {
    System.out.println("The health of hero is " + hero.health);
  }
}

Thread t = new DumbThread(aHero); // create a thread
t.start(); // Start thread t (start() calls run())
t.join(); // wait until thread t terminates
System.out.println("The thread has terminated");
```

Note aHero is shared between the main thread and the new DumbThread

What if we call t.run(), instead of t.start()?

implement interface Runnable
- must implement run() method

```
public class DumbThread implements Runnable {

  private Hero hero;

  public DumbThread(Hero hero) {
    this.hero = hero;  // now the thread shares a Hero object
  }                    // with its creator

  public void run() {
    System.out.println("The health of hero is " + hero.health);
  }
}

Thread t = new Thread(new DumbThread(aHero)); // create a thread
t.start(); // start thread t
t.join(); // wait for thread t to terminate
System.out.println("The thread has terminated");
```

This is more desirable than extending class Thread. Why?

* Coordination

Lecture 10 Concurrency

Coordination Mechanism

mutex (mutual exclusion object)
lock()
unlock()

Java object and Monitor

monitorenter
monitorexit } obtain/release unlock on object ⇒ synchronized block

aObject.wait() make current thread wait
aObject.notify() notify one thread wait
aObject.notifyAll() notify all thread wait

Synchronized block

- it is as if the method body is a synchronized(this) block

```
// hero must be accessed in mutual exclusion
private Hero hero;
public synchronized void decreaseHealth(){ // this.monitorenter()
 // critical region
 if (hero.getHealth() > 0) {
  hero.decreaseHealth(1);
 }
}                                  // this.monitorexit()
public synchronized void increaseHealth(){ ... }
```

- Equivalent to

```
public void decreaseHealth(){
 synchronized(this){
  if (hero.getHealth() > 0) { hero.decreaseHealth(1); }
 }
}
...
```

Static

- The monitor associated with the class is used for locking

```
class Test{

 public static synchronized void test(){ // Test.class.monitorenter()
  ...
 }                                  // Test.class.monitorexit()

}
```

- Equivalent to

```
class Test{
 public static void test(){
  synchronized (Test.class){ ... }
 }
}
```

* P22-24 Producer-Consumer Example

monitor: lack of concurrency,

Thread.sleep(int t) interrupt occurs

termination: stop() destroy()
garbage c cleanup release lock
no cleanup no release lock

cancel interrupt()
interrupted() check

* synchronize block
choose lock
(more granularity)
sync (object)

method
this → lock

Synchronized

Lecture 11 Swing & Event-Driven Programming.

Nested class

class can be defined as member of another class

→ increase encapsulation
→ readable
→ logically group class

static → `OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();`

non-static →

reference

`OuterClass.InnerClass innerObject = outerObject.new InnerClass();`

Local Class

Lambda Expression

(parameters) → expression
{ statement ... }