CSE214 Data Structures
Object-Oriented Design

YoungMin Kwon

Object-Oriented Design Goals

- Robustness
  - Correctness: correct outputs for all anticipated correct inputs
  - Robustness: handling unexpected inputs
  - E.g.) A program expecting a positive integer should be able to recover gracefully when a negative integer is given

Object-Oriented Design Goals

- Adaptability
  - Software needs to be able to evolve over time to cope with changing environments
    - E.g.) Web browsers, Internet search engines are used for many years while evolving over time.

  - Portability: ability of software to run with minimal change on different platforms (hardware and operating system)

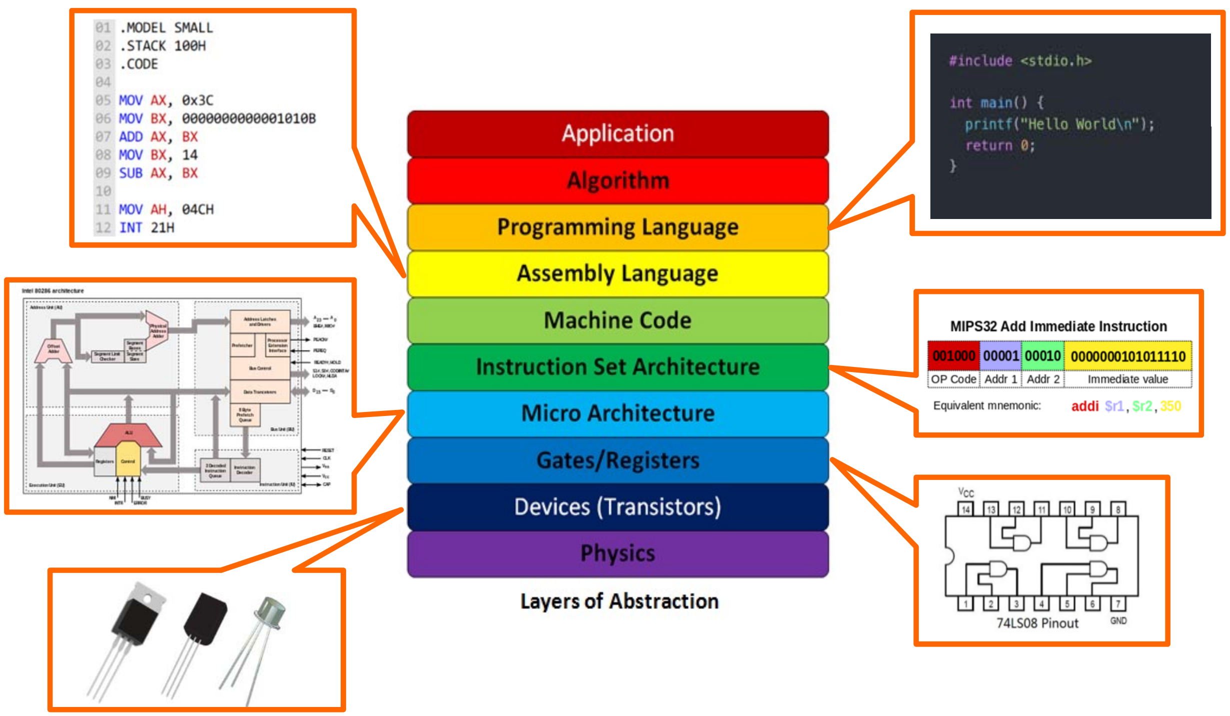
SUNY Korea

Object-Oriented Design Goals

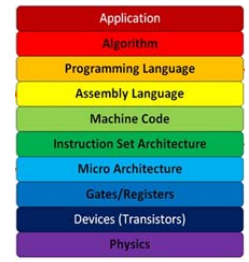
- Reusability
  - The same code should be usable as a component of different systems in various applications
  - Developing quality software can be expensive
  - The cost can be offset if the software is designed to be reused

Object-Oriented Design Principles

- Abstraction
  - Hide unwanted details and provide the most essential information



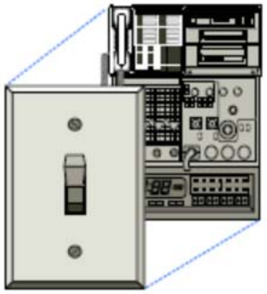
Object-Oriented Design Principles



- Abstract Data Type (ADT)
  - Abstraction in the design of data structures
    - Type of data stored
    - Operations supported on them (what but not how)
    - Type of the parameters of the operations
  - In Java, interfaces can provide ADT
    - Classes realize ADTs by implementing interfaces

SUNY Korea

Object-Oriented Design Principles



- Encapsulation
  - Provides a protection by hiding implementation details from other components
  - The only constraint a programmer should maintain is the public interfaces
    - Frees a programmer from the concern that others may depend on his/her implementations
  - It yields the robustness and the adaptability

Object-Oriented Design Principles

- Modularity
  - Organizing principle in which different components of a software system are divided into separate functional units
  - Robustness can be improved
    - Easier to test and debug separate components before they are integrated into a larger software system

Design Patterns

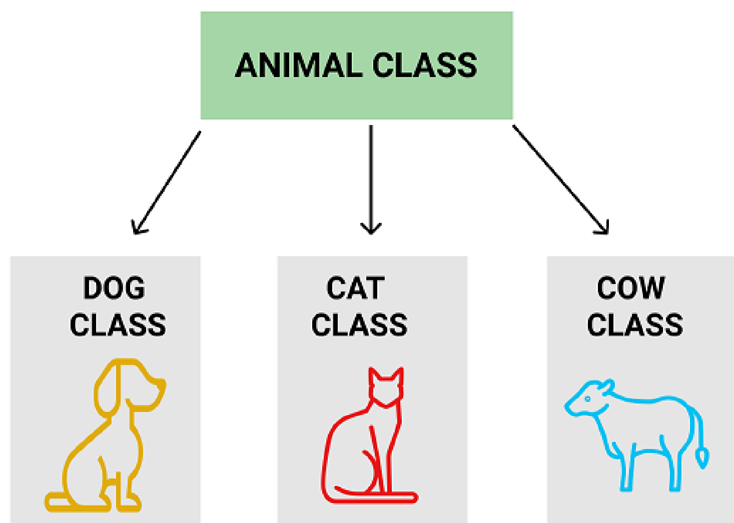
- Design Pattern
  - Pattern provides a general template for a solution that can be applied in many different situations

- Algorithm design patterns
  - Recursion
  - Amortization
  - Divide-and-conquer
  - Prune-and-search
  - Brute force
  - Greedy method
  - Dynamic Programming

- Software engineering design patterns
  - Template method
  - Factory method
  - Composition
  - Adapter (aka wrapper)
  - Position
  - Iterator
  - Comparator

Inheritance

- Inheritance
  - Define a new class based upon an existing class as a starting point
  - Organize software components in a hierarchy



Inheritance

- Terminology
  - Existing class: base class, parent class, super class
  - New class: subclass, child class
  - Subclass extends super class
    - "is a" relation: subclass is a superclass
    - Subclass can augment superclass by adding new fields or new methods
    - Subclass can specialize existing behaviors by overriding existing methods

```java
public class Animal {
    public String sound() {
        throw new UnsupportedOperationException("Not implemented");
    }

    public static class Dog extends Animal {
        public String sound() { return "Bow Bow"; }      //specialize
        public String swim()  { return "Like"; }         //augment
    }

    public static class Cat extends Animal {
        public String sound() { return "Meow Meow"; }    //specialize
        public String swim()  { return "Hate"; }         //augment
    }

    public static class Duck extends Animal {
        public String sound() { return "Quack Quack"; } //specialize
        public String swim()  { return "Love"; }         //augment
    }

    public static void main(String[] args) {
        Animal a = new Dog();                      //a is a Dog
        System.out.println(a.sound());             //Bow Bow
        System.out.println(((Dog)a).swim()); //Need casting
    }
}
```

Polymorphism

- Polymorphism (many forms)
  - Ability of a reference variable to take different forms
  - Liskov substitution principle: a variable of a class can be assigned an instance of its subclasses

```
Animal a = new Cat(); //Liskov substitution
a = new Dog();        //Liskov substitution
Dog d = (Dog) a;      //need to cast
```

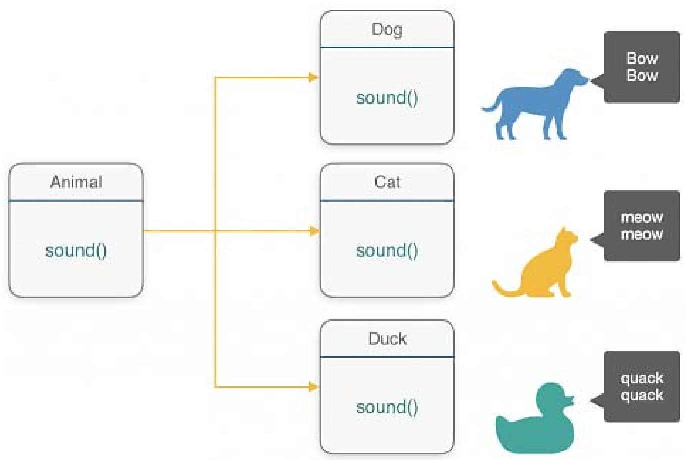
  - instanceof operator: whether "is a" relation is true

```
a instanceof Animal //true
a instanceof Dog    //true
a instanceof Cat    //false
```

SUNY Korea

Polymorphism

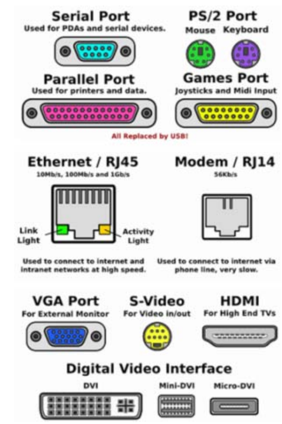
- Polymorphism
  - Dynamic dispatch: the method that is closest to the actual instance is decided at runtime




```
Animal a = new Dog();
a.sound(); //Bow Bow
```

Application Programming Interface (API)

- API
  - For two objects to interact, they know the messages that each will accept
  - In object-oriented design, the knowledge about the messages is specified as an API



- ADTs can provide API
  - An interface defining an ADT is specified as
    - A type definition
    - A collection of methods for this type
  - Strong typing: at compile time or at runtime, the types of the parameters actually passed are rigorously checked

Interfaces

- Interface
  - A main structural element in Java that enforces API
  - A concrete class has bodies of all of the methods of the interfaces it implements
    - Interfaces enforce that an implemented class has methods with certain specified signatures
- In Java, multiple inheritance is
  - Allowed for interfaces
  - Not allowed for classes
    - Diamond inheritance: confusion can arise if two base classes have fields/methods with the same name/signature

Interfaces (multiple inheritance)

```java
public interface Ring {
    public Ring add(Ring a);
    public Ring addIdentity();
    public Ring addInverse();
    public Ring mul(Ring a);
}

public interface Ordered {
    public boolean ge(Ordered a);   //greater than or equal to
}

public interface OrderedField extends Ring, Ordered {
    public Ring mulIdentity();
    public Ring mulInverse() throws ArithmeticException;
}
```

Abstract Classes

- Abstract classes
  - Serves a role in between classes and interfaces
  - Can have fields and some implemented methods
  - Can have unimplemented methods
  - Single inheritance only

```java
public abstract class Container {//abstract class
    //load in percent of volume
    protected double percentLoad;

    //abstract methods
    public abstract double volume();
    public abstract Container create();

    public double load() {
        //template method pattern
        return percentLoad / 100 * volume();
    }

    public Container split() {
        //factory method pattern
        Container c = create(); //create the same container
        double newLoad = percentLoad / 2;
        percentLoad = newLoad;
        c.percentLoad = newLoad;
        return c;
    }
}
```

```java
public static class Box extends Container {
    protected double h, w, l;

    public Box(double h, double w, double l) {
        this.h = h; this.w = w; this.l = l;
    }

    public double volume() {
        return h * w * l;
    }

    public Box create() {  //factory pattern
        return new Box(h, w, l);
    }

    public String toString() {
        return String.format("Box: h:%f, w: %f, l: %f, load: %f",
                                     h, w, l, load());
    }
}
```

```
public static class Cylinder extends Container {
    protected double r, l;

    public Cylinder(double r, double l) {
        this.r = r; this.l = l;
    }

    public double volume() {
        return 3.141592 * r * r * l;
    }

    public Cylinder create() {  //factory pattern
        return new Cylinder(r, l);
    }

    public String toString() {
        return String.format("Cylinder: r:%f, l: %f, load: %f",
                                        r, l, load());
    }
}
```

```java
    public static void main(String[] args) {
        Container c = new Box(1/*h*/, 2/*w*/, 3/*l*/);
        c.percentLoad = 100;
        Container d = c.split();
        System.out.println(c);
        System.out.println(d);

        c = new Cylinder(1/*r*/, 2/*l*/);
        c.percentLoad = 100;
        d = c.split();
        System.out.println(c);
        System.out.println(d);
    }
}
```

Result

```
Box: h:1.000000, w: 2.000000, l: 3.000000, load: 3.000000
Box: h:1.000000, w: 2.000000, l: 3.000000, load: 3.000000
Cylinder: r:1.00000, l: 2.00000, load: 3.14159
Cylinder: r:1.00000, l: 2.00000, load: 3.14159
```

Design Patterns

- Template method pattern
  - Container uses volume() that will be implemented by Container's subclasses
- Factory method pattern
  - Container uses create() that creates an instance of a subclass type

Exceptions

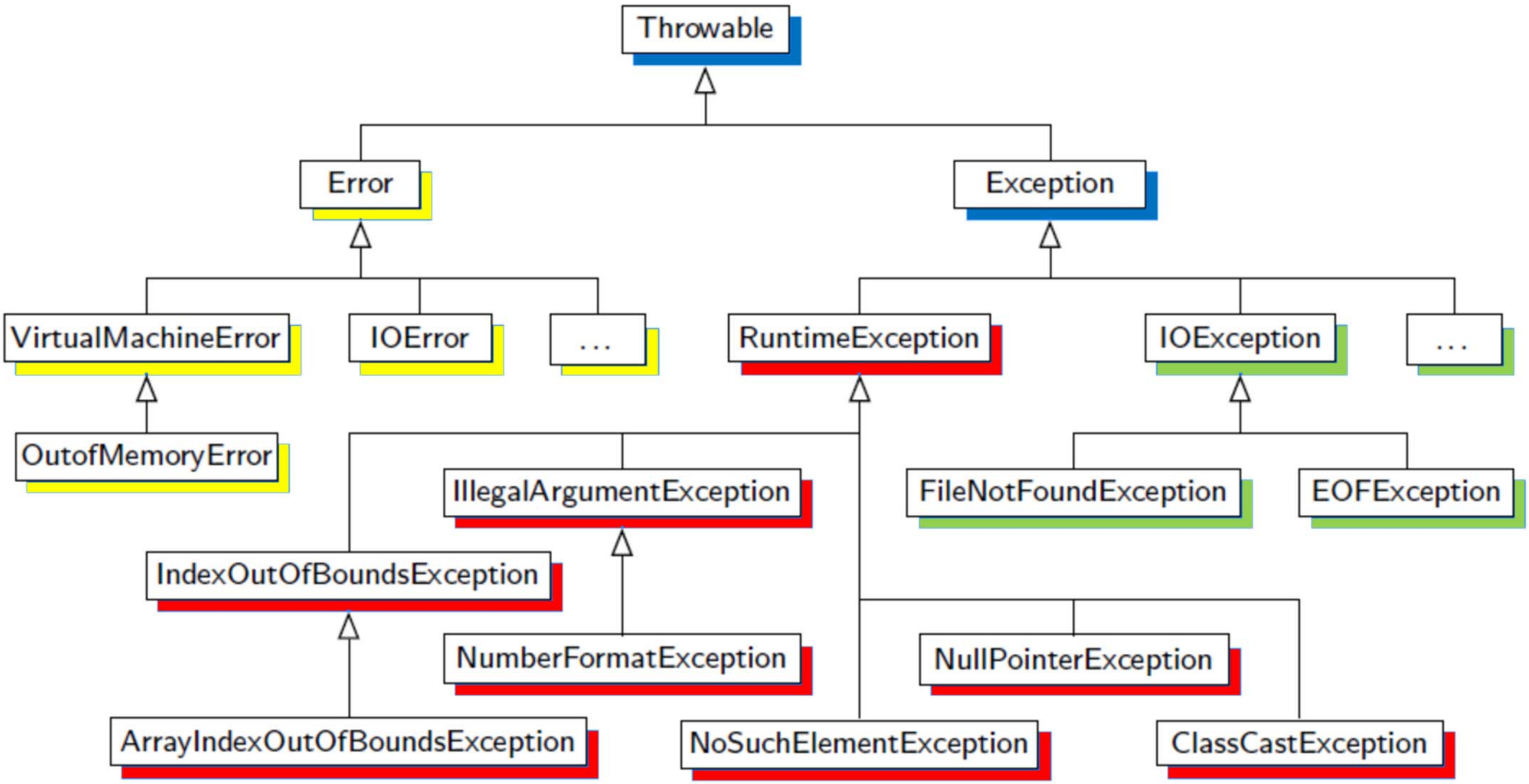
- Exceptions
  - Unexpected events that occurred (unavailable resource, unexpected input, program error,…)
- Exceptions in Java
  - Exceptions are an Object that can be thrown by
    - the code or
    - the Java Virtual Machine (run out of memory)
  - Exceptions can be caught by a surrounding block of code
    - Exception can be caught by the method caller’s surrounding block
    - Uncaught exceptions cause Java virtual machine to stop running the program

Exceptions

- Errors
  - Errors are typically thrown by JVMs for situations unlikely to be recoverable.
- Unchecked exceptions
  - Subtypes of RuntimeException
  - Due to programming logic errors
  - No need to be declared in the signature
- Checked exceptions
  - All checked exceptions that might propagate upwards from a method must be declared in its signature

```java
public class Container {
    //load in percent of volume
    protected double percentLoad;

    //unchecked exception
    public double volume() {
        throw new UnsupportedOperationException("not implemented");
    }

    //checked exception
    public double load() throws IllegalAccessException {
        throw new IllegalAccessException("you don’t have access");
    }

    public boolean isOverloaded() throws IllegalAccessException {
        return load() > volume();
    }
```

```java
    public Container add(double amount) {
        percentLoad += amount / volume() * 100;

        try {
            if(isOverloaded())
                return split();
        } catch(IllegalAccessException e) {
            e.printStackTrace();
            return null;
        } catch(Exception e) {
            e.printStackTrace();
            throw e;
        }
        return null;
    }
}
```



Throwable
Error
Exception
VirtualMachineError
IOError
...
RuntimeException
IOException
...
OutofMemoryError
IllegalArgumentException
FileNotFoundException
EOFException
IndexOutOfBoundsException
NumberFormatException
NullPointerException
ArrayIndexOutOfBoundsException
NoSuchElementException
ClassCastException

Casting (type conversion)

- Suppose that P is a super class (parent class) of C
- Widening conversion: type C $\rightarrow$ type P
  - Needs for no explicit casting

```
Container c = new Box();
```

- Narrowing conversion: type P $\rightarrow$ type C
  - Needs an explicit casting
  - May throw a ClassCastException when unsuccessful

```
void foo(Container c) {
    Box b = (Box) c; …
}
```

  - instanceof operator can check if an object is a certain type
    - `if(c instanceof Box) …`

Generics

- Java supports generic classes and methods
  - Operating on a variety of types while avoiding explicit casting
  - Use formal type parameters
    - The type parameters are used when declaring variables, parameters, and return values
    - The type parameters are specified when using the generic classes

```java
public class ObjectPair {       //without generics
    private Object first, second;
    public ObjectPair(Object a, Object b) {
        first = a; second = b;
    }
    public Object getFirst()  { return first; }
    public Object getSecond() { return second; }
}

public ObjectPair foo() {
    return new ObjectPair("YM", 10);     //composition pattern
}

public void print() {
    ObjectPair p = foo();
    String name = (String) p.getFirst(); //explicit casting
    int id = (Integer) p.getSecond();    //explicit casting
    System.out.format("%s: %s\n", name, id);
}
```

- Composition design pattern
  - To return multiple values, define a class that can hold those values

```java
public class Pair<F,S> { //generic class: type parameters F and S
    private F first;
    private S second;
    public Pair(F a, S b) { first = a; second = b; }
    public F getFirst()   { return first; }
    public S getSecond()  { return second; }
}

public Pair<String,Integer> foo() {
    //return new Pair<String,Integer>("YM", 10);
    return new Pair<>("YM", 10);
}

public void print() {
    Pair<String,Integer> p = foo();
    String name = p.getFirst();
    int id = p.getSecond();
    System.out.format("%s: %s\n", name, id);
}
```

```java
//generic function: F and S are type parameters
public static <F,S> String toStr(
    Pair<? extends F /*subclass of F*/, ? super S /*superclass of S*/> pair) {
    F name = pair.getFirst();
    Object id = pair.getSecond(); //Object is a superclass of all classes
    return String.format("%s: %s", name.toString(), id.toString());
}

public void print() {
    Pair<String,Integer> p = foo();

    //String s = Pair.<String,Integer>toStr(p);
    String s = toStr(p); //types of F, S are inferred from p
    System.out.println(s);
}
```

Nested Classes

- Nested class
  - A class defined within the definition of another class
  - Increase encapsulation

- static nested class
  - Similar to traditional classes
  - Its instance has no association with any specific instance of the outer class

- Non-static nested class (inner class)
  - Can be created from within a non-static method of an outer class
  - Inner class instance is associated with the outer class instance that creates it

```java
public class Outer {
    static int count;
    int c;
    public static class A { //nested class
        public void foo() { count++; }
    }
    public static class B { //nested class
        public static void foo() { count++; }
    }
    public class C {        //inner class
        public void foo() { c++; }
    }
    public C newC() { return new C(); }
    public static void main(String[] args) {
        A a = new A();
        a.foo();
        B.foo();
        System.out.println("count: " + count);
        //C c = new C(); error
        Outer o = new Outer();
        C c1 = o.newC();
        C c2 = o.new C();
        c1.foo();
        c2.foo();
        System.out.println("o.c: " + o.c);
    }
}
```

Programming Assignment 2

- A polynomial over a ring is a ring. For this appointment, implement the following three classes
  - PolyDbl (polynomial of double): easier one of the two
  - Poly (polynomial of fields)
  - CRC (Cyclic Redundancy Check)
- Unit test cases are provided and your implementation should pass all test cases (you still need IntMod.java, Rat.java, and Euclidean.java from the previous assignment)
- Zip PolyDbl.java, Poly.java, and CRC.java and submit the zip file through blackboard
- Due date: 3/10/2022, 11:59 pm

Programming Assignment 2

- A polynomial is represented by a coef array s.t. `coef[i]` is the coefficient for x^i.
  - E.g. $2x^3 + 5x^2 + x + 7$ is represented as `coef[0]=7, coef[1]=1, coef[2]=5, coef[3]=2`
  - Leading 0s in the coefficient array should be trimmed out (from constructors): [7, 1, 5, 2, 0, 0, 0] → [7, 1, 5, 2]
- For the remainder and quotient, use the long division algorithm

$$
\begin{array}{r}
x - 10 \\
x^2 - 2x + 1 \overline{) x^3 - 12x^2 + 0x - 42} \\
\underline{x^3 - 2x^2 + x } \\
-10x^2 - x - 42 \\
\underline{-10x^2 + 20x - 10} \\
-21x - 32
\end{array}
$$

Programming Assignment 2

- Ordered: for polynomials p and q, p ≽ q iff
  - p equals q OR
    - E.g.: [1, 2, 3] ≽ [1, 2, 3]
  - The degree of p is larger than the degree of q OR
    - E.g.: [1, 2, 3, 4] ≽ [1, 2, 3]
  - If their degrees are equal, compare the coefficients from the highest degree term
    - Let cp and cq are the first coefficients that differ, then p ≽ q iff cp ≽ cq
    - E.g.: [1, 2, 3,4] ≽ [1, 0, 3,4]

```java
public class App {
    public static void main(String[] args) {
        UnitTest.testPolyDbl();
        UnitTest.testPolyRat();
        UnitTest.testPolyIntMod();
    }
}
public class UnitTest {
    ...
    public static void testPolyRat() {
        System.out.println("testPolyRat...");
        Poly a = new Poly(new Rat[] {
                                      new Rat( 1,1), new Rat(2,1), new Rat(1,1)});
        Poly b = new Poly(new Rat[] {
                                      new Rat(-1,1), new Rat(0,1), new Rat(1,1)});
        Poly c = new Poly(new Rat[] {
                                      new Rat( 1,1), new Rat(1,1), new Rat(1,1)});
        testOrdered(a, b, c);
        testRing(a, b, c);
        testEuclidean(a, b, c);
        System.out.println("testPolyRat done");
    }
    ...
}
```

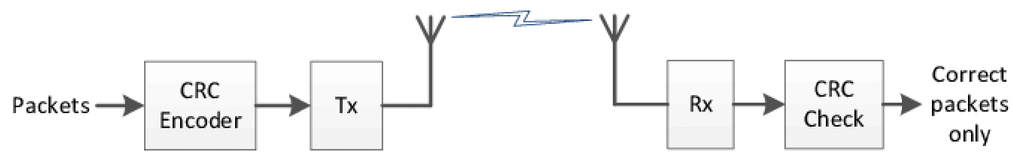
```java
public class PolyDbl implements Ring, Modulo, Ordered {
    //x^2 + 2*x + 3 is stored in coef array as [3, 2, 1]
    private double[] coef;

    public PolyDbl(double[] coef) {
        //TODO: implement the constructor
        //unnecessary zero terms should be trimmed off:
        //i.e. [3, 2, 1, 0, 0] should be [3, 2, 1]
    }

…
public class Poly implements Ring, Modulo, Ordered {
    // x^2 + 2*x + 3 is stored in coef array as [3, 2, 1]
    private Field[] coef;

    public Poly(Field[] coef) {
        //TODO: implement the constructor
        //unnecessary zero terms should be trimmed off
        int n = coef.length;
        while(n >= 2 && Comp.eq(coef[n-1], coef[0].addIdentity()))
            n--;
        this.coef = (Field[])new Field[n];
        for(int i = 0; i < n; i++)
            this.coef[i] = coef[i];
    }
```

Optional: CRC

- Cyclic Redundancy Check
  - Checks whether transmitted message has an error

  - Polynomial code
    - bit strings → polynomials with coefficients of 0 and 1
    - E.g.: 1, 1, 0, 0, 0, 1 → $x^5 + x^4 + x^0$
  - Polynomial arithmetic is done modulo 2
    - +, -, *, / on modulo 2 system
    - 0: IntMod(0, 2), 1: IntMod(1, 2)

Optional: CRC



- Sender and Receiver agree on a generator polynomial G(x)
  - G(x) begins with x^r and ends with 1: $x^r + \ldots + 1$
  - Given a G(x) a shift S(x) is x^r
- Sender: to send a message M(x)
  - Checksum C(x) = S(x) * M(x) mod G(x)
  - Transmit T(x) = S(x) * M(x) - C(x) such that T(x) mod G(x) = 0
- Receiver: receive T(x)
  - Check if T(x) mod G(x) = 0
  - M(x) = T(x) quo S(x)

```java
//Cyclic Redundancy Check
public class CRC {
    static final IntMod I = new IntMod(1, 2);
    static final IntMod O = new IntMod(0, 2);

    public static Poly sendMessage(Poly msg, Poly gen) {…}
    public static boolean checkMessage(Poly rxMsg, Poly gen) {…}
    public static Poly receiveMessage(Poly rxMsg, Poly gen) {…}
    protected static Poly shiftPoly(Poly gen) {…}
    protected static void checkPoly(Poly poly) {…}

    public static void testCRC() {
        /* expected output
          msg:       [1%2, 1%2, 0%2, 1%2, 1%2, 0%2, 1%2, 0%2, 1%2, 1%2, ]
          gen:       [1%2, 1%2, 0%2, 0%2, 1%2, ]
          shift:     [0%2, 0%2, 0%2, 0%2, 1%2, ]
          shiftMsg:  [0%2, 0%2, 0%2, 0%2, 1%2, 1%2, 0%2, 1%2, 1%2, 0%2, 1%2, 0%2, 1%2, 1%2, ]
          checksum:  [0%2, 1%2, 1%2, 1%2, ]
          txMsg:     [0%2, 1%2, 1%2, 1%2, 1%2, 1%2, 0%2, 1%2, 1%2, 0%2, 1%2, 0%2, 1%2, 1%2, ]
          rem:       [0%2, ]
          shift:     [0%2, 0%2, 0%2, 0%2, 1%2, ]
          msg:       [1%2, 1%2, 0%2, 1%2, 1%2, 0%2, 1%2, 0%2, 1%2, 1%2, ]
          testCRC Success!
         */
         …
    }

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