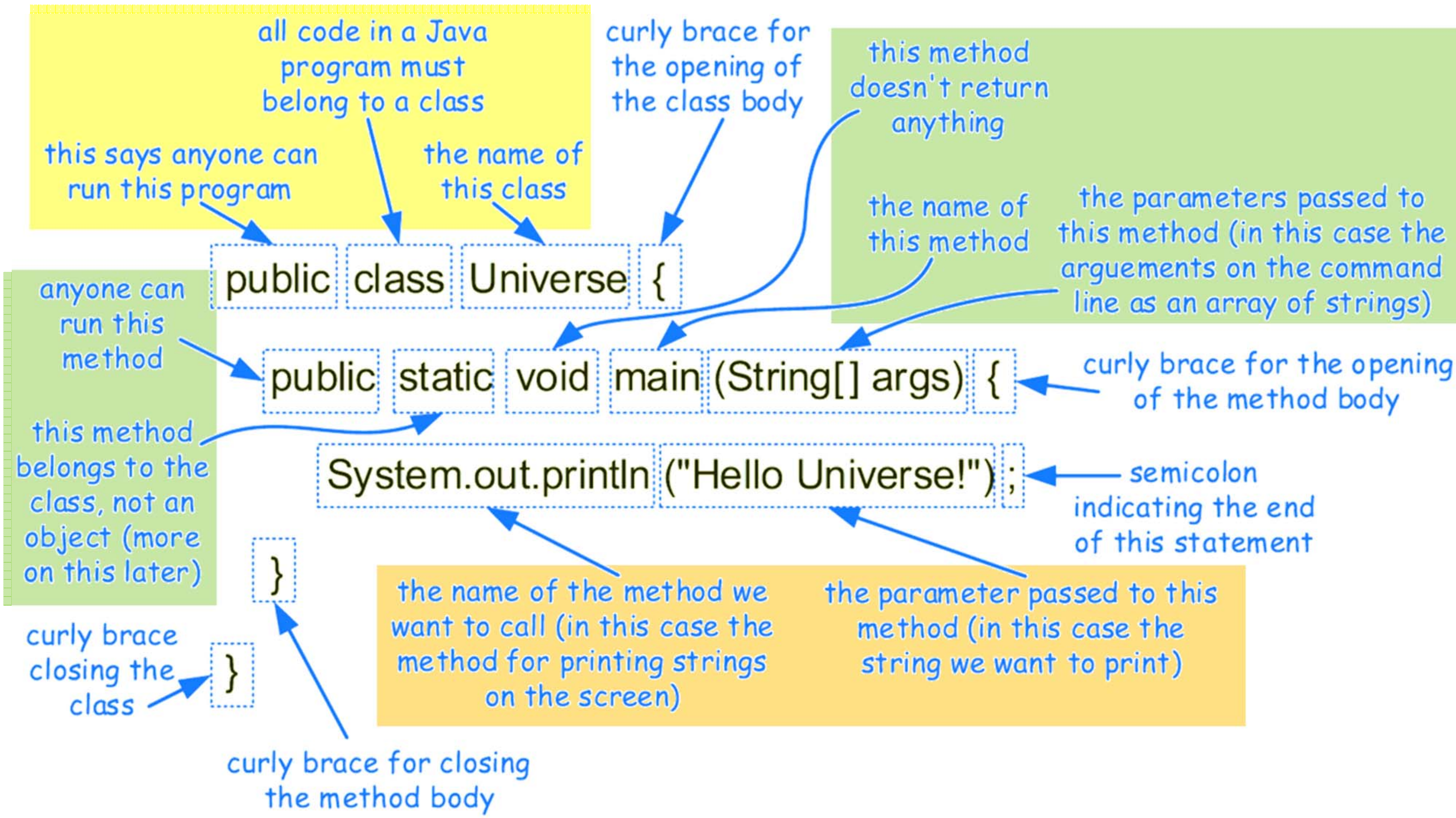
CSE214 Data Structures
Java Objects & Classes

YoungMin Kwon

Hello Universe Program



Primitive Types (Values)

boolean	a boolean value: true or false
char	16-bit Unicode character
byte	8-bit signed two's complement integer
short	16-bit signed two's complement integer
int	32-bit signed two's complement integer
long	64-bit signed two's complement integer
float	32-bit floating-point number (IEEE 754-1985)
double	64-bit floating-point number (IEEE 754-1985)

```
boolean flag = true;
boolean verbose, debug; // two variables declared, but not yet initialized
char grade = 'A';
byte b = 12;
short s = 24;
int i, j, k = 257;            // three variables declared; only k initialized
long l = 890L;                // note the use of "L" here
float pi = 3.1416F;           // note the use of "F" here
double e = 2.71828, a = 6.022e23;   // both variables are initialized
```

Objects and Classes (Addresses)

- Object
  - An instance of a class
  - Reference type: address of an object instance
- Class
  - Type of a object
  - Fields: data associated with an object
  - Methods: blocks of code to perform actions

Objects and Classes

```
public class Counter {
  private int count;      // a simple integer instance variable
  public Counter( ) { } // default constructor (count is 0)
  public Counter(int initial) { count = initial; }      // an alternate constructor
  public int getCount( ) { return count; }              // an accessor method
  public void increment( ) { count++; }                 // an update method
  public void increment(int delta) { count += delta; }  // an update method
  public void reset( ) { count = 0; }                   // an update method
}
```

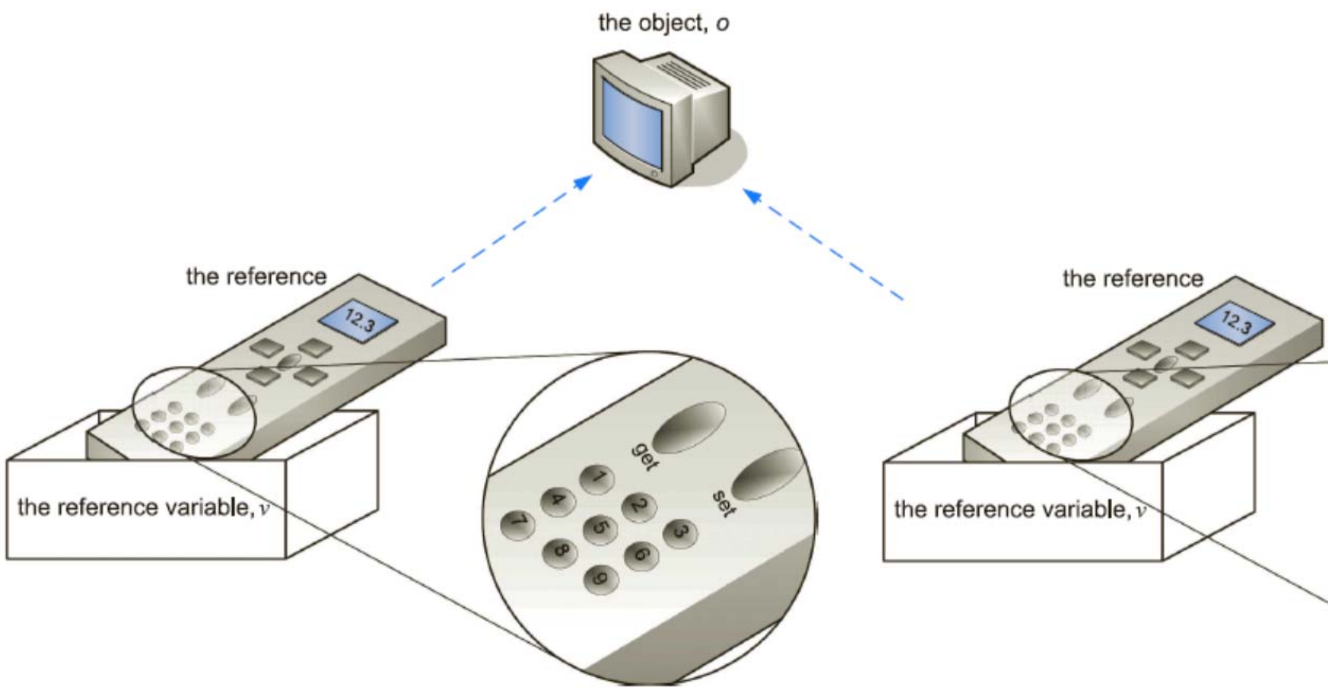
Creating and Using Objects

```
public class CounterDemo {
  public static void main(String[ ] args) {
    Counter c;                      // declares a variable; no counter yet constructed
    c = new Counter( );             // constructs a counter; assigns its reference to c
    c.increment( );                 // increases its value by one
    c.increment(3);                 // increases its value by three more
    int temp = c.getCount( );       // will be 4
    c.reset( );                     // value becomes 0
    Counter d = new Counter(5);     // declares and constructs a counter having value 5
    d.increment( );                 // value becomes 6
    Counter e = d;                  // assigns e to reference the same object as d
    temp = e.getCount( );           // will be 6 (as e and d reference the same counter)
    e.increment(2);                 // value of e (also known as d) becomes 8
  }
}
```

Reference Variable

- Reference variable
  - Holds the location of the class object instance
  - E.g. c, d, e in the previous code
- Dot operator
  - To access the members of an instance of a class
    - E.g. `c.increment(3);`
- Signature of a method
  - Method's name combined with the number and the type of its parameters
  - Signature does not include the return type of a method

Object and Reference Variable



Counter d = **new** Counter(5);

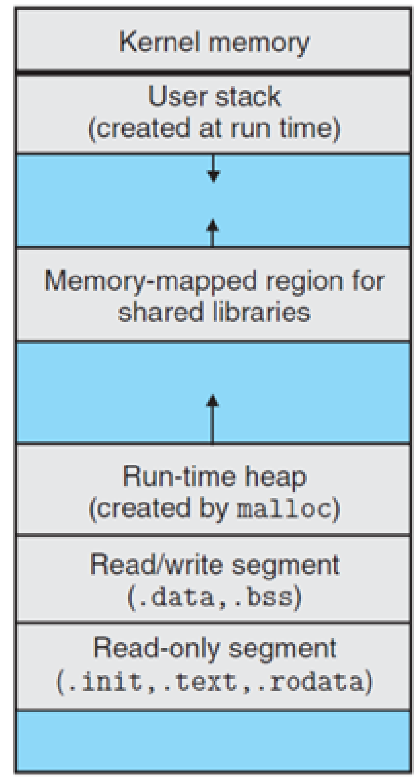
Counter e = d;

Creating an Object (3 Events)

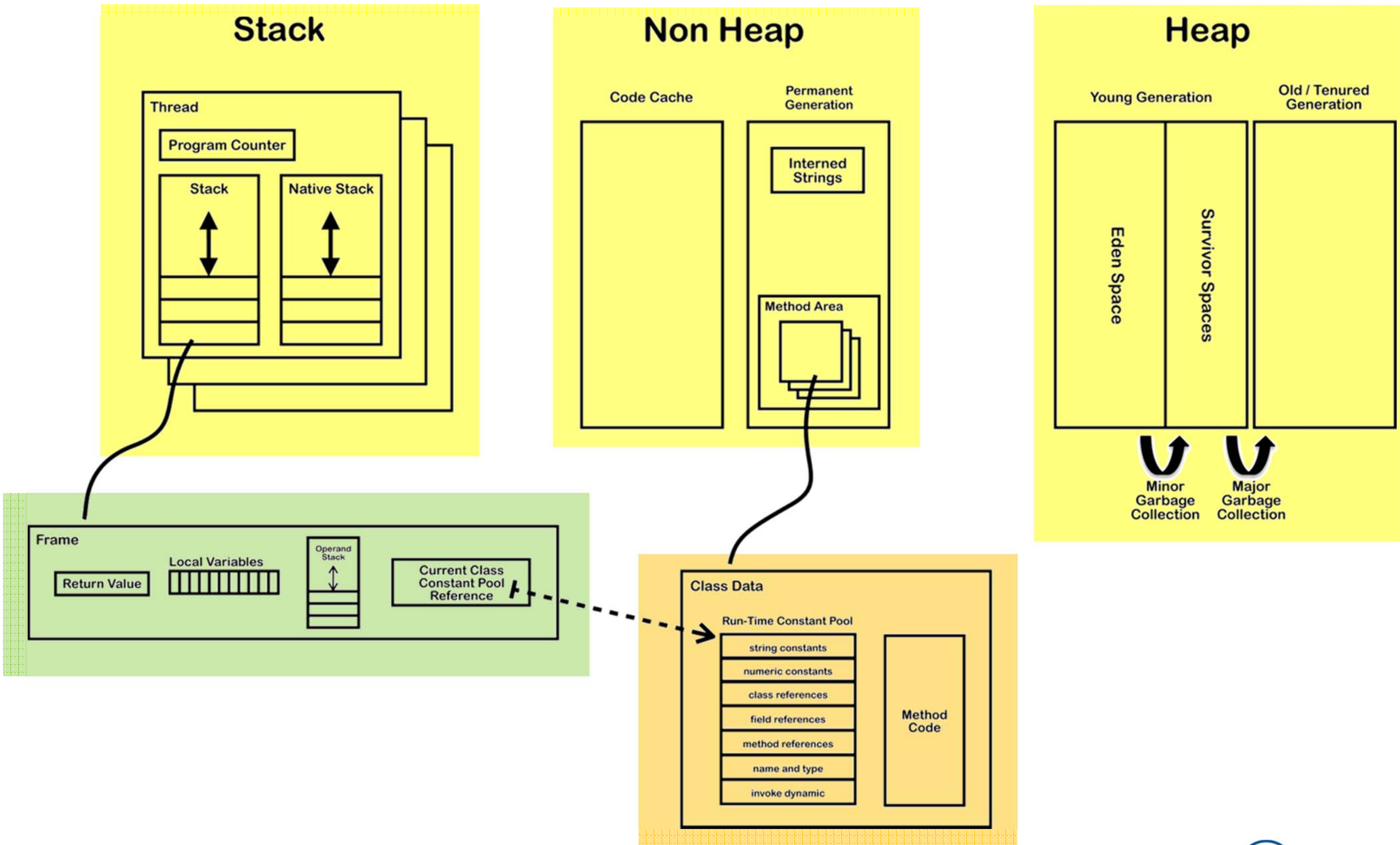
- A new object is dynamically allocated in memory
  - Fields are initialized to their standard default values
    - Numeric types: 0, boolean type: false, char type: null char, reference type: null
- The constructor for the new object is called with the parameter specified
- new operator returns a reference (memory address) to the newly created object

Memory Layout of a Process

- Every process has the same *virtual* memory space
  - 4 GB in 32bit Linux system
  - .text: assembly instructions
  - .rodata: read only data
    - E.e. "Hello World"
  - .data, .bss: global variables
  - heap: dynamic allocation
  - stack: local variables, parameters,…



JVM Architecture



Modifiers: public/protected/private

- Access control modifiers
  - public class modifier: all classes may access the defined class
    - public class Counter { …
    - Create a new instance, declare variable, declare parameter…
  - public method modifier: allow other classes to call the method
    - public int getCount() { …
  - public field modifier: allow other classes to use the field directly using the dot operator
    - public int count; …

Modifiers: public/protected/private

- Access control modifiers
  - protected modifier: access to the defined class is granted to
    - Classes in the same package
    - Subclasses of the class
  - private modifier: access to the defined class is granted only to the class
  - Package-private modifier (no explicit modifier): access to the defined class is granted to
    - Classes in the same package
    - Subclasses in different package cannot see

Modifiers: static

- static modifier
  - Fields, methods, and classes are associated with the class itself rather than its instances
  - static fields: the variables can be accessed without any object instances
  - static methods: the methods can be called without any object instances
  - static class (nested classes only): an instance can be created without an instance of the nesting class

```java
public class Example {
    static int count;
    int c;
    public static class A {
        public static void foo() {
            count++;
        }
    }
    public static class B {
        public void foo() {
            count++;
        }
    }
    public class C {
        public void foo() {
            c++;
        }
        public int getC() {
            return c;
        }
    }
    public C newC() {
        return new C();
    }
```

```java
public static void main(String[] args) {
    A.foo();

    //B.foo(); Error!
    B b = new B();
    b.foo();
    System.out.println("count: " + count);

    //C c = new C(); Error!
    Example e = new Example();
    C c1= e.newC();
    C c2= e.new C();
    c1.foo();
    c2.foo();
    System.out.println("e.c: " + e.c);
    System.out.println("c1.c: " + c1.getC());
    System.out.println("c2.c: " + c2.getC());
  }
}
```

Modifiers: abstract

- abstract modifier
  - abstract method: its signature is provided but without an implementation of the body
  - abstract class: a class with one or more abstract methods; must be declared as abstract
- interfaces
  - A type declaration with a list of method signatures without bodies

Modifiers: final

- final modifier
  - final variables
    - Initialized as a part of declaration
    - cannot be assigned a value again
  - final methods: cannot be overridden by a subclass
  - final classes: cannot be subclassed

Special Types (String)

- char: stores a single text character value
  - Alphabet (set of all possible characters): 16 bit unicode encoding that covers most of written languages

- String: represents a sequence of zero or more characters

```
String str = "hello";         //string literal
char chr   = str.charAt(0); //char indexing
int len    = str.length();  //length
str = str + " world";         //concatenation
```

Special Types (StringBuilder)

- String instance is immutable
  - To perform str = str + " world"; //str was "hello"
    - A new buffer for 11 characters is allocated
    - The contents of str ("hello") is copied to the new buffer
    - " world" is copied to the new buffer

- StringBuilder (mutable version of string)
  - setCharAt(k,c): changes the char at k to c
  - insert(k,s): inserts sting s to k, suffix from k is shifted
  - append(s): append s to the end of the string
  - reverse(): reverse the current string
  - toString(): return the immutable String instance

Special Types (Wrapper Types)

- Wrapper : classes derived from Object for primitive types
  - Some data structures and algorithms require Object instances, not primitive types

Base Type	*Class Name*	*Creation Example*	*Access Example*
boolean	Boolean	obj = new Boolean(true);	obj.booleanValue()
char	Character	obj = new Character('Z');	obj.charValue()
byte	Byte	obj = new Byte((byte) 34);	obj.byteValue()
short	Short	obj = new Short((short) 100);	obj.shortValue()
int	Integer	obj = new Integer(1045);	obj.intValue()
long	Long	obj = new Long(10849L);	obj.longValue()
float	Float	obj = new Float(3.934F);	obj.floatValue()
double	Double	obj = new Double(3.934);	obj.doubleValue()

Special Types (Wrapper Types)

- Automatic boxing and unboxing
  - Java provides a support for implicit conversion between base types and their wrapper types

```
int j = 8;
Integer a = new Integer(12);
int k = a;                               // implicit call to a.intValue()
int m = j + a;                           // a is automatically unboxed before the addition
a = 3 * m;                               // result is automatically boxed before assignment
Integer b = new Integer("-135");         // constructor accepts a String
int n = Integer.parseInt("2013");        // using static method of Integer class
```

Programming Assignment 1

- Implement the following 3 java classes to refresh your Java programming knowledge
  - IntMod (Integer Modulo)
  - Rat (Rational number)
  - Euclidean
- Download hw1.zip from the course website
  - Implement IntMod and Rat classes
  - Implement Euclidean algorithm
- Unit test cases are provided and your implementation should pass all test cases
- Zip IntMod.java, Rat.java, and Euclidean.java and submit the zip file through blackboard
- Due date: 3/3/2022, 11:59pm (NY time)

```java
public class App {
    public static void main(String[] args) {
        UnitTest.testInt();
        UnitTest.testIntMod();
        UnitTest.testRat();
    }
}

public class UnitTest {
...
    //test for each type
    public static void testInt() {
        System.out.println("testInt...");
        Int a = new Int(12);
        Int b = new Int(30);
        Int c = new Int(7);
        testOrdered(a, b, c);
        testRing(a, b, c);
        testEuclidean(a, b, c);
        System.out.println("testInt done");
    }
...
}
```

Interface Ordered

```java
//Ordered defines a total order
public interface Ordered {
    public boolean ge(Ordered a);   //greater than or equal to

    //default methods
    public default boolean gt(Ordered a) {      //greater than
        return this.ge(a) && this.ne(a);
    }
    public default boolean le(Ordered a) {      //less than or equal to
        return a.ge(this);
    }
    public default boolean lt(Ordered a) {      //less than
        return a.ge(this) && a.ne(this);
    }
    public default boolean eq(Ordered a) {      //equal
        return this.ge(a) && a.ge(this);
    }
    public default boolean ne(Ordered a) {      //not equal
        return !this.eq(a);
    }
}
```

Interface Ordered

```
//A helper class for comparison
public class Comp {
    public static boolean ge(Ring a, Ring b) {    //greater than or equal to
        return ((Ordered) a).ge((Ordered) b);
    }
    public static boolean gt(Ring a, Ring b) {    //greater than
        return ((Ordered) a).gt((Ordered) b);
    }
    public static boolean le(Ring a, Ring b) {    //less than or equal to
        return ((Ordered) a).le((Ordered) b);
    }
    public static boolean lt(Ring a, Ring b) {    //less than
        return ((Ordered) a).lt((Ordered) b);
    }
    public static boolean eq(Ring a, Ring b) {    //equal
        return ((Ordered) a).eq((Ordered) b);
    }
    public static boolean ne(Ring a, Ring b) {    //not equal
        return ((Ordered) a).ne((Ordered) b);
    }
}
```

Interface Ring

```
//Ring is an algebra that supports + - * operations
public interface Ring {
    public Ring add(Ring a);
    public Ring addIdentity();
    public Ring addInverse();
    public Ring mul(Ring a);

    //default methods
    public default Ring sub(Ring a) {
        return this.add(a.addInverse());
    }
}
```

Interface Field

```
//Field is an algebra that supports + - * / operations
public interface Field extends Ring {
    public Ring mulIdentity();
    public Ring mulInverse() throws ArithmeticException;

    //default methods
    public default Field div(Field a) {
        return (Field)this.mul(a.mulInverse());
    }
}
```

Interface Modulo

```
public interface Modulo {
    public Ring quo(Ring d);            //quotient

    //default methods
    public default Ring mod(Ring d) {   //remainder
        Ring q = this.quo(d);
        return ((Ring)this).sub(d.mul(q));
    }
}
```

- When a is divided by n, the quotient q and the remainder r should satisfy
  - q: integer
  - $a = n \cdot q + r$
  - $|r| < |n|$

```java
//Int represents integer
public class Int implements Ring, Modulo, Ordered {
    private int n;

    public Int(int n)              { this.n = n;}

    //interface Ring
    public Ring add(Ring a)        { return new Int(n + ((Int)a).n); }
    public Ring addIdentity()      { return new Int(0); }
    public Ring addInverse()       { return new Int(-n); }
    public Ring mul(Ring a)        { return new Int(n * ((Int)a).n); }

    //interface Modulo
    public Ring mod(Ring a)        { return new Int(n % ((Int)a).n); }
    public Ring quo(Ring a)        { return new Int(n / ((Int)a).n); }

    //interface Ordered
    public boolean ge(Ordered a) { return n >= ((Int)a).n; }

    public int getInt()            { return n; }
    public String toString()       { return String.format("%d", n); }
}
```

```java
//IntMod represents integer modulo system
//IntMod(n, m) means n modulo m, where m is a prime number.
//  IntMod(7, 5) means 7 in modulo 5 system
//  IntMod(7, 5) is equivalent to IntMod(2, 5)

public class IntMod implements Field, Ordered {
    private int n, m;

    public IntMod(int n, int m) {
        if(m <= 0)
            throw new IllegalArgumentException("Not a positive division");
        n = n % m;
        n = n < 0 ? n + m : n;  //if n is negative, make it positive
        this.n = n;
        this.m = m;
    }
    ...
    //TODO: implement interface Field
    public Ring mulInverse() throws ArithmeticException {
        //TODO: find and return x such that (x * n) % m = 1
    }
    ...
}
```

```java
//Rat represents rational number
//  Rat(10, 15) is 10/15 and is equivalent to Rat(2,3)

public class Rat implements Field, Modulo, Ordered {
    private int n, d;   //numerator, denominator
    public Rat(int numerator, int denominator) {
        if(denominator == 0)
            throw new ArithmeticException("Division by zero");
        //TODO Using Euclidean and Int, reduce numerator/denominator
        //to its lowest terms (divide them by their gcd)
    }
    ...
    //interface Modulo
    public Ring quo(Ring a) throws ArithmeticException {
        Rat r = (Rat)a;
        if(r.n == 0)
            throw new ArithmeticException("Division by zero");
        return new Rat((n*r.d) / (d*r.n), 1);
    }
}
```

```java
public class Euclidean {
    protected static Ring GCDImpl(Ring a, Ring b) {
        //TODO: implement GCDImpl by remainder
        // if a is equal to its add identity return b;
        // if b is equal to its add identity return a;
        // if a >= b then return gcd of a % b and b
        // if b >= a then return gcd of b % a and b
    }

    //Greatest Common Denominator
    public static Ring GCD(Ring a, Ring b) {
        return GCDImpl(Arith.abs(a), Arith.abs(b));
    }

    //Least Common Multiplier
    public static Ring LCM(Ring a, Ring b) {
        Ring gcd = GCD(a, b);
        Ring q = ((Modulo)b).quo(gcd);
        return a.mul(q);
    }
}
```

SUNY Korea

Expected result

```
testInt...
testOrdered: Success.
testRing: Success.
testEuclidean: Success.
testInt done
testIntMod...
testOrdered: Success.
testRing: Success.
testField: Success.
testIntMod done
testRat...
testOrdered: Success.
testRing: Success.
testField: Success.
testEuclidean: Success.
testRat done
...
```

Optional: Total Order

- Partial order $\succcurlyeq \subseteq A \times A$
  - Reflexivity: $a \succcurlyeq a$
  - Antisymmetry: $a \succcurlyeq b \Rightarrow b \not\succcurlyeq a$ if $a \neq b$
  - Transitivity: $a \succcurlyeq b$ and $b \succcurlyeq c \Rightarrow a \succcurlyeq c$

- Total order
  - For any elements $a, b \in A$, either $a \succcurlyeq b$ or $b \succcurlyeq a$

Optional: Ring

- The structure $(A, +, \cdot)$ is a ring if for all a, b, c $\in$ A, the following is satisfied
  - $a + b = b + a$ commutative law of +
  - $a + (b + c) = (a + b) + c$ associative law of +
  - Exists $z \in A$ s.t. $a + z = z + a = a$ for every $a \in A$ identity for +
  - For each $a \in A$, there is $b \in A$ s.t. $a + b = b + a = z$ inverse for +
  - $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ associative law of $\cdot$
  - $a \cdot (b + c) = a \cdot b + a \cdot c$, $(b + c) \cdot a = b \cdot a + c \cdot a$ distributive laws of $\cdot$ over +

Optional: Field

- A ring structure is called a field if every nonzero element has a multiplicative inverse
  - $a \cdot u = u \cdot a = a$ for all $a \in A$ and $a \neq z$

    u: unity or multiplicative identity
  - $a \cdot b = b \cdot a = u$

    multiplicative inverse