CSCI 4180 – Tutorial 1

Introduction to Java Programming

REN, Yanjing

yjren22@cse.cuhk.edu.hk

2022.09.14

Acknowledgement

- Slides are modified from CSCI4180 Tutorial 1, Fall 2021 by Keyun Cheng.

Outline

- Basic Topics
  - Development environment
  - Language basics
  - Classes and objects
  - `String`
  - Exceptions
- Advanced Topics
  - Nested class
  - Generics
  - Collection classes: Set, list, queue and map

Assumption:
Students should have strong **C/C++ programming** background after taking CSCI/CENG 3150.

Disclaminer:
For Java Programming, we will only cover topics that are **useful for this course**.

Basic topics

- Development environment
- Language basics
- Classes and objects
- `String`
- Exceptions

Development environment

- To install Oracle Java 8 in Ubuntu:

```
$  Download JAVA 8 from
https://www.oracle.com/java/technologies/javase/javase8u211-
later-archive-downloads.html (say jdk-8u333-linux-x64.tar.gz)
$ Unpack it to your specific path (say /opt/java)
$ Append the JAVA environment variable to ~/.bashrc
        export JAVA_HOME=/opt/java
        export PATH=$JAVA_HOME/bin:$PATH
$ “source ~/.bashrc”, then use “java –version” to test
```

- Editors:
  - **Windows:** Sublime Text, Notepad, Notepad++, gVim…
  - **Mac:** Sublime Text, Xcode, vim…
  - **Linux:** Sublime Text, gedit, vim…
  - **IDE (all platforms)**: IDEA, NetBeans, Eclipse…

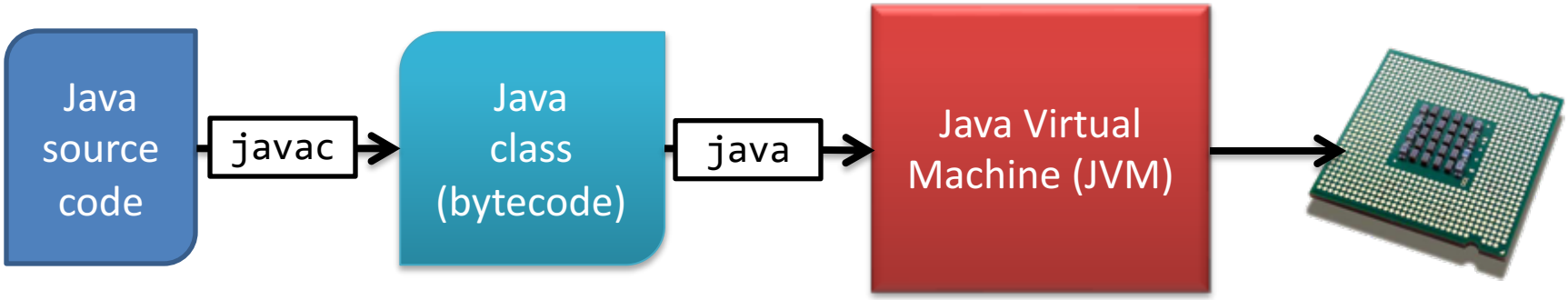
Development environment

- To compile a Java class in command-line (Mac / Linux):

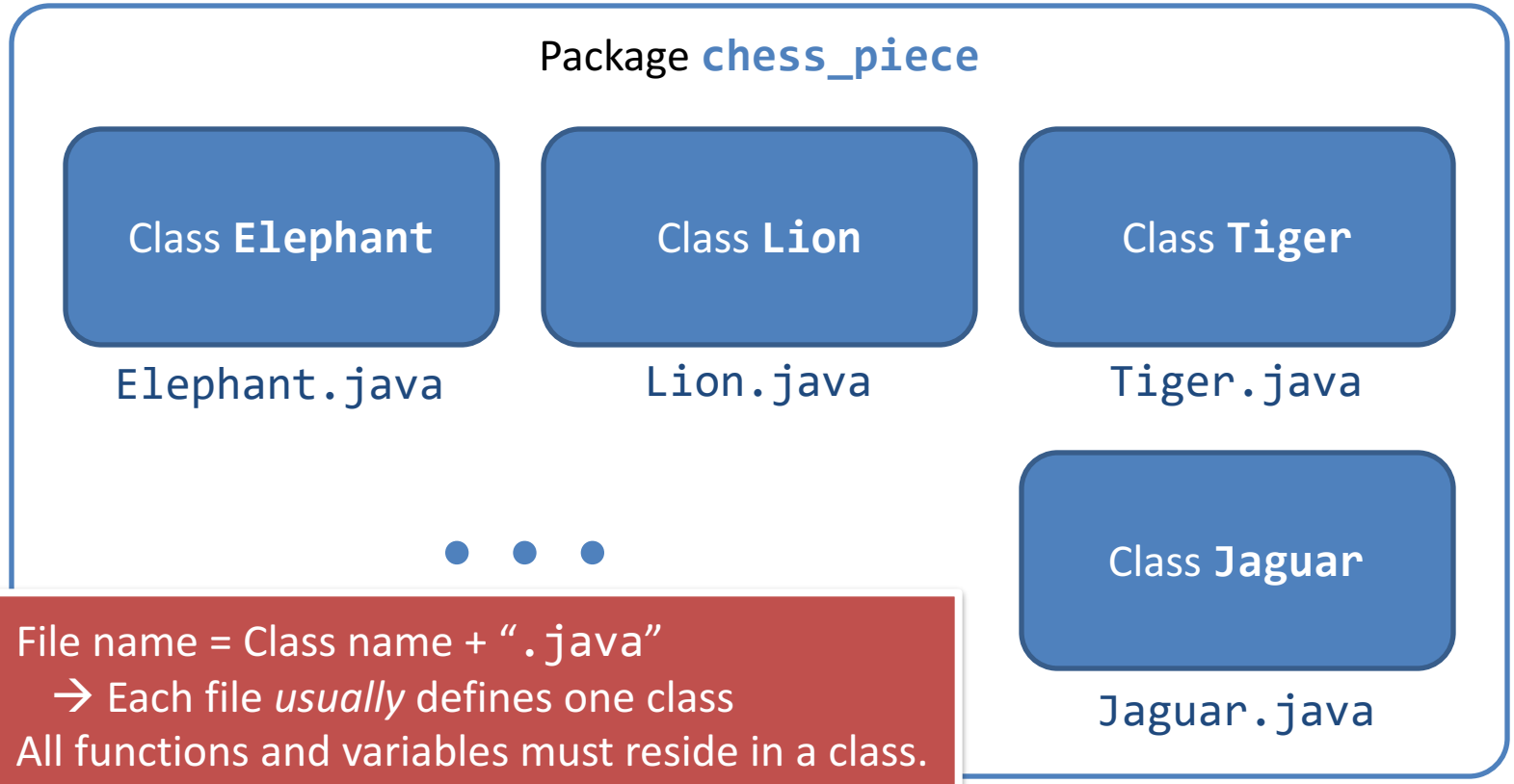
```
$ javac [Java source code(s)]
```

- To execute:

```
$ java [Java class name (without extension)]
```



Language basics: Project structure



- File name = Class name + "`.java`"
  → Each file *usually* defines one class
- All functions and variables must reside in a class.
- Class is used to create **objects**.

Language basics: Defining a class

```
package chess_piece;

import java.lang.*;
import java.util.Scanner;

public class Elephant {
        int ranking;
        static String name = "Elephant";
        public void setRanking( int ranking ) {
                this.ranking = ranking;
        }

        public int getRanking() {
                return this.ranking;
        }
}
```

Language basics: Defining a class

```
package chess_piece;

import java.lang.*;
import java.util.Scanner;

public class Elephant {
        int ranking;
        static String name
        public void setRan
                this.ranking = ranking;
        }

        public int getRanking() {
                return this.ranking;
        }
}
```

Package name

- There are many Java libraries to make life easier!
- Similar to "`#include`" in C.
- Format:
  `import` ***<package name>.<class name>;***
- Use "*" in the place of *<class name>* if you want to import all classes in this package.

Language basics: Defining a class

```
package chess_piece;

import java.lang.*;
import java.util.Scanner;

public class Elephant {
        int ranking;
        static String
        public void s
                this.r
        }

        public int ge
                return
        }
}
```

- This is the **modifier** of the class and its fields and methods.
- The default modifier is "`protected`".
- It is used to control access to other classes.
- Sorry…I won't talk about them. Just use "`public`" or simply ignore them in your assignment!
- Ref.: http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Language basics: Defining a class

```
package chess_piece;

import java.lang.*;
import java.util.Scanner;

public class Elephant {
        int ranking;
        static String name
        public void setRank
                this.rankin
        }

        public int getRanking() {
                return this.ranking;
        }
}
```

- This is an **instance variable** (non-static field).
- Their values are **unique** to each object.
- We also have **instance methods**.
  - They are invoked on an object.
  - They can access the instance variables.

Language basics: Defining a class

```
package chess_piece;

import java.lang.*;
import java.util.Scanner;

public class Elephant {
        int ranking;
        static String name = "Elephant";
        public void setRank
                this.rankin
        }

        public int getRanki
                return this
        }
}
```

- This is an **class variable** (static field).
- Its value is shared among all objects.
- Of course, we also have **class methods**.
  - They are invoked on a class.
  - They can access the class variables.
  - But, they **CANNOT** access instance variables!

Language basics: Defining a class

```
package chess_piece;

import java.lang.*;
import java.util.Scanner;

public class Elephant {
        int ranking;
        static String name =
        public void setRankin
                this.ranking =
        }

        public int getRanking
                return this.ra
        }
}
```

- They are the data types of the variables / return types of the methods.
- They can be primitive types or a class.
- Primitive types:
  - SIGNED Integer:
    - `byte` (8 bit)
    - `short` (16 bit)
    - `int` (32 bit, default)
    - `long` (64 bit)
  - Real number:
    - `float` (32 bit)
    - `double` (64 bit, default)
  - Character: `char`
  - Boolean: `boolean`

Language basics: Defining a class

```
package chess_piece;

import java.lang.*;
import java.util.Scanner;

public class Elephant {
        int ranking;
        static String name = "Elephant";
        public void setRanking( int ranking ) {
                this.ranking = ranking;
        }

        public int getRanking() {
                return this.ranking;
        }
}
```

Use “this” to access the current object.

Language basics: Operators

- Simple assignment operator: **=**
- Arithmetic operators: **+**, **-**, *****, **/**, **%**
- Unary operators: **+**, **-**, **++**, **--**, **!**
- Equality and relational operators: **==**, **!=**, **>**, **>=**, **<**, **<=**
- Conditional operators: **&&**, **||**, **? :**
  - *[True/false statement]* **?** *[Expression if true]* **:** *[Expression if false]*
- * Type comparison operator: **`instanceof`**
- * Bitwise and bit shift operators: **~**, **<<**, **>>**, **>>>**, **&**, **^**, **|**

Ref.: http://docs.oracle.com/javase/tutorial/java/nutsandbolts/opsummary.html

Language basics: Operators

Type Comparison Operator	
`instanceof`	Compares an object to a specified type
Bitwise and Bit Shift Operators	
~	Unary bitwise complement
<<	Signed left shift
>>	Signed right shift
>>>	Unsigned right shift
&	Bitwise AND
^	Bitwise exclusive OR *(XOR)*
|	Bitwise inclusive OR

Language basics: Control flow statements

- Branching
  - `if-then`, `if-then-else`
  - `switch`
- Repetition
  - `while`, `do-while`
  - `for`
  - `break`, `continue` statements
- *Same as C, so you must know them well* ☺

Language basics: The `main()` method

```
public class Main {
    public static void main( String[] args ) {
        System.out.println( "Hello World!" );
    }
}
```

“Hello World!” in Java

```
#include <stdio.h>

int main( int argc, char **argv ) {
    printf( "Hello World!\n" );
    return 0;
}
```

“Hello World!” in C

Language basics: The `main()` method

```
public class Main {
    public static void main( String[] args ) {
        System.out.println( "Hello World!" );
    }
}
```

"Hello World!" in Java

In Java, we **do not** have a return type in `main()`.
Question: How to notify the system for the error in the program?

```
#include <stdio.h>

int main( int argc, char **argv ) {
    printf( "Hello World!\n" );
    return 0;
}
```

"Hello World!" in C

Language basics: The `main()` method

```
public class Main {
    public static void main( String[] args ) {
        System.out.println( "Hello World!" );
    }
}
```

"Hello World!" in Java

In Java, the command-line arguments are passed into a `String` array.

```
#include <stdio.h>

int main( int argc, char **argv ) {
    printf( "Hello World!\n" );
    return 0;
}
```

"Hello World!" in C

Language basics: Command-line arguments

```
public class CmdArgs {
    public static void main( String[] args ) {
        System.out.print("Number of command-line arguments: ");
        System.out.println( args.length );

        for ( int i = 0; i < args.length; i++ ) {
            System.out.print( "Argument #" + i + ": " );
            System.out.println( args[ i ] );
        }
    }
}
```

Number of arguments

The *i*-th argument

`args/CmdArgs.java`
(C version: **`args/CmdArgs.c`**)

Language basics: Command-line arguments

```
$ javac CmdArgs.java                    # Compile the Java program
$ make CmdArgs                          # Compile the C program
cc     CmdArgs.c   -o CmdArgs
$ java CmdArgs there are some arguments
Number of command-line arguments: 4
Argument #0: there
Argument #1: are
Argument #2: some
Argument #3: arguments
$ ./CmdArgs there are some arguments
Number of command-line arguments: 5
Argument #0: ./CmdArgs
Argument #1: there
Argument #2: are
Argument #3: some
Argument #4: arguments
```

Can you see the difference?

Language basics: Arrays

- Arrays can be of **primitive** types, **class** types, or even another **array** types
- Declaration:
  - `int[] intArray;`
- Creation:
  - `intArray = new int[ 10 ];`
  - Different from C, you can use a variable to specify the array size!
  - Creation by enumerating its values:
    `char[] vowels = { 'a', 'e', 'i', 'o', 'u' };`
- Getting the size of an array: "`intArray.length`"
- Other operations are similar to C

Language basics: Multi-dimensional arrays

- Declaration:
  - `int[][] intArray;`
- Creation:
  - `intArray = new int[ 10 ][ 10 ]; // An 10x10 matrix`
  - Creation by enumerating its values:

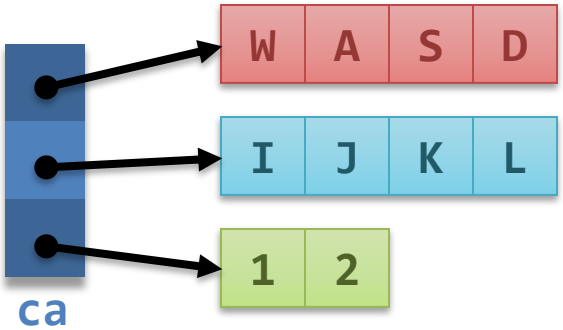
```
char[][] ca = { { 'W', 'A', 'S', 'D' },
                { 'I', 'J', 'K', 'L' },
                { '1', '2' }
              };
```

array/Array.java

Unequal size!? What's happening?

Language basics: Multi-dimensional arrays

- `ca` is actually an array of `char[]`…




```
char[][] ca = { { 'W', 'A', 'S', 'D' },
                { 'I', 'J', 'K', 'L' },
                { '1', '2' }
              };
```

array/Array.java

- So, a “matrix” is not really a matrix…(`array/Matrix.java`)

Language basics: Summary

- Project structure
- Defining a Java class
  - Defining and importing packages
  - Instance/Class fields & methods
- Primitive data types
- Operators
- Control flow statements
- The `main()` method
  - Processing command-line arguments
- Arrays

Classes and objects: Object creation

- You already know how to define a class ☺
- To create (*instantiate*) an object, use "new"
  - `Elephant e = new Elephant();`
- If the constructor asks for some arguments, invoke like this:
  - `Elephant e = new Elephant( "Dumbo", 5 );`
- <u>Note:</u> Constructor is a "special" method for initializing an object.

I will talk about what **e** is later.

```
public class Elephant {
    // ...
    public Elephant( String name, int ranking ) {
        this.name = name;
        this.ranking = ranking;
    }
// ...
```

No return type

Arguments of the constructor

Classes and objects: Inheritance

- I will skip the OOP concepts!
- Yet, you need to know how to **inherit** a class (i.e., defining a subclass).
  - You need to inherit classes in writing MapReduce programs.
- Suppose that we want to create a **subclass** of **`Elephant`** called "**`ElephantBaby`**".
  - Define the **`ElephantBaby`** class as follows:

```
public class ElephantBaby extends Elephant {
        // ...
}
```

Classes and objects: Inheritance

- "All" fields and methods will be inherited from the **superclass**, namely **`Elephant`**
  - In other words, **`ElephantBaby`** contains "all" variables (`ranking`, `name`) and methods of Elephant (`getRanking()`, `setRanking()`)
  - "All" = fields with modifier "`protected`" or "`public`"
    - `private` fields are NOT inherited
    - But, you can forget about `private` in this course, and always use `public`

Classes and objects: Object reference

- Pointers in C:

```
int num = 4180;
int *num_ptr = &num;
printf( "%p\n", num_ptr );
printf( "%d\n", *num_ptr );
```

- In Java, **NO POINTERS**!
  - A variable holds either a **primitive value** or an **object reference**
- An object reference variable holds the **address** of an object
  - E.g., Elephant **e** = new Elephant();
  - **e** is the object reference

Classes and objects: Object reference

```
public class CloneDemo {
    public static void main( String[] args ) {
        Elephant e1 = new Elephant( 5 );
        Elephant e2;

        e2 = e1; // Copy e1 to e2
        e2.setAge( 3 );

        System.out.println( "e1's age: " + e1.getAge() );
        System.out.println( "e2's age: " + e2.getAge() );
    }
}
```

reference/wrong/CloneDemo.java

```
$ java CloneDemo
e1's age: 3
e2's age: 3
```

What's wrong!?

Classes and objects: Object reference

Correct Version

```
public class CloneDemo {
    public static void main( String[] args ) {
        Elephant e1 = new Elephant( 5 );
        Elephant e3;

        e3 = e1.clone();
        e3.setAge( 3 );

        // ...
    }
}
```

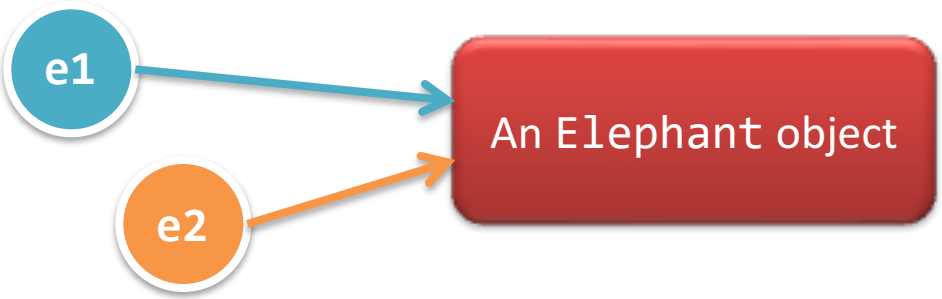
reference/right/CloneDemo.java

```
public class Elephant {
  // ...
  public Elephant clone() {
      Elephant copy =
          new Elephant( this.age );
      return copy;
  }
}
```

reference/right/Elephant.java

Classes and objects: Object reference

- Lessons learnt:
  - Copying an object reference ≠ copying an object
  - Copied object reference points to the **same object**



  - To copy an object, you need to implement a **dedicated** method (e.g., `clone()` in the `Elephant` class)
    - Step 1: **`new`** an object
    - Step 2: Copy the fields from the current object to the new object

Classes and objects: Object reference

- `free()`?
  - **new** = `malloc()` in C
  - How to **release** allocated memory (`free()` in C) for an object?
- In Java, there is **NO WAY** to "`free`"!
  - The Java Virtual Machine has a **garbage collector (GC)**
  - The GC frees the objects that are not assigned with object references **periodically**

```
Elephant e1 = new Elephant( 5 );
// ...
e1 = null;
```

Tell the JVM that you don't need the Elephant object anymore such that the JVM can kill it later.

String

Common string operations	*(Note: All of them are object methods)*
`charAt( int index )`	Returns the char value at the specified index
`compareTo(String anotherString)`	Compares two strings lexicographically
`concat(String str)`	Concatenates the specified string to the end of this string (Alternatively, you can write: "**`str1 + str2`**" to concatenate `str2` to `str1`)
`equals(Object anObject)`	Compares this string to the specified object
`length()`	Returns the length of this string

Ref.: http://docs.oracle.com/javase/8/docs/api/java/lang/String.html

Exceptions

- How do you handle unexpected conditions or errors?
- In C, you ignore (or forget to handle) them!
  - And by **Murphy's Law**, the program crashes during demo ☹
- In Java, you must handle them, or…**compilation error**
  - Ignoring them is one possible way to *handle* them (!?)
- `Exception` in Java
  - **Unexpected conditions or errors**
  - Some common exceptions are already defined: `IOException`, `ArithmeticException`, `EOFException`, etc.

Exceptions

- In any method, we may:
  - **Raise** exception (**`throw`**)
    - `throw new` *<Exception Class Name>*`();`
    - Used for indicating the errors
    - Propagate until being handled
    - The program terminates if not handled
  - **Detect** (**`try`**) and **handle** (**`catch`**) exceptions
    - `try { … } catch (` *<Exception Class Name to be Caught>* `) { … }`
    - Stop the propagation of the exception
    - Save the program from terminating
  - **Ignore** exceptions (**`throws`**)
    - `throws` *<Exception Class Name>*
    - Allow the exception to propagate

Exceptions

As we want the caller to handle this exception, we simply **propagate** it instead of handling it.

```
import java.io.IOException;

public class Processor {
    public void check( String[] args ) throws IOException {
        if ( args.length == 0 ) {
            throw new IOException( "Insufficient arguments." );
        }
    }
}
```

Error occurs! Let's throw an exception.

exception/Processor.java

Exceptions

```
import java.io.IOException;

public class Demo {
    public static void main( String[] args ){
        Processor proc = new Processor();

        try {
            proc.check( args );
        } catch ( Exception e ) {
            System.err.println( e );
            System.exit( 1 );
        }
        // ...
    }
}
```

Statement(s) that may cause exception.

If exception is thrown, the remaining statements of the `try` block will be skipped and the corresponding `catch` block will be executed.

exception/Demo.java

Advanced topics

- Nested class
- Generics
- Collection classes
  - Set
  - List
  - Queue
  - Map

Nested class

- Nested class = The class defined within another class
  - Nested classes declared static = **static nested classes**
  - Non-static nested classes = **inner classes** *(not required in this course)*

```
class OuterClass {
    // ...
    static class StaticNestedClass {
        // ...
    }
}
```

Static nested class

```
class OuterClass {
    // ...
    class InnerClass {
        // ...
    }
}
```

Inner class

- In MapReduce programs, we define a job as a class, with the **Mapper** and **Reducer** defined as **static nested classes**

Static nested class

```
public class Computer {
    CPU intel, amd;
    RAM ram;

    // Static nested class "CPU"
    public static class CPU {
        public int run( int a, int b, char operation ) {
            // ...
        }
    }
// ...
```

nested_class/Computer.java

Just put the class (declared `static`) inside another class. Simple enough?

Static nested class

- So, what's happening behind the static nested class?

```
$ javac Computer.java
$ ls
./  ../  Computer.class  Computer$CPU.class  Computer.java
Computer$RAM.class
```

The static nested classes are compiled into a separate `.class` file (<Outer class name>**$**<Inner class name>`.class`)

Generics

- Suppose we want to implement a linked list of `int`, `char` and `String`
- Isn't it ***silly*** to implement the same thing into three classes: `LinkedListInt`, `LinkedListChar` and `LinkedListString`!?
  - Can we **customize** the data type(s) (in this example, `int`, `char` and `String`) at **compile-time**?
- Yes, it is called **Generics** in Java
  - **Type parameters** are used to represent the customizable data types

Generics

```
public class Data<CustomType> {
    public CustomType data;

    public Data( CustomType d ) {
        data = d;
    }

    public CustomType getData() {
        return data;
    }

    public void setData( CustomType d ) {
        data = d;
    }
}
```

generics/Data.java

CustomType is the type parameter of this class (Surrounded by "<>"). Its value is replaced by a real type later.

We use CustomType to represent the *not-yet-known* data type in the class.

Generics

```
public class GenericsDemo {
    public static void main( String[] args ) {
        Data<String> s = new Data<String>( "String" );
        Data<Integer> i = new Data<Integer>( new Integer( 4180 ) );
        System.out.println( "Data<String> s : " + s.getData() );
        System.out.println( "Data<Integer> i : " + i.getData() );
    }
}
```

generics/Data.java

Supply the data type when using the generic class using "<>".

Detour

Sometimes you need a class to represent a primitive type.

In Java, you don't need to write it yourself – there are many **primitive wrapper classes**:

`Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, `Boolean`.

See http://en.wikipedia.org/wiki/Primitive_wrapper_class for their operations.

Generics

- Of course, you can use multiple type parameters:

```
public class Data<CustomTypeA, CustomTypeB> {
    public CustomTypeA dataA;
    public CustomTypeB dataB;
    // ...
}
```

```
Data<String, Integer> s = new Data<String, Integer>();
```

Collection classes

- The **Java Collections Framework** provides four major categories of data structures:
  - **Set**: A collection that contains **no duplicate elements**
    - E.g., `HashSet`
  - **List**: An **ordered** collection (or sequence) which allow duplications
    - E.g., `ArrayList`, `LinkedList`
  - **Queue**: A collection for holding elements **prior to processing**
    - E.g., `PriorityQueue`
  - **Map**: An object that **maps keys** (which are unique) **to values**
    - E.g., `HashMap`

Ref.: http://docs.oracle.com/javase/tutorial/collections/implementations/index.html

Collection classes: Demo (HashMap)

```
import java.util.HashMap;

public class HashMapDemo {
    public static void main( String[] args ) {
        HashMap<String, String> map = new HashMap<String, String>();
        map.put( "Facebook", "1 Hacker Way" );
        map.put( "Microsoft", "One Microsoft Way" );
        map.put( "Apple", "1 Infinite Loop" );

        for ( String key : map.keySet() ) {
            System.out.println( key + " --> " + map.get( key ) );
        }
    }
}
```

collections/HashMapDemo.java

Collection classes: Demo (HashMap)

```
import java.util.HashMap;

public class HashMapDemo {
    public static void main( String[] args ) {
        HashMap<String, String> map = new HashMap<String, String>();
        map.put( "Facebook", "1 Hacker Way" );
        map.put( "Microsoft"
        map.put( "Apple", "1

        for ( String key : map.keySet() ) {
            System.out.println( key + " --> " + map.get( key ) );
        }
    }
}
```

Specify the **type parameters**. In this example, a String-to-String map is created.

collections/HashMapDemo.java

Collection classes: Demo (HashMap)

```
import java.util.HashMap;

public class HashMapDemo {
    public static void main( String[] args ) {
        HashMap<Strin                                    tring>();
        map.put( "Fac
        map.put( "Mic
        map.put( "App

        for ( String key : map.keySet() ) {
            System.out.println( key + " --> " + map.get( key ) );
        }
    }
}
```

put() and keySet() are the operations provided in HashMap. You need to check the Javadoc (like manpage in C) when you use them.

collections/HashMapDemo.java

Collection classes: Demo (HashMap)

```
import java.util.HashMap;

public class HashMapDemo {
    public static void main( String[] args ) {
        HashMap<Strin                                                    ();
        map.put( "Fac
        map.put( "Mic
        map.put( "App

        for ( String key : map.keySet() ) {
            System.out.println( key + " --> " + map.get( key ) );
        }
    }
}
```

This is a `for-each` loop which is designed for iterating a collection of elements. The meaning of this expression is:

For each element *key* in `map.keySet()`, …

collections/HashMapDemo.java

Supplementary Notes

Topics that are basic, but not required in this course :

- Bitwise and bit shift operators
- `instanceof`
- Console I/O

Bitwise and bit shift operators

- You should know most of them after taking CSCI/CENG 3420 (for CS/CE major students) or the digital logic course…

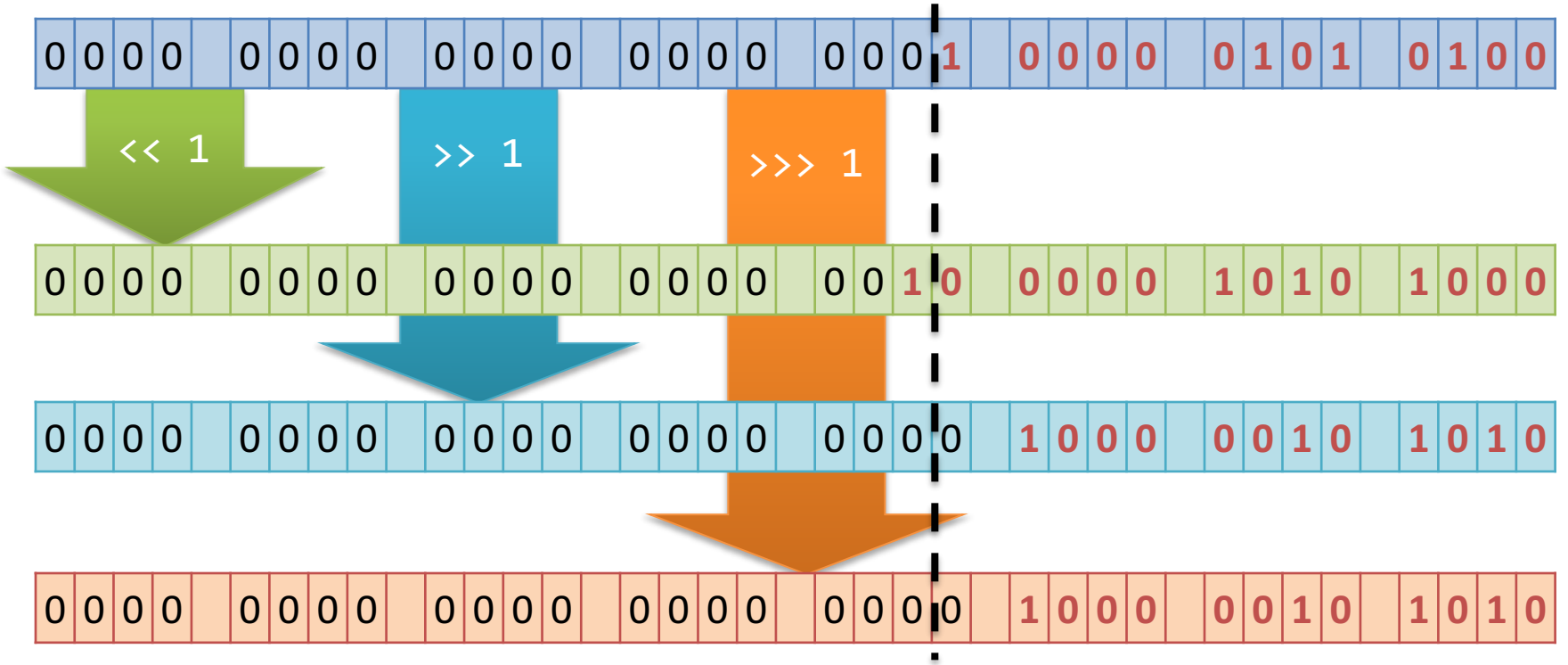
~	Unary bitwise complement
<<	Signed left shift
>>	Signed right shift
>>>	**Unsigned** right shift
&	Bitwise AND
^	Bitwise exclusive OR *(XOR)*
|	Bitwise inclusive OR

What is this?

- Demo program: `bitshift_ops/BitDemo.java`

Bitwise and bit shift operators: Examples

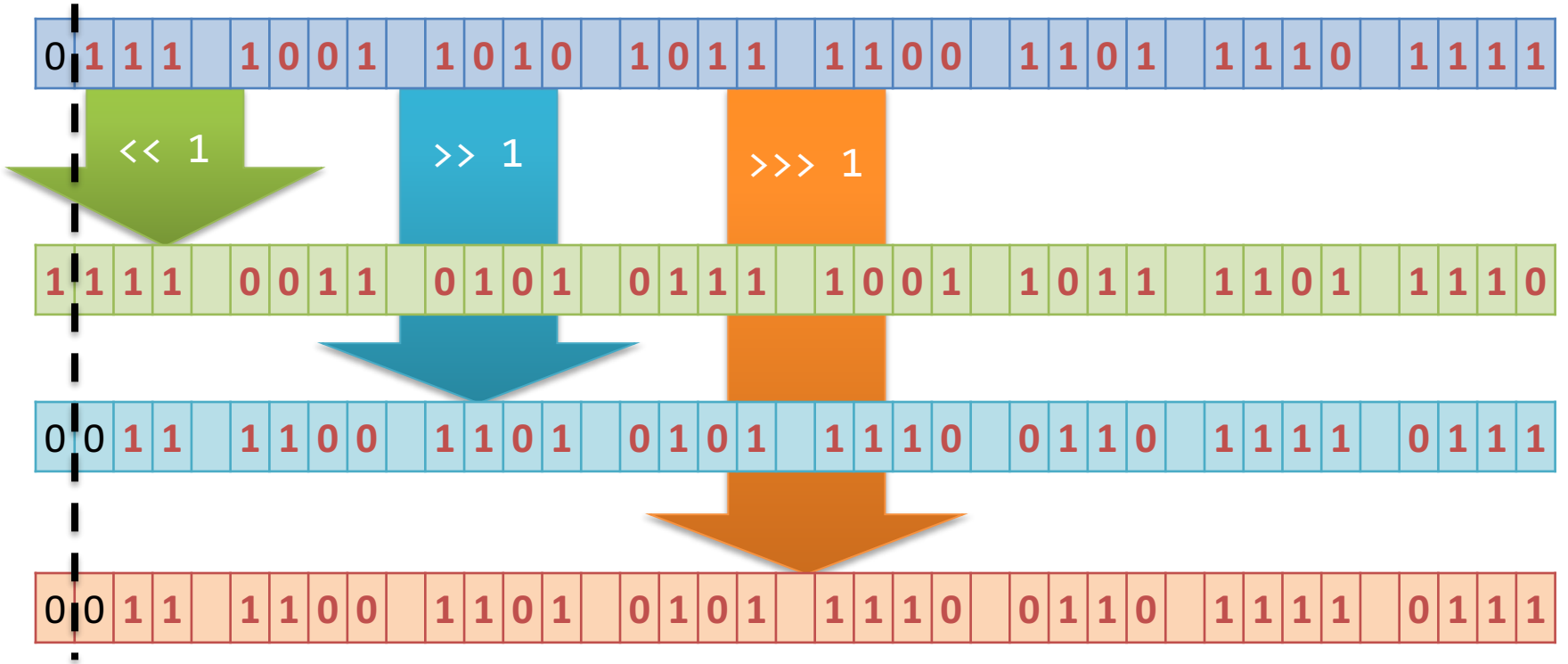
Number: $(4180)_{10}$ = (0000 0000 0000 0000 0001 0000 0101 0100)$_2$



What is the difference between “>>” and “>>>”?

Bitwise and bit shift operators: Examples

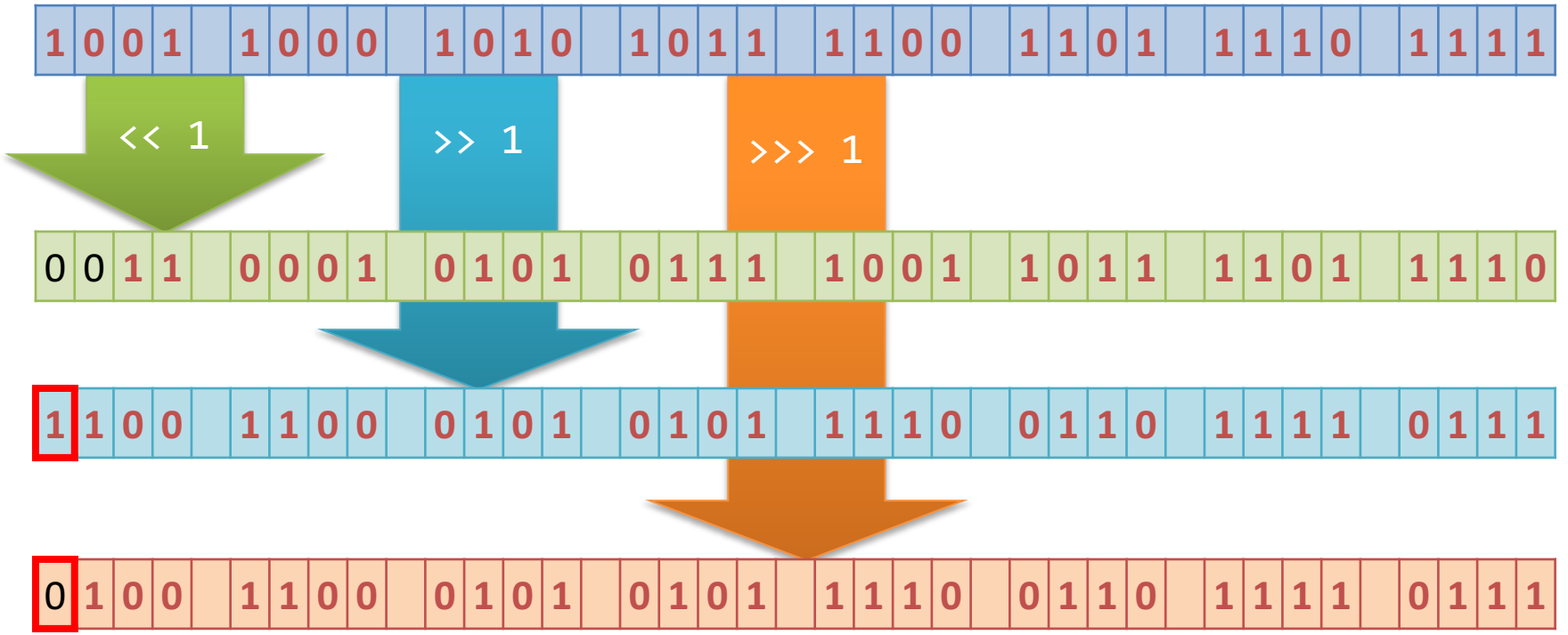
Number: $(2041302511)_{10} = (0111\ 1001\ 1010\ 1011\ 1100\ 1101\ 1110\ 1111)_2$



Again, what is the difference between “>>” and “>>>”?

Bitwise and bit shift operators: Examples

Number: $(-1733571089)_{10} = (1001\ 1000\ 1010\ 1011\ 1100\ 1101\ 1110\ 1111)_2$



You should see the difference between “>>” and “>>>” now!

Bitwise and bit shift operators: Examples

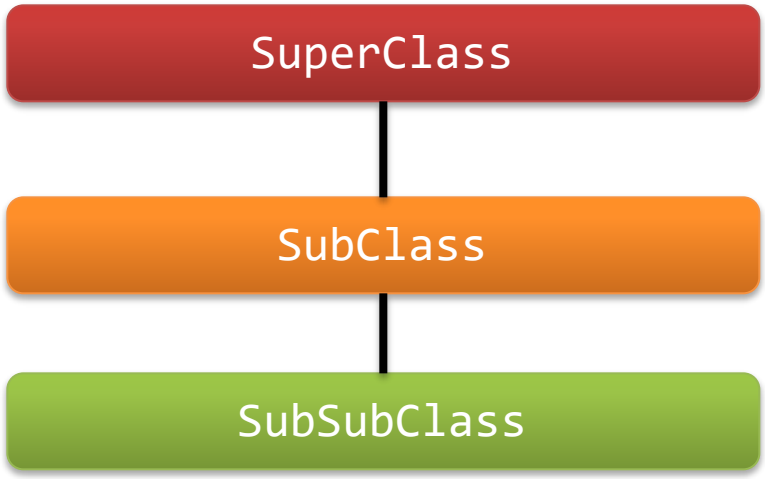
Number: $(-1733571089)_{10} = (1001\ 1000\ 1010\ 1011\ 1100\ 1101\ 1110\ 1111)_2$

- >>: Signed right shift
  - When the number is right-shifted, the sign is **preserved**.
  - In other words, negative integer will still be negative after right-shifted.
  - Yet, there is **no difference** from “>>>” for **positive integers**.
- >>>: ***Unsigned*** right shift
  - After right-shifted, the **most significant bit** is always 0.
  - In other words, negative integer will become positive after right-shifted.
  - It **does not consider the sign**, so it is “unsigned” operations.

You should see the difference between “>>” and “>>>” now!

`instanceof`

- As mentioned before, `instanceof` is used for checking whether an object is an instance of a class
- Suppose we have the following **class hierarchy**. If we use these classes to create objects, are they also the instance of their **superclasses**?



instanceof

- Demo program: `instanceof/*.java`

instanceof	SuperClass	SubClass	SubSubClass
SuperClass object	True	False	False
SubClass object	True	True	False
SubSubClass object	True	True	True

Console I/O

- Do you remember them?
  - `stdin`
  - `stdout`
  - `stderr`
- In Java, they are represented as objects in the `System` class
  - `System.in`: The standard input stream
  - `System.out`: The standard output stream
  - `System.err`: The standard error output stream

Console I/O: `System.[out|err]`

- `System.out` and `System.err` are both `PrintStream` objects
- Three commonly used methods:
  - `printf()`: The C-style `printf()` (which you know it well)
  - `print()`: Print anything that can be converted to String without a line separator at the end
  - `println()`: Print anything that can be converted to String with a line separator ("`\r\n`" or just "`\n`" depending on the OS) at the end
  - http://docs.oracle.com/javase/8/docs/api/java/io/PrintStream.html
- Demo program: **`console_io/ConsoleOutputDemo.java`**

Console I/O: `System.in`

- `System.in` is a `InputStream` object
- To read input from the console, you can use the "low level" `read()` method to read it into a `byte` array
- You can also use the `Scanner` class to help you (and this is the recommended way)!
  - Create a `Scanner` object from `System.in`:

```
Scanner scanner = new Scanner( System.in );
```

  - Check if the next input exists and is an integer:

```
if ( scanner.hasNextInt() ) {
        // ...
}
```

Console I/O: `System.in`

- You can also use the `Scanner` class to help you (and this is the recommended way)!
  - Read the next integer in the input:

```
i = scanner.nextInt();
```

  - Check if the input is not ended:

```
if ( scanner.hasNextLine() ) {
        // ...
}
```

  - Read the next line in the input:

```
i = scanner.nextLine();
```

Console I/O: `System.in`

- Demo program: **`console_io/ConsoleInputDemo.java`**
- *Reminder:* You need to **import** the `Scanner` class before you use it!

```
import java.util.Scanner;
```

- Reference:
  - http://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html
  - http://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html

CSCI 4180 – Tutorial 1

Introduction to Java Programming

– End –

References:

- http://docs.oracle.com/javase/tutorial/index.html
- http://docs.oracle.com/javase/8/docs/api/overview-summary.html